ATTACK OF THE CLONES

Speeding Up Coroutine Compilation

```
$ clang++ -std=c++20 -g2 -c coroutine.cpp
```

Artem Pianykh
Meta — Dev Infra

EuroLLVM 2026
April 14-15, 2026

Context

At Meta:

Large C++ codebase
Backend services, infrastructure, and more

Thousands of engineers writing, building, and debugging C++ daily

C++20 coroutines as the standard async pattern

Centralized developer infrastructure
Every change is felt globally

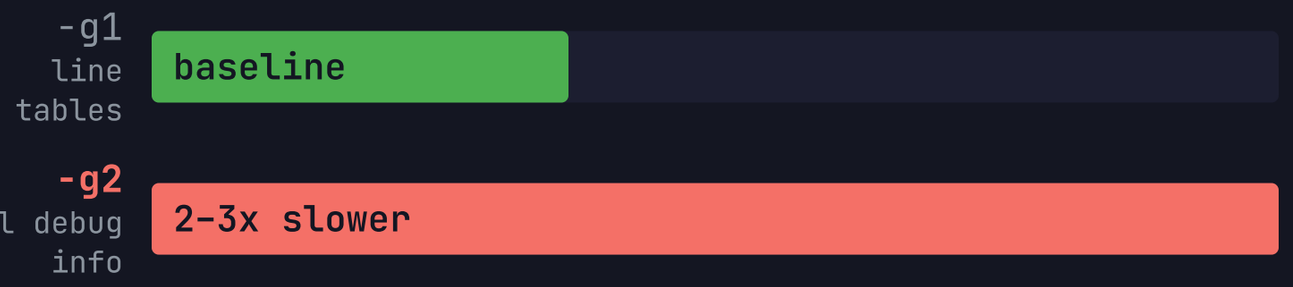
The Problem

To improve the dev workflow, we wanted to switch from `-g1` to `-g2` in dev builds — any binary debuggable out of the box.

This required work across the stack:

- ✓ Build-level optimizations
- ✓ Split DWARF rollout
- ✓ LLDB performance improvements

✗ **Compile times for sources with coroutines**



Simulations showed a significant end-to-end build time regression.

Profiling: Narrowing to a Pass

Profiling pointed to a single pass. A representative file:

`-g1` **~3 ms**

CoroSplitPass with line tables

`-g2` **~306 ms**

CoroSplitPass with full debug info

100x

SLOWER

What is CoroSplitPass doing that is so much more expensive with `-g2`?

CoroSplit: One Coroutine Becomes Many

coro [initial]
Function with coroutine intrinsics

transform →
clone + transform →
clone + transform →
clone + transform →

coro [ramp]
Entry point, runs until first suspend

coro.resume
Coroutine body (state machine)

coro.destroy
Destroy coroutine state

coro.cleanup
Cleanup coroutine resources

3 clones per coroutine* × **100s** of coroutines = **1000s of clones** via CloneFunctionInto

The slowdown was **not** in the coroutine transformation — it was in **cloning**.

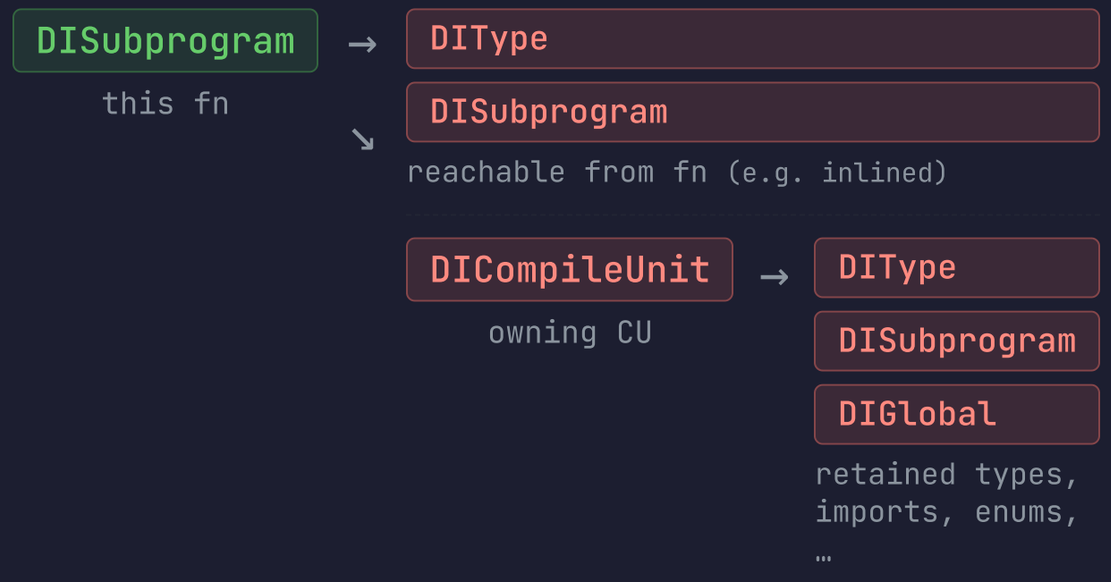
** Switch ABI (C++). Other ABIs (e.g. Swift retcon) may produce more clones.*

Function Cloning & Debug Info

Instructions, BBs
Explicit ownership

Debug info metadata
"Sea of nodes" — implicit ownership, discovered by traversal

TO CLONE A FUNCTION, TRAVERSE FROM…



CLONING POLICY

Clone only the function's `own` metadata.

Keep everything else `not ours` as-is.

Non-trivial for metadata: ownership is implicit, so implementing this policy requires eagerly traversing the graph to discover what's `not ours`.

The Root Cause: O(CompileUnit) Per Clone

HOW THE CLONING POLICY WAS IMPLEMENTED

```
// For each clone:

// 1. Traverse from subprogram — cascades into
//    processCompileUnit, pulling in everything
DIFinder.processSubprogram(SP);  // O(CU)

// 2. Identity-map "not ours" into VMap
for (auto *Ty : DIFinder.types())
  VMap[Ty] = Ty;
for (auto *S : DIFinder.subprograms())
  if (S ≠ SP)
    VMap[S] = S;
// ...more categories

// 3. Clone the function
CloneFunctionInto(NewFn, OldFn, VMap, ...);
```

O(CU)

PER CLONE

should be O(Function)

- × `processSubprogram` cascades into the entire CU
- × Pointer chasing across scattered heap nodes
- × Clone policy baked into mutable VMap — can't share

The Optimization Journey: Four Iterations

Baseline
Eager VMap pre-population — O(CU) per clone
306 ms
1x

1

Step 1: Decouple
Separate IdentityMD set, lazy identity-mapping via ValueMapper
221 ms
1.4x

2

Step 2: Reuse
Build the set once, share across all clones of a coroutine
68 ms
4.5x

?

Detour: Cache SUPERSEDED
Module-level analysis pass — worked, but invalidation was unclear
17 ms
18x

✓

Step 3: Simplify
From set to predicate — no traversal. O(Function) restored.
3.8 ms
80x

Step 1: Decouple Clone Policy From Remapping State

BEFORE: IDENTITY NODES STUFFED INTO VMAP

```
// Insert identity nodes into VMap upfront
for (Node in FindDebugInfoToIdentityMap(Fn)) {
  VMap[Node] = Node;  // expensive tracking
}
CloneFunctionInto(NewFn, Fn, VMap, ...);
```

AFTER: SEPARATE IMMUTABLE SET, LAZY MAPPING

```
// Build set separately (still O(CU))
IdentityMD = FindDebugInfoToIdentityMap(Fn);

// New parameter flows through to ValueMapper
CloneFunctionInto(NewFn, Fn, VMap,
    ..., &IdentityMD);
  ↓
ValueMapper(VMap, ..., &IdentityMD);
  // identity-maps matching metadata on first use
```

✓ IdentityMD is **immutable** — decoupled from VMap
✓ Can now be **shared** across clones (enables Step 2)
✗ Still traverses module to build the set per clone

306ms → **221ms** 1.4x

Step 2: Reuse Policy Across Clones

BEFORE: SET REBUILT INSIDE CLONE LOOP

```
for (CloneKind : {Ramp, Resume, Destroy}) {
  // O(CU) traversal — once per clone!
  IdentityMD = CollectCommonDebugInfo(Fn);
  CloneFunctionInto(Clone, Fn, VMap,
      IdentityMD, ...);
}
```

AFTER: BUILD ONCE, SHARE ACROSS CLONES

```
// O(CU) traversal — once per coroutine
CommonDI = CollectCommonDebugInfo(Fn);

for (CloneKind : {Ramp, Resume, Destroy}) {
  CloneFunctionInto(Clone, Fn, VMap,
      /*IdentityMD=*/CommonDI, ...);
}
```

Key observation: all clones of the same coroutine share the same module-level metadata. No reason to rebuild the set for each clone.

✓ Eliminates redundant traversals per coroutine
× Still O(CU) once per coroutine
× Hundreds of coroutines × O(CU) still adds up

306ms → **68ms** 4.5x

Detour: Cache It

SUPERSEDED

DEBUGINFOCACHEANALYSIS

Primed DIFinder per compile unit

computed once at module level

↓ ↓ ↓ ↓

copy & reuse

coro_1	coro_2	coro_3	⋯	coro_N
.resume .destroy .cleanup	.resume .destroy .cleanup	.resume .destroy .cleanup		.resume .destroy .cleanup

306ms → **17ms** 18x

✓ Fast in practice

× No clean invalidation story

They say there are two hard problems in CS...

Thanks to @felipepiovezan for the invalidation question that prompted rethinking this approach

Step 3: Express Policy as a Predicate

Before
```
const MetadataSetTy *IdentityMD
```

→

After
```
const MetadataPredicate *IdentityMD
```

- ✗ No eager CU traversal
- ✗ No Set
- ✗ No Cache
- ✓ **O(Function)**

80x

306ms → 3.8ms

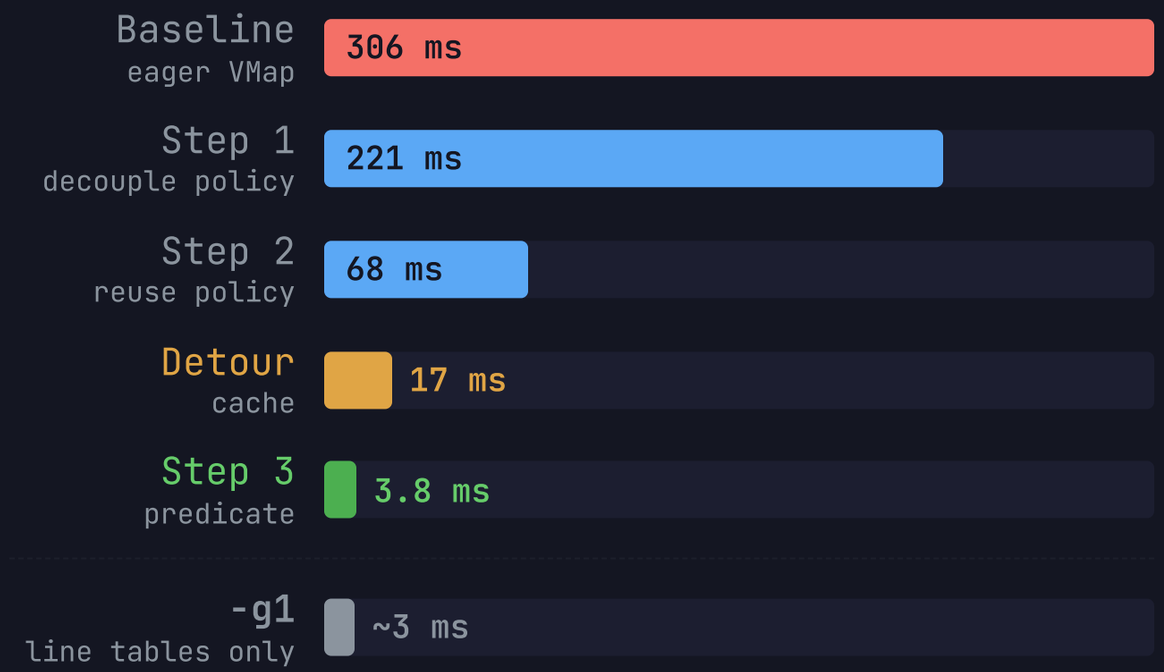
CONCEPTUAL SKETCH OF THE PREDICATE

```
auto createIdentityMDPredicate(const Function &F, Changes) {
  auto *SP = F.getSubprogram();
  return [=](const Metadata *MD) {
    if (isa<DICompileUnit>(MD) || isa<DIType>(MD))
      return true;
    if (auto *S = dyn_cast<DISubprogram>(MD))
      return S ≠ SP;
    if (auto *LS = dyn_cast<DILocalScope>(MD))
      return LS→getSubprogram() ≠ SP;

    return false;
  };
}
```

Performance Results

COROSPLITPASS TIME (SAMPLE FILE, FULL DEBUG INFO)



80x

FASTER

Full debug info, nearly as fast as **-g1**

Not Just Coroutines

CoroSplit

Primary motivation — high clone volume made the O(CU) cost painfully visible.

Also benefits

`FunctionSpecialization` — IPSCCP-driven specialization in default O1+ pipelines

`ThinLTO function aliases` — imported aliases are materialized via cloning

`MemProf cloning` — context-specific clones when enabled

** The fix landed in CloneFunctionInto / ValueMapper — generic same-module cloning infrastructure, not a coroutine-specific fast path.*

Takeaways

Cloning was cheap. Ownership recovery was expensive.

The O(CU) cost came from reconstructing `"not ours"` from debug-info graph shape.

Separate remapping state from clone policy.

Once policy stopped living inside `VMap`, it became immutable, shareable, and finally a predicate.

The sharp edges are still in the representation.

Metadata ownership is still implicit, so cloning still relies on node-kind heuristics and special cases.

Thank You!

May the source be with you

Artem Pianykh
Meta — Dev Infra

pianykh.com/blog/talks/eurollvm26.html

REFERENCES

RFC discourse.llvm.org — "Amortizing debug info processing cost in CoroSplit"

PR #109032 — Umbrella PR

PR #118627, #129147, #129148 — Key upstream patches