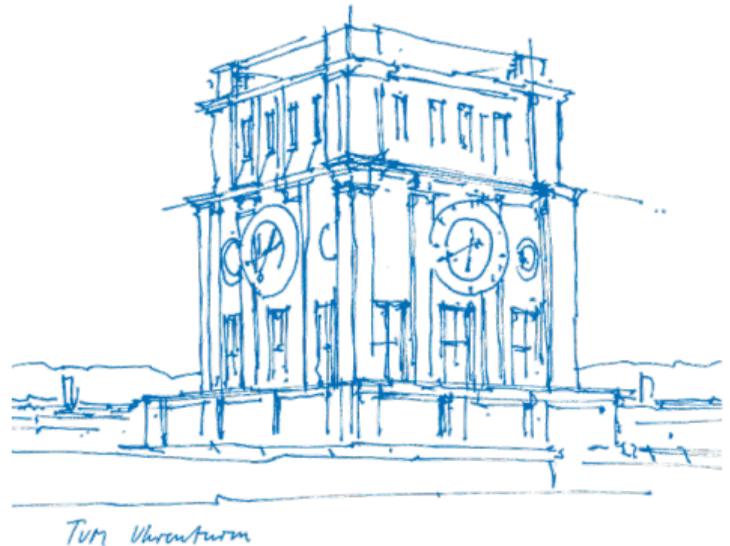# A Wishlist for Faster LLVM Back-Ends

Alexis Engelke
engelke@tum.de

(with contributions from Tobias Stadler)

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

EuroLLVM '24, Vienna, AT, 2024-04-11

TUM Uhrenturm

1

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 ⇒ low startup latency
▶ Developer experience: faster develop–test roundtrip
  ▶ (Also needs to consider front-end)

This talk:

Analyze -O0 back-end pipeline and outline possible improvements

Step 1: LLVM-IR Passes

- ▶ Prepare LLVM IR for back-end, 15–20 passes
    - ▶ Lower constant intrinsics (`is.constant`, `objectsize`),
      expand atomic operations, large divisions, . . .
    - ▶ x86: lower AMX types, float conversions

- ▶ Passes typically look for some simple instruction pattern and rewrite it
- ▶ Iterating over LLVM-IR is not free: ∼0.3% of compile time per iter.
- ▶ Many of the patterns occur rarely/not at all, but passes always run

- ⤳ Merge passes with shared pattern matching infrastructure?
- ⤳ Only run passes when required (or add an option to disable)?

IR
Pass

Step 2: Instruction Selection

▶ Transform LLVM IR into SSA-based Machine IR
  ▶ FastISel: handle common cases in single step   *←we want this*
  ▶ SelectionDAG: rewrite to graph, match patterns, schedule into MIR
  ▶ GlobalISel: rewrite to generic MIR, rewrite gMIR twice, rewrite to MIR

▶ ISel performance is only *ok-ish* when staying on the happy FastISel path

⤳ Somehow derive single-step ISel for GlobalISel?
  ▶ Downsides: maintenance effort, testing, etc.
⤳ Please don't prematurely replace FastISel with GlobalISel

IR Pass	ISel

Step 3: (Up To) Register Allocation

▶ Several passes to assign registers and stack slots
  ▶ Allocate stack slots, destruct SSA, handle two-address instructions
  ▶ Actual register allocation: linear and greedy (`RegAllocFast`)
  ▶ x86: handle flag copies (needs DomTree), AMX tiles, FPU stack

▶ Multiple rewrites of Machine IR are expensive

⤳ Don't rewrite MIR that often?
  ▶ Would require larger effort, probably not realistic

IR Pass	ISel	RegAlloc

Step 4: Miscellaneous Changes and Fix-ups

ΠΠΠ

- ▶ Insert prologue/epilogue and rewrite stack references
- ▶ Dozens of mostly target-specific passes
  - ▶ Insert CFI instructions, `patchable-function`
  - ▶ x86: add vzeroupper, compress encoding / AArch64: errata workarounds, . . .

- ▶ Most passes are individually cheap, several do typically nothing
- ▶ But: adds up nonetheless – are all passes strictly required?
  - ▶ Example: at `-O0` we don't care about EVEX-to-VEX compression

⤳ Reduce number of passes?

IR Pass	ISel	RegAlloc	Other Passes

Interlude: Miscellaneous

- ▶ ∼5% spent in legacy pass manager infrastructure
  - ▶ @paperchalice and others restarted porting efforts towards new PM
- ▶ ∼3% spent in `MachineInstr::addOperand`
- ▶ ∼1% spent in de-allocating LLVM IR
- ▶ ∼1% spent in de-allocating Machine IR

- ▶ ∼2% overhead due to time measurements

IR Pass	ISel	RegAlloc	Other Passes	Misc. Overhead

Step 5: Emit Machine Code to Object File

ПШ

▶ AsmPrinter: encode instructions and create object (or asm) file

▶ Fairly slow, especially on x86
▶ Every instruction transformed MIR→MC→Binary
▶ Lots of hooks and virtual function calls *per instruction*
  ▶ Abstraction comes at a price...
▶ All basic blocks get string labels, even for object files

⤳ Reduce hooking points and abstractions?

IR Pass	ISel	RegAlloc	Other Passes	Misc. Overhead	AsmPrinter

Step 6: JIT-Linking

ПΙΠ

- ▶ Standard back-end pipeline creates in-memory (ELF) object file
- ▶ JITLink maps and relocates object files into a process

- ▶ ELF file generation and parsing unnecessary
- ▶ Processing symbols and relocations is slow

- ⤳ `MCJITStreamer` for compiling to process memory?
    - ▶ Benefits: directly resolve symbols, keep fixups in same data structures, . . .
    - ▶ Focus on common subset – many JIT-codes don't use complex features

IR Pass	ISel	RegAlloc	Other Passes	Misc. Overhead	AsmPrinter	JIT-Link

Key Take-Aways

- Keep number of passes in `-O0` back-end low
  - Omission, merging, or feature-sensitive execution
- Finish porting back-end to new pass manager
- Keep FastISel(-like) instruction selector

- Rewriting IR is fairly expensive
- Iterating over IR is not cheap

- JIT: Better integration of AsmPrinter and linker

IR Pass	ISel	RegAlloc	Other Passes	Misc. Overhead	AsmPrinter	JIT-Link

One more thing...

- Over 20+ years, LLVM accumulated features and abstractions
- Most programs don't need most of that

Should we start over from scratch?

- Prototypical LLVM back-end:[1] 10–20x comptime speedup, -O0 performance
- Focus on common subset; 3 passes; single-step LLVM-IR $\rightarrow$ machine code

[1] https://llvm.org/devmtg/2024-03/slides/llvm-fast-backend.pdf