LLVM DEVELOPERS' MEETING · OCTOBER 2026

ClangIR in 2026

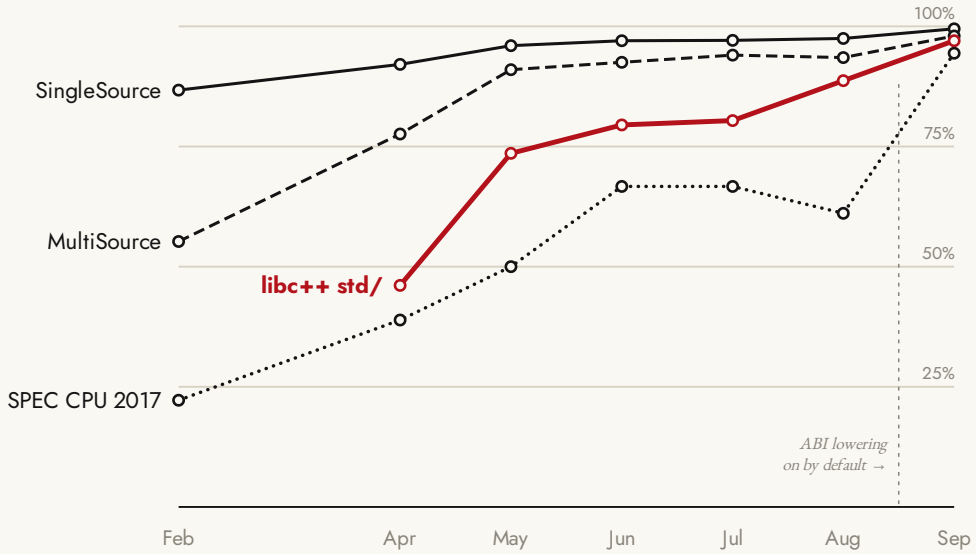
Benchmarks, Self-Hosting, and Compile Time

BRUNO CARDOSO LOPES

Meta

The test suites have nearly caught up

Pass rates on upstream main since the incubator was archived. x86_64, -O2.



SEPTEMBER, SIDE BY SIDE

SUITE	CIR	CLASSIC
SingleSource	**99.5%**	99.7%
MultiSource	**98.0%**	98.0%
libc++ std/	**97.0%**	97.4%
SPEC CPU 2017	**17/18**	18/18
SPEC CPU 2026	**40/47**	47/47
coreutils 9.11	**1527**	1527

SPEC CPU 2026 and coreutils are new this fall; coreutils matches Classic test for test.

Meta

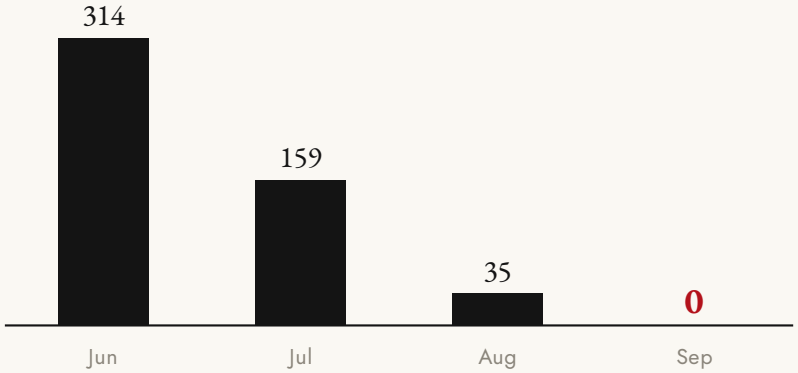
Clang now compiles all of itself

Every LLVM and Clang translation unit, built one at a time by a -fclangir clang.

4,744 / 4,744

translation units compile in September

UNITS THAT FAIL TO COMPILE



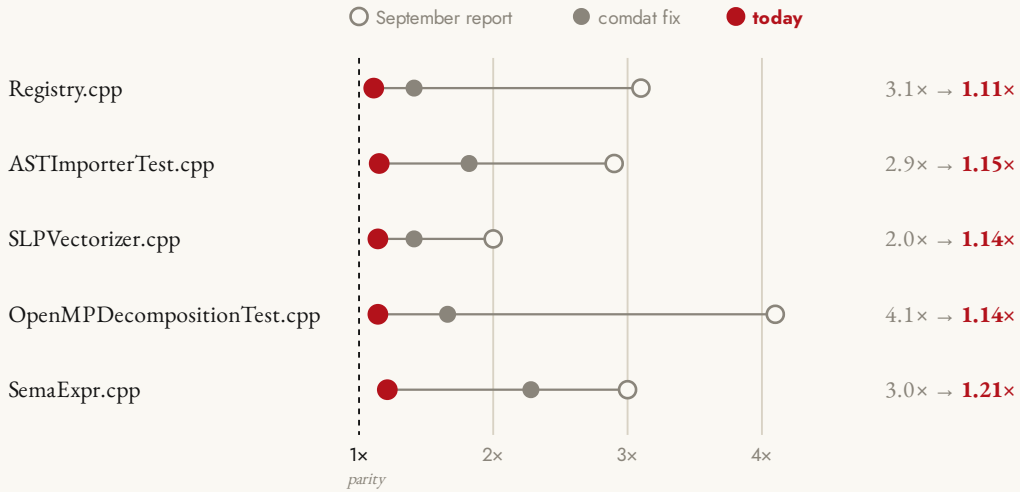
THE FULL STAGE 2

- ✓ Every unit compiles
- ✓ The CIR-built clang links
- ✓ Its TableGen output is byte-identical
  221/221 generated headers
- ✓ It passes `check-clang`
  every test — the self-built clang checks out

Meta

The compile-time gap is closing

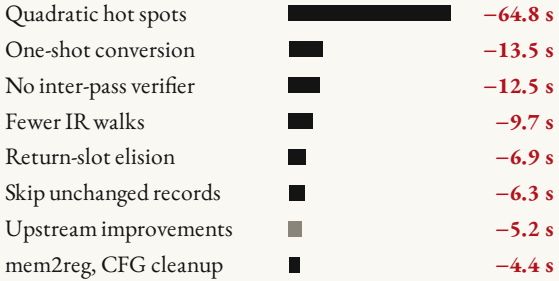
The five slowest files in the self-host build: CIR vs Classic



−78%

of CIR's extra time

WHERE THE 123 S CAME FROM



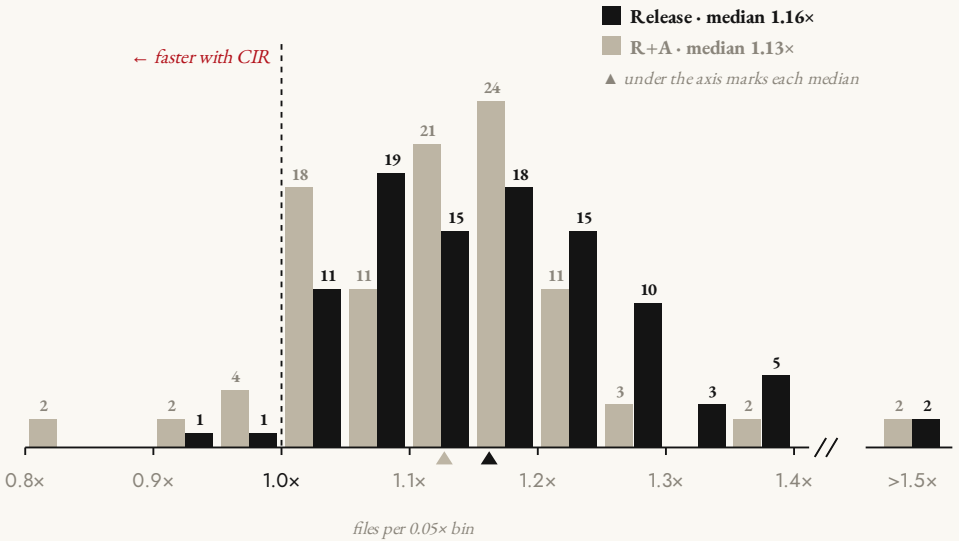
All on one branch: 30 fixes, 3 upstreamed so far
bcardosolopes/llvm-project · compile-time-investigations

Assertion-enabled build (serial 3-run medians). **Release build: 1.15–1.20×**

Meta

Beyond the worst five, the typical file costs 1.16×

The 100 slowest files during selfhost: CIR vs Classic.



1.16×

for the median file

65 of 100 within 1.2×

1.18× overall

2,137 s vs 1,814 s

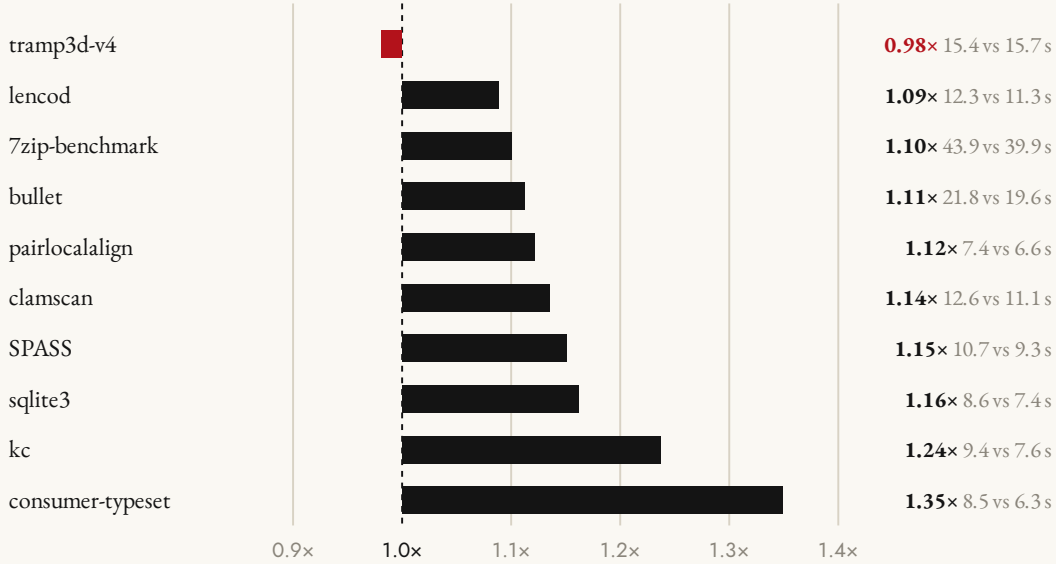
6 s to 66 s per file

9 s to 87 s with CIR

Meta

On LLVM's compile-time benchmark: 1.12×

CTMark, CIR vs Classic.



1.12×

across all 10 programs

150.6 s vs 134.8 s
total build time

tramp3d-v4 already faster with CIR
the template-heavy C++ one

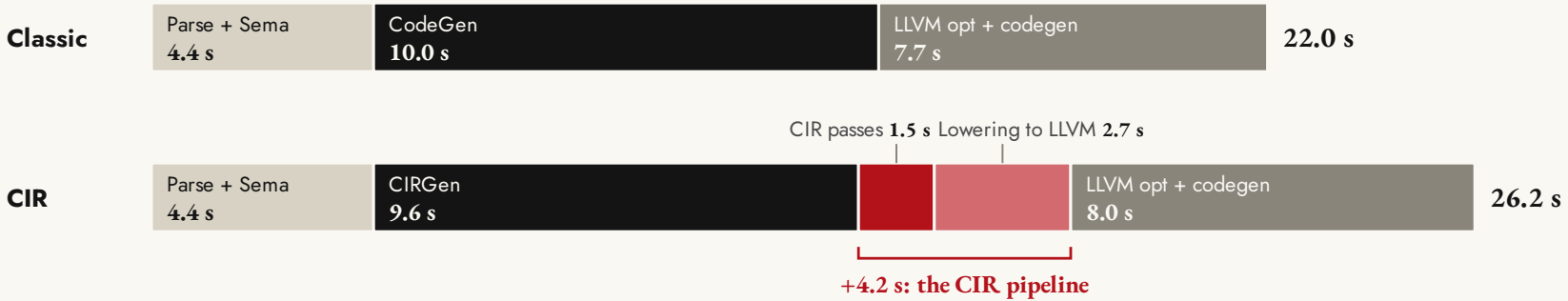
All 10 build with CIR

Release build

Meta

CIRGen keeps pace; the pipeline is the cost

`OpenMPDecompositionTest.cpp`, phase by phase (Release build)



IR EMISSION

CIRGen −0.4 s

Emitting CIR costs about the same as Classic emitting LLVM IR directly.

CIR PIPELINE

The whole gap

CIR-to-CIR passes, then lowering to LLVM IR: what's left to optimize.

LLVM BACK END

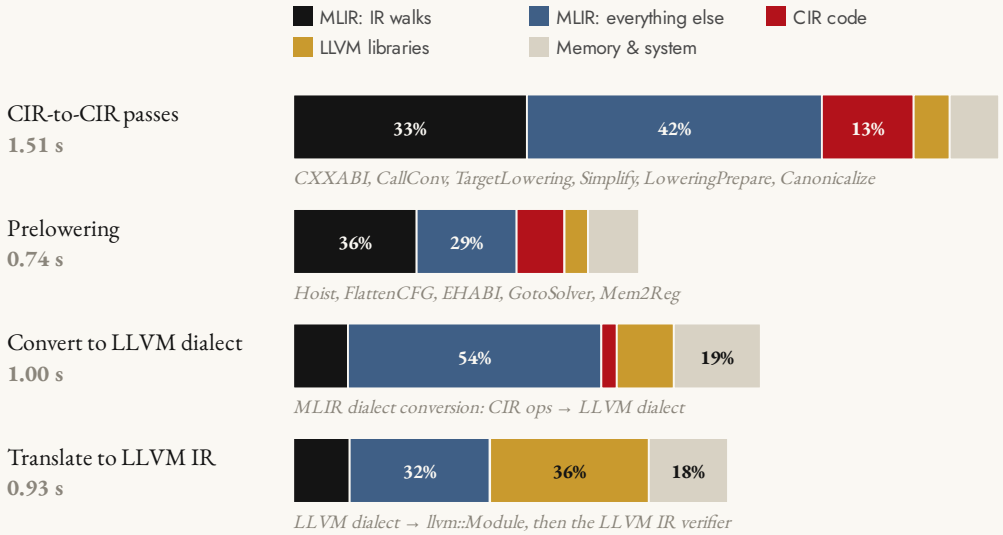
A wash

8.0 vs 7.7 s: LLVM handles CIR's IR almost as fast as Classic's.

Meta

Inside the CIR pipeline, the time goes to MLIR

The +4.2 s from the previous slide, CPU breakdown



8%

is CIR's own code (0.3 s)

MLIR: 2.7 s (64%)
1.0 s walking the IR; the rest: dialect conversion, uniquing, symbol lookups

LLVM libraries: 0.6 s
hashing, building the LLVM IR

Memory & system: 0.6 s
allocator, page faults

Meta

Next: make the simpler IR pay off

LLVM OPTIMIZER

Stop the inliner churn

CIR hands LLVM 30% fewer instructions, yet the inliner revisits functions 33% more often. Fix that and the optimizer gets cheaper.

CIR PIPELINE

Faster lowering

The CIR pipeline is the whole remaining gap. Its lowering time goes mostly to MLIR's conversion machinery, not CIR code: speed up the framework, not just CIR.

UPSTREAM

Land the branch

30 fixes on one branch, 3 upstreamed so far: one small PR per fix.

"Hand LLVM simpler IR, and the rest of the pipeline gets cheaper."