Qualcomm

Inliner in MLIR

Javed Absar, Principal Engineer
UK Reservoir Labs R&D
Qualcomm Technologies International, Ltd.

Snapdragon and Qualcomm branded products are products of Qualcomm Technologies, Inc. and/or its subsidiaries.

Contents

- MLIR – highly versatile, extensible compiler infrastructure
- Inlining in MLIR world
- MLIR Inliner
- Conclusion

MLIR: versatile IR

Affine Dialect

```
func.func private @foo_0(%a : memref<10xf32>, %b : memref<10xf32>) {
  affine.for %i = 0 to 10 {
    %v0 = affine.load %b[%i] : memref<10xf32>
    %v1 = affine.load %a[%i] : memref<10xf32>
    %v2 = arith.addf %v0, %v1 : f32
    affine.store %v2, %b[%i] : memref<10xf32>
  }
  return
}
```

LLVM

```
9:                                    ; preds = %9, %6
  %10 = phi i64 [ 0, %6 ], [ %16, %9 ]
  %11 = getelementptr inbounds float, float* %0, i64 %10
  %12 = load float, float* %11, align 4, !tbaa !2
  %13 = getelementptr inbounds float, float* %1, i64 %10
  %14 = load float, float* %13, align 4, !tbaa !2
  %15 = fadd float %12, %14
  store float %15, float* %13, align 4, !tbaa !2
  %16 = add nuw nsw i64 %10, 1
  %17 = icmp eq i64 %16, %7
  br i1 %17, label %8, label %9
```

MLIR: Affine.for (ODS)

```
def AffineForOp : Affine_Op<"for",
    [AutomaticAllocationScope, ImplicitAffineTerminator, ConditionallySpeculatable,
     RecursiveMemoryEffects, DeclareOpInterfaceMethods<LoopLikeOpInterface,
     ["getSingleInductionVar", "getSingleLowerBound", "getSingleStep",
      "getSingleUpperBound"]>,
     DeclareOpInterfaceMethods<RegionBranchOpInterface,
     ["getSuccessorEntryOperands"]>]> {
  let summary = "for operation";
  let description = [{ …}];

  let arguments = (ins Variadic<AnyType>);
  let results = (outs Variadic<AnyType>:$results);
  let regions = (region SizedRegion<1>:$region);
  …
}
```

MLIR: Affine.for (Generic)

```
affine.for %i = 0 to 10 {
  %v0 = affine.load %b[%i] : memref<10xf32>
  %v1 = affine.load %a[%i] : memref<10xf32>
  %v2 = arith.addf %v0, %v1 : f32
  affine.store %v2, %b[%i] : memref<10xf32>
}
```

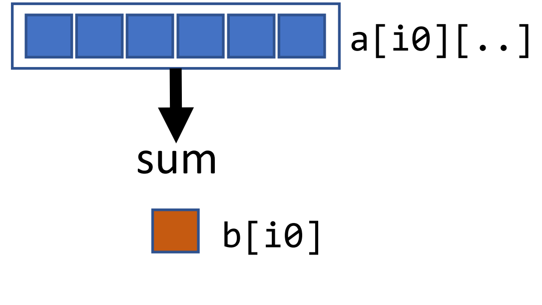
```
mlir-opt  -mlir-print-op-generic file.mlir
```

```
#map = affine_map<(d0) -> (d0)>
#map1 = affine_map<() -> (0)>
#map2 = affine_map<() -> (10)>
"builtin.module"() ({
  "func.func"() ({
  ^bb0(%arg0: memref<10xf32>, %arg1: memref<10xf32>):
    "affine.for"() ({
    ^bb0(%arg2: index):
      %0 = "affine.load"(%arg1, %arg2) {map = #map} : (memref<10xf32>, index) -> f32
      %1 = "affine.load"(%arg0, %arg2) {map = #map} : (memref<10xf32>, index) -> f32
      %2 = "arith.addf"(%0, %1) {fastmath = #arith.fastmath<none>} : (f32, f32) -> f32
      "affine.store"(%2, %arg1, %arg2) {map = #map} : (f32, memref<10xf32>, index) -> ()
      "affine.yield"() : () -> ()
    }) {lower_bound = #map1, step = 1 : index, upper_bound = #map2} : () -> ()
    "func.return"() : () -> ()
  }) {function_type = (memref<10xf32>, memref<10xf32>)
          -> (), sym_name = "foo_0", sym_visibility = "private"} : () -> ()
}) : () -> ()
```

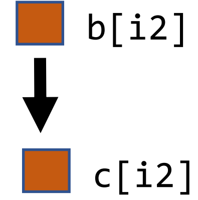
Inlining in MLIR world

```
func.func @foo(%a : memref<10x10xf32>, %b : memref<10xf32>, %c : memref<10xf32>) {
  func.call @foo_0(%a, %b) : (memref<10x10xf32>, memref<10xf32>) -> ()
  func.call @foo_1(%b, %c) : (memref<10xf32>, memref<10xf32>) -> ()
  return
}
```

```
func.func private @foo_0(%a : memref<10x10xf32>, %b : memref<10xf32>) {
  affine.for %i0 = 0 to 10 {
    affine.for %i1 = 0 to 10 {
      %v0 = affine.load %b[%i0] : memref<10xf32>
      %v1 = affine.load %a[%i0, %i1] : memref<10x10xf32>
      %v3 = arith.addf %v0, %v1 : f32
      affine.store %v3, %b[%i0] : memref<10xf32>
    }
  }
  return
}
```




```
func.func private @foo_1(%b : memref<10xf32>, %c : memref<10xf32>) {
  affine.for %i2 = 0 to 10 {
    %v4 = affine.load %b[%i2] : memref<10xf32>
    affine.store %v4, %c[%i2] : memref<10xf32>
  }
  return
}
```



Inlining in MLIR world

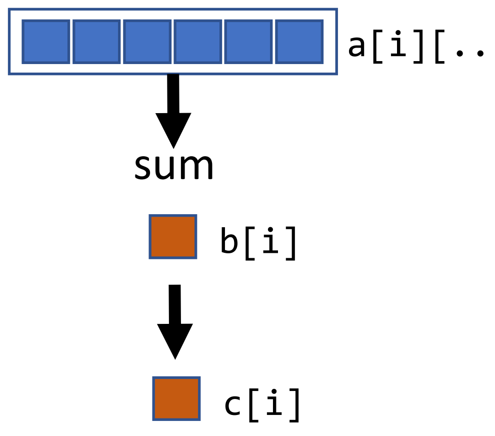
```
mlir-opt -inline='op-pipelines=func.func(canonicalize,cse)’ ex.mlir –o inlined.mlir
```

```
module {
  func.func @foo(%a: memref<10x10xf32>, %b: memref<10xf32>,
                 %c: memref<10xf32>) {
    affine.for %i = 0 to 10 {
      affine.for %j = 0 to 10 {
        %0 = affine.load %b[%i] : memref<10xf32>
        %1 = affine.load %a[%i, %j] : memref<10x10xf32>
        %2 = arith.addf %0, %1 : f32
        affine.store %2, %b[%i] : memref<10xf32>
      }
    }
    affine.for %i = 0 to 10 {
      %0 = affine.load %b[%i] : memref<10xf32>
      affine.store %0, %c[%i] : memref<10xf32>
    }
    return
  }
}
```

Inlining in MLIR world

```
mlir-opt  inlined.mlir  -pass-pipeline='builtin.module(func.func(affine-loop-fusion))'
```

```
module {
  func.func @foo(%a: memref<10x10xf32>, %b: memref<10xf32>,
                 %c: memref<10xf32>) {
    affine.for %i = 0 to 10 {
      affine.for %j = 0 to 10 {
        %1 = affine.load %b[%i] : memref<10xf32>
        %2 = affine.load %a[%i, %j] : memref<10x10xf32>
        %3 = arith.addf %1, %2 : f32
        affine.store %3, %b[%i] : memref<10xf32>
      }
      %0 = affine.load %b[%i] : memref<10xf32>
      affine.store %0, %c[%i] : memref<10xf32>
    }
    return
  }
}
```



How to Inline ? MLIR Interfaces

- Generic way of interacting with the IR
- Corner-stone of MLIR extensibility
- Provide information to the transformation/analysis pass
- Interfaces (properties of interest) defined by the transformation
- Transformation/Analyses in terms of interfaces

```
TestInterfaces, ArithOpsInterfaces,OpenMPOpsInterfaces,LLVMInterfaces, MatchInterfaces,
TransformInterfaces,TosaInterfaces, BuiltinAttributeInterfaces,SymbolInterfaces,
AffineMemoryOpInterfaces, LinalgInterfaces, VectorInterfaces, ShapedOpInterfaces,
CallInterfaces, CastInterfaces, DataLayoutInterfaces, …
```

DialectInlinerInterface

```
InliningUtils.h
```

```
/// This is the interface that must be implemented by the dialects of operations
/// to be inlined. This interface should only handle the operations of the …
class DialectInlinerInterface
    : public DialectInterface::Base<DialectInlinerInterface> {
public:
  //===----------------------------------------------------------------------===//
  // Analysis Hooks
  //===----------------------------------------------------------------------===//

  /// Returns true if the given operation 'callable', that implements the
  /// 'CallableOpInterface', can be inlined into the position given call…
  virtual bool isLegalToInline(Operation *call, Operation *callable,
                               bool wouldBeCloned) const;
   ...
  //===----------------------------------------------------------------------===//
  // Transformation Hooks
  //===----------------------------------------------------------------------===//
  virtual void handleTerminator(Operation *op, Block *newDest) const;
  ...
```

DialectInlinerInterface

LLVMInlining.cpp

```
struct LLVMInlinerInterface : public DialectInlinerInterface {
  using DialectInlinerInterface::DialectInlinerInterface;
  LLVMInlinerInterface(Dialect *dialect)
      : DialectInlinerInterface(dialect),
        // Cache set of StringAttrs for fast lookup in `isLegalToInline`.
        disallowedFunctionAttrs({
            StringAttr::get(dialect->getContext(), "noduplicate"),
            StringAttr::get(dialect->getContext(), "noinline"),
            StringAttr::get(dialect->getContext(), "optnone"),
            StringAttr::get(dialect->getContext(), "presplitcoroutine"),
            StringAttr::get(dialect->getContext(), "returns_twice"),
            StringAttr::get(dialect->getContext(), "strictfp"),
        }) {}
  bool isLegalToInline(Operation *call, Operation *callable,
                       bool wouldBeCloned) const final {
    auto callOp = dyn_cast<LLVM::CallOp>(call);
    if (!callOp) {
      LLVM_DEBUG(llvm::dbgs()
                 << "Cannot inline: call is not an LLVM::CallOp\n");
      return false;
    }
```

CallOp Interface

CallInterfaces.td

```
def CallOpInterface : OpInterface<"CallOpInterface"> {
  let description = [{ ...transfers control from one sub-routine to.. }];
  …
  let methods = [
    InterfaceMethod<[{Returns the callee of this call-like operation. …either a
        reference to a symbol, or a reference to a defined SSA value. }],
      "::mlir::CallInterfaceCallable", "getCallableForCallee"
    >,
    InterfaceMethod<[{… get the operands …to the callee. }],
      "::mlir::Operation::operand_range", "getArgOperands"
    >,
  ];
  let extraClassDeclaration = [{/// Resolve the callable operation for given callee
      Operation *resolveCallable(SymbolTableCollection *symbolTable = nullptr);
  }];
}
```

CallableOp Interface

CallInterfaces.td

```
def CallableOpInterface : OpInterface<"CallableOpInterface"> {
  let description = [{ … represents a potential sub-routine, and may
    be a target for those providing the CallOpInterface  }];
  …
  let methods = [
    InterfaceMethod<[{
        Returns the region on the current operation that is callable. This may
        return null in the case of an external callable object, e.g. an external
        function.
      }],
      "::mlir::Region *", "getCallableRegion"
    >,
    InterfaceMethod<[{
        The results types that the callable region produces when executed.
      }],
      "::llvm::ArrayRef<::mlir::Type>", "getCallableResults"
    >,
  ];
}
```

Using call interfaces

FuncOps.td

```
def CallOp : Func_Op<"call",
    [CallOpInterface, MemRefsNormalizable,
     DeclareOpInterfaceMethods<SymbolUserOpInterface>]> {
  let summary = "call operation";
  let description = [{... a direct call to a function ...
    Example:

    ```mlir
    %2 = func.call @my_add(%0, %1) : (f32, f32) -> f32
    ```
  }];

  let arguments = (ins FlatSymbolRefAttr:$callee, Variadic<AnyType>:$operands);
  let results = (outs Variadic<AnyType>);
  ...
}
```

Inline Legality

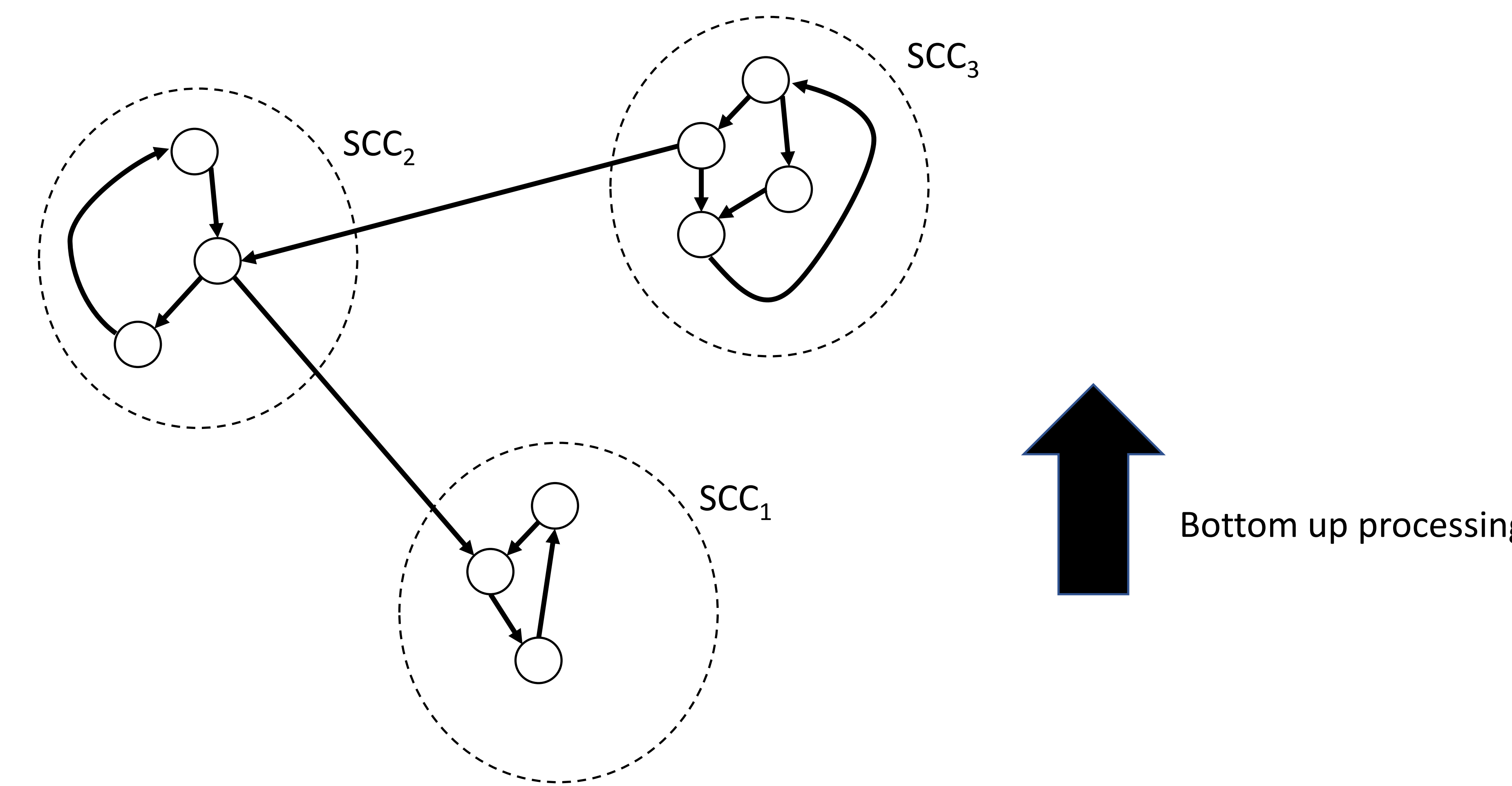
InliningUtils.h

```
/// Utility to check that all of the operations within 'src' can be inlined.
static bool isLegalToInline(InlinerInterface &interface, Region *src,
                            Region *insertRegion, bool shouldCloneInlinedRegion,
                            IRMapping &valueMapping) {
  for (auto &block : *src) {
    for (auto &op : block) {
      // Check this operation.
      if (!interface.isLegalToInline(&op, insertRegion,
                                     shouldCloneInlinedRegion, valueMapping)) {
        ...
        });
        return false;
      }
      // Check any nested regions.
      ...
}
```

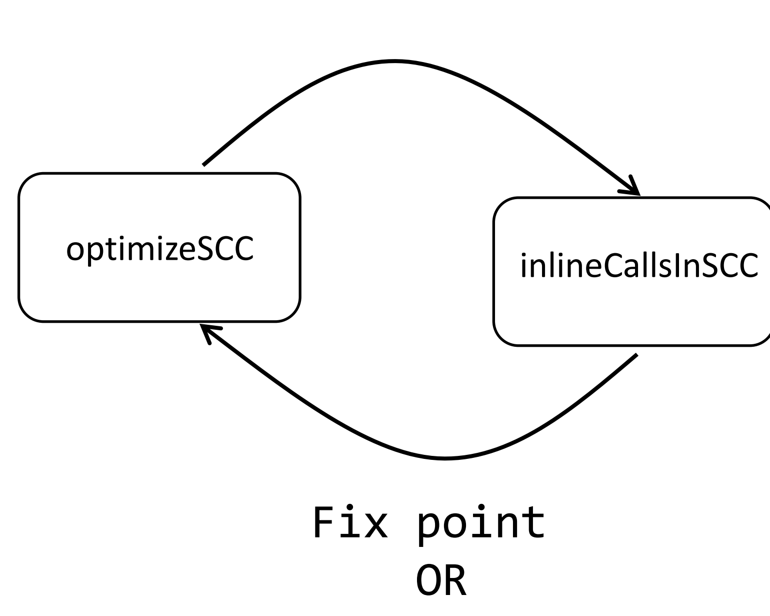
How to Inline ? MLIR Interfaces

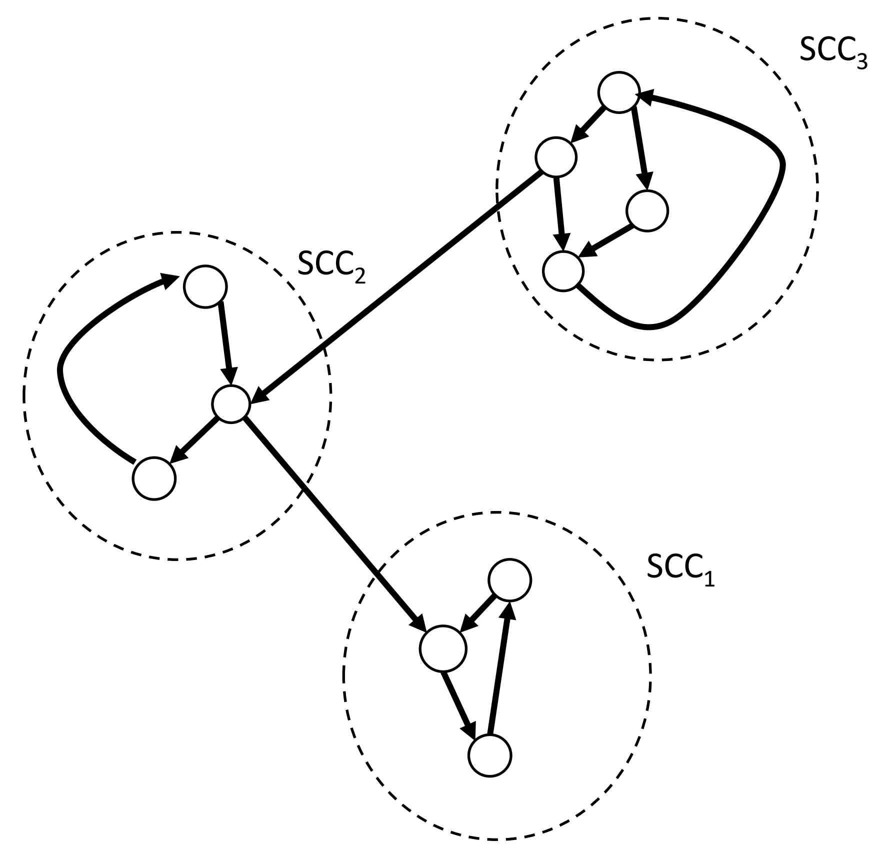
- Dialect Inliner Interface
- CallOp Interface
- CallableOp Interface

Inliner Implementation



Inliner in MLIR





Inline History

```
func.func @A(%x : i32) -> i32 {
  %res = func.call @B(%x) : (i32) -> i32
  return %res : i32
}

func.func @B(%x : i32) -> i32 {
  %res = func.call @C(%x) : (i32) -> i32
  return %res : i32
}

func.func @C(%x : i32) -> i32 {
  %cst_1 = arith.constant 1 : i32
  %y = arith.addi %x, %cst_1 : i32
  %res = func.call @A(%y) : (i32) -> i32
  return %res : i32
}
```

```
func @A( ) {
 C2: call @B
}
```

```
func @B( ) {
 C1: call @C
}
```

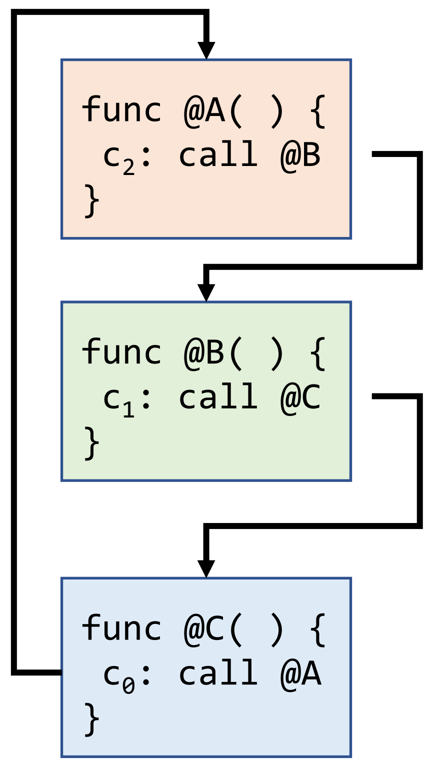
```
func @C( ) {
 C0: call @A
}
```

```
func.func @A(%x : i32) -> i32 {
  %res = func.call @B(%x) : (i32) -> i32
  return %res : i32
}

func.func @B(%x : i32) -> i32 {
  %res = func.call @C(%x) : (i32) -> i32
  return %res : i32
}

func.func @C(%x : i32) -> i32 {
  %cst_1 = arith.constant 1 : i32
  %y = arith.addi %x, %cst_1 : i32
  %res = func.call @A(%y) : (i32) -> i32
  return %res : i32
}
```

Inline History



Calls

c_0. call @A
c_1. call @C
c_2. call @B

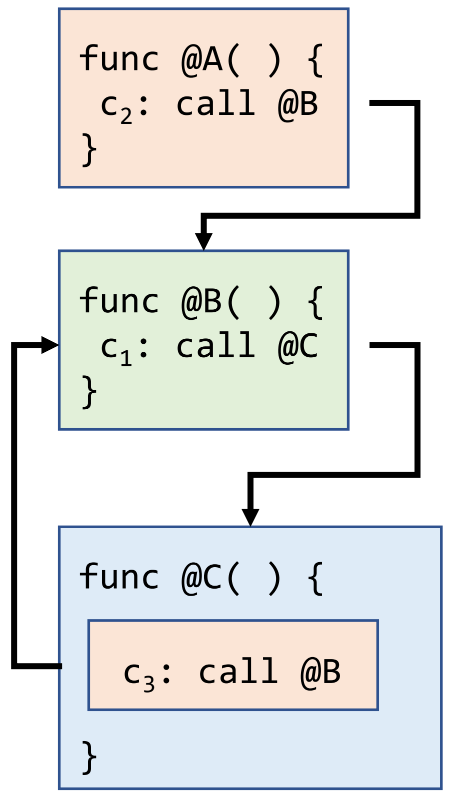
Call-History

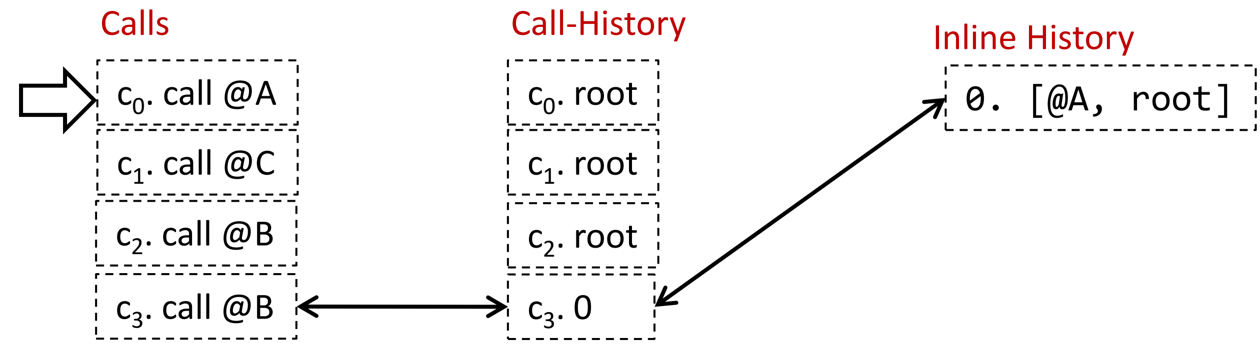
c_0. root
c_1. root
c_2. root

Inline History

```
* Inliner: Initial calls in SCC are: {
  0. %1 = func.call @A(%0) : (i32) -> i32,
  1. %0 = func.call @C(%arg0) : (i32) -> i32,
  2. %0 = func.call @B(%arg0) : (i32) -> i32,
}
```

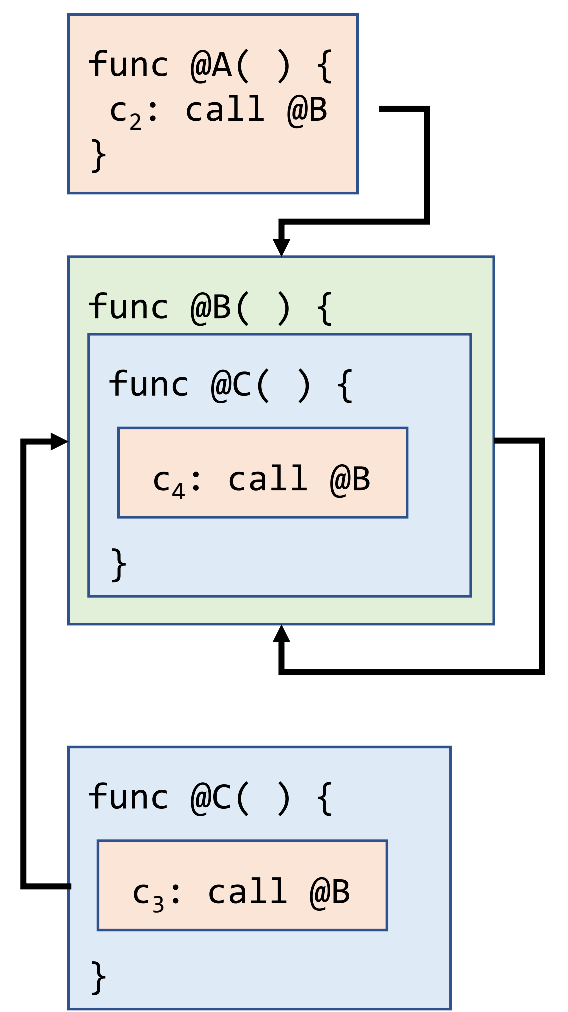
Inline History



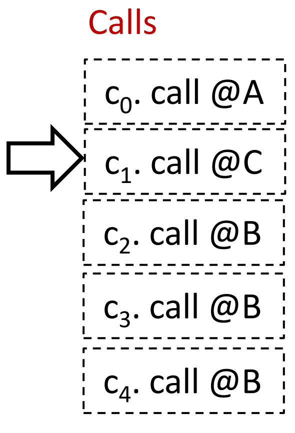



```
* Inlining call: 0. %1 = func.call @A(%0) : (i32) -> i32
* new inlineHistory entry: 0. [%1 = func.call @A(%0) : (i32) -> i32, root]
* new call 3 {%2 = func.call @B(%0) : (i32) -> i32}
   with historyID = 0, added due to inlining of
  call {%1 = func.call @A(%0) : (i32) -> i32}
 with historyID = root
```

Inline History



Calls



Call-History

- c_0. root
- c_1. root
- c_2. root
- c_3. 0
- c_4. 1

Inline History

- 0. [@A, root]
- 1. [@C, root]

```
* Inlining call: 1. %0 = func.call @C(%arg0) : (i32) -> i32
* new inlineHistory entry: 1. [%0 = func.call @C(%arg0) : (i32) -> i32, root]
* new call 4 {%2 = func.call @B(%1) : (i32) -> i32}
   with historyID = 1, added due to inlining of
  call {%0 = func.call @C(%arg0) : (i32) -> i32}
 with historyID = root
```

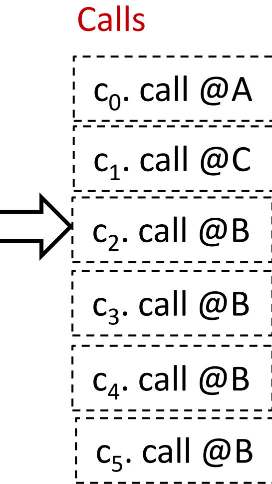
Inline History

```
func @A( ) {
  func @B( ) {
    func @C( ) {
      c5: call @B
    }
  }
}
```

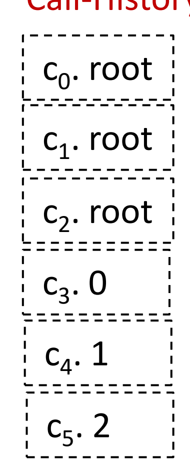
```
func @B( ) {
  func @C( ) {
    c4: call @B
  }
}
```

```
func @C( ) {
  c3: call @B
}
```

Calls



Call-History

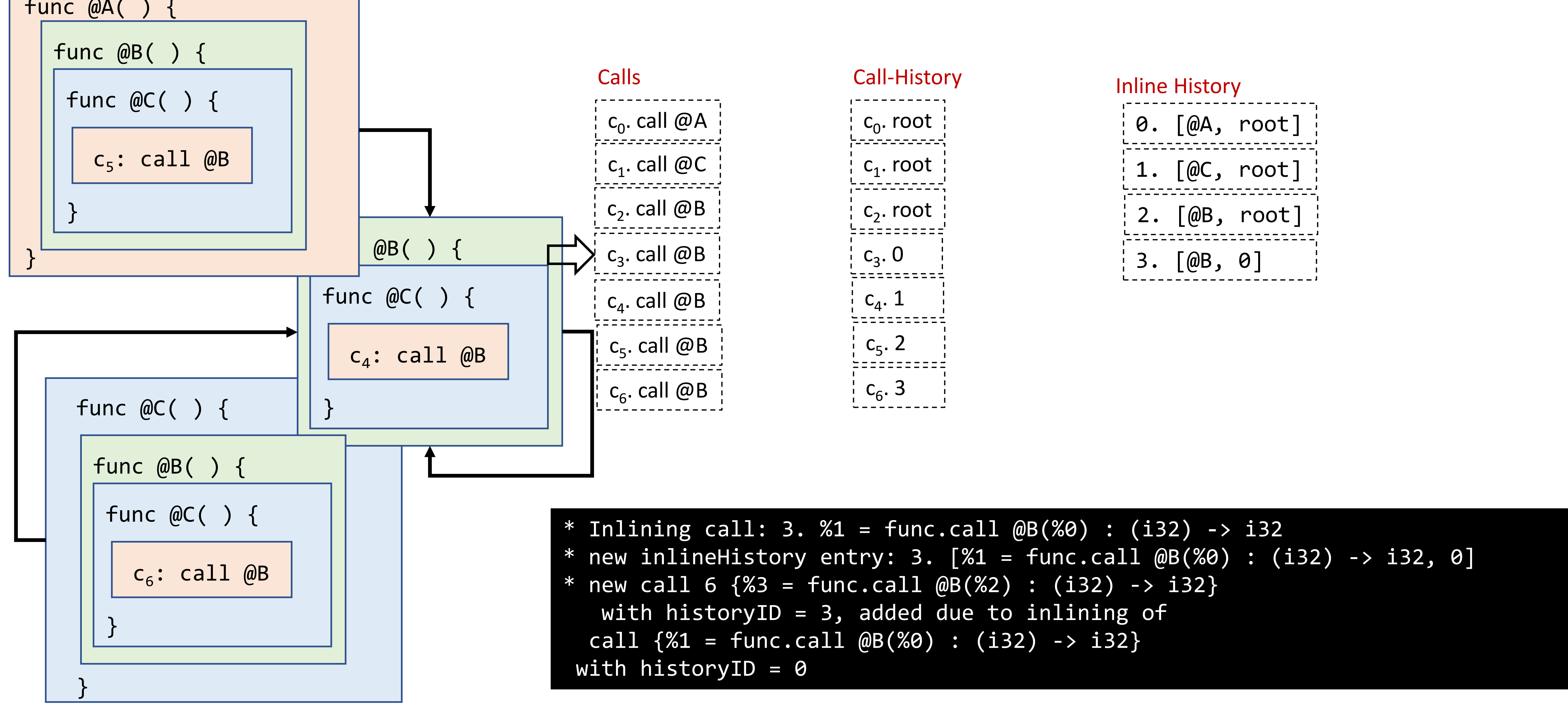


Inline History

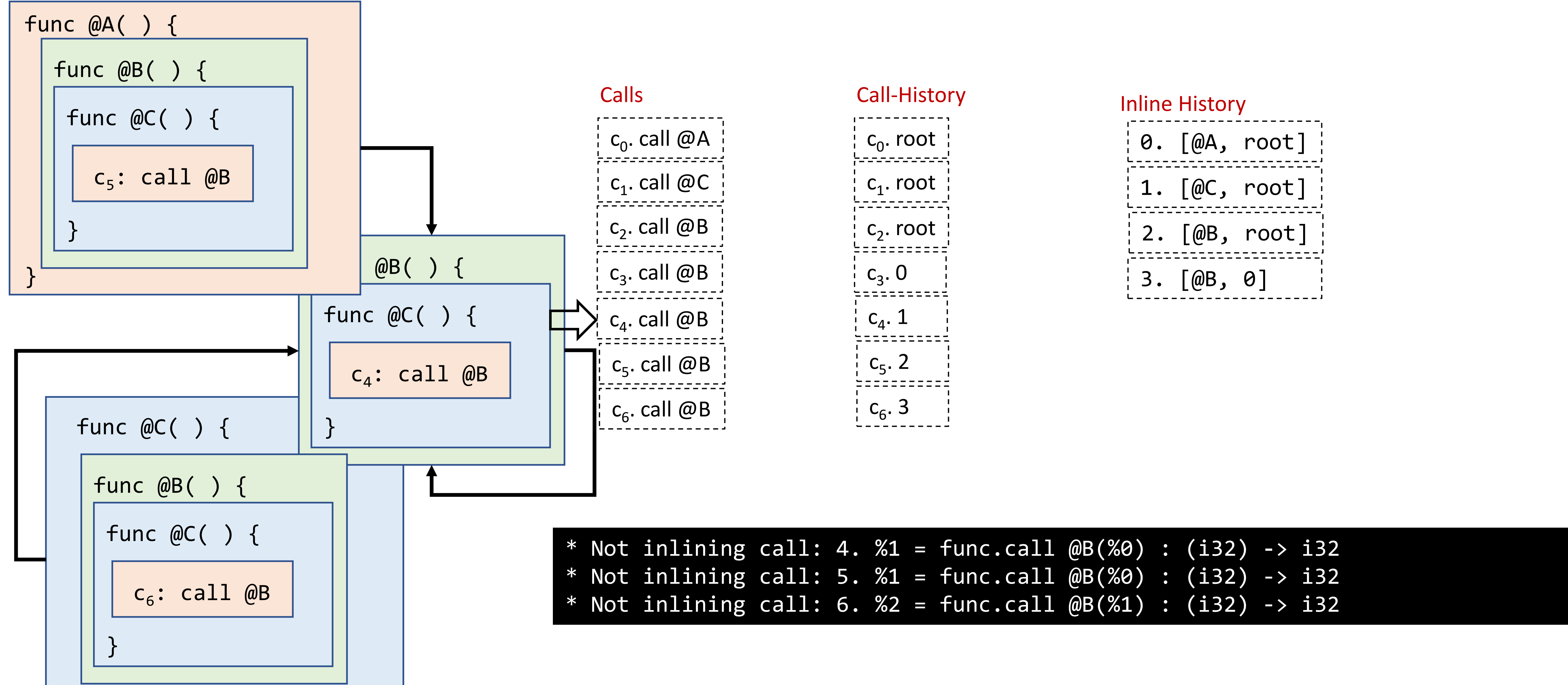
```
0. [@A, root]
1. [@C, root]
2. [@B, root]
```

```
* Inlining call: 2. %0 = func.call @B(%arg0) : (i32) -> i32
* new inlineHistory entry: 2. [%0 = func.call @B(%arg0) : (i32) -> i32, root]
* new call 5 {%2 = func.call @B(%1) : (i32) -> i32}
   with historyID = 2, added due to inlining of
  call {%0 = func.call @B(%arg0) : (i32) -> i32}
 with historyID = root
```

Inline History



Inline History



Inline History

func @A( ) {
func @B( ) {
func @C( ) {
c_5: call @B
}
}

@B( ) {
func @C( ) {
c_4: call @B
}

func @C( ) {
func @B( ) {
func @C( ) {
c_6: call @B
}
}

```
module {
  func.func @A(%arg0: i32) -> i32 {
    %c1_i32 = arith.constant 1 : i32
    %0 = arith.addi %arg0, %c1_i32 : i32
    %1 = call @B(%0) : (i32) -> i32
    return %1 : i32
  }
  func.func @B(%arg0: i32) -> i32 {
    %c1_i32 = arith.constant 1 : i32
    %0 = arith.addi %arg0, %c1_i32 : i32
    %1 = call @B(%0) : (i32) -> i32
    return %1 : i32
  }
  func.func @C(%arg0: i32) -> i32 {
    %c1_i32 = arith.constant 1 : i32
    %0 = arith.addi %arg0, %c1_i32 : i32
    %c1_i32_0 = arith.constant 1 : i32
    %1 = arith.addi %0, %c1_i32_0 : i32
    %2 = call @B(%1) : (i32) -> i32
    return %2 : i32
  }
}
```

Conclusion

- MLIR : versatile but some common transformations can be useful
- Inliner: works well across dialects
- Inlining Interface
- Details of Inliner operation

Thank you

Follow us on:

For more information, visit us at:

qualcomm.com & qualcomm.com/blog

Nothing in these materials is an offer to sell any of the components or devices referenced herein.

©2018-2023 Qualcomm Technologies, Inc. and/or its affiliated companies. All Rights Reserved.

Qualcomm is a trademark or registered trademark of Qualcomm Incorporated. Other products and brand names may be trademarks or registered trademarks of their respective owners.

References in this presentation to "Qualcomm" may mean Qualcomm Incorporated, Qualcomm Technologies, Inc., and/or other subsidiaries or business units within the Qualcomm corporate structure, as applicable. Qualcomm Incorporated includes our licensing business, QTL, and the vast majority of our patent portfolio. Qualcomm Technologies, Inc., a subsidiary of Qualcomm Incorporated, operates, along with its subsidiaries, substantially all of our engineering, research and development functions, and substantially all of our products and services businesses, including our QCT semiconductor business.

Snapdragon and Qualcomm branded products are products of Qualcomm Technologies, Inc. and/or its subsidiaries. Qualcomm patented technologies are licensed by Qualcomm Incorporated.