Reducing Code Size with Speculative Inlining

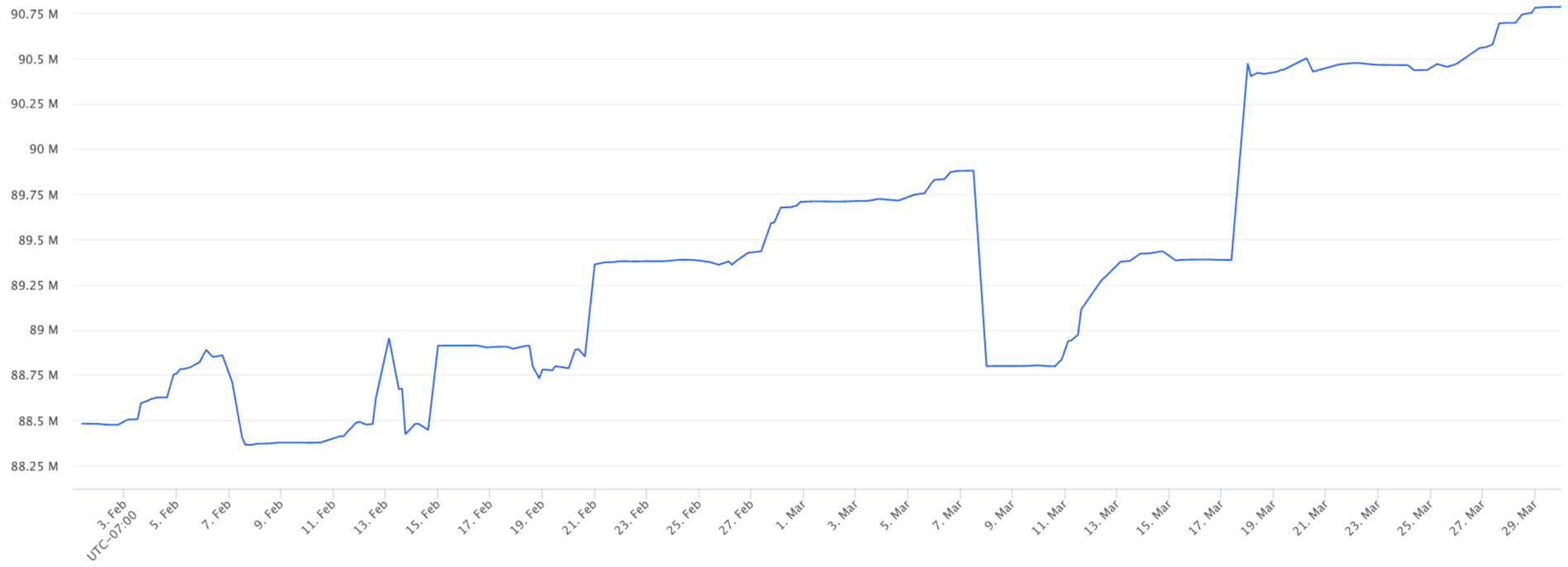
Vincent Lee

Meta

Meta

Growth in App Size

- Reducing app size is important for mobile applications
- Large apps impact user experience and user retention
- Employ optimizations (e.g. inliner) to reduce code size



Meta

Inlining for Code Size

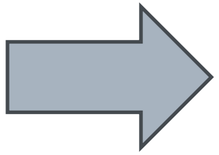
- LLVM generally tuned for performance and not code size
- Performance often viewed at the expense of increased code size
- Inlining is critical for compiler optimizations
- Leverage inlining to reduce code size may help with performance
  - Potential to speed up programs by maximizing amount of hot code in instruction cache

Example

```c
int f2(int *ptr, int a, int b, int n, int scale) {
    int valid = ptr ? 1 : 0;
    int s = 0;
    for (int i = 0; i < a; i++)
        for (int j = 0; j < b; j++)
            if (n * valid)
                s += scale * ptr[i];
    return s;
}

void f1(int *arr, int a, int b, int n, int t) {
    for (int i = 0; i < n; i++) {
        arr[i] = f2(0, a, b, n, t);
    }
}

void f1_optimized(int *arr, int a, int b, int n, int t) {
    for (int i = 0; i < n; i++) {
        arr[i] = 0;
    }
}
```



Under -Oz

```
define dso_local void @f1(...)
  %6 = tail call i32 @llvm.smax.i32(i32 %3, i32 0)
  %7 = zext nneg i32 %6 to i64
  br label %8

8:
  %9 = phi i64 [ %15, %12 ], [ 0, %5 ]
  %10 = icmp eq i64 %9, %7
  br i1 %10, label %11, label %12

11:
  ret void

12:
  %13 = tail call noundef i32 @f2(...)
  %14 = getelementptr inbounds nuw i32, ptr %0, i64 %9
  store i32 %13, ptr %14, align 4
  %15 = add nuw nsw i64 %9, 1
  br label %8
}
```

Under -O3

```
define dso_local void @f1_optimized(...)
  %6 = tail call i32 @llvm.smax.i32(i32 %3, i32 0)
  %7 = zext nneg i32 %6 to i64
  br label %8

8:
  %9 = phi i64 [ %14, %12 ], [ 0, %5 ]
  %10 = icmp eq i64 %9, %7
  br i1 %10, label %11, label %12

11:
  ret void

12:
  %13 = getelementptr inbounds nuw i32, ptr %0, i64 %9
  store i32 0, ptr %13, align 4
  %14 = add nuw nsw i64 %9, 1
  br label %8
}
```

https://godbolt.org/z/E71E9xqjK

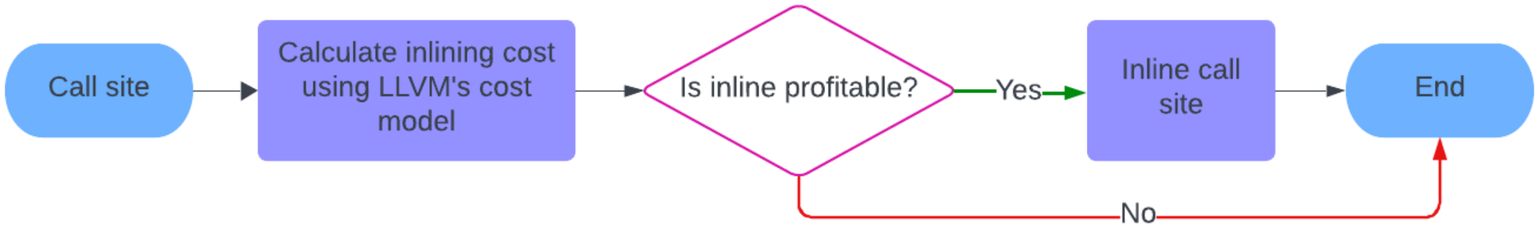
Meta

Speculative Inliner

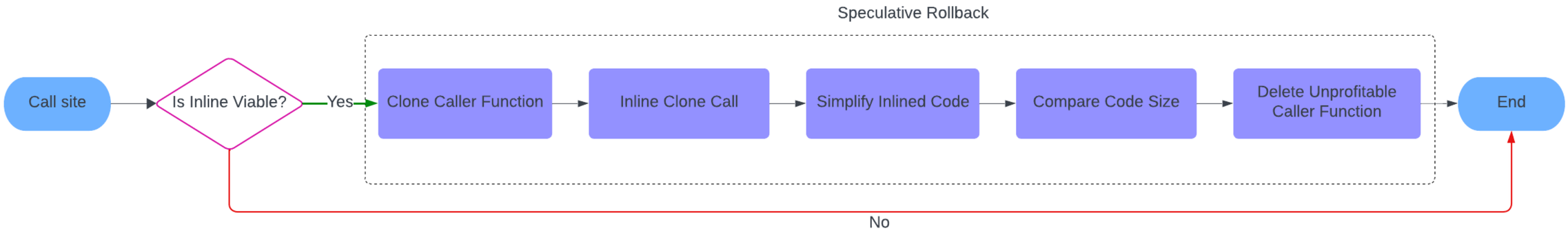
- Inliner pass that measures the cost of inlining based on the material outcome of the post-inliner optimizations on the inlined code
  - Consider all inline viable call sites for speculation and ignoring LLVM’s standard inline thresholds
- Post-inline optimizations (i.e. simplification) determines whether an inline is profitable or not

Meta

LLVM Inliner



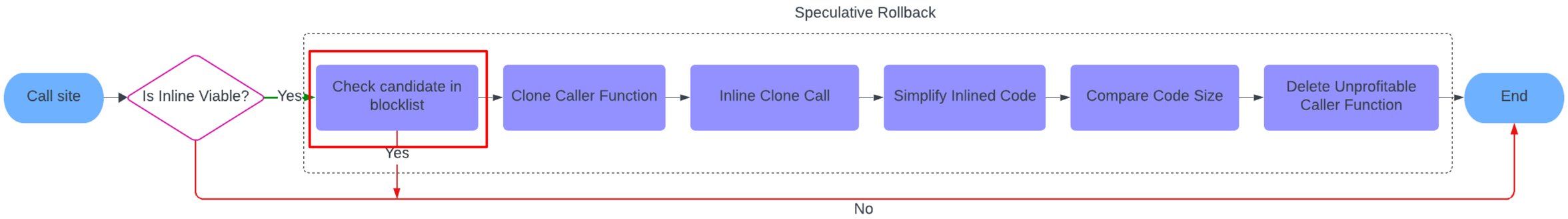
Speculative Inliner



Meta

Problem #1: Uninlineable callsites

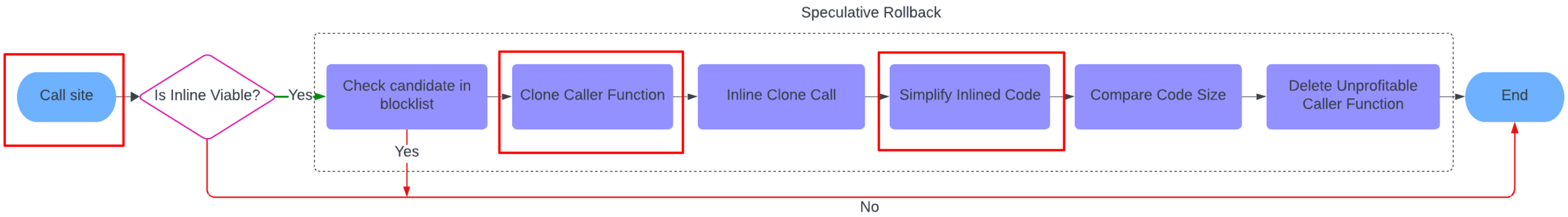
- Inline assembly
- Known mis-optimizations deeper in the pipeline
- Added a blocklist
  - Prevent inlining callee into caller
  - Prevent inlining **all** callees into caller



Meta

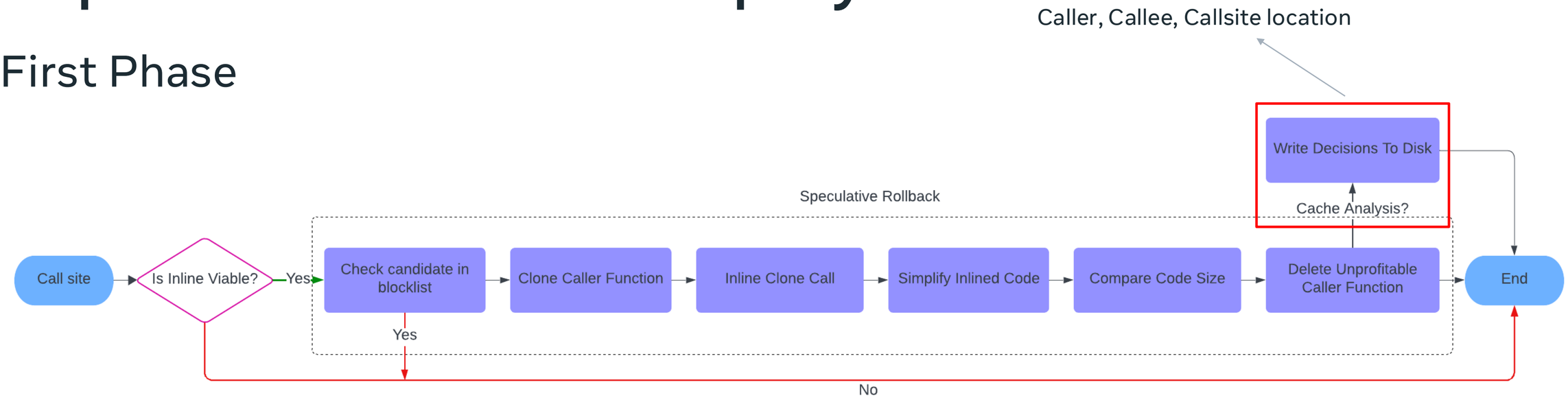
Problem #2: Searching

- Build time is very expensive
  - Large amount of callsites
  - Cloning operation can be costly
  - Simplifying large functions do not scale linearly

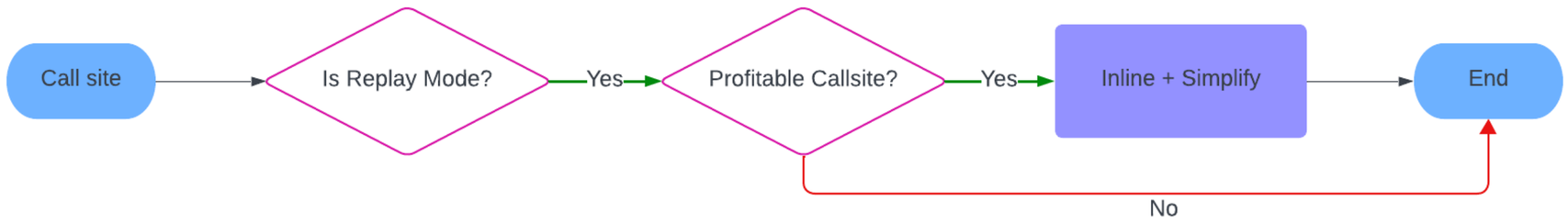


Speculative Inliner Replay

First Phase

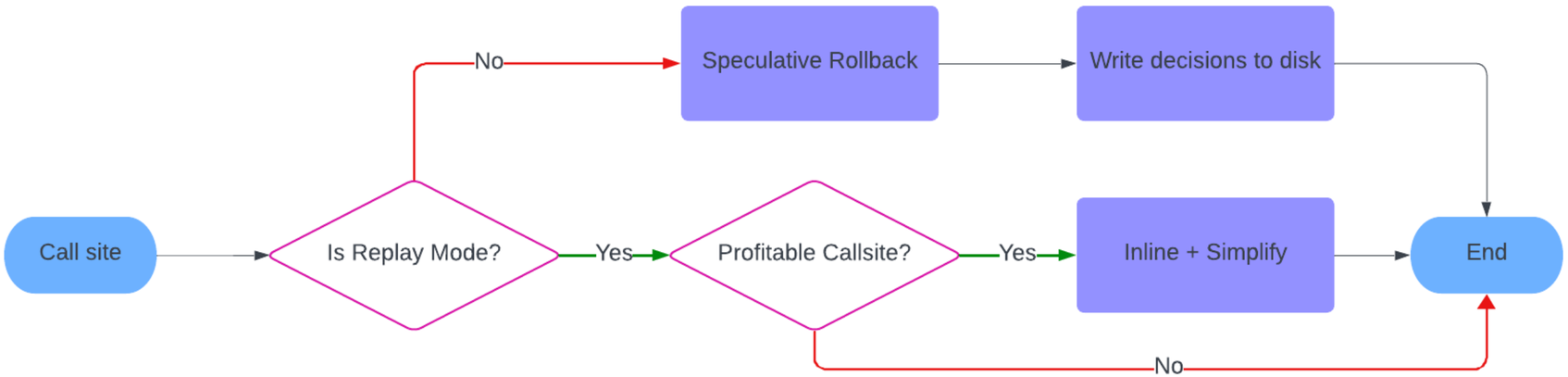


Second Phase



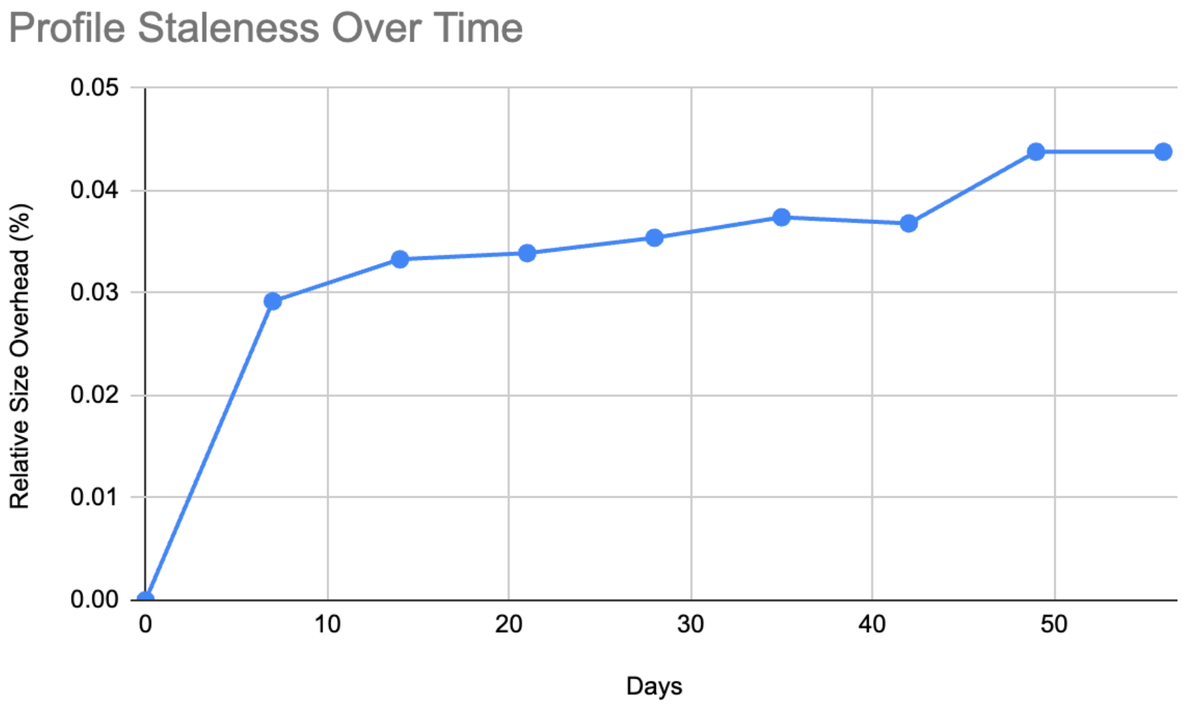
Meta

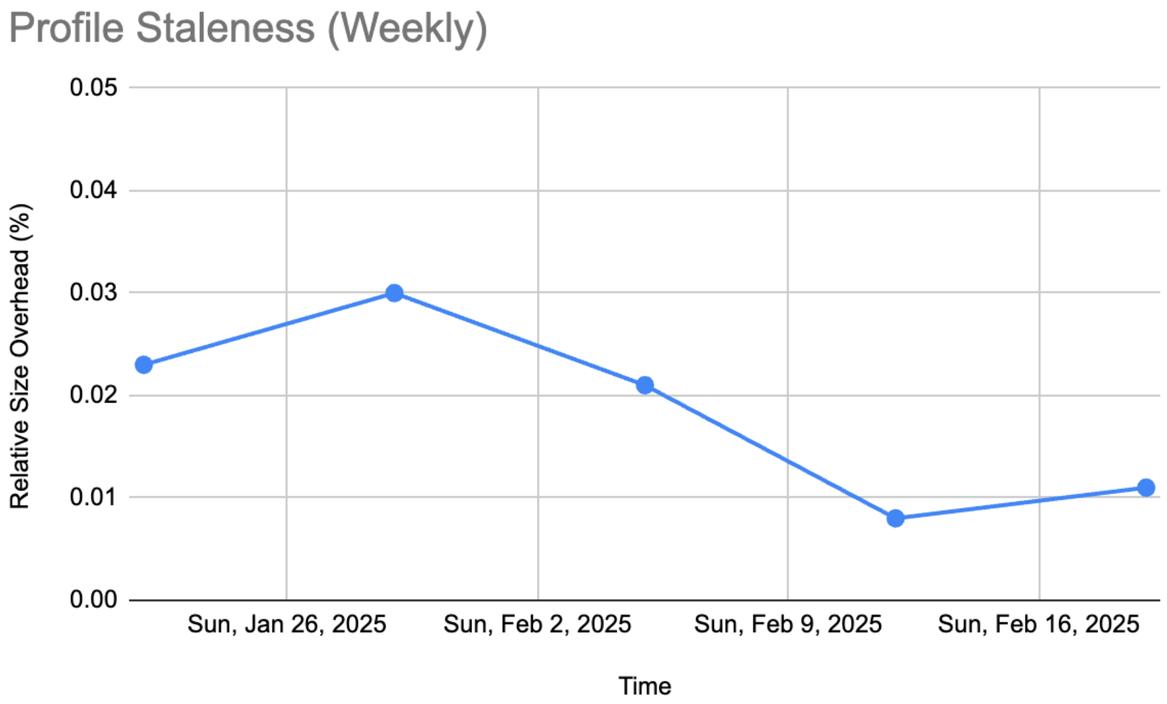
Speculative Inliner Replay



Meta

Profile Staleness





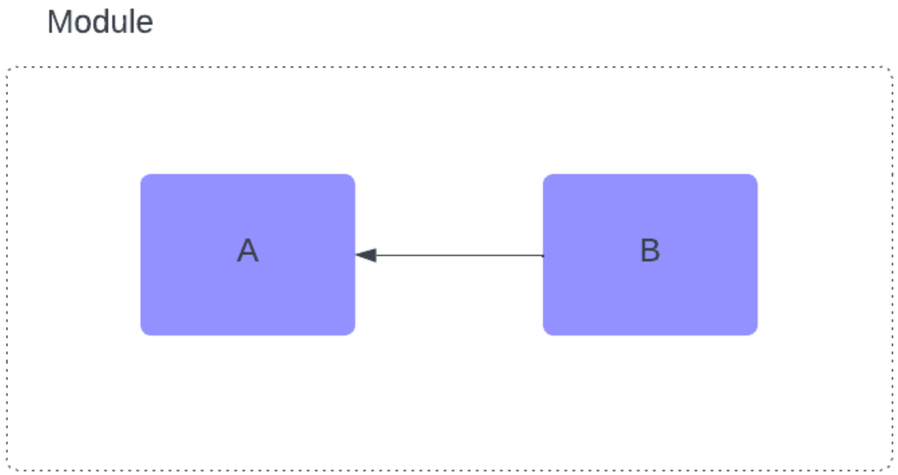
Meta

Size Beneficial Inlining

- Obtained when the callee can be removed from the program after inlining
  - Dropping its use count to zero
  - Internal linkage function

Meta

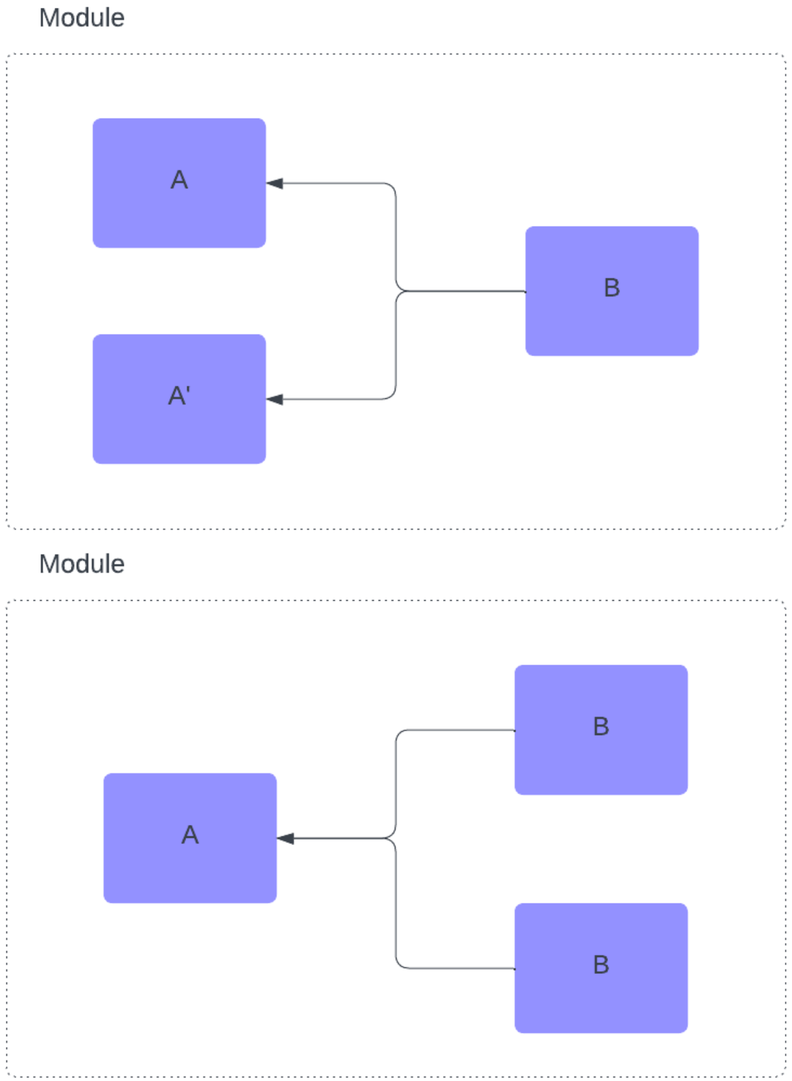
N = 1 callsite



- Guaranteed code size win once B is inlined into A

Meta

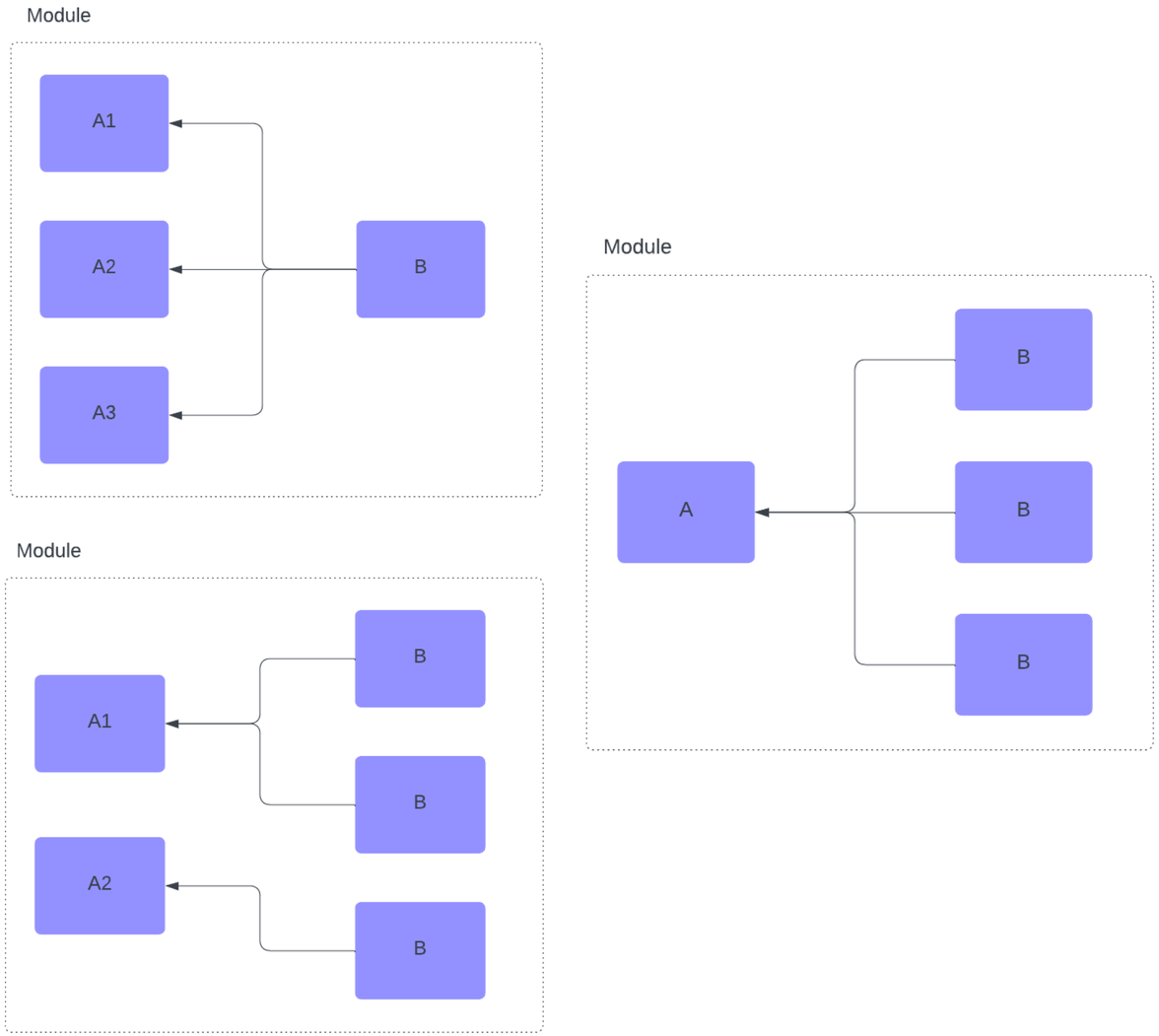
N = 2 callsites



- Not always guaranteed a win, but has been mostly true for small functions
  - Register pressure on larger functions
- Gated behind an arbitrary threshold
  - ~15 LLVM IR instructions through empirical testing

Meta

N = 3+ callsites

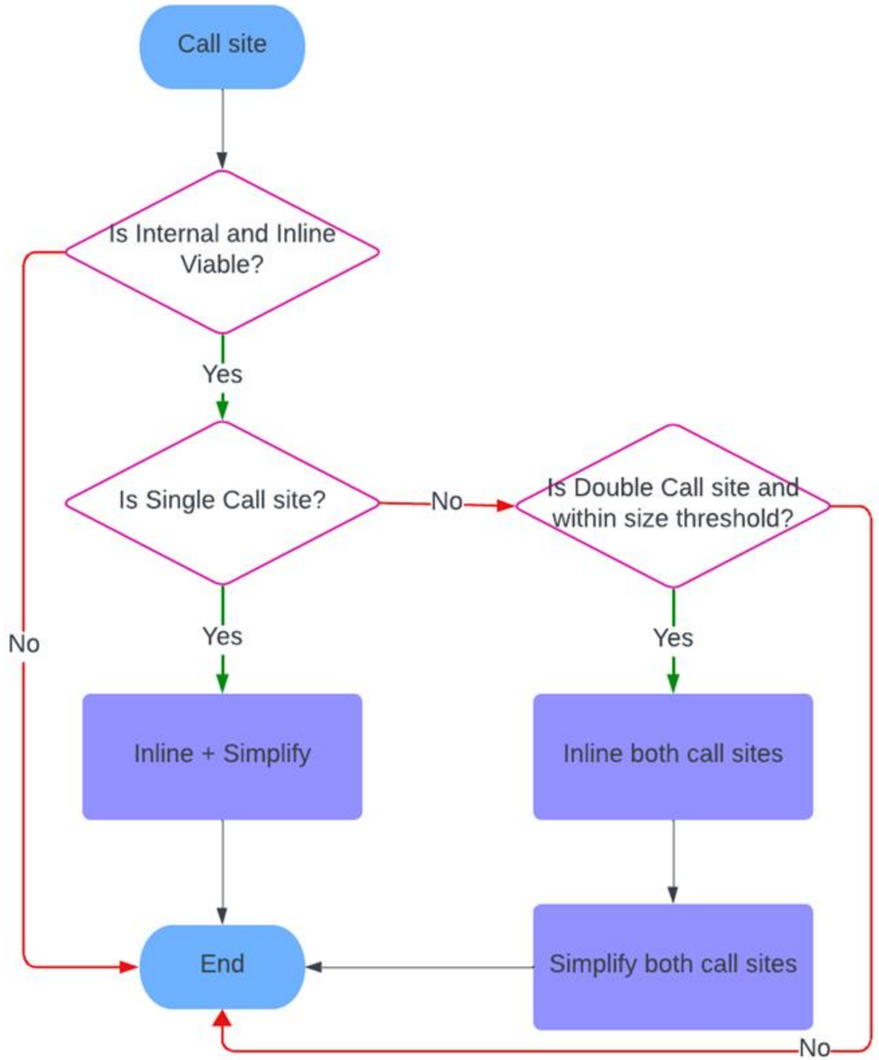


- Largely a size regression due to duplication
- Non-trivial heuristic

Meta

Internal Simplification Inliner

- Inliner pass that optimistically inlines and simplifies internal linkage functions based on the number of uses from a given call site
  - Similar to the speculative inliner - all inline viable call sites and ignores LLVM's standard inline thresholds

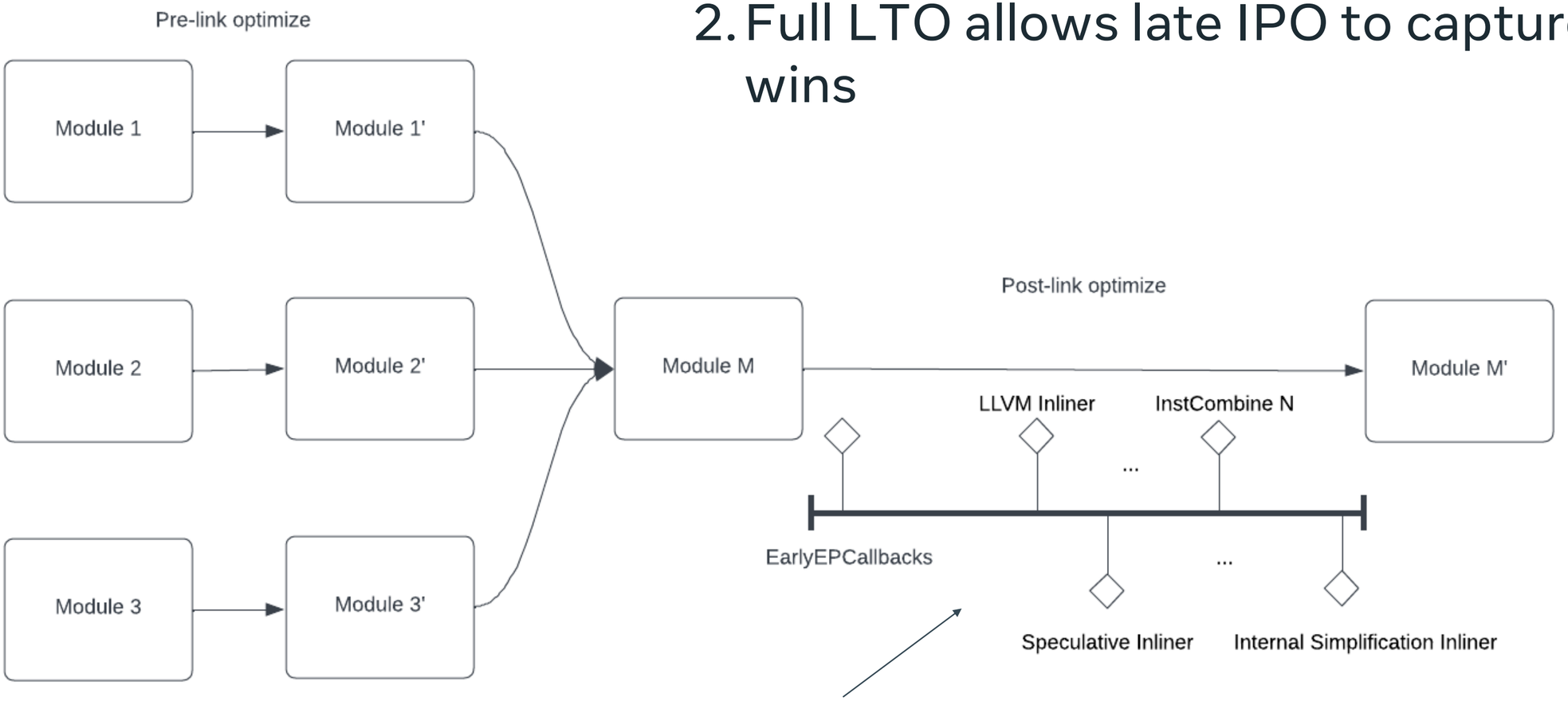


- No rollback mechanism
- Pass runs significantly faster!

Meta

FullLTO

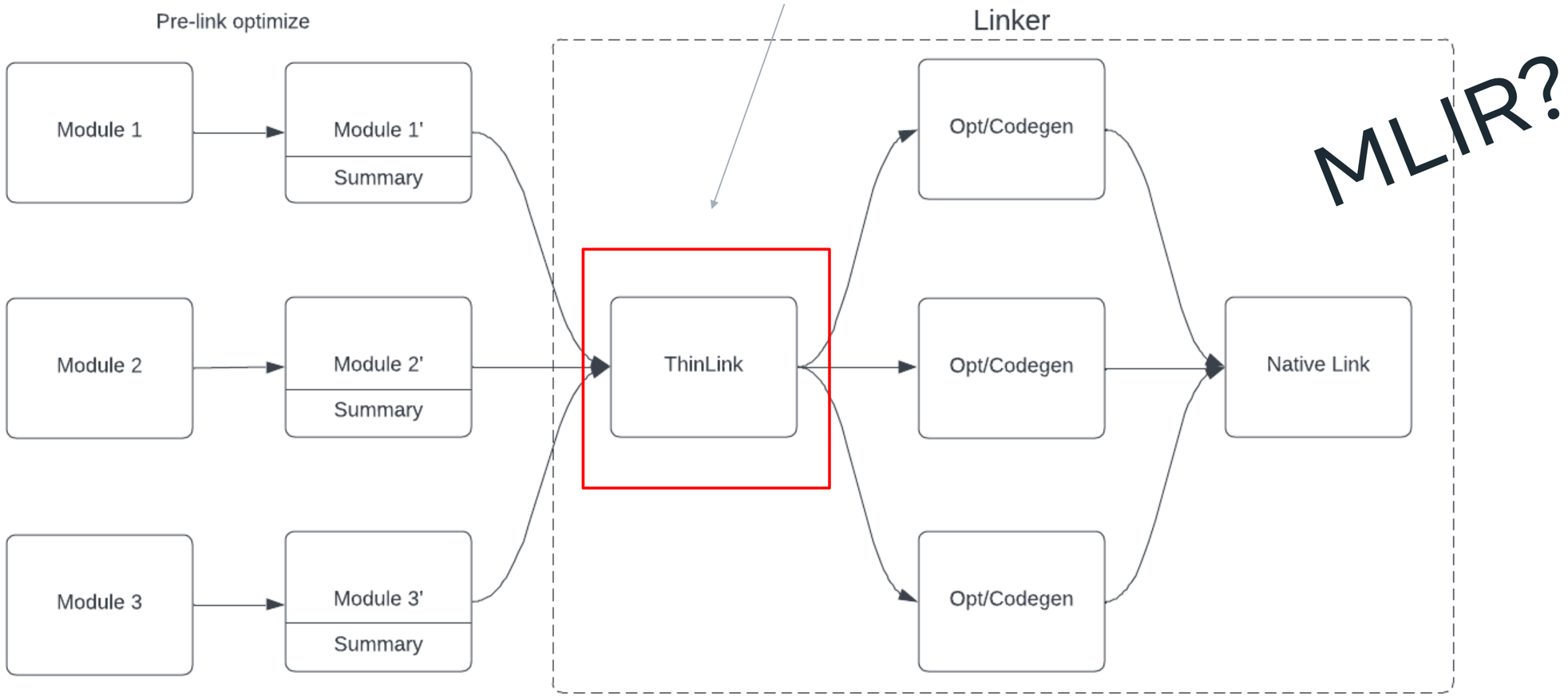
1. Inlining compounds with other passes
2. Full LTO allows late IPO to capture more wins



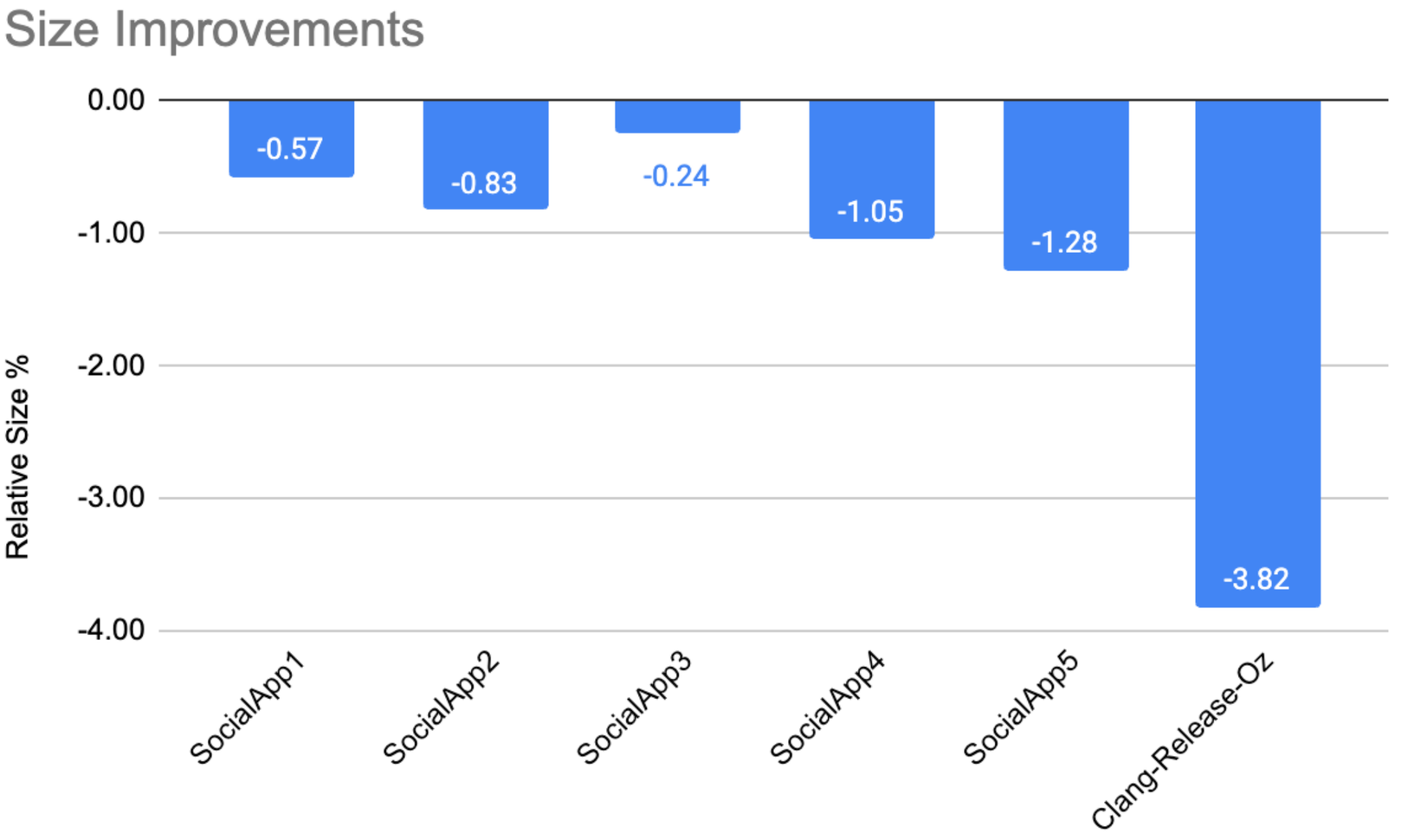
Meta

ThinLTO

Cross module import happens in the beginning



Meta


Size Improvements
0.00
-1.00
-2.00
-3.00
-4.00
Relative Size %
-0.57
-0.83
-0.24
-1.05
-1.28
-3.82
SocialApp1
SocialApp2
SocialApp3
SocialApp4
SocialApp5
Clang-Release-Oz

Meta

Conclusions

- Current LLVM paradigm has limitations in modeling the benefits of downstream simplifications for size
- Speculative inlining addresses this limitation, which can provide up to ~4% in code size reduction*
  - Size tradeoff with build time costs solved by replaying
- More wins to be captured when running IPO at different points of the pass pipeline
  - Experiments show running it early results in size regression, and running it late is more profitable


Meta