

Analyzing AArch64 and x86-64 machine code with armlint and x86lint

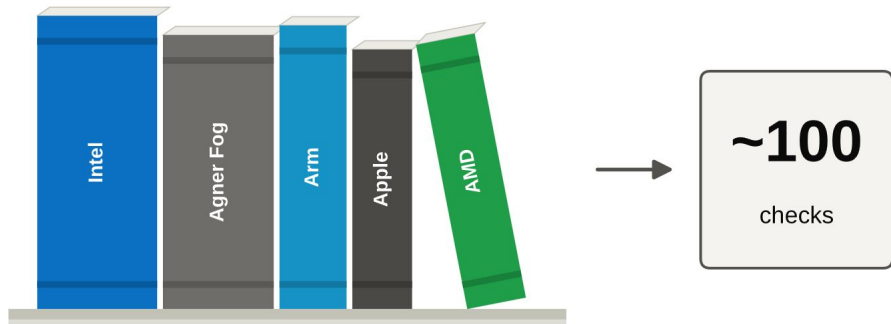
Andrew Gaul

<https://gaul.org/>

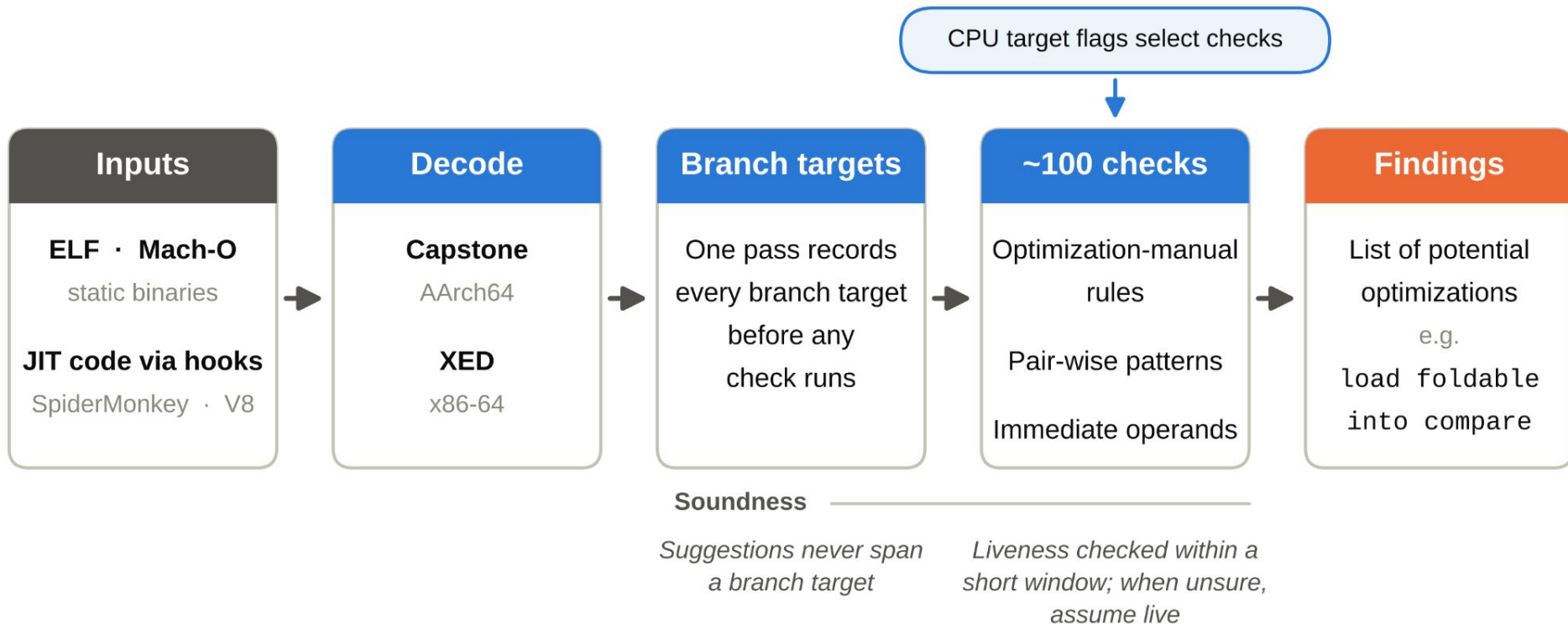
LLVM Developers' Meeting
October 2026

What are armlint and x86lint?

- Tools to analyze compiler machine code output
- ~100 optimization checks derived from Agner Fog, Apple, ARM, AMD, and Intel optimization manuals
 - Plus pair-wise instruction analyses
- Immediate-operand analysis to find constants that cost extra instructions
- Other safety and build configuration analyses
- Apache 2.0-licensed:
 - <https://github.com/gaul/armlint>
 - <https://github.com/gaul/x86lint>



How do they work?



A simple armlint finding: test a bit and branch

- `and w6, w3, #0x400 ; cmp w6, #0 ; b.ne target`
 - 3 instructions
- `and w6, w3, #0x400 ; cbnz w6, target`
 - 2 instructions – compare with zero + branch → `cbnz`
- `tbnz w3, #10, target`
 - 1 instruction – single-bit mask + `cbnz` → `tbnz`; `w6` is freed
- armlint checks that `w6` and `NZCV` are dead afterwards and the target is within ± 32 KB
- Many recent cores fuse `cmp` + `b.cond`, so the first form often costs 2 μ ops rather than 3; `tbnz` is 1 μ op and 8 bytes shorter either way
- Not emitted by LLVM – but previously in SpiderMonkey [bug 2070355](#) and in V8 [CL 8477411](#)

Example LLVM x86-64 optimization

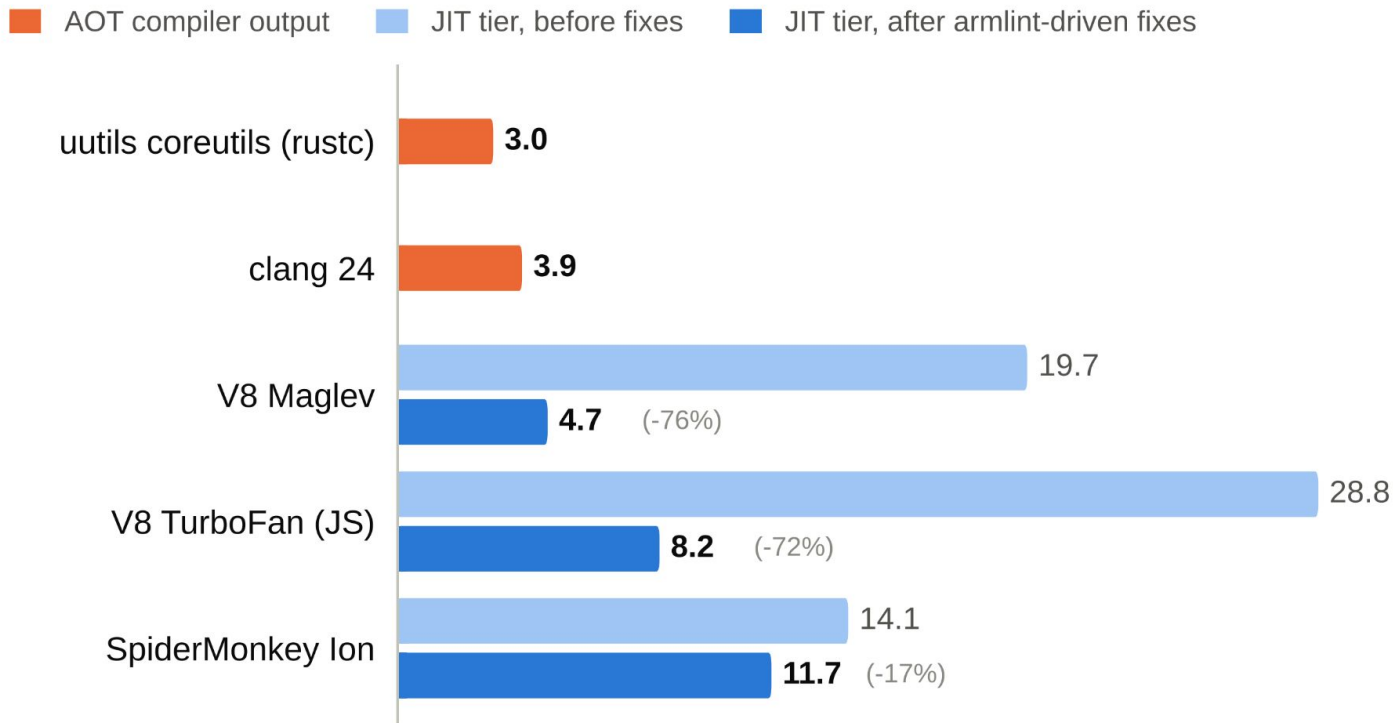
- [X86] Fold atomic loads into compares - [LLVM #221290](#) (merged)
 - x86 ALU instructions accept a memory operand
 - But LLVM <= 23 emitted an extra MOV for atomics
 - 5,854 affected sites in libxul.so
- Previously emitted 2 instructions (9 bytes) and clobbered one register
 - `movzbl guard_var(%rip), %eax`
 - `testb %al, %al`
- Now emits 1 instruction (7 bytes) and clobbers no registers
 - `cmpb $0, guard_var(%rip)`
- 43-line fix in X86ISelLowering.cpp (EmitCmp) and X86InstrCompiler.td (X86cmp atomic-load patterns)

Example LLVM AArch64 optimization

- [AArch64] Fold cmp #0 into its definition when the flags feed a CCMP - [LLVM #221858](#) (under review)
 - The peephole already folds it for branch and CSEL users, but its flag-use analysis didn't know where CCMP keeps its condition code, so a CCMP user blocked the fold
 - This is the head of every CCMP chain: 489 sites in librustc_driver, 408 in libxul.so
- Previously emitted 3 instructions:
 - `and w8, w8, #0x7 ; cmp w8, #0 ; ccmp w8, #3, #4, ne`
- Proposing 2 - ANDS sets N and Z from its result, so the CMP disappears:
 - `ands w8, w8, #0x7 ; ccmp w8, #3, #4, ne`
- Reuses the existing safety rules: fold only when the consumer reads N/Z (cmp #0 sets C, ANDS clears it)
- 39-line fix in AArch64InstrInfo.cpp (findCondCodeUseOperandIdx, now covers CCMP/CCMN/FCCMP)

How prevalent are optimization opportunities?

Findings per 1,000 static instructions (AArch64)



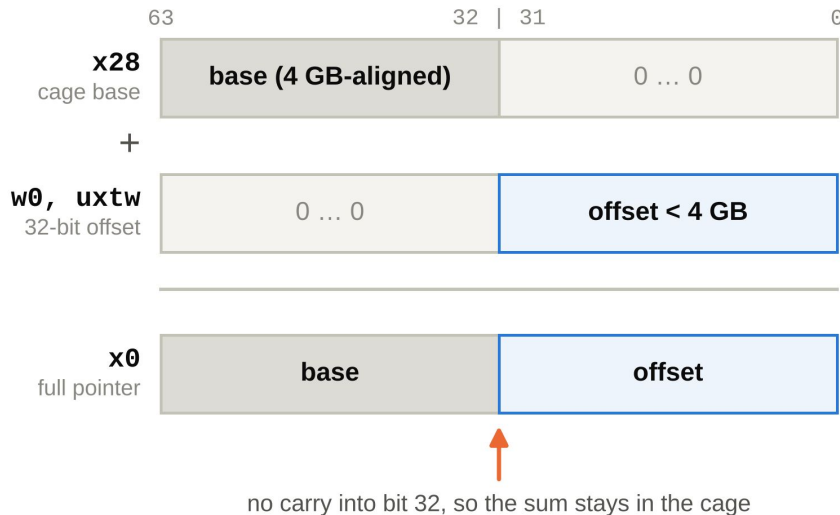
Immediate operand analysis

- SpiderMonkey NaN boxes type tags in doubles via: $(0x1FFF0 \mid \text{type}) \ll 47$
- Materializing Int32(5):
 - `movz x0, #0x5 ; movk x0, #0x8000, lsl #32 ; movk x0, #0xfff8, lsl #48`
- Renumbering common types to even numbers removes the movk that set bit 47, the tag's low bit:
 - `movz x0, #0x5 ;` `movk x0, #0xfff9, lsl #48`
- Saves 659K out of 54.7M static instructions (1.2%) from the JetStream 3 corpus

TYPE CODE	0x0	0x1	0x2	0x3	0x4	0x5	0x6	0x7	0x8	0x9	0xa	0xb	0xc	0xd	0xe	0xf
INSTRUCTIONS	1	2	1	2	1	2	1	2	1	2	1	2	1	2	1	1
Before	Double	Int32	Boolean	Undef	Null	Magic	String	Symbol	Private	BigInt			Object			
After	Double		Int32		Null		Undef		Boolean		Magic	Private	Object	BigInt	String	Symbol

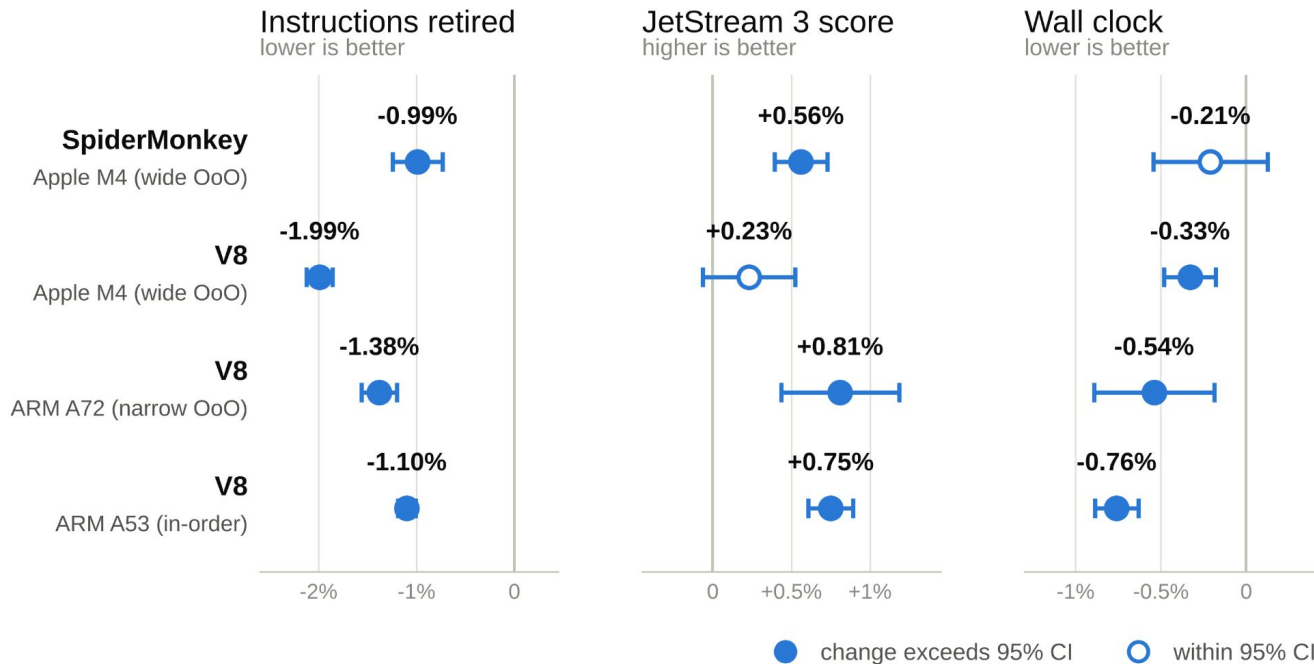
V8 caged pointer regression

- V8 compresses pointers into 4 GB cages (base in x28)
- A security hardening change replaced ADD with ORR so an out-of-range offset can't address memory outside the cage
- But ORR has no zero-extending register form, so decompressing became:
 - `ubfx x16, x0, #0, #32`
 - `orr x0, x28, x16`
- armlint identified this regression and recommended ADD with a zero-extended 32-bit index:
 - `add x0, x28, w0, uxtw`
 - Still can't escape the cage (see right)
- Two 1-line changes improved JetStream 3 by +0.58% (ARM A72) and +0.26% (ARM A53)



What is the impact?

- Merged commits: SpiderMonkey 50+, V8 25+, LLVM 4 (+7 under review)
- Measurably improved JetStream 3 scores of SpiderMonkey and V8



Limitations

- Measurements – individually micro-optimizations are within noise but dozens compound to 0.2–0.8% on JetStream 3
 - Largest gains on in-order and narrow cores; wide OoO cores hide many inefficiencies
- Static counts – tools emit unweighted findings so users must manually combine with perf samples to prioritize fixes
- Precision – checks are conservative but conventions like linker fixups and patch points emit a handful of false positives that require per-runtime flags
- Tuning – tools have an -march flag but not a -mtune flag to filter checks like the INC-vs-ADD-1 and LDP/STP fusion

But wait, there's more!

- Instruction census – was a binary compiled for the intended sub-architecture, e.g., Armv8.9 CSSC, x86-64-v3 (AVX2)?
- Safety analysis – was a binary compiled with the intended control-flow protections, e.g., AArch64 BTI/PAC, x86-64 IBT/CET shadow stack?
- Erratum analysis – does a jump or macro-fused branch cross a 32-byte boundary on Skylake-family cores?
- ISA extension analysis – what would future and proposed extensions buy, e.g., Armv9.6 fused compare-and-branch (CMPBR), RISC-V indexed integer loads (Zilx)?

Conclusion

- armlint and x86lint give developers and users visibility into compiler output
- They codify optimization-manual guidance that otherwise lives in PDFs and ad-hoc analyses
- Using them can measurably improve code generation

- Question: what other analyses would benefit LLVM?
- Question: could LLVM use these tools to prevent regressions?
- Question: what is the best way to upstream identified micro-optimizations?
 - How to balance small benefits against reviewer toil?