PGO and LLVM
Status and Current Work

Bob Wilson
Diego Novillo
Chandler Carruth

PGO: What Is It?

PGO: What Is It?

- PGO = Profile Guided Optimization

PGO: What Is It?

- PGO = Profile Guided Optimization
- More information -> better optimization

PGO: What Is It?

- PGO = Profile Guided Optimization
- More information -> better optimization
- Profile data
  - Control flow: e.g., execution counts
  - Future extensions: object types, etc.

What Is It Good For?

- Some examples:

What Is It Good For?

- Some examples:
  - Block layout

What Is It Good For?

- Some examples:
  - Block layout
  - Spill placement

What Is It Good For?

- Some examples:
  - Block layout
  - Spill placement
  - Inlining heuristics

What Is It Good For?

- Some examples:
  - Block layout
  - Spill placement
  - Inlining heuristics
  - Hot/cold partitioning

What Is It Good For?

- Some examples:
  - Block layout
  - Spill placement
  - Inlining heuristics
  - Hot/cold partitioning
- Can significantly improve performance

What’s the Catch?

- Assumes program behavior is always the same
- PGO may hurt performance if behavior changes
- May require some extra build steps

History of PGO in LLVM

History of PGO in LLVM

- Instrumentation, profile info and block placement (2004, Chris Lattner)

History of PGO in LLVM

- Instrumentation, profile info and block placement (2004, Chris Lattner)
- Branch weights and block frequencies (2011, Jakub Staszak)

History of PGO in LLVM

- Instrumentation, profile info and block placement (2004, Chris Lattner)
- Branch weights and block frequencies (2011, Jakub Staszak)
- Setting branch weights from execution counts (2012, Alastair Murray)

Outline

- Front-end instrumentation
- Profiles from sampling
- Using profile info in the optimizer and back-end

Profiling with Instrumentation

Profiling with Instrumentation

- Pros:
  - Detailed information
  - Predictability
  - Resilient against changes

Profiling with Instrumentation

- Pros:
  - Detailed information
  - Predictability
  - Resilient against changes
- Cons:
  - Need to build instrumented version
  - Running with instrumentation is slower

Design Goals

Design Goals

- Degrade gracefully when code changes

Design Goals

- Degrade gracefully when code changes
- Profile data not tied to specific compiler version

Design Goals

- Degrade gracefully when code changes
- Profile data not tied to specific compiler version
- Minimize instrumentation overhead

Design Goals

- Degrade gracefully when code changes
- Profile data not tied to specific compiler version
- Minimize instrumentation overhead
- Execution counts accurately mapped to source

Dealing with Change

Dealing with Change

- Project source code changes
  - Detect functions that have changed
  - Ignore profile data for those functions only

Dealing with Change

- Project source code changes
  - Detect functions that have changed
  - Ignore profile data for those functions only
- Some changes are OK
  - Minimum requirement: same control-flow structure

Compiler Changes

- Compiler updates should not invalidate profiles
- LLVM IR generated by front-end often changes
- Associating profiles with IR can be a problem

Source-level Accuracy

- PGO vs. code coverage testing
- Should only have one profile format for both
- Profile data for PGO should be viewable
- Requires profiles to map accurately to source

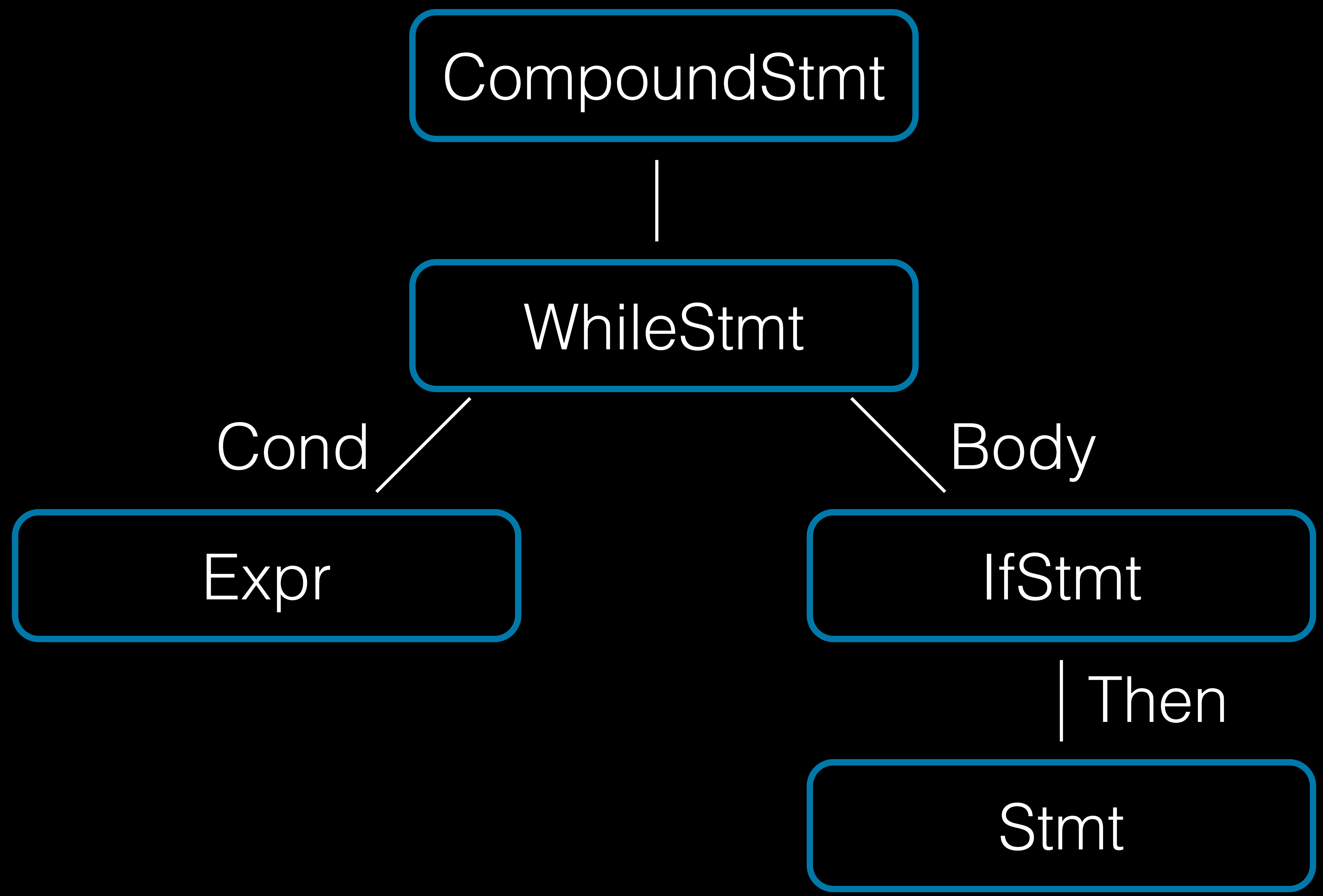
Use the Source

- Solution: associate profile data with clang ASTs
- Compiler changes are (almost) irrelevant
- Provides info to detect source changes
- Independent of optimization and debug info

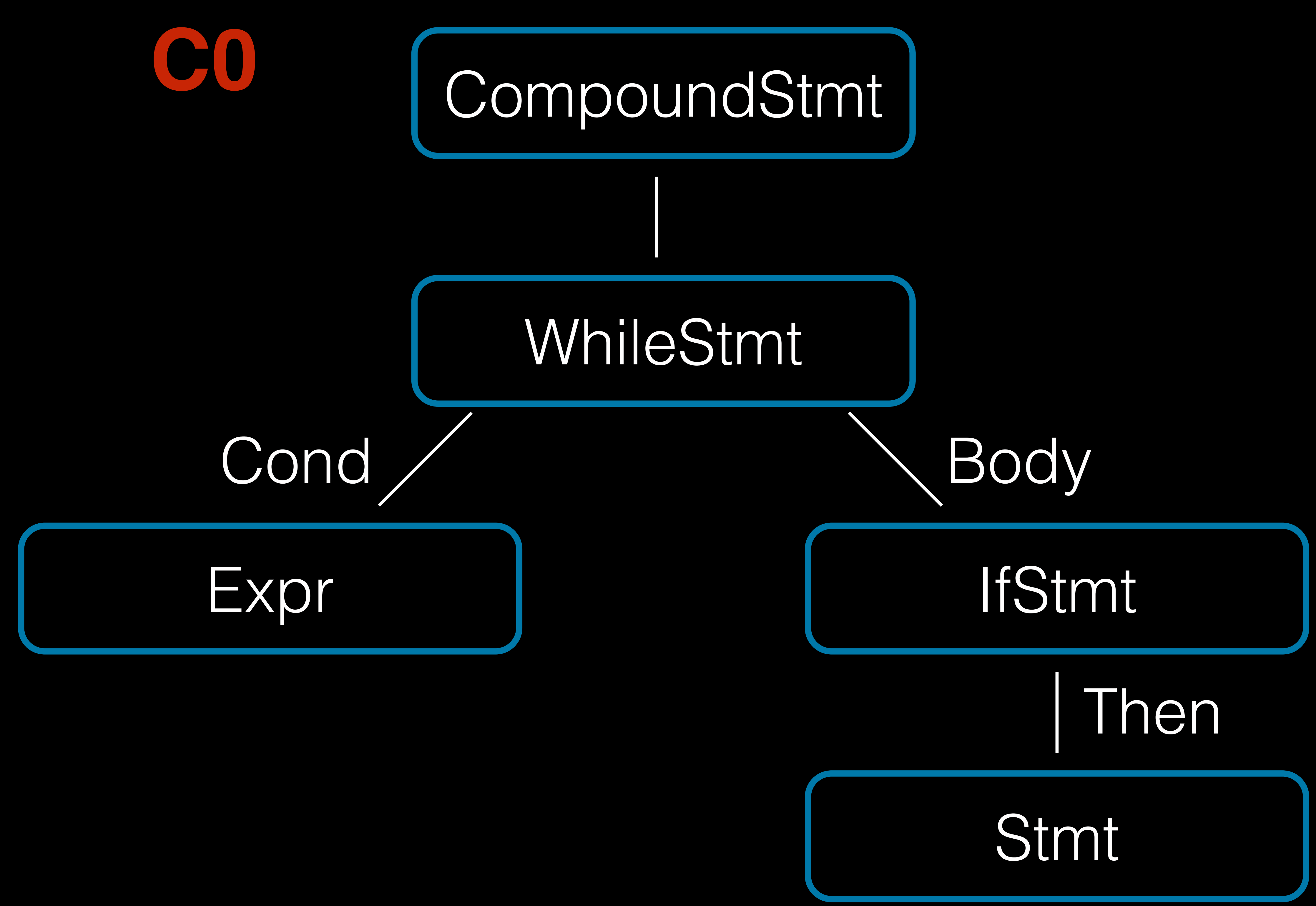
Counters on ASTs

- Walk through ASTs in program order
- Assign counters to control-flow constructs
- Compare number of counters to detect changes
- Can add a hash of ASTs to be more sensitive

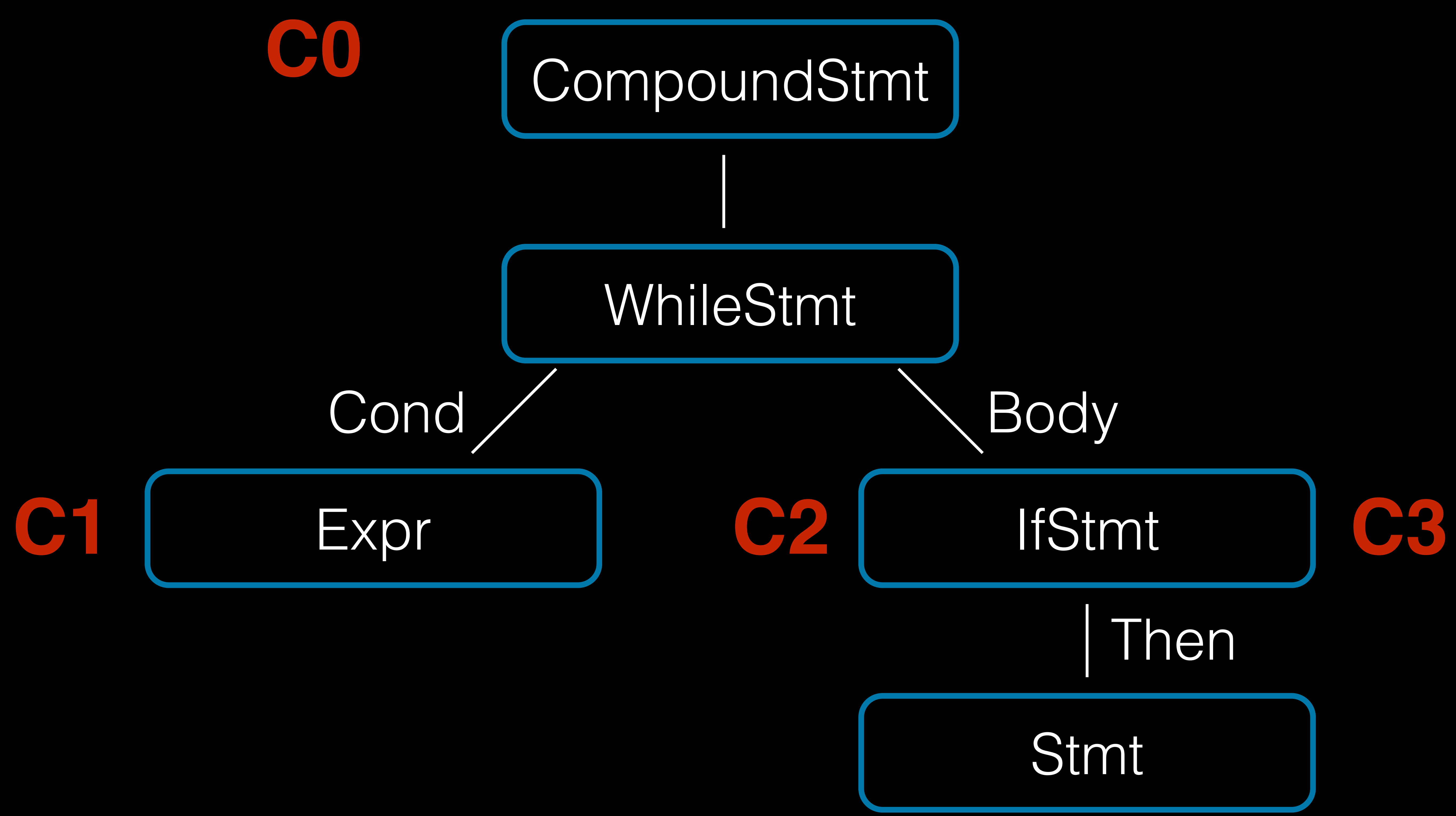
Example



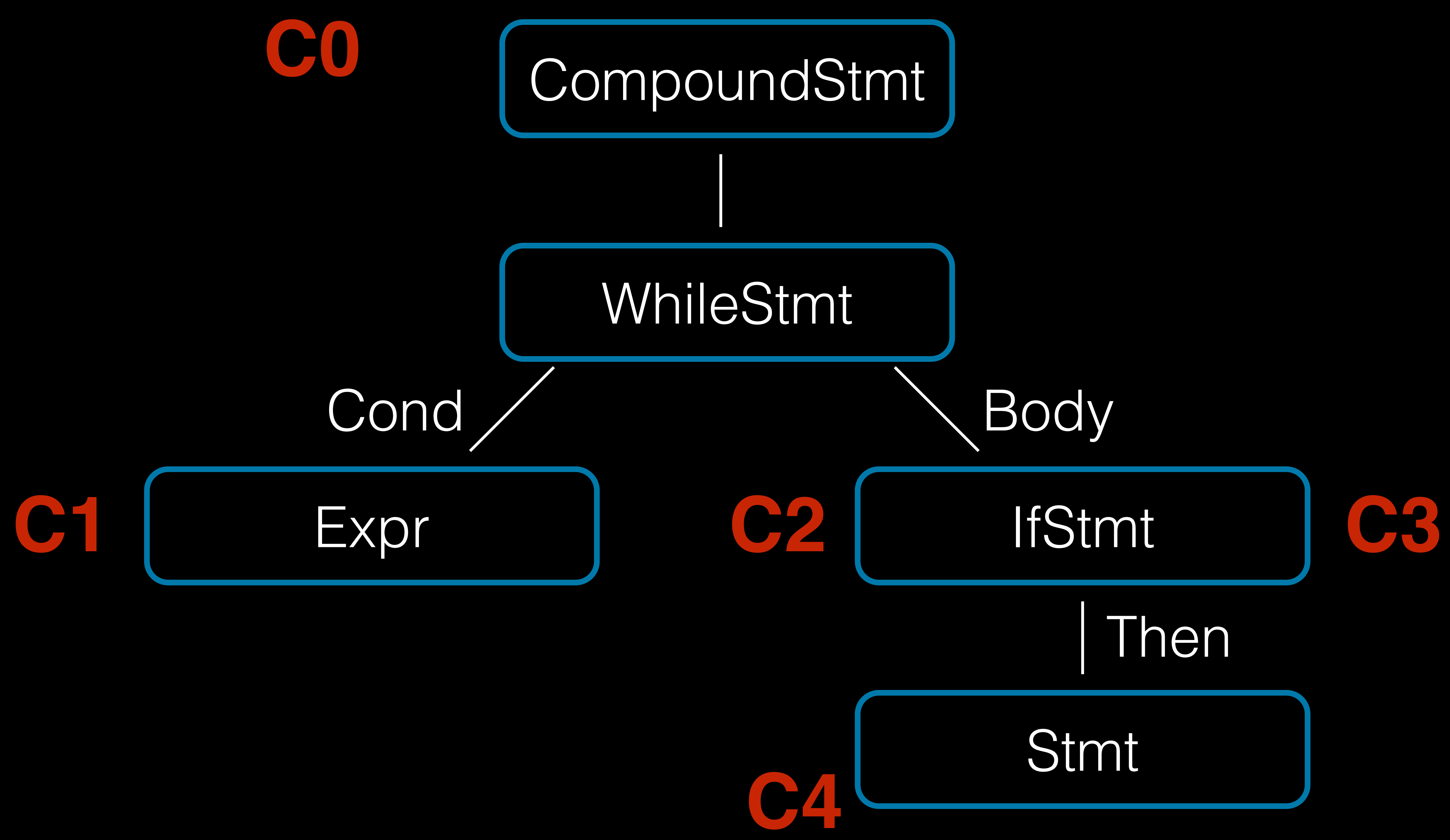
Example



Example


C0
CompoundStmt
WhileStmt
Cond
Body
C1
Expr
C2
IfStmt
C3
Then
Stmt

Example

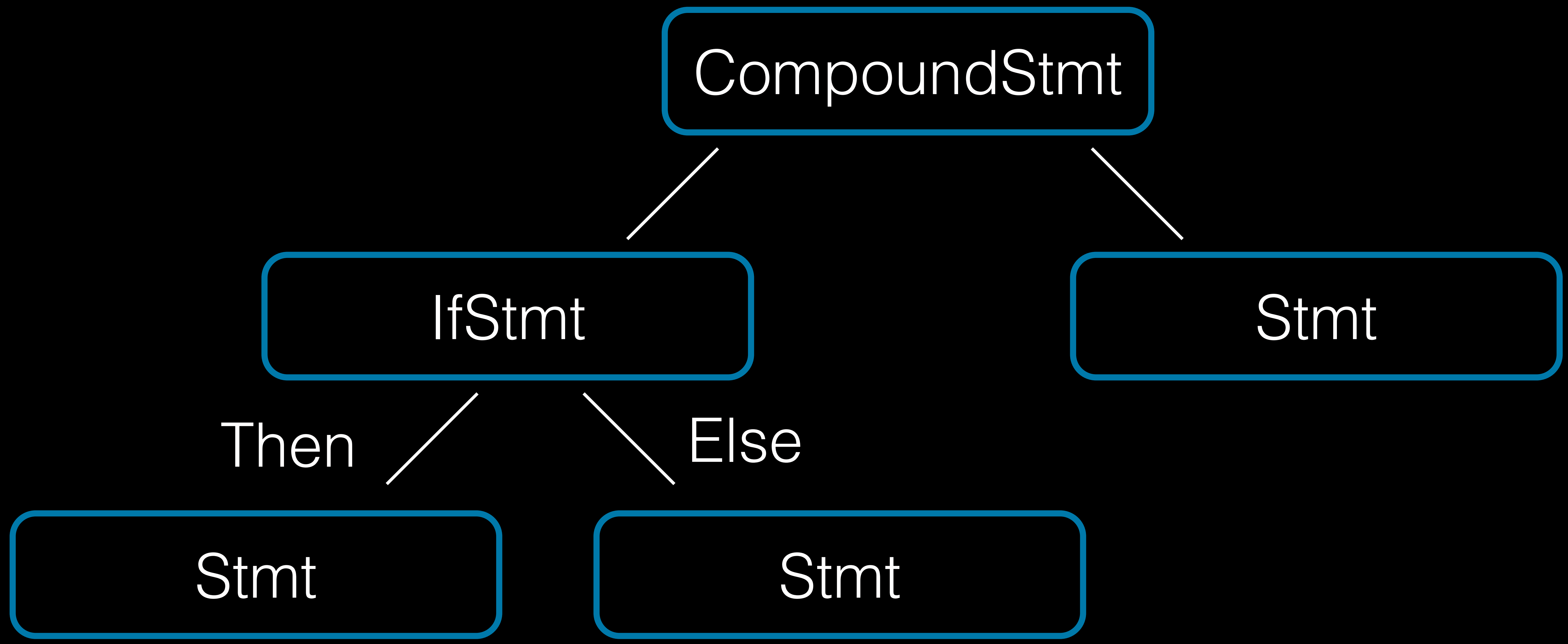


C0
CompoundStmt
WhileStmt
Cond
Body
C1
Expr
C2
IfStmt
C3
Then
C4
Stmt

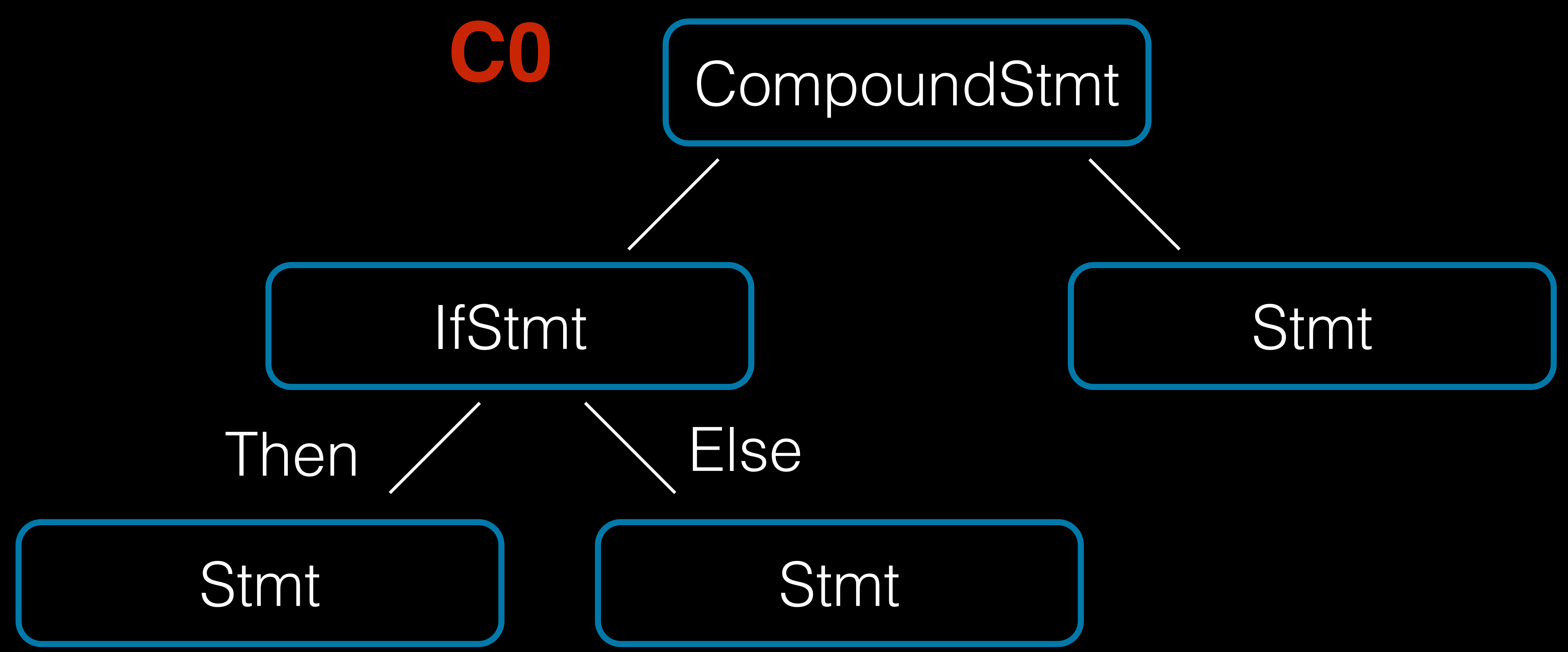
Minimizing Overhead

- Not every block needs a counter
- CFG-based approach: compute a spanning tree
- Can often do as well by following AST structure

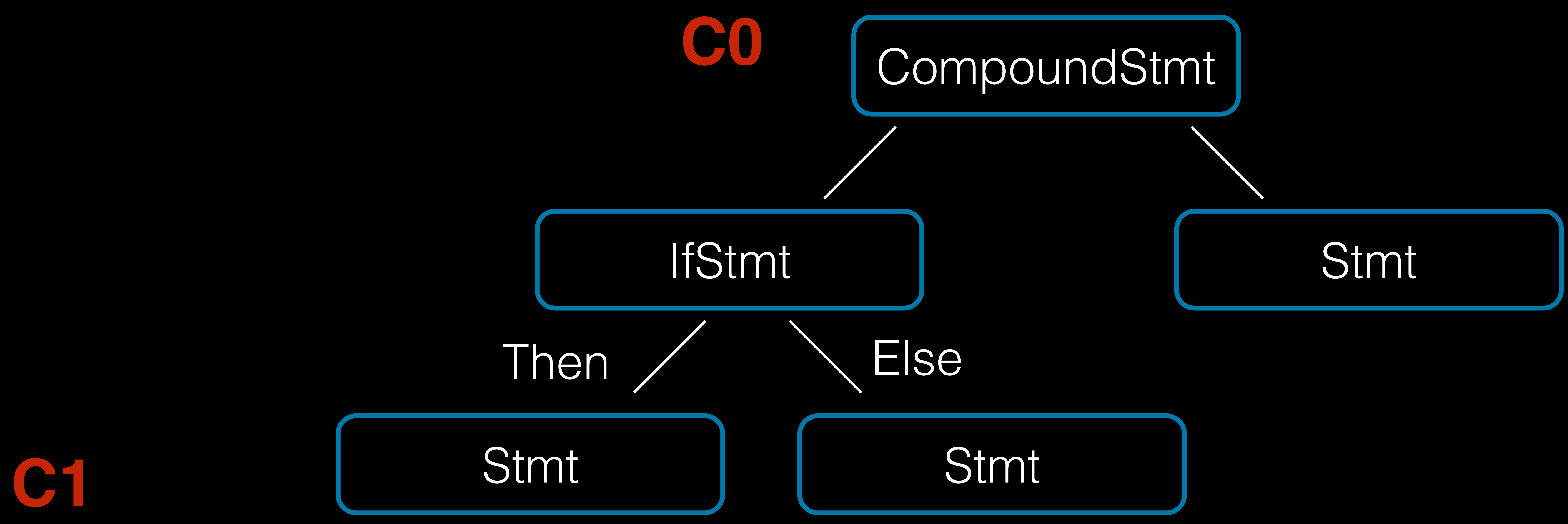
Example



Example



Example

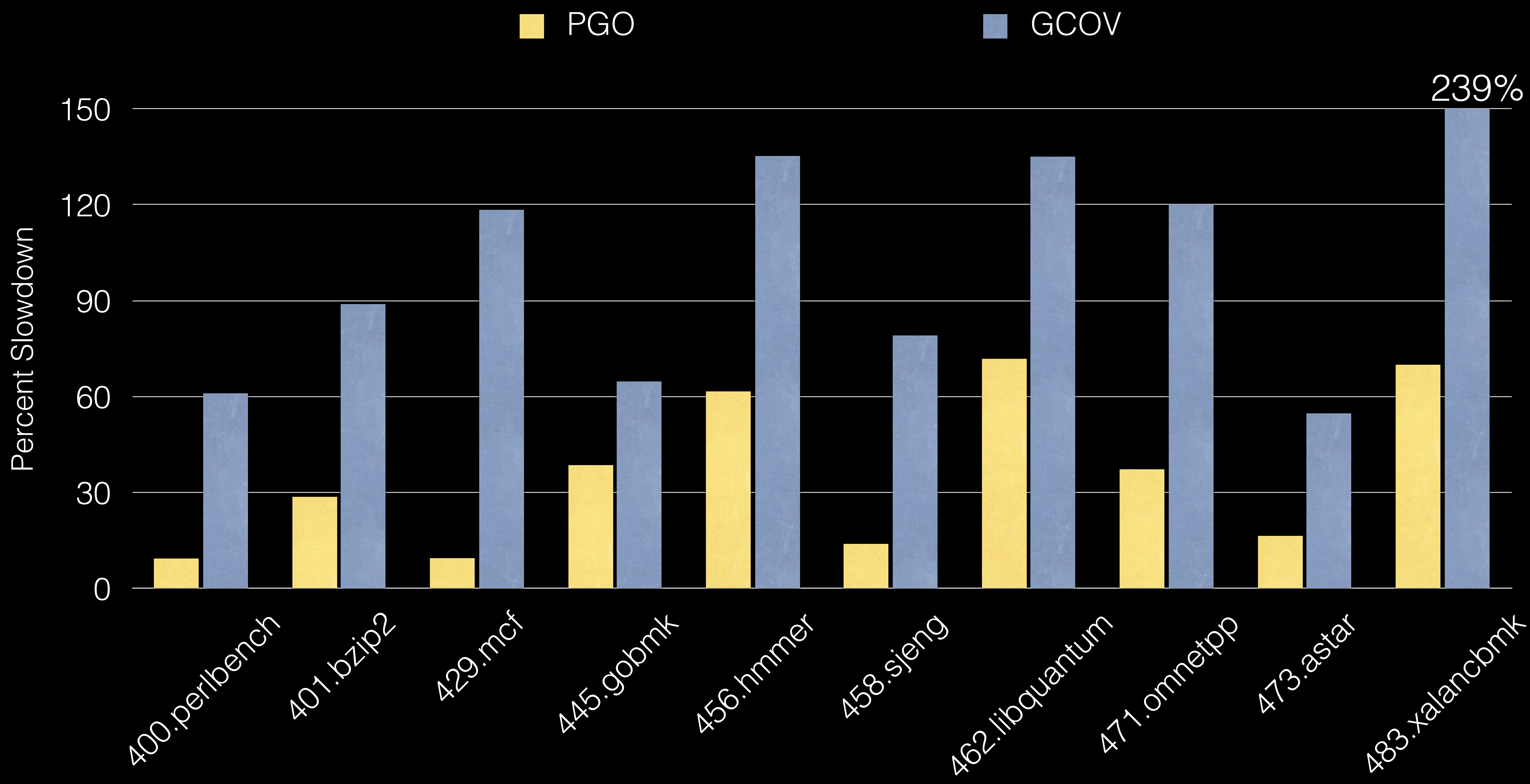


No-Return Calls

- Important for code coverage
- Not an issue for PGO
  (we don’t have a “likely no-return” attribute)
- A counter after every call would be expensive
- Can we get away with ignoring this?

Instrumentation Overhead: Compile Time

Instrumentation Overhead: Execution Time



PGO with External Profiling

Diego Novillo

External Profilers

- No changes needed to user application
- Binary runs under control of profiler
  - binary instrumentation (valgrind, cachegrind)
  - hardware counters (perf, oprofile)
- Profilers using HW counters → low overhead
- Profiler saves profile results in a file
  - Used as input to analysis tools
  - Why not use it as input to the compiler?

```
$ perf annotate -l
[ … ]
         :        for (int i = 0; i < N; i++) {
         :          A *= i / 32;
 /home/dnovillo/prog.cc:5
    9.18% :        400520:       mov    %eax,%ecx
    0.00% :        400522:       sar    $0x1f,%ecx
    0.00% :        400525:       shr    $0x1b,%ecx
    0.00% :        400528:       add    %eax,%ecx
    7.89% :        40052a:       sar    $0x5,%ecx
    0.00% :        40052d:       xorps  %xmm0,%xmm0
    0.00% :        400530:       cvtsi2sd %ecx,%xmm0
    8.23% :        400534:       mulsd  0x200aec(%rip),%xmm0        # 601028 <A>
   66.10% :        40053c:       movsd  %xmm0,0x200ae4(%rip)        # 601028 <A>
[ … ]
```

GOAL: Use all the collected runtime knowledge as input to the optimizers

Why External Profiler?

- No need for instrumented builds
  - Simplifies build rules for user application
  - No build time overhead

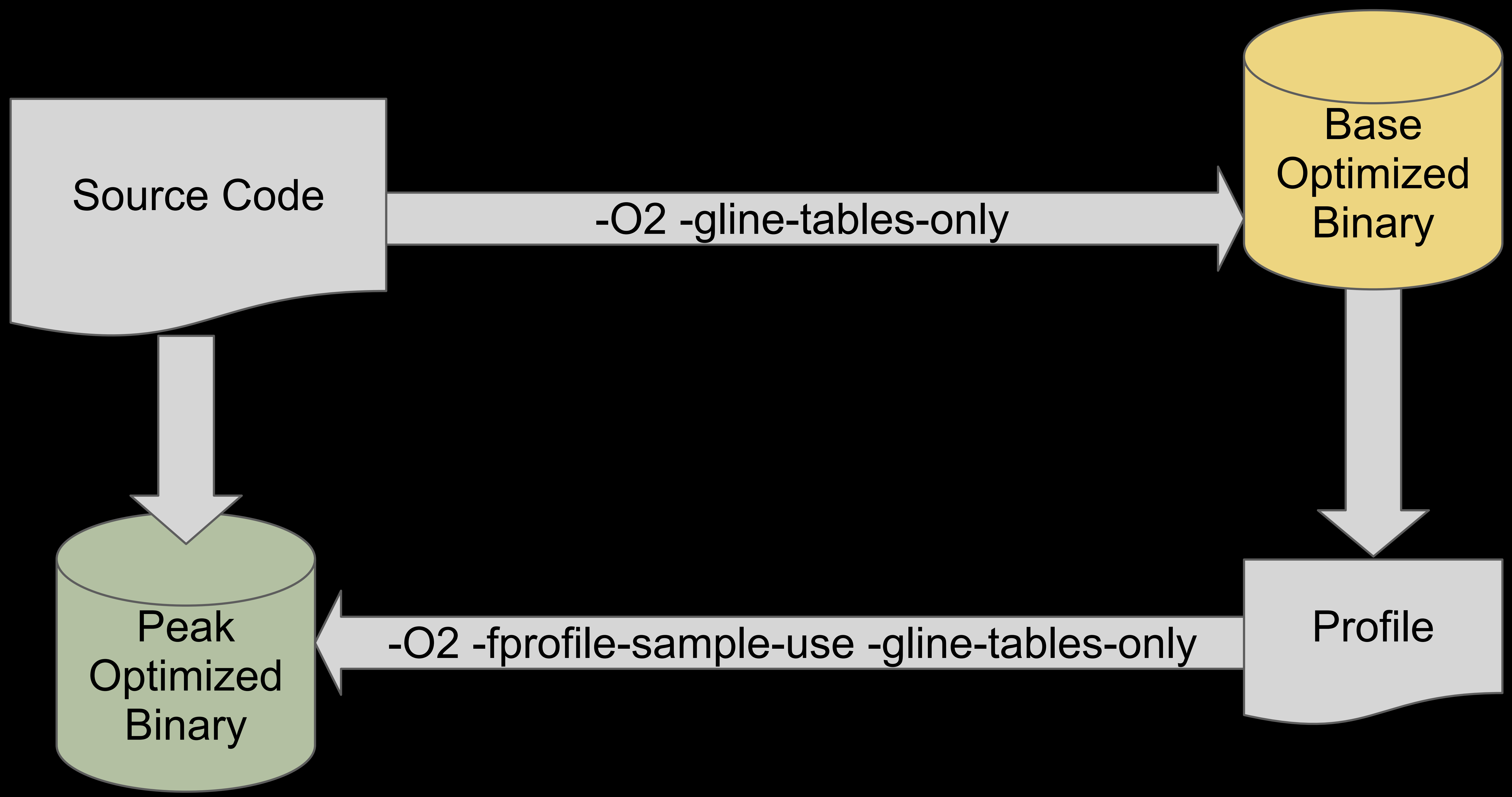
Why External Profiler?

- Very low runtime overhead (< 1%)
  - Profiles can be collected in production environments
  - Profile data is more representative
  - Training is done on actual production loads

Why External Profiler?

- Allows application-specific profilers
  - e.g., game engines
  - ***Anything*** that can be converted into hints to the compiler

User Model



Execute under profiler (low overhead)

Design

- Profile data often needs conversion
  - Samples are associated with processor instructions
  - External tool converts into mapping to source LOCs
- Bad/stale/missing profiles
  - Never affect correctness
  - Only affect performance
- Scalar pass incorporates profile into IR
  - Source locations mapped to IR instructions
  - Profile kind dictates representation
  - Optimizers query via standard analysis pass API
  - Analysis routines fallback on static heuristics

Current Implementation

1. Conversion tool for Linux Perf
   (Sample-based profiles)
2. Samples converted to branch weights
3. Profile pass simply annotates the IR
4. Analysis uses IR metadata for estimates
5. Optimizers automatically adjust cost models
   (Provided they use the Analysis API properly)
   (Work is needed in this area)

Limitations & Restrictions

Profile says "LIAR!"

```
foo(int x) {
  if (__builtin_expect(x > 100, 1))
    hot();
  else
    cold();
}

main() {
  while (true) foo(rand() % 100);
}
```

- Program behaviour must coincide with profile
  - Stale profiles degrade performance (significantly)
  - Non-representative runs mislead optimizers
- Who do we listen to?
- Warn the user?
- Silently override?
- Is the profile representative?

Limitations & Restrictions

Line 2 is HOT according to profile

Need to know where in the line

- Column numbers
- DWARF discriminators

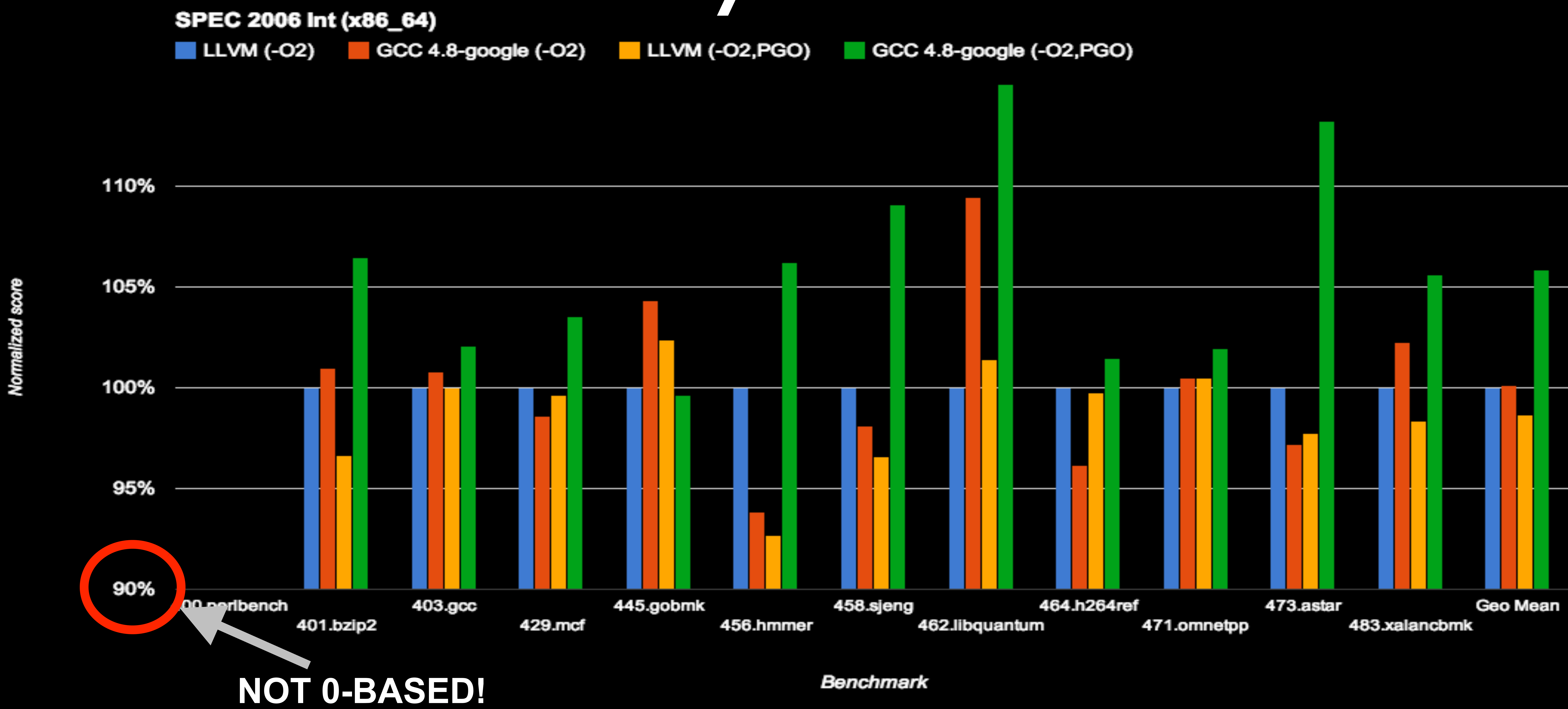
```
1  foo(int x) {
2    if (x < 100) hot(); else cold();
3  }
4
5  main() {
6    while (true) foo(rand() % 100);
7  }
```

- HW counters → IR mapping is lossy
- Requires good line table information
- Many instructions on the same line of code

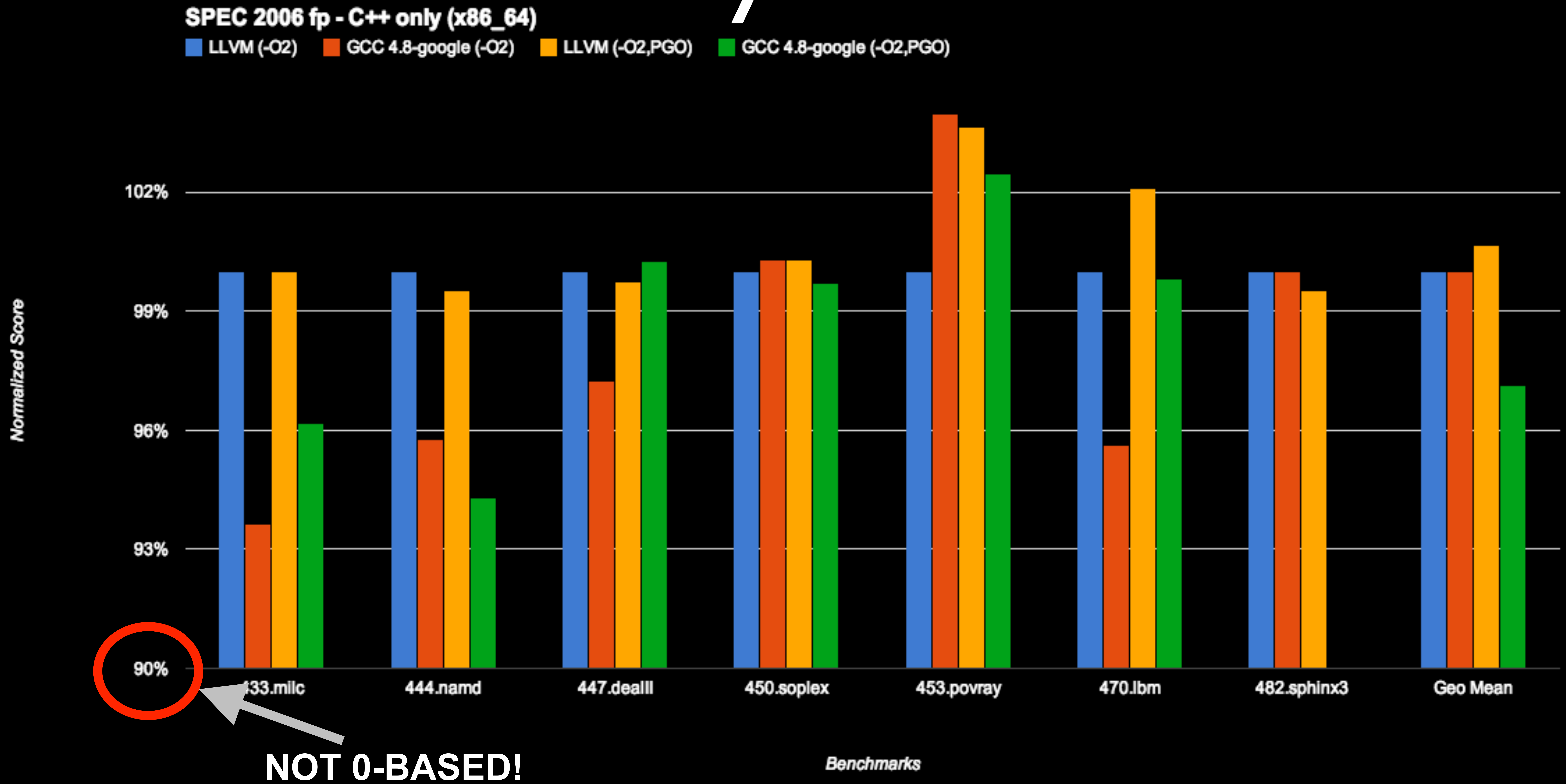
Limitations & Restrictions

- The optimizer must use profiles!
  - Notably, the inliner

Early Results



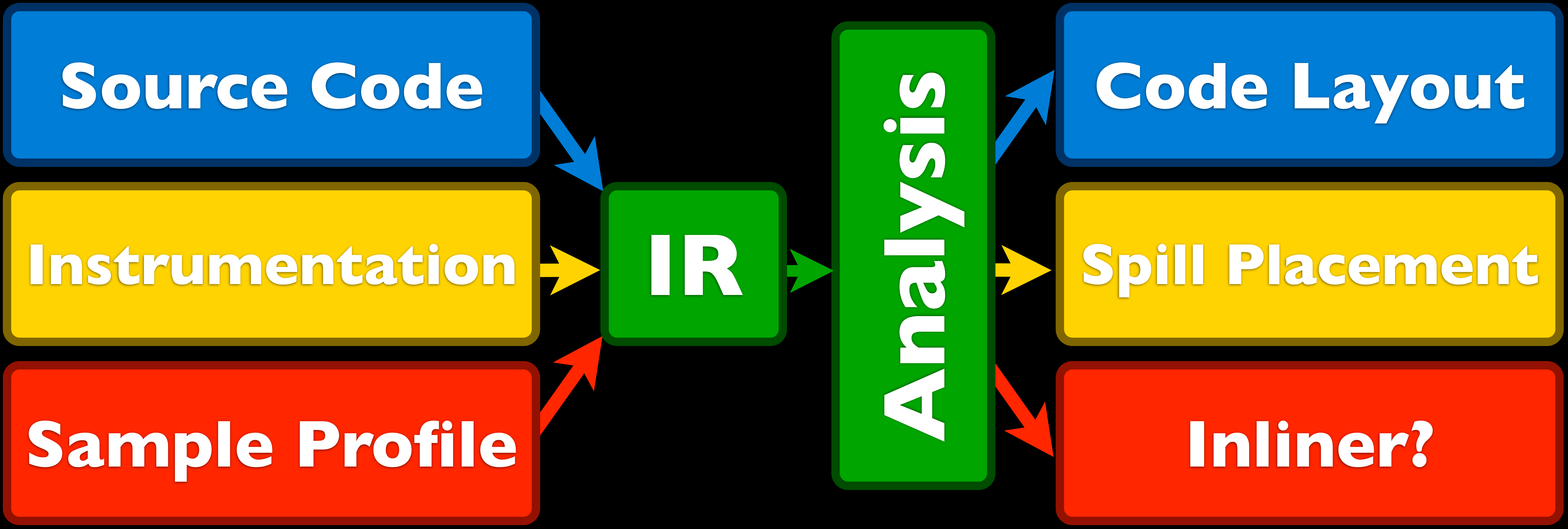
Early Results



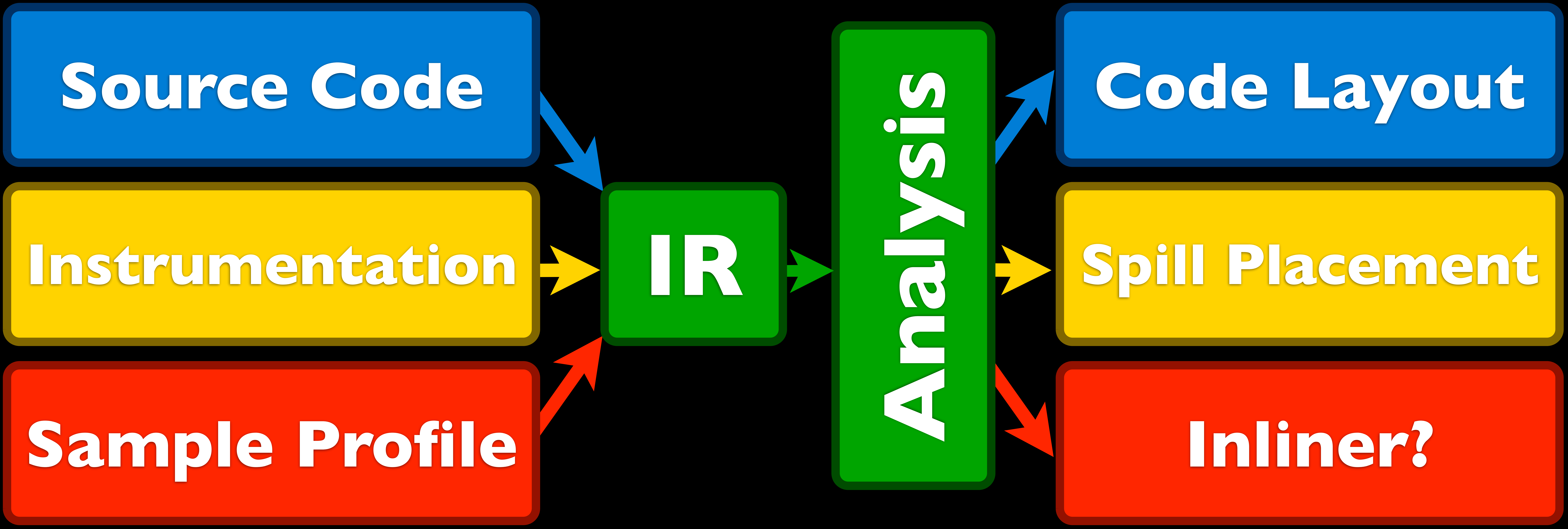
Status

- Profile conversion tool for Linux Perf Events
  - Writes flat profiles to text file
  - Working on release
- Scalar pass works with SPEC2006
  - Produces branch weights
  - Trunk patches under review
- In the works
  - Other function attributes (e.g. cold)
  - More efficient profile encoding (bitcode)
  - Context aware profiles
  - Other profile types
    - value profiles to disambiguate indirect calls

So, we have some profile data...
Now what?

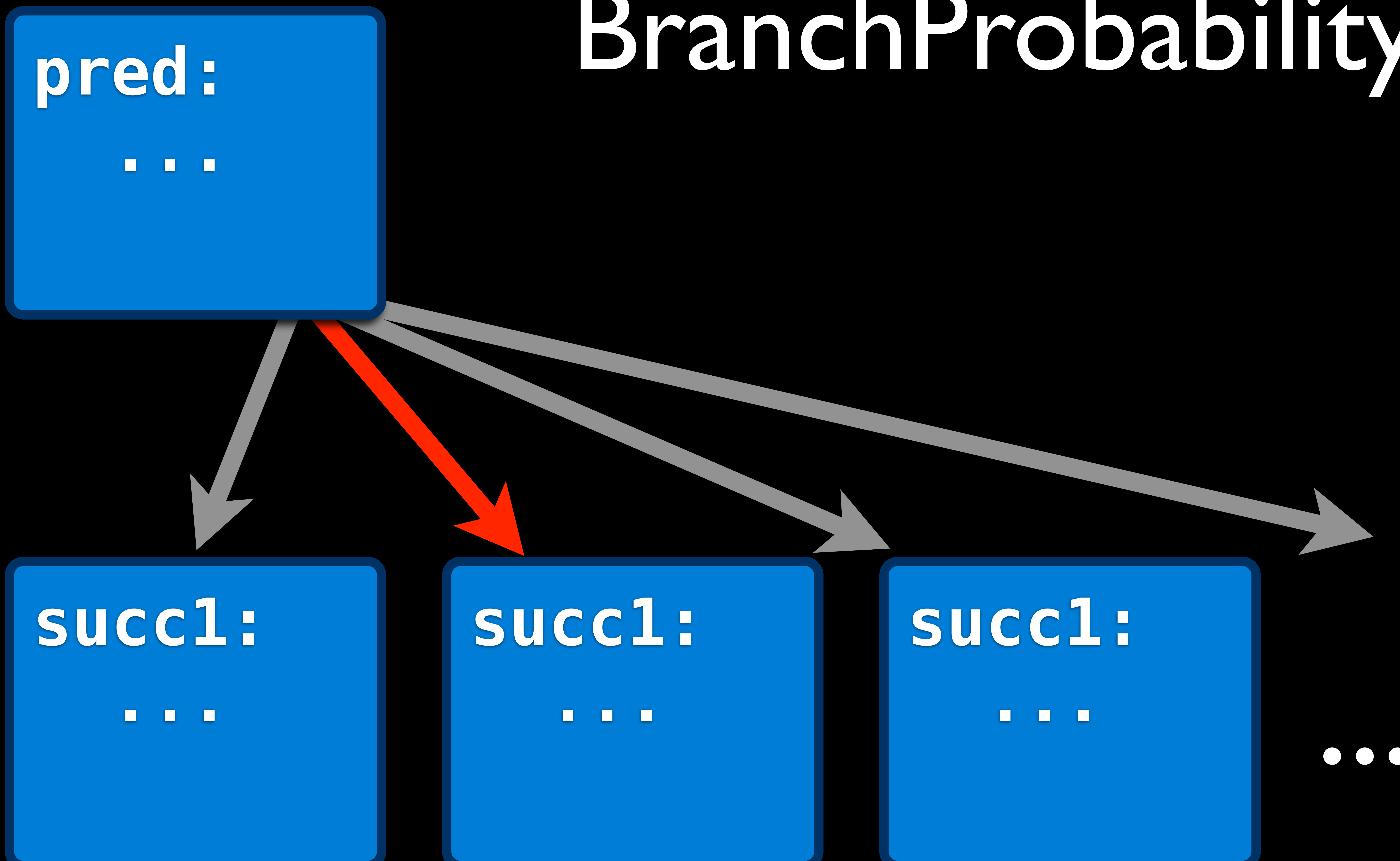


All profile info ends up in a common IR annotation



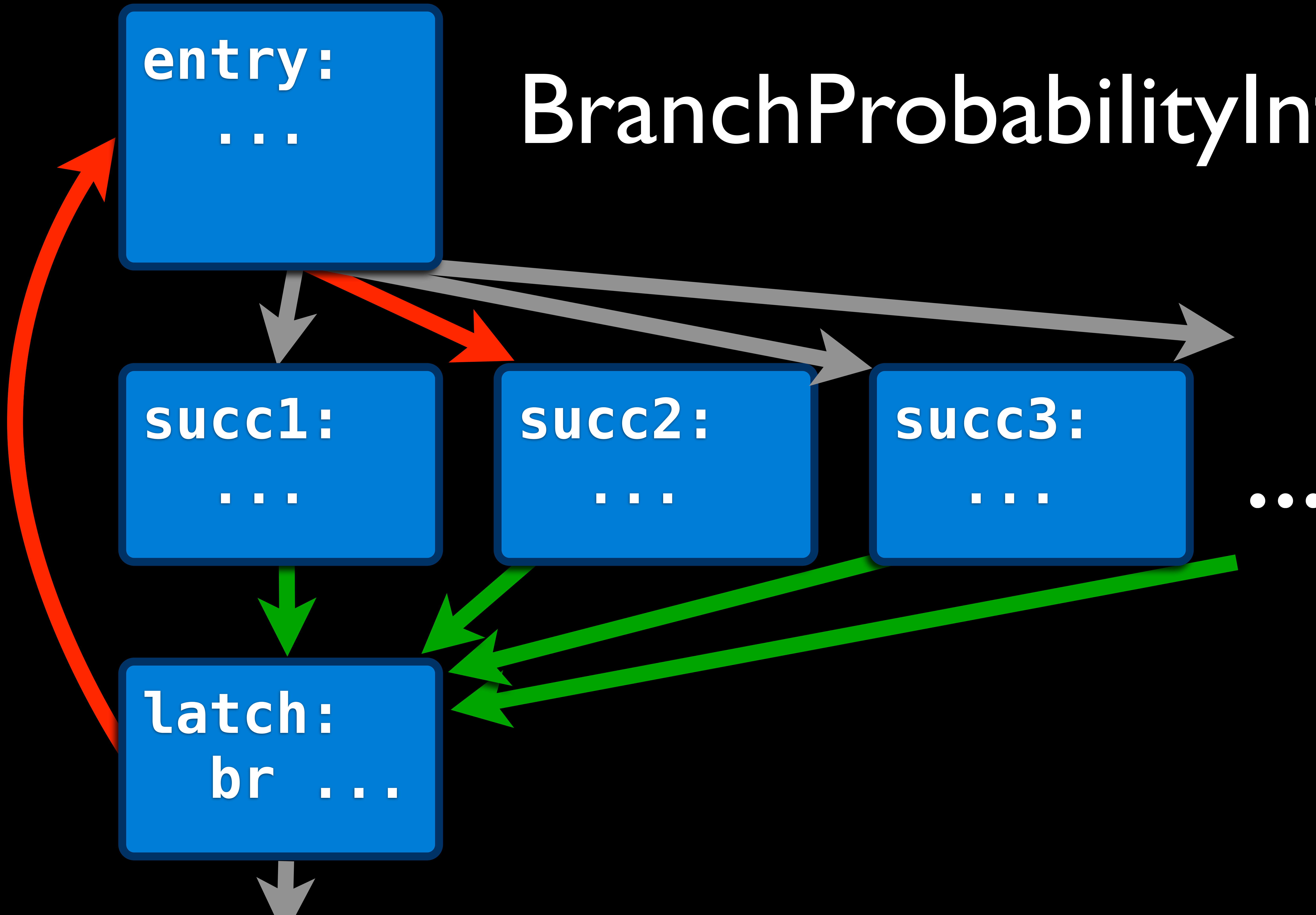
Passes access it through a common analysis API

BranchProbabilityInfo



```
define void @f(i1 %a) {
entry:
  ...
  br i1 %a, label %t, label %f, !prof !0

t:
  ...
  br label %exit

f:
  ...
  br label %exit

exit:
  ret void
}
!0 = metadata !{metadata !"branch_weights", i32 64, i32 4}
```

BranchProbabilityInfo



```
define void @f(i1 %a) {
entry:
  ...
  br i1 %a, label %t, label %f, !prof !0

t:
  ...
  unreachable

f:
  ...
  br label %exit

exit:
  ret void
}
!0 = metadata !{metadata !"branch_weights", i32 64, i32 4}
```

```
define void @f(i1 %a) {
entry:
  ...
  br i1 %a, label %t, label %f

t:
  ...
  call coldcc void @g()
  ...
  br label %exit

f:
  ...
  br label %exit

exit:
  ret void
}

declare coldcc void @g()
```

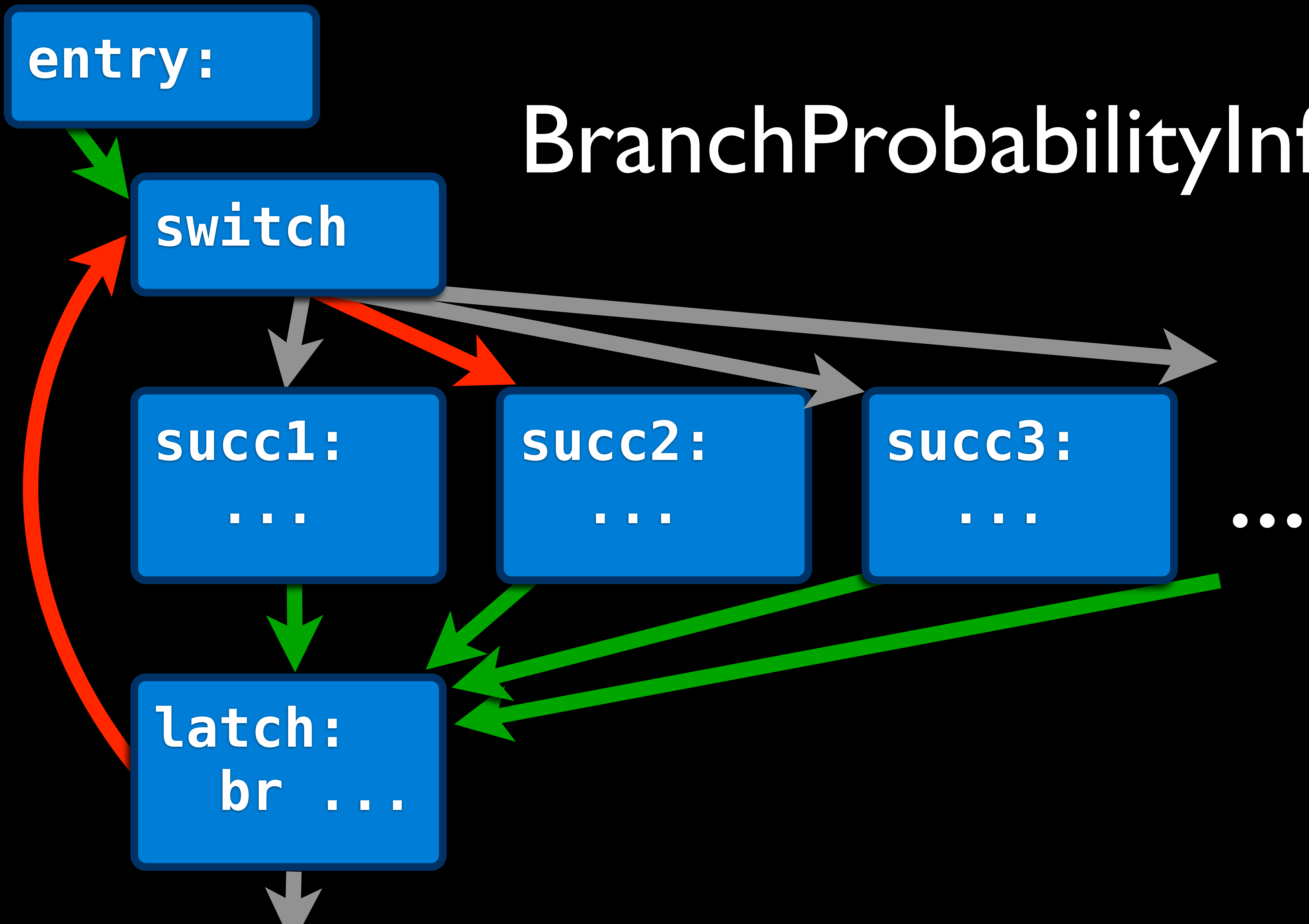
```
define void @f(i32 %i) {
entry:
  %a = icmp eq i32 %i, 0
  br i1 %a, label %t, label %f

t:
  ...
  br label %exit

f:
  ...
  br label %exit

exit:
  ret void
}
```

```
define void @f(i32 %i) {
entry:
  %a = icmp ne i32 %i, 0
  br i1 %a, label %t, label %f

t:
  ...
  br label %exit

f:
  ...
  br label %exit

exit:
  ret void
}
```

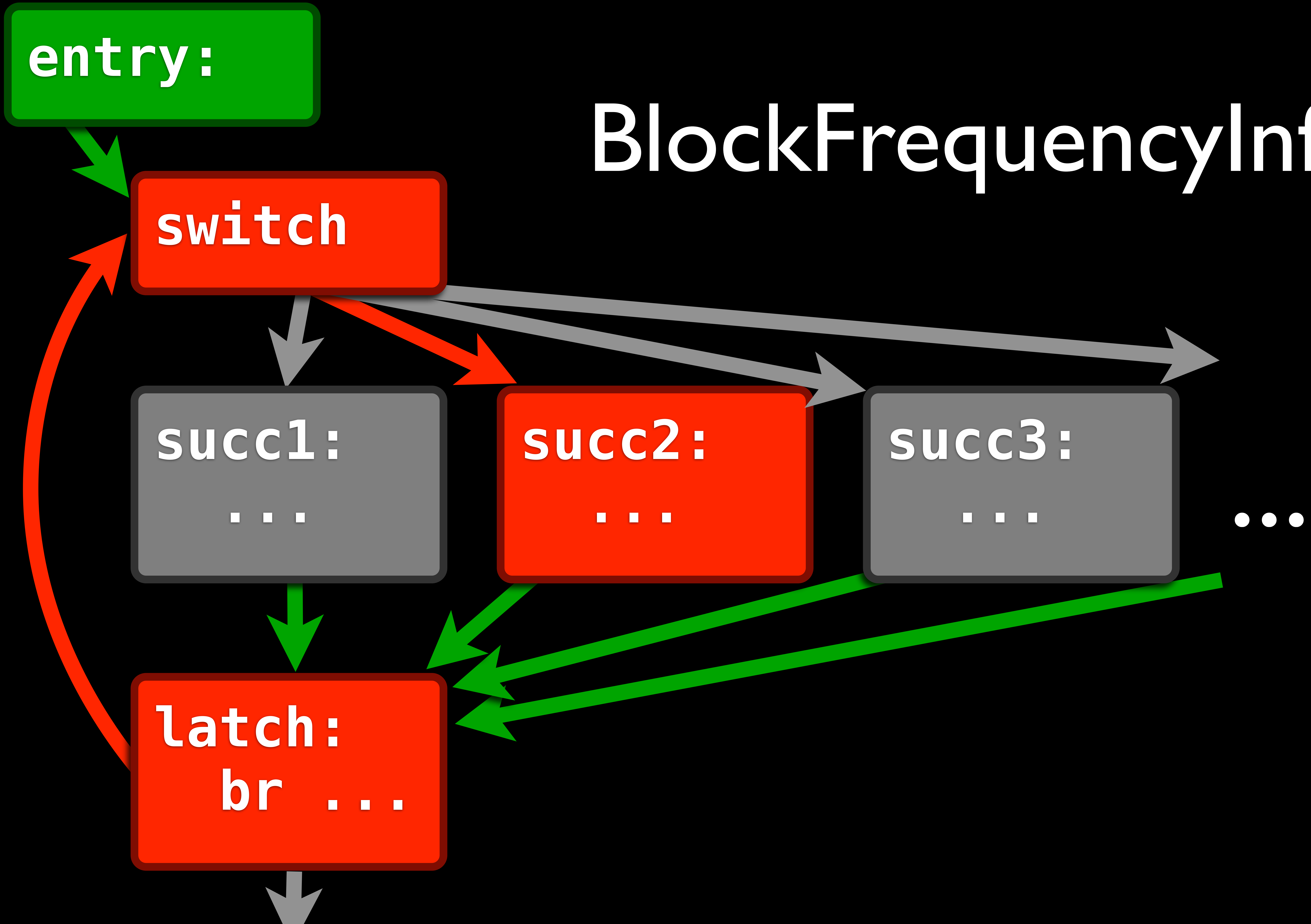
```
define void @f(i32 %i) {
entry:
  %a = icmp slt i32 %i, 0
  br i1 %a, label %t, label %f

t:
  ...
  br label %exit

f:
  ...
  br label %exit

exit:
  ret void
}
```

```
define void @f(i8* %p) {
entry:
  %a = icmp eq i8* %p, null
  br i1 %a, label %t, label %f

t:
  ...
  br label %exit

f:
  ...
  br label %exit

exit:
  ret void
}
```

BranchProbabilityInfo



BlockFrequencyInfo



What about MI?
Everything is there too.

Resolving Conflicts

- Some times the profile will directly conflict with other information:
  - Static heuristics may be contradicted
  - Other profiles may be incompatible
- Need to be extremely cautious when disregarding profile information, but may be necessary
  - When we have bad profiles, bounding the bad impact is both hard and important

The hard part: cache invalidation!

- What happens when an optimization pass transforms the CFG in a way that invalidates annotations on the IR?
- The analyses are easy -- we re-run them
- Annotations are hard

Before...

```
define void @f(i1 %a) {
entry:
  ...
  br i1 %a, label %t, label %f, !prof !0

t:
  ...
  br label %exit

f:
  ...
  br label %exit

exit:
  %phi = phi i32 [ ..., %t ], [ ..., %f ]
  ret void
}
!0 = metadata !{metadata !"branch_weights", i32 64, i32 4}
```

After...

```
define void @f(i1 %a) {
entry:
  ...
  br i1 %a, label %f, label %t, !prof !0

t:
  ...
  br label %exit

f:
  ...
  br label %exit

exit:
  %phi = phi i32 [ ..., %t ], [ ..., %f ]
  ret void
}
!0 = metadata !{metadata !"branch_weights", i32 4, i32 64}
```

Before...

```
define void @f(i1 %a) {
entry:
  ...
  br i1 %a, label %t, label %f, !prof !0

t:
  ...
  br label %exit

f:
  ...
  br label %exit

exit:
  %phi = phi i32 [ ..., %t ], [ ..., %f ]
  ret void
}
!0 = metadata !{metadata !"branch_weights", i32 64, i32 4}
```

After...

```
define void @f(i1 %a) {
entry:
  ...
  ...
  ...
  %phi = select i1 %a, i32 ..., ...
  br i1 %a, label %t, label %f, !prof !0

t:
  br label %exit

f:
  br label %exit

exit:
  ret void
}
!0 = metadata !{metadata !"branch_weights", i32 64, i32 4}
```

Before...

```
define void @f(i32 %a, i32 %b, i32 %c, i32 %d) {
entry:
  ...
  %x = icmp eq i32 %a, %b
  %y = icmp eq i32 %c, %d
  %xy = and i1 %x, %y
  br i1 %xy, label %t, label %f, !prof !0
t:
  ...
  br label %exit
f:
  ...
  br label %exit
exit:
  %phi = phi i32 [ ..., %t ], [ ..., %f ]
  ret void
}
!0 = metadata !{metadata !"branch_weights", i32 64, i32 4}
```

After...

```
define void @f(i32 %a, i32 %b, i32 %c, i32 %d) {
entry:
  ...
  %x = icmp eq i32 %a, %b
  br i1 %x, label %entry2, label %f, !prof !0
entry2:
  %y = icmp eq i32 %c, %d
  br i1 %y, label %t, label %f, !prof !0
t:
  ...
  br label %exit
f:
  ...
  br label %exit
exit:
  %phi = phi i32 [ ..., %t ], [ ..., %f ]
  ret void
}
!0 = metadata !{metadata !"branch_weights", i32 64, i32 4}
```

Need other annotations?

- While we believe that block frequency can and should be derived from branch weight, there are other things being profiled
- May need module-wide call site or function definition annotation
- May need value-based annotation for value profiling

Profile Guided Transforms

Spill Placement

- RA has a collection of potential values to spill from registers onto the stack to satisfy the allocation problem
- Which spill is chosen will cause a spill inside of different blocks
- Can use profile information to prioritize the hot path's in-register values

Code Layout

- Called MachineBlockPlacement
- Runs at the very end of MI to lay out the code of a single function
- Primarily layout is driven based on the topological structure of the CFG and loop nest structure
  - Ties are broken using profile information
  - Cold regions of code are extracted out-of-line

Hot/Cold Partitioning?

- GCC picks a partition point in the layout of the function and emits the two halves under different sections
- The linker can then group the hot regions together, fully isolating the cold code frem the hot code even at an IP level

The Inliner

- Today, the inliner doesn't even know profile information exists. Oops.
- LLVM's inliner is also unusual: mostly focused on enabling simplifications: constant propagation, combining, etc.
- Consequentially the primary expected change is to avoid inlining into cold regions unhelpfully.

Outlining & Merging

- The more radical change we would like is to do function outlining for cold regions
- This will in turn allow a significantly larger set of non-cold paths to be considered for simplifying inlining
- Forms in essence a partial inliner by splitting it into two steps
- Outlining in the middle-end allows merging of common cold regions (perhaps expanded via macros) by outlining them to functions and then running merge functions.

PGO Summary

- Strong analysis support from annotations down
- Two parallel and complementary efforts to annotate with profile information, this is going on right now!
- Most basic profile guided transformations in place
- Still a lot of work to do on other transforms (inlining, etc)

Questions?