LLVM → Transforms → Scalar

Loop Unswitching Pass

c👩‍💻de wa🚶 k

by

Saket U. and Khyati K.

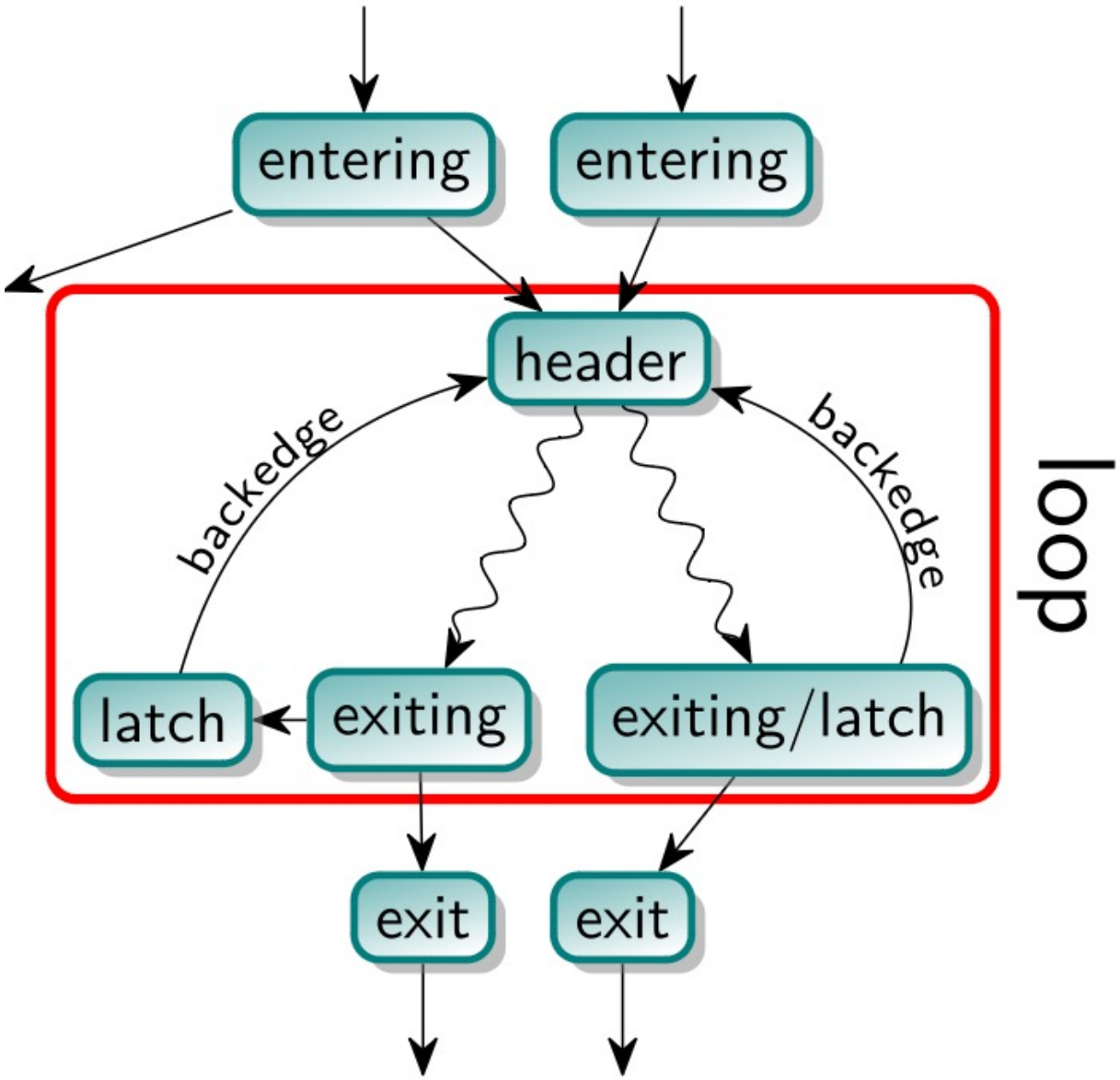
CS6620 | Sp. 2023

Base Theory → Code Walk → Example.

Loops in LLVM

LLVM Loop Terminology

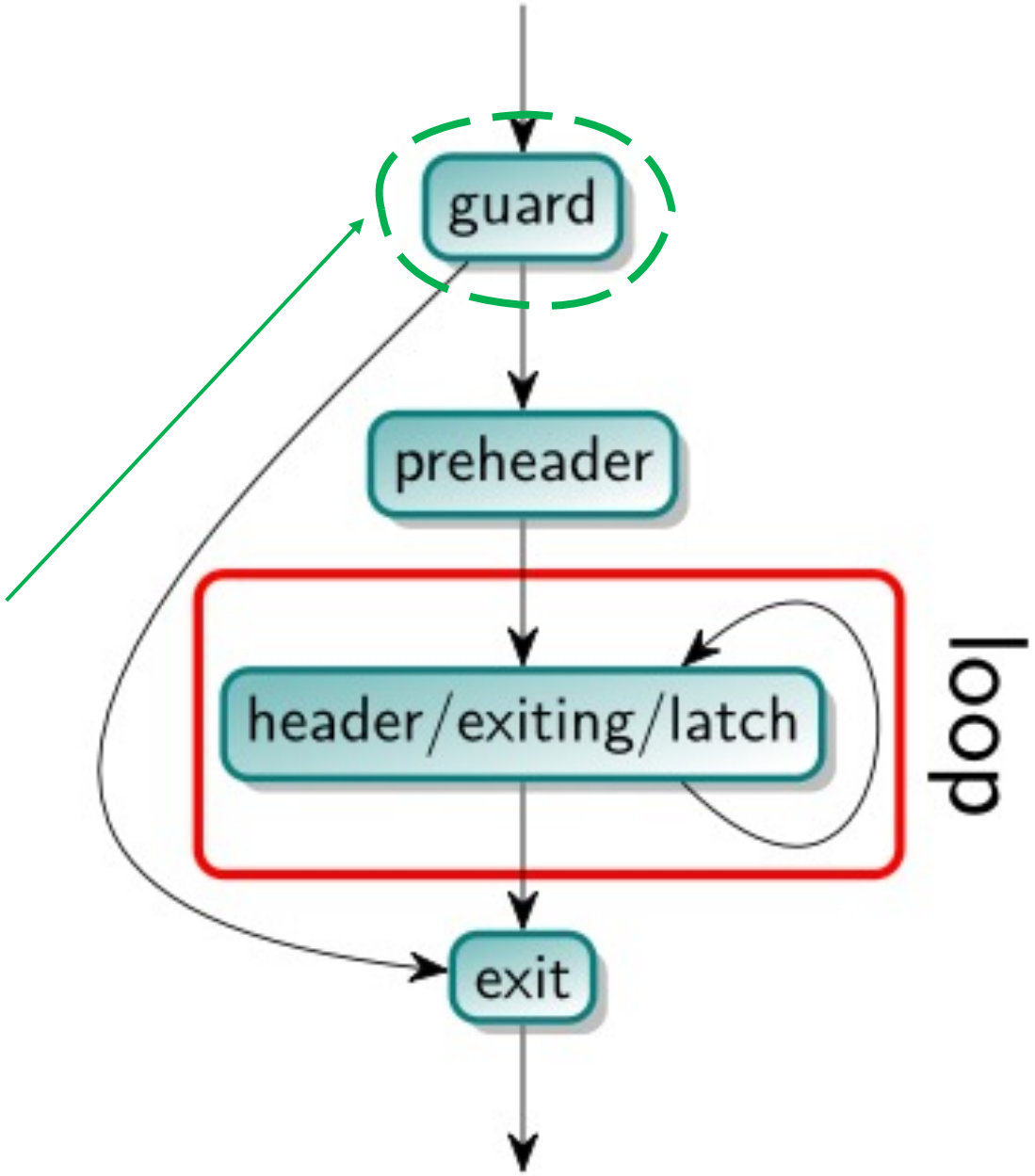
- An entering block (or loop predecessor) is a non-loop node that has an edge into the loop (necessarily the header). If there is only one entering block, and its only edge is to the header, it is also called the loop's preheader. The preheader dominates the loop without itself being part of the loop.
- A latch is a loop node that has an edge to the header.
- A backedge is an edge from a latch to the header.
- An exiting edge is an edge from inside the loop to a node outside of the loop. The source of such an edge is called an exiting block, its target is an exit block.



The Loop Gaurd

- The number of executions of the loop header before leaving the loop is the loop trip count (or iteration count).

- If the loop is not executed, a loop guard must skip the entire loop.



Understanding Unswitching

What is “Loop Unswitching”?

An elementary example

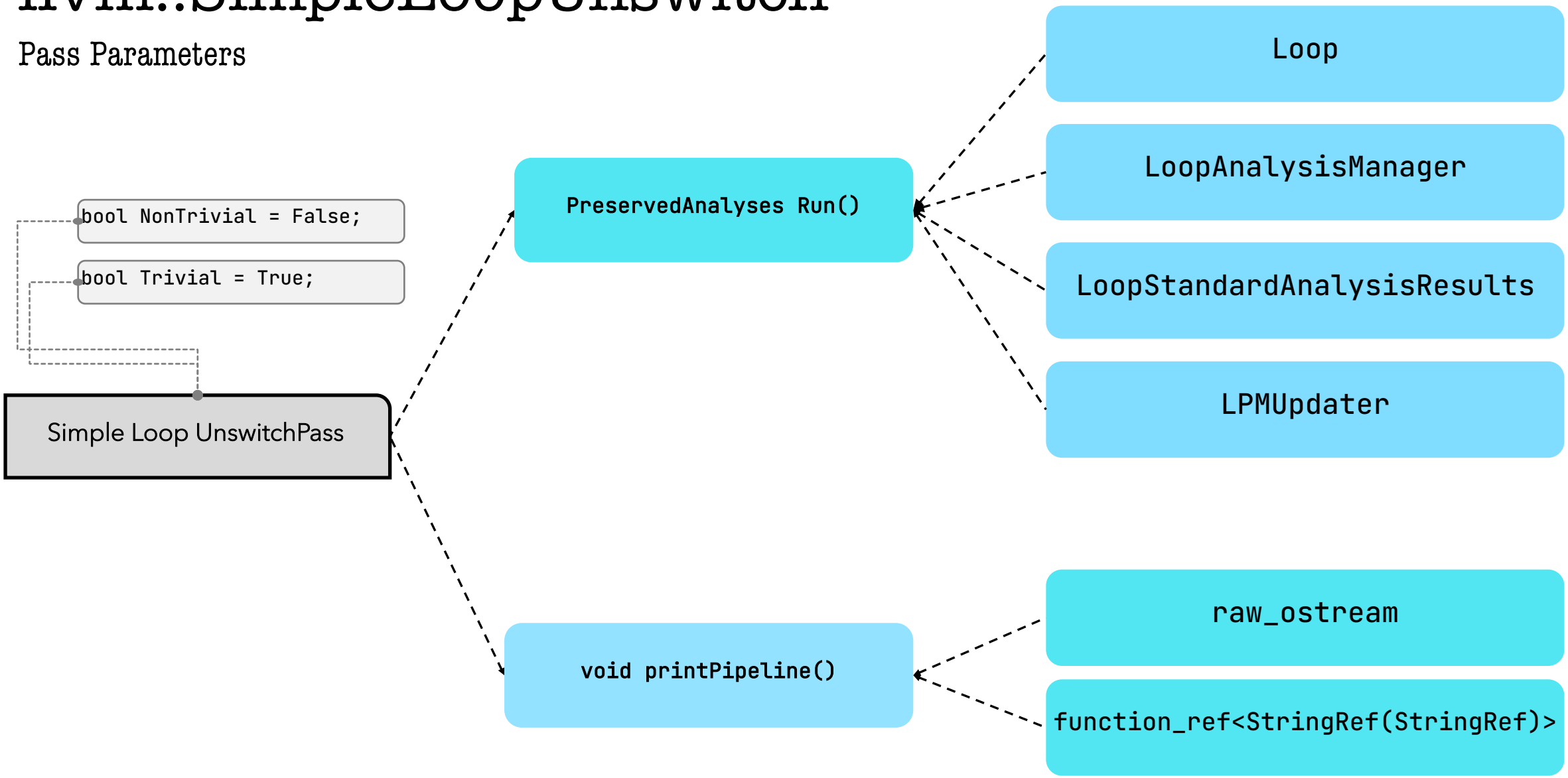
It simplifies the loop by taking the *invariant loop conditionals* outside the loop, preventing them from being computed at each iteration.

```
bool B = false;
int X = 0, Y = 0, Z = 0;
for (int itr = 0; itr < 10000; itr++) {
  if (B) {
    X += 1;
  } else {
    Y += 1;
    Z += 1;
  }
}
```

```
bool B = false;
int X = 0, Y = 0, Z = 0;
if (B) {
  for (int itr = 0; itr < 10000; ++itr) {
    X += 1;
  }
}
else {
  for (int itr = 0; itr < 10000; ++itr) {
    Y += 1;
    Z += 1;
  }
}
```

llvm::SimpleLoopUnswitch

Pass Parameters



But why it’s called 'Simple' Unswitching? ಠ_ಠ

Why 'Simple' Unswitching?

Trivial v/s Non-trivial Unswitching

- A trivial unswitching is when the condition can be unswitched without cloning any code from inside the loop. This is possible when all the code is encapsulated in the conditionals.
- A non-trivial unswitch requires code duplication, this happens when some code is outside the condition body.

```
for (int i = 0; i < 10; i++) {
    // Non-Trivial
  if (flag) {
    // do IF something TRIVIAL
  } else {
    // do something ELSE TRIVIAL
  }
}
```

Trivial

```
for (int i = 0; i < 10; i++) {
  if (flag) {
    // do something
  } else {
    // do something else
  }
}
```

```
if (flag) {
  for (int i = 0; i < 10; i++) {
    // do something
  }
} else {
  for (int i = 0; i < 10; i++) {
    // do something else
  }
}
```

Non-Trivial

```
for (int i = 0; i < 10; i++) {
  // do something anyways
  if (flag) {
    // do something IF
  } else {
    // do something ELSE
  }
}
```

```
if (flag) {
  for (int i = 0; i < 10; i++) {
    // do something anyways
    // do something IF
  }
} else {
  for (int i = 0; i < 10; i++) {
    // do something anyways
    // do something ELSE
  }
}
```

Why 'Simple' Unswitching?

Full v/s Partial Unswitching

- Full unswitching is when the branch or switch is completely moved from inside the loop to outside the loop.

- Partial unswitching removes the branch from the clone of the loop but must leave a (somewhat simplified) branch in the original loop.

- While theoretically partial unswitching can be done for switches, the requirements are extreme - *we need the loop invariant input to the switch to be sufficient to collapse to a single successor in each clone.*

Example of Partial Unswitch

```
for (int i = 0; i < n; i++) {
  if (a[i] < threshold && Flag) {
    // do something
  } }
```

```
if (Flag) {
  for (int i = 0; i < n; i++) {
    if (a[i] < threshold) {
      // do something
    }}
} else {
  for (int i = 0; i < n; i++) {
    // do nothing
  }
}
```

Complexity matrix?

	FULL	**PARTIAL**
TRIVIAL	All conditions outside the loop, no duplication of code. 😇	Only some conditions outside the loop, but no duplication of code. 😕
NON-TRIVIAL	All conditions are outside the loop, but duplication of code. 😅	Only some conditions outside the loop, also code duplication. 😣

Why 'Simple' Unswitching?

What does it take to make it complex?

This pass always does -

Trivial, Full unswitching. → Branches & Switches.

Trivial, Partial unswitching. → Switches.

If enabled (via the constructor's `NonTrivial` parameter) -

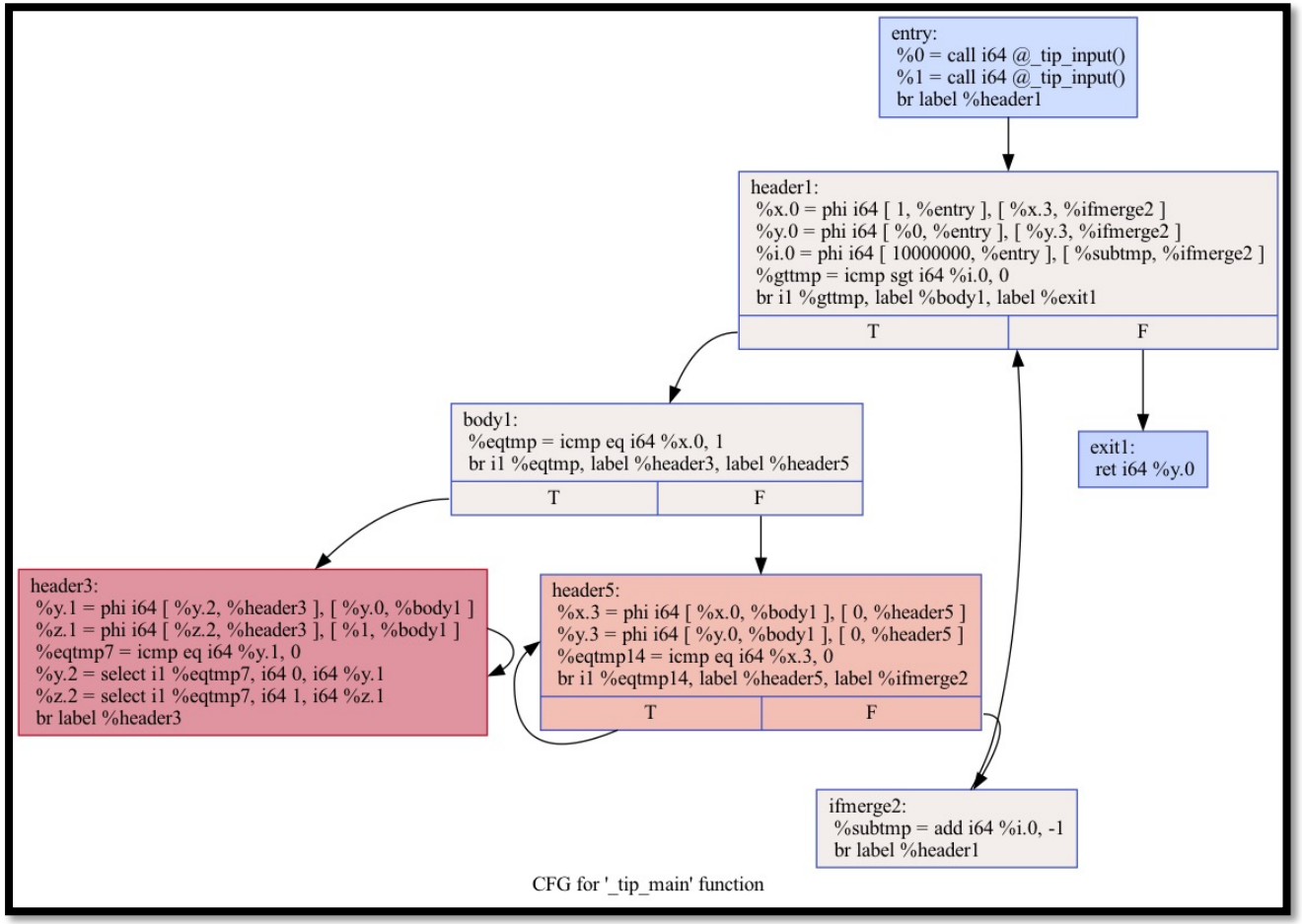
Non-trivial, Full unswitching. → Branches & Switches.

Non-trivial, Partial unswitching. → Branches.

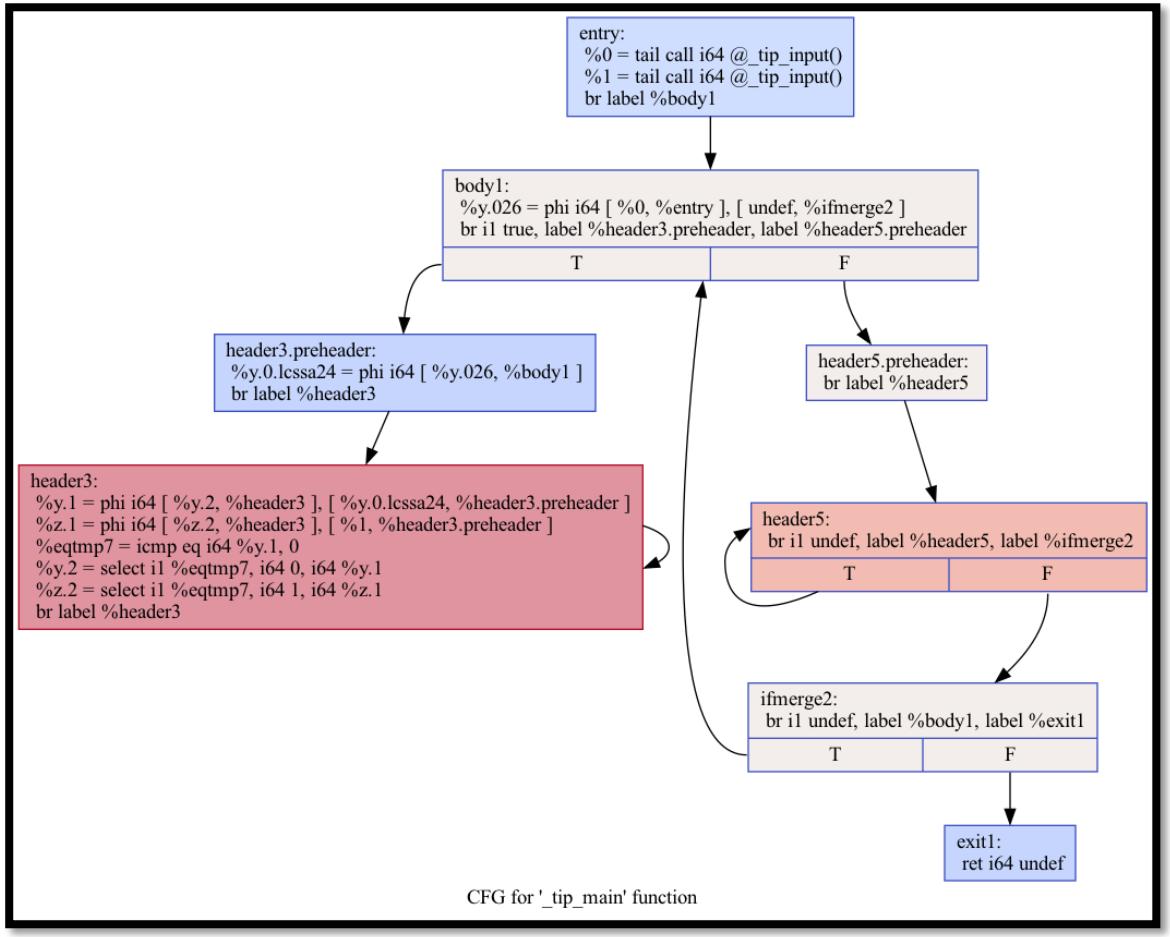
Because partial unswitching of switches is extremely unlikely to be possible in practice & significantly complicates the implementation, this pass does not currently implement that in any mode.

Example in TIPC

(Taken from assignment)



Base Optimizations

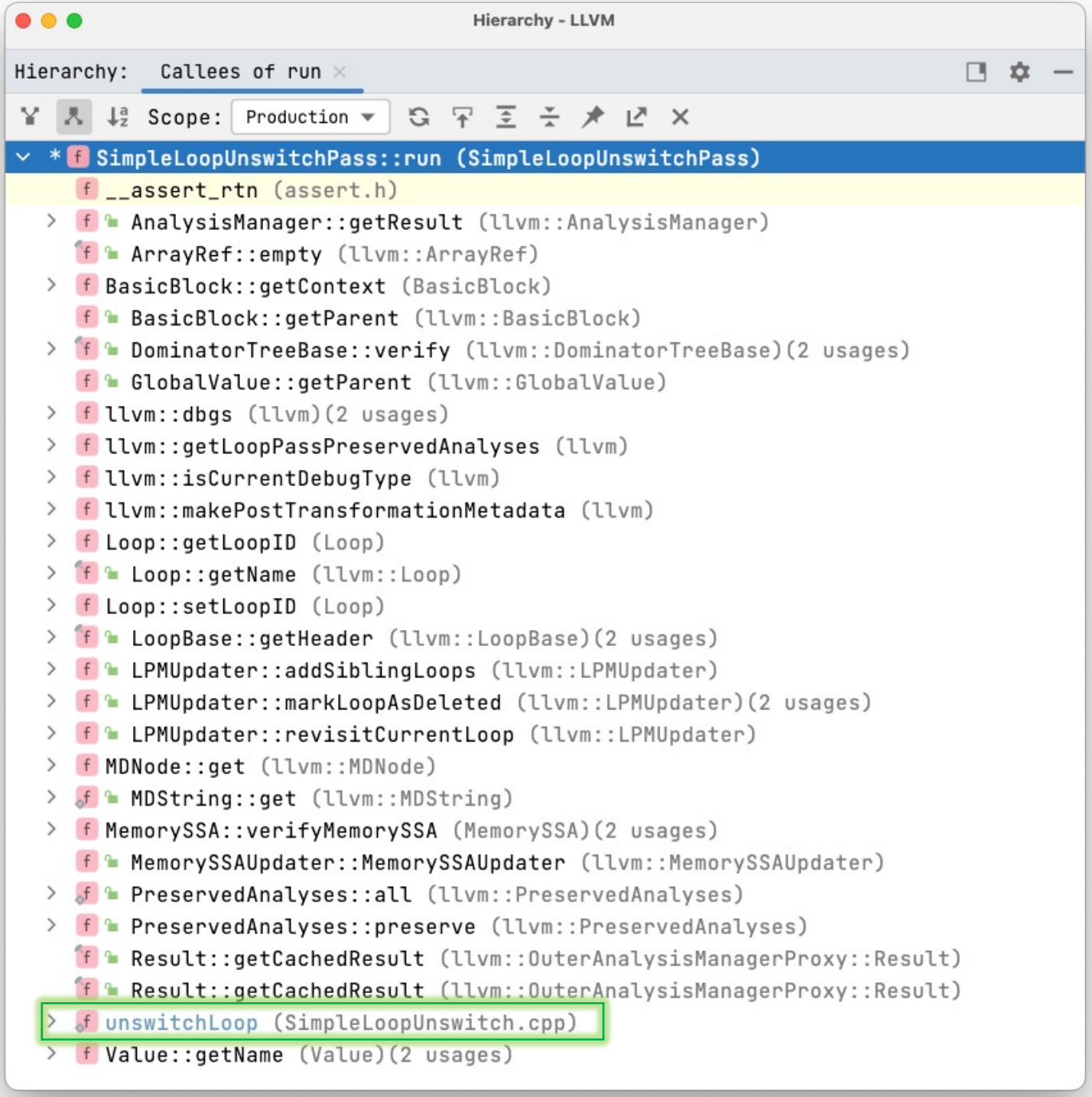


+5 Extra Optimizations

Exploring the LLVM Code

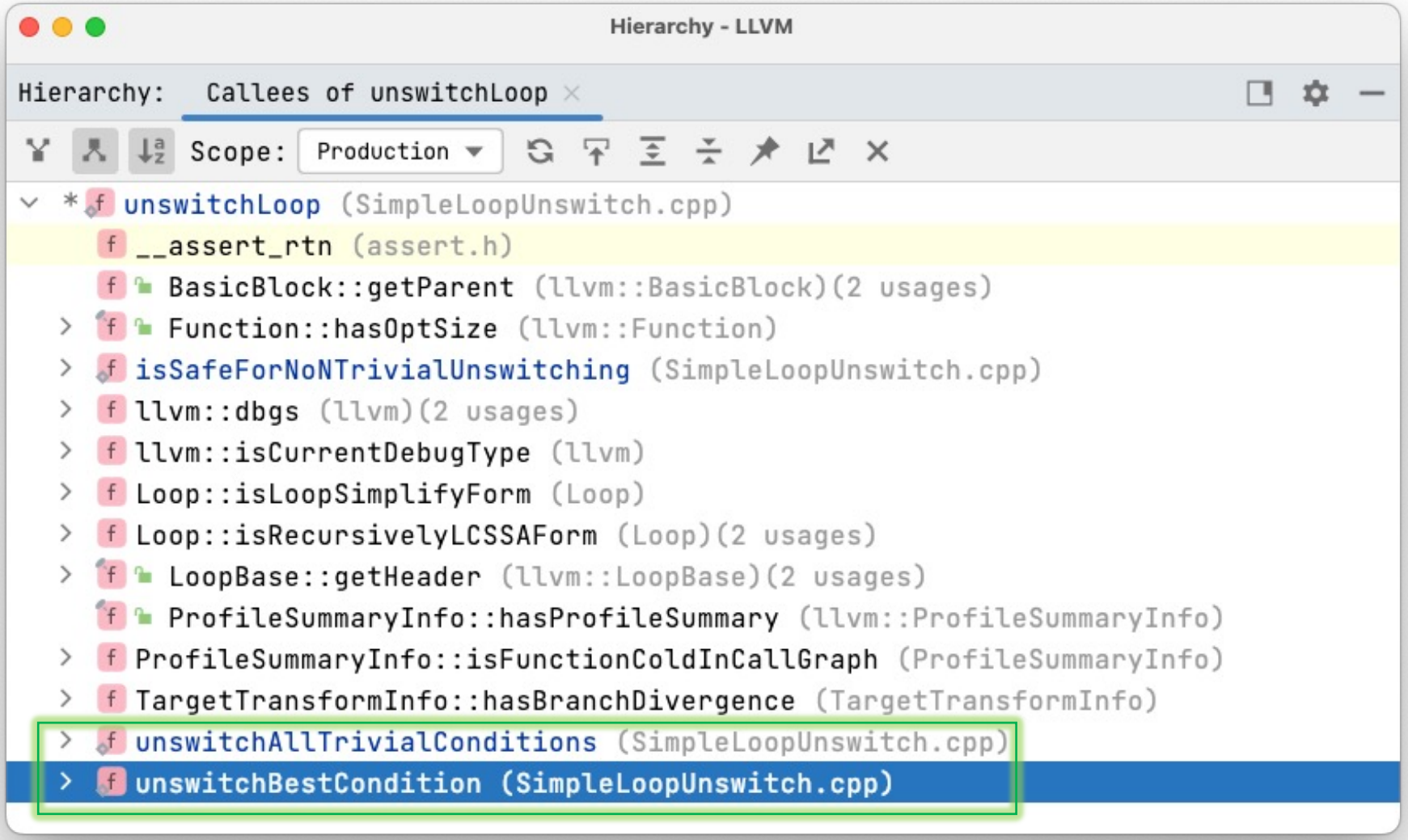
run()

|Callee Hierarchy|



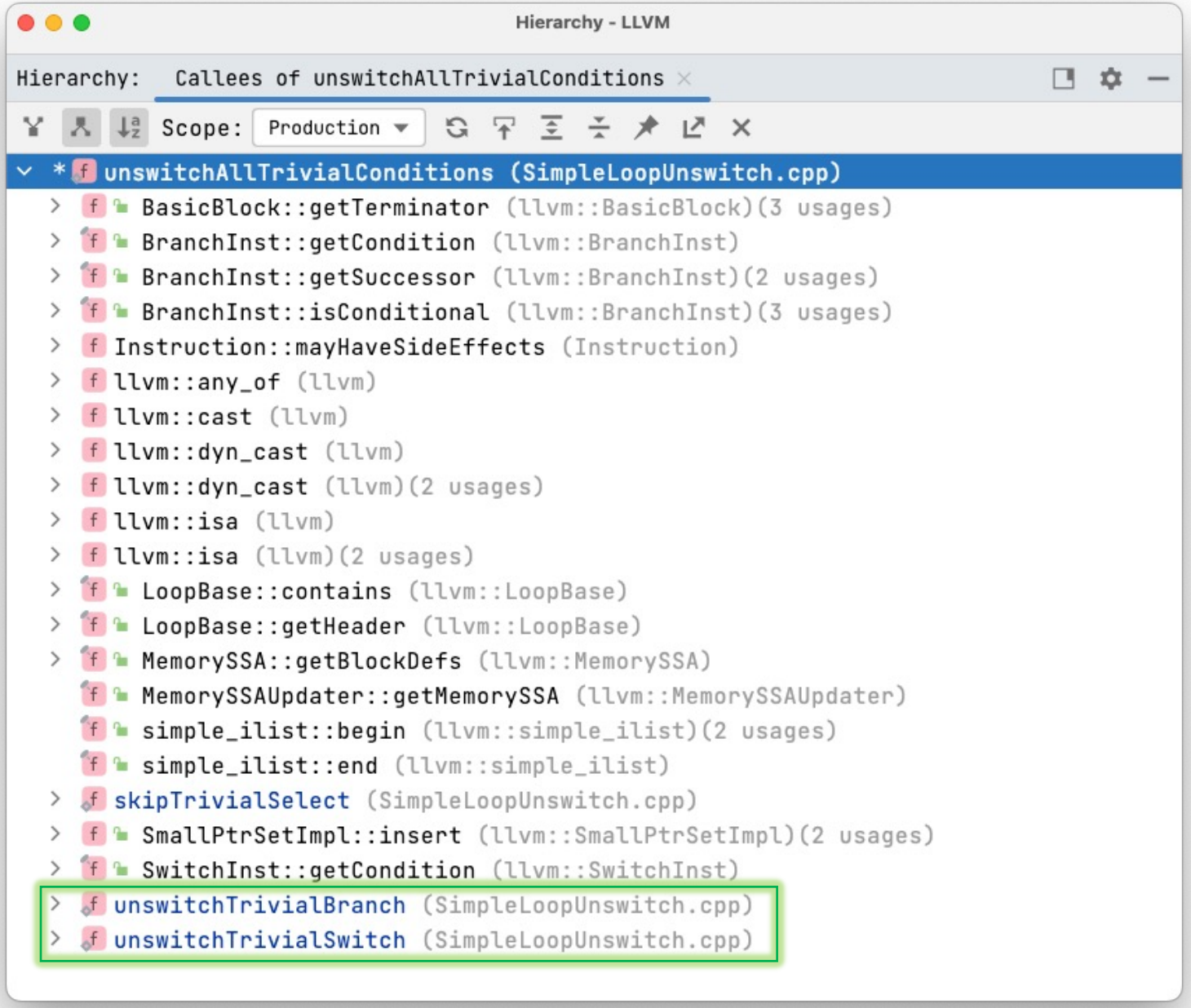
unswitchLoop()

|Callee Hierarchy|



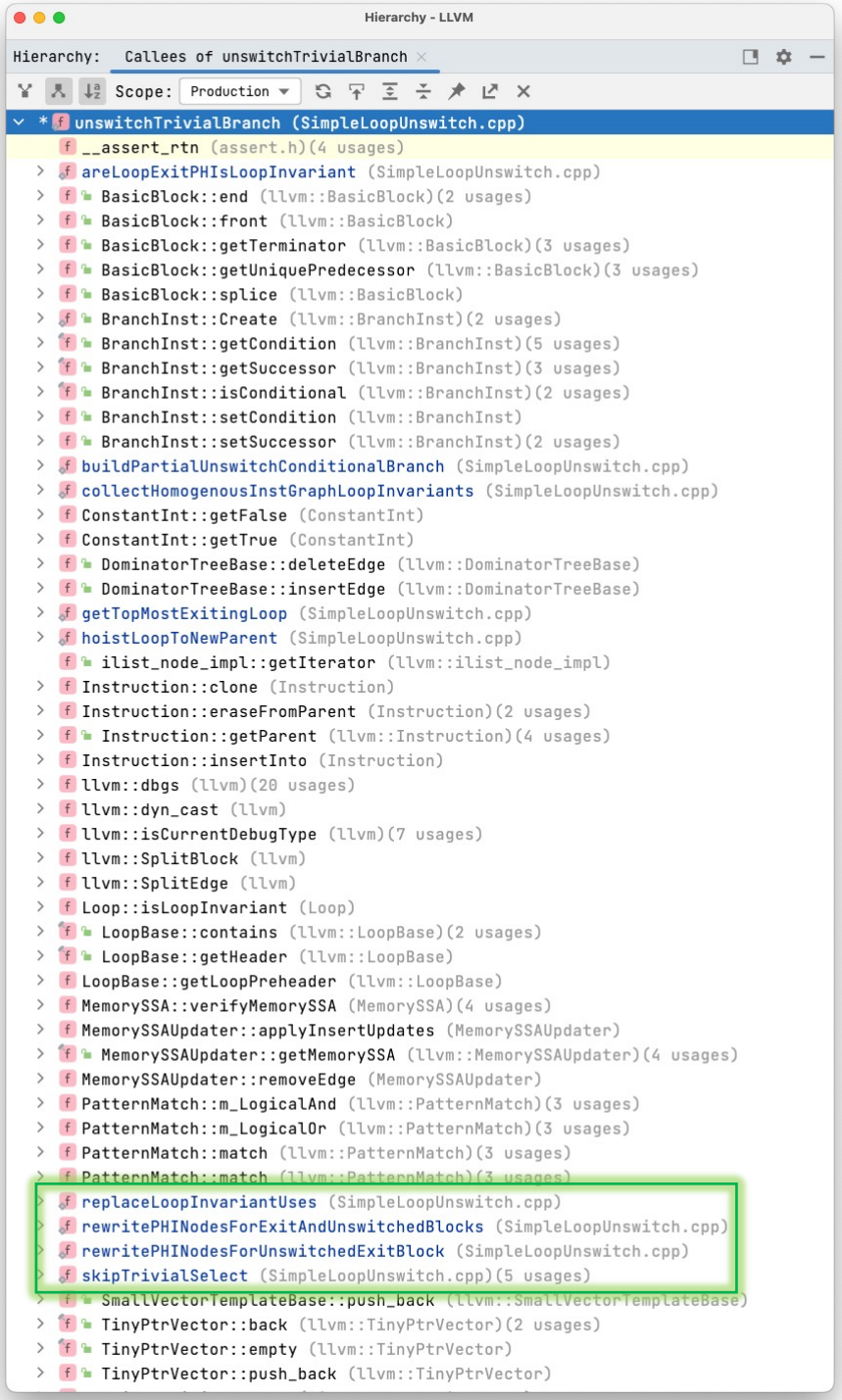
unswitchAllTrivialConditions()

|Callee Hierarchy|



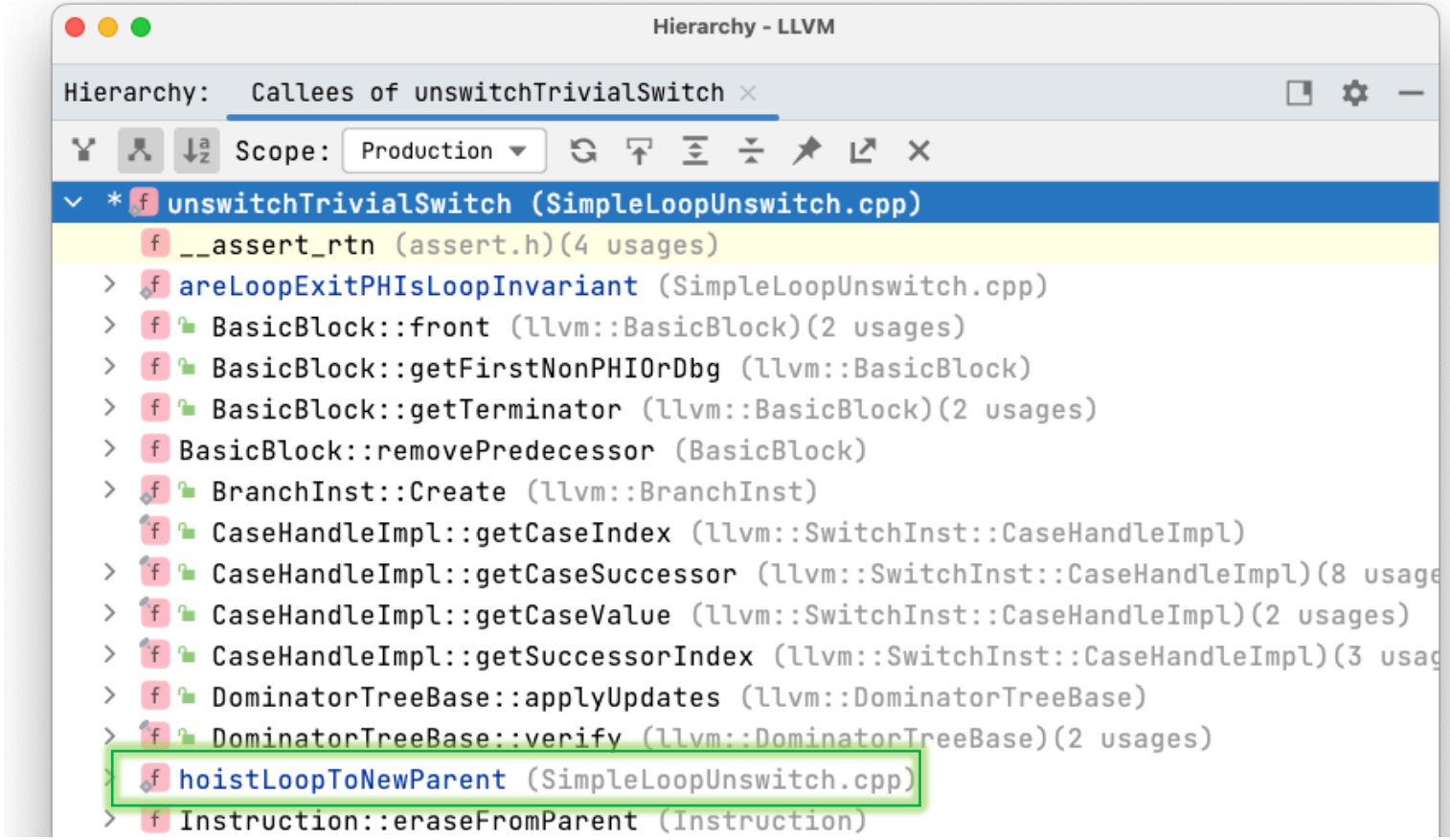
unswitchTrivialBranch()

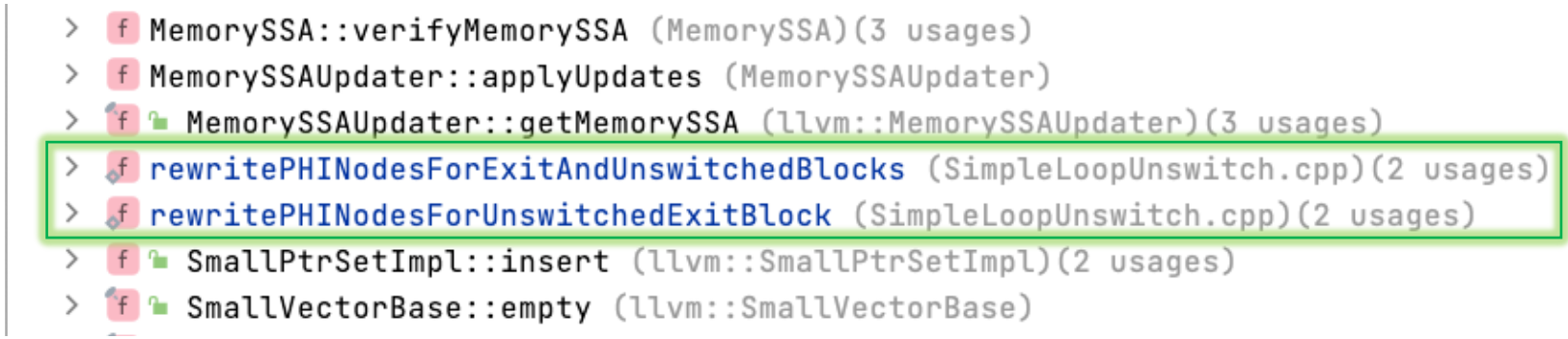
|Callee Hierarchy|



unswitchTrivialSwitch()

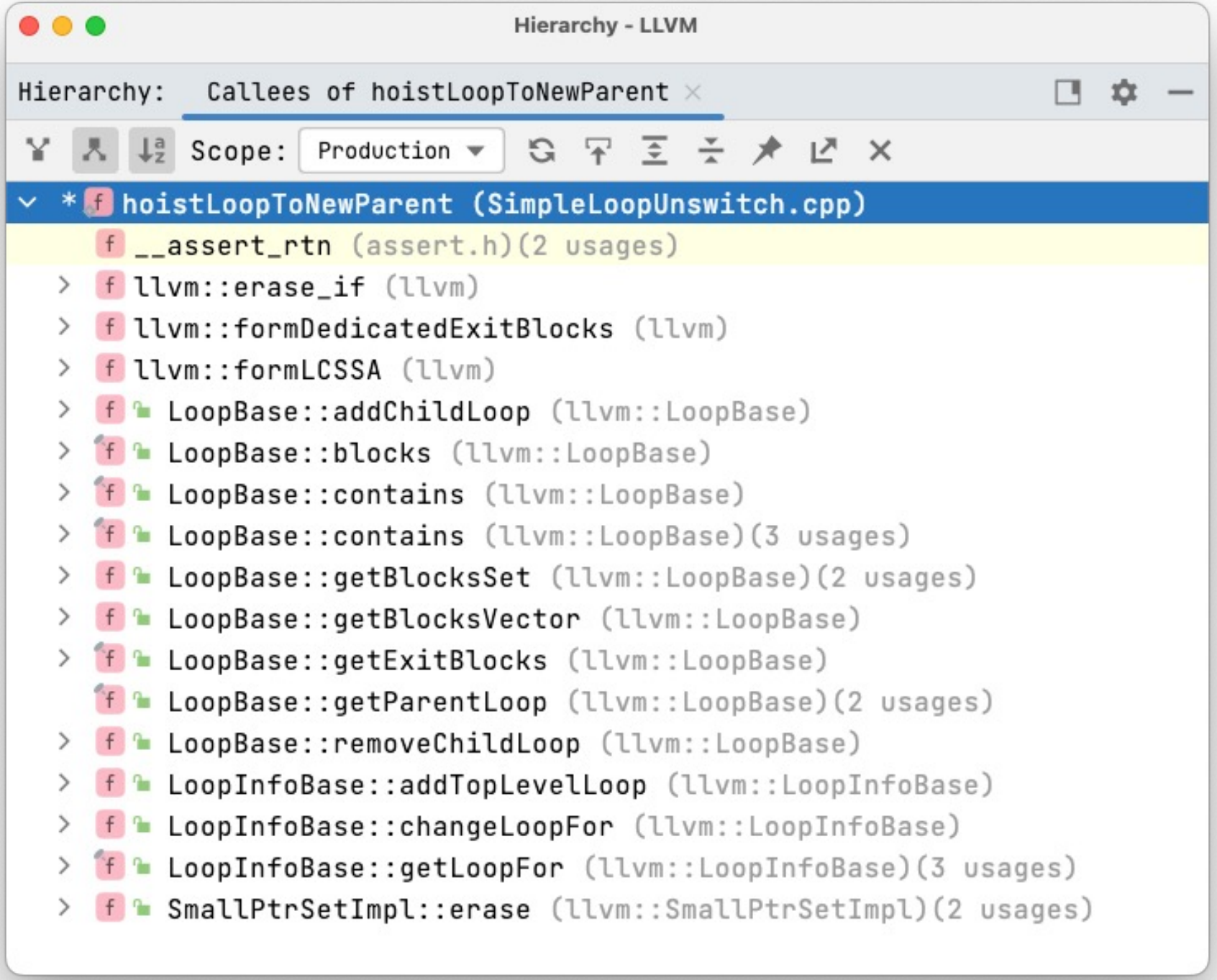
|Callee Hierarchy|





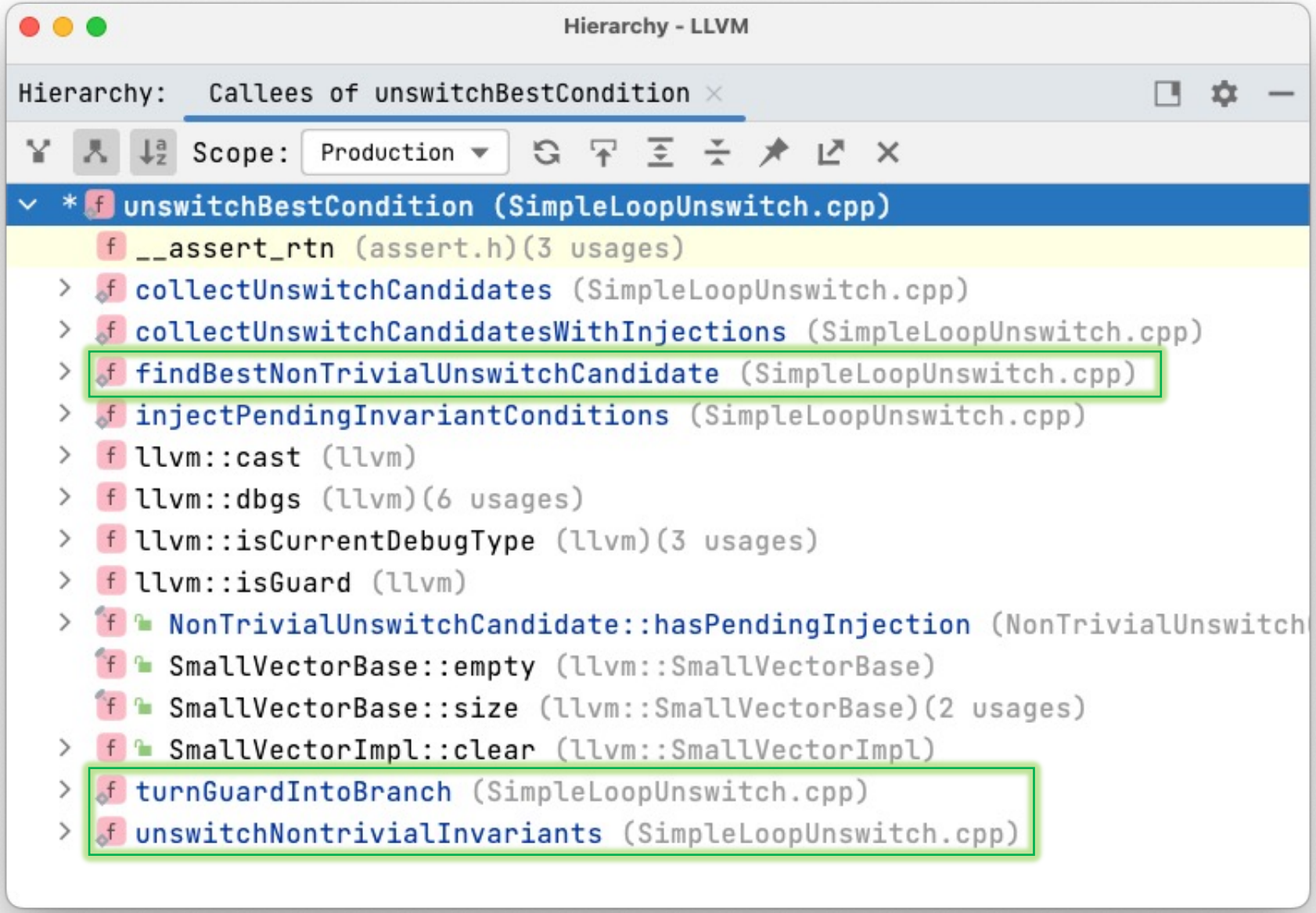
hoistLoopToNewParent()

|Callee Hierarchy|



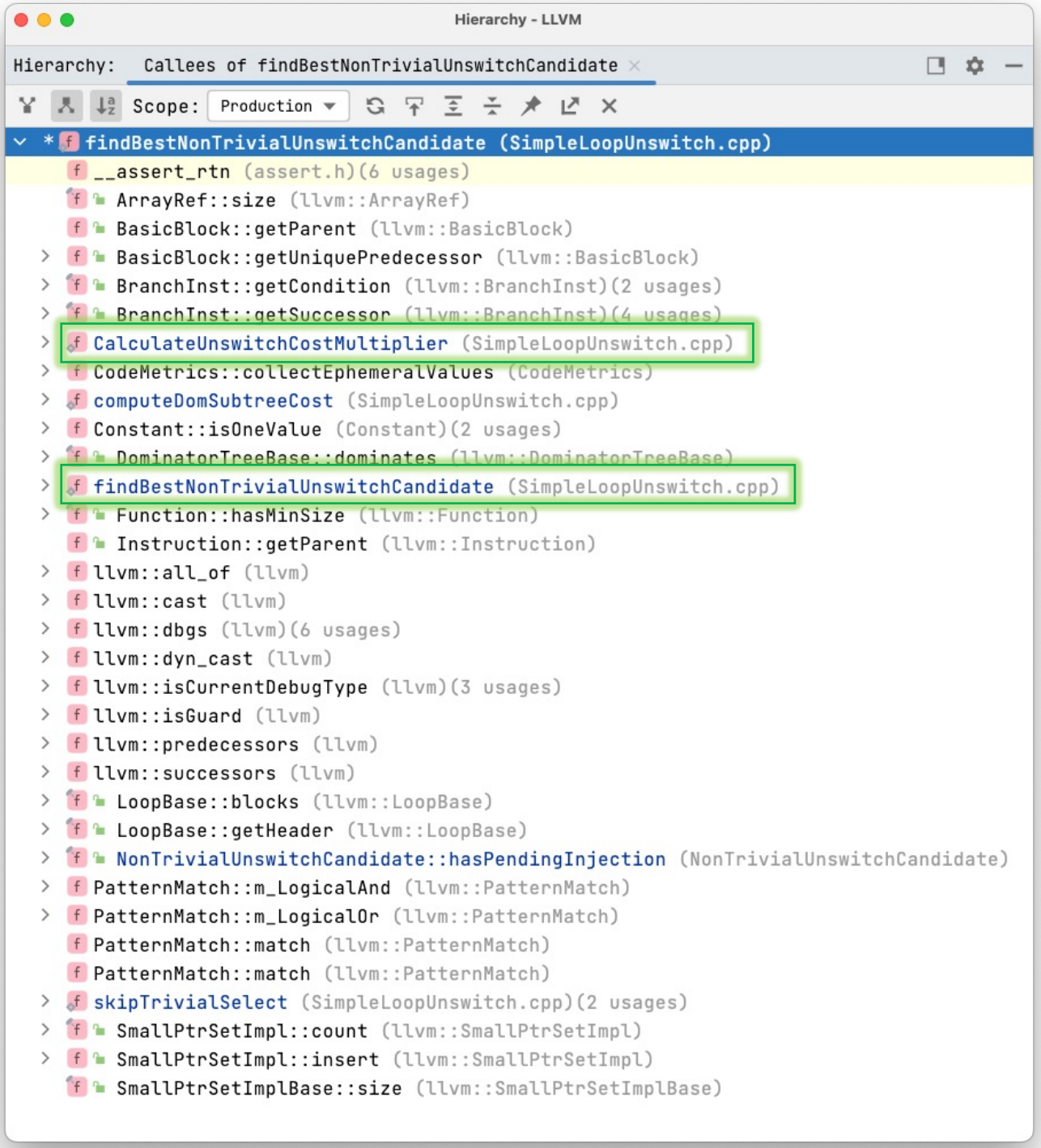
unswitchBestCondition()

|Callee Hierarchy|



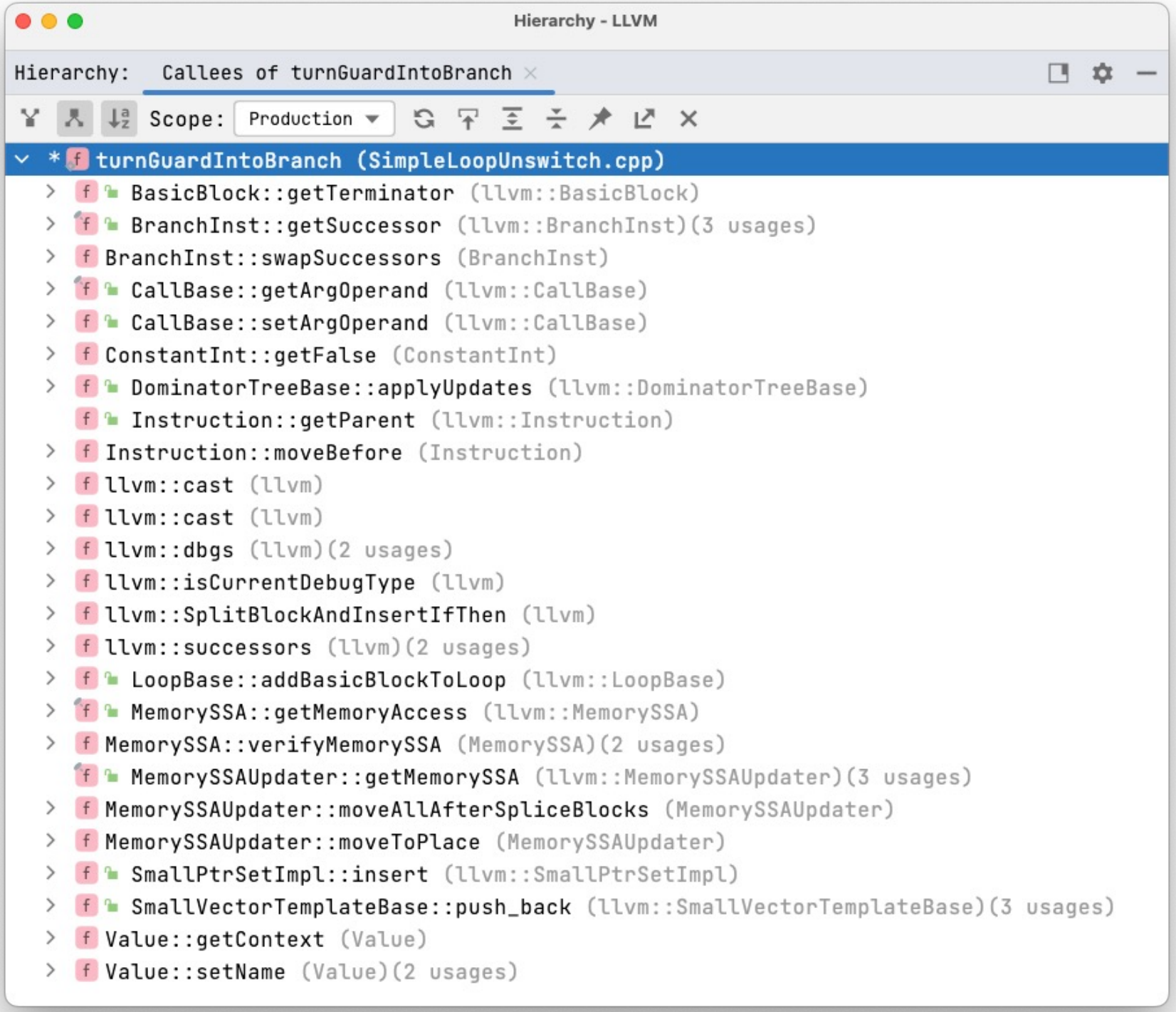
findBestNonTrivialCandidates()

|Callee Hierarchy|



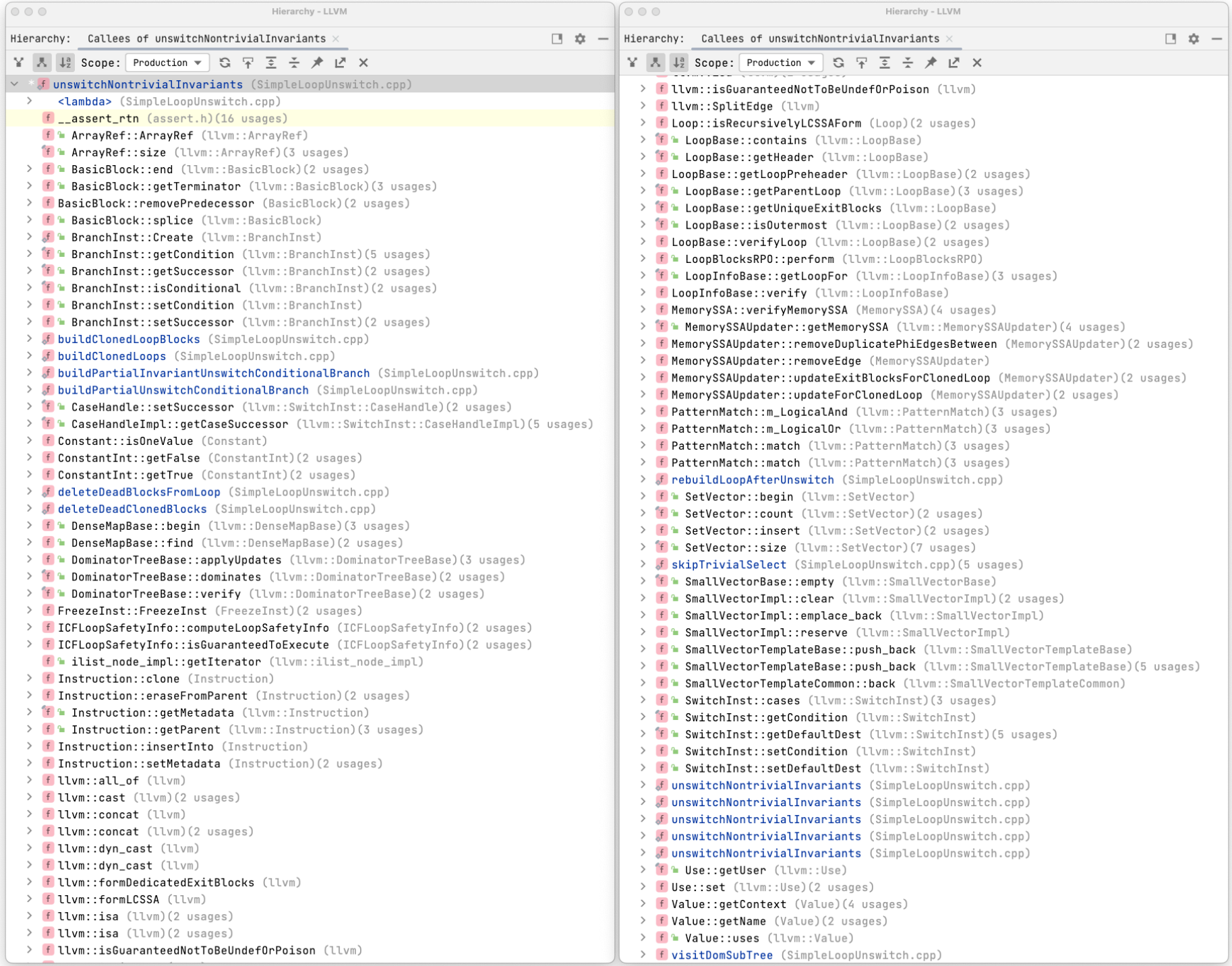
turnGaurdIntoBranch()

|Callee Hierarchy|



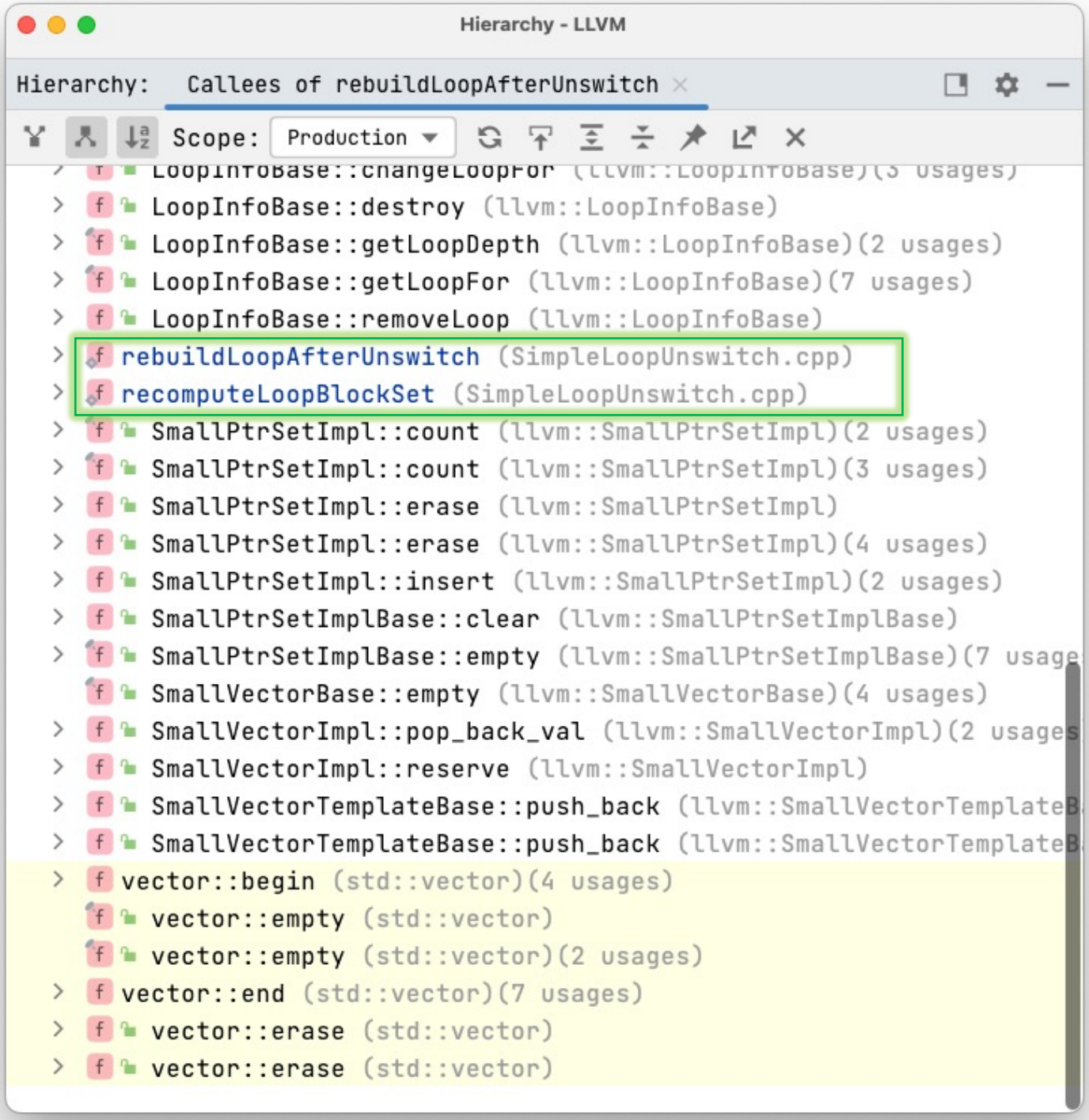
unswitchNontrivialInvariants

|Callee Hierarchy|

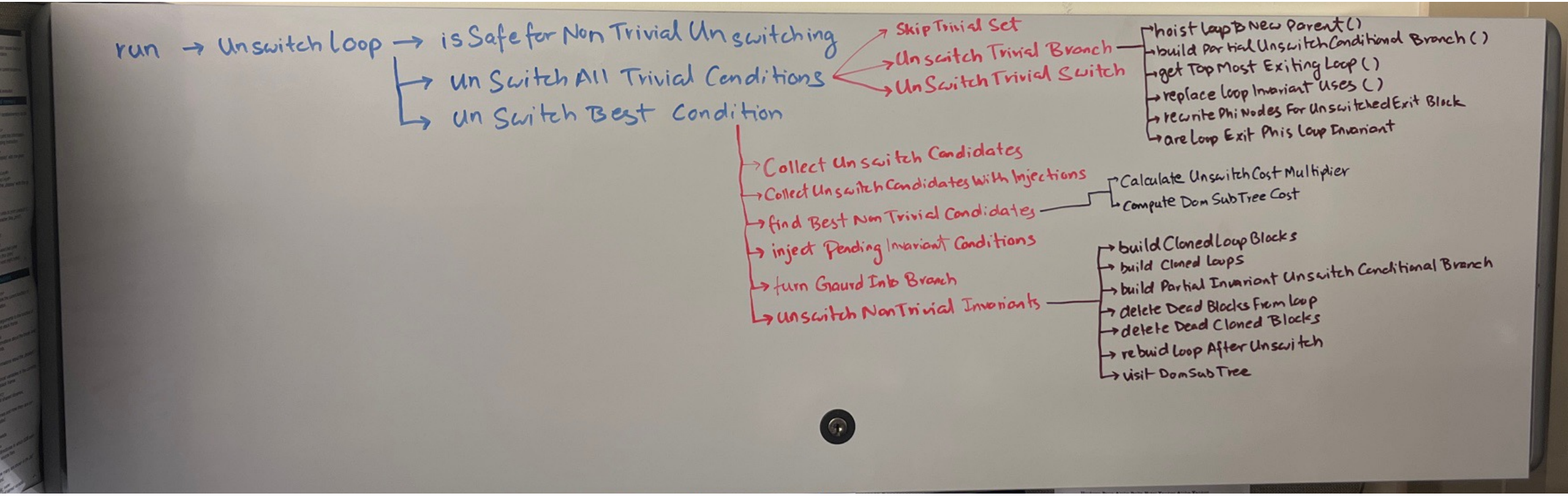


rebuildLoopAfterUnswitch()

|Callee Hierarchy|



Identifying interesting functions.



unswitchTrivialBranch()

Unswitch a trivial branch if the condition is loop invariant

- Should only be called when loop code leading to the branch has been validated as trivial
- Checks if the condition is invariant and one of the successors is a loop exit (implying no non-trivial blocks)
- Returns *false* if fails to unswitch
- If it can unswitch
  - Split the preheader
  - Hoist the branch above the split
- Can only unswitch conditional branches
  - If condition is Loop invariant -> Add it to the Invariants, FullUnswitch is set to true
  - Else -> collect loop invariants used by homogenous instruction graph from the given root

CollectHomogenousInstGraphLoopInvariants()

```
/// Collect all of the loop invariant input values transitively used by the
/// homogeneous instruction graph from a given root.
///
/// This essentially walks from a root recursively through loop variant operands
/// which have perform the same logical operation (AND or OR) and finds all
/// inputs which are loop invariant. For some operations these can be
/// re-associated and unswitched out of the loop entirely.
static TinyPtrVector<Value *>
collectHomogenousInstGraphLoopInvariants(const Loop &L, Instruction &Root,
                                         const LoopInfo &LI) {
  assert(!L.isLoopInvariant(&Root) &&
         "Only need to walk the graph if root itself is not invariant.");
  TinyPtrVector<Value *> Invariants;
```

hoistLoopToNewParent()

Hoist the current loop up to the innermost loop containing a remaining exit

- If the loop is already at the top level -> can't hoist
- Get exit loop for each exit block of the loop and update the newParent
- Return: when newParent = oldParent //nothing inside the loop
- Move the preheader to newParent
- Remove this loop from oldParent and add to newParent or toplevelLoop
- Create PHI nodes for new exit paths

Is called when the nesting relation for the loop might have changed

getTopMostExitingLoop()

Returns the top-most loop containing the Exit BB and having Exit BB as exiting block

- Search through the loop parents recursively until it reaches the topmost

```
478  static const Loop *getTopMostExitingLoop(const BasicBlock *ExitBB,
479                                            const LoopInfo &LI) {
480    const Loop *TopMost = LI.getLoopFor(ExitBB);
481    const Loop *Current = TopMost;
482    while (Current) {
483      if (Current->isLoopExiting(ExitBB))
484        TopMost = Current;
485      Current = Current->getParentLoop();
486    }
487    return TopMost;
488  }
```

replaceLoopInvariantUses()

Replaces uses of LIC in the loop with the given constant

- Finds the users of LIC
- Replaces the user instruction in the loop with the constant

```
234  static void replaceLoopInvariantUses(const Loop &L, Value *Invariant,
235                                        Constant &Replacement) {
236    assert(!isa<Constant>(Invariant) && "Why are we unswitching on a constant?");
237
238    // Replace uses of LIC in the loop with the given constant.
239    // We use make_early_inc_range as set invalidates the iterator.
240    for (Use &U : llvm::make_early_inc_range(Invariant->uses())) {
241      Instruction *UserI = dyn_cast<Instruction>(U.getUser());
242
243      // Replace this use within the loop body.
244      if (UserI && L.contains(UserI))
245        U.set(&Replacement);
246    }
247  }
```

areLoopExitPHIsLoopInvariant()

Checks that all the LCSSA PHI nodes in the loop exit block have trivial incoming value along the edge.

```
251  static bool areLoopExitPHIsLoopInvariant(const Loop &L,
252                                           const BasicBlock &ExitingBB,
253                                           const BasicBlock &ExitBB) {
254    for (const Instruction &I : ExitBB) {
255      auto *PN = dyn_cast<PHINode>(&I);
256      if (!PN)
257        // No more PHIs to check.
258        return true;
259
260      // If the incoming value for this edge isn't loop invariant the unswitch
261      // won't be trivial.
262      if (!L.isLoopInvariant(PN->getIncomingValueForBlock(&ExitingBB)))
263        return false;
264    }
265    llvm_unreachable("Basic blocks should never be empty!");
266  }
```

rewritePHInodesForUnswitchedExitBlock()

Rewrites the PHI nodes in an unswitched loop exit BB

such that they become trivial PHI nodes from the old preheader that now contains the unswitched terminator

- Conditions:
  - Loop exit and unswitched BB should be same
  - Exiting block is a unique predecessor of this basic block
- Update the incoming basic block of the PHI node

```
340  static void rewritePHINodesForUnswitchedExitBlock(BasicBlock &UnswitchedBB,
341                                                    BasicBlock &OldExitingBB,
342                                                    BasicBlock &OldPH) {
343    for (PHINode &PN : UnswitchedBB.phis()) {
344      // When the loop exit is directly unswitched we just need to update the
345      // incoming basic block. We loop to handle weird cases with repeated
346      // incoming blocks, but expect to typically only have one operand here.
347      for (auto i : seq<int>(0, PN.getNumOperands())) {
348        assert(PN.getIncomingBlock(i) == &OldExitingBB &&
349               "Found incoming block different from unique predecessor!");
350        PN.setIncomingBlock(i, &OldPH);
351      }
352    }
353  }
```

buildPartialUnswitchConditionalBranch()

Only unswitch some invariant conditions and not all

```
268   /// Copy a set of loop invariant values \p ToDuplicate and insert them at the
269   /// end of \p BB and conditionally branch on the copied condition. We only
270   /// branch on a single value.
271   static void buildPartialUnswitchConditionalBranch(
272       BasicBlock &BB, ArrayRef<Value *> Invariants, bool Direction,
273       BasicBlock &UnswitchedSucc, BasicBlock &NormalSucc, bool InsertFreeze,
274       const Instruction *I, AssumptionCache *AC, const DominatorTree &DT) {
275     IRBuilder<> IRB(&BB);
276
277     SmallVector<Value *> FrozenInvariants;
278     for (Value *Inv : Invariants) {
279       if (InsertFreeze && !isGuaranteedNotToBeUndefOrPoison(Inv, AC, I, &DT))
280         Inv = IRB.CreateFreeze(Inv, Inv->getName() + ".fr");
281       FrozenInvariants.push_back(Inv);
282     }
283
284     Value *Cond = Direction ? IRB.CreateOr(FrozenInvariants)
285                             : IRB.CreateAnd(FrozenInvariants);
286     IRB.CreateCondBr(Cond, Direction ? &UnswitchedSucc : &NormalSucc,
287                      Direction ? &NormalSucc : &UnswitchedSucc);
288   }
```

findBestNonTrivialUnswitchCandidate()

- The best candidate is the one with the following properties in order:
  1. An unswitching cost below the **threshold**
  2. The smallest number of **duplicated** unswitch candidates
     - To avoid creating redundant subsequent unswitching
  3. The smallest **cost** after unswitching.

- Prioritize reducing **fanout** of unswitch candidates
  - The cost remains below the threshold => this has a multiplicative effect.

unswitchNonTrivialInvariants()

If (**FullUnswitch** is False)

- We can only unswitch switches or conditional branches with:
  - an invariant condition or partially invariant instructions

- However, when unswitching a set of invariants combined with `and` or `or` or partially invariant instructions,
  - The combining operation determines the **best direction** to unswitch
  - We want to unswitch the direction that will **collapse the branch**.

deleteDead*Blocks()

Delete dead blocks from the loop

1. deleteDeadClonedBlocks()
   - Find all the dead clones and remove them from their successors.
   - Now, we have an accurate dominator tree, first delete the dead cloned blocks, then we can accurately build any cloned loops.
   - It is important to not delete the blocks from the original loop yet we still want to reference the original loop to understand the cloned loop's structure.

2. deleteDeadBlocksFromLoops()
   Find all the dead blocks tied to this loop and remove them from their successors.

rebuildLoopAfterUnswitch()

Rebuild a loop after unswitching removes some subset of blocks and edges.

- The removal may have removed some child loops but not all
  - However, they may need to be hoisted to the parent loop (or to be top-level loops).
  - The original loop may be completely removed

- This update creates a list of Siblings loops
  - The original loop will be the first entry of the list, if it is valid

- Returns
  - True: if the loop remains a loop after unswitching
  - False: if it is no longer a loop after unswitching

An (Interesting) Example.

overcomplicated_program_to_add_all_digits.cpp

```
int main() {
const int n = 1000000;
int arr[n];
int sum1 = 0, sum2 = 0;

for (int i = 0; i < n; i++) {
arr[i] = i;
}

for (int i = 0; i < n; i++) {
if (i % 2 == 0) {
sum1 += arr[i];
} else {
sum2 += arr[i];
}
}
int summ=sum1+sum2;
return summ;
}
```

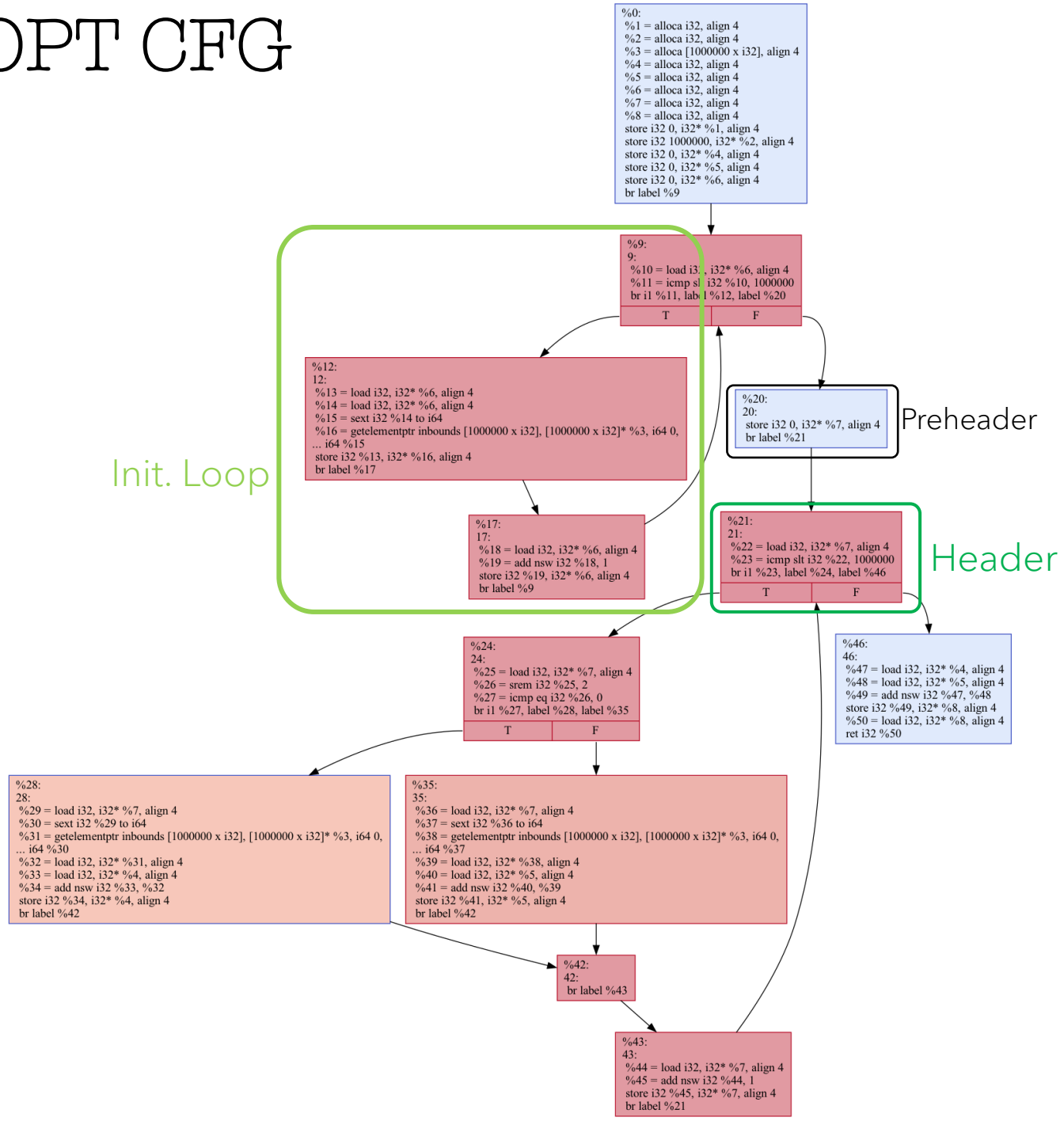
Simple Init. Loop

Switched Main Loop

Clang and OPT arguments

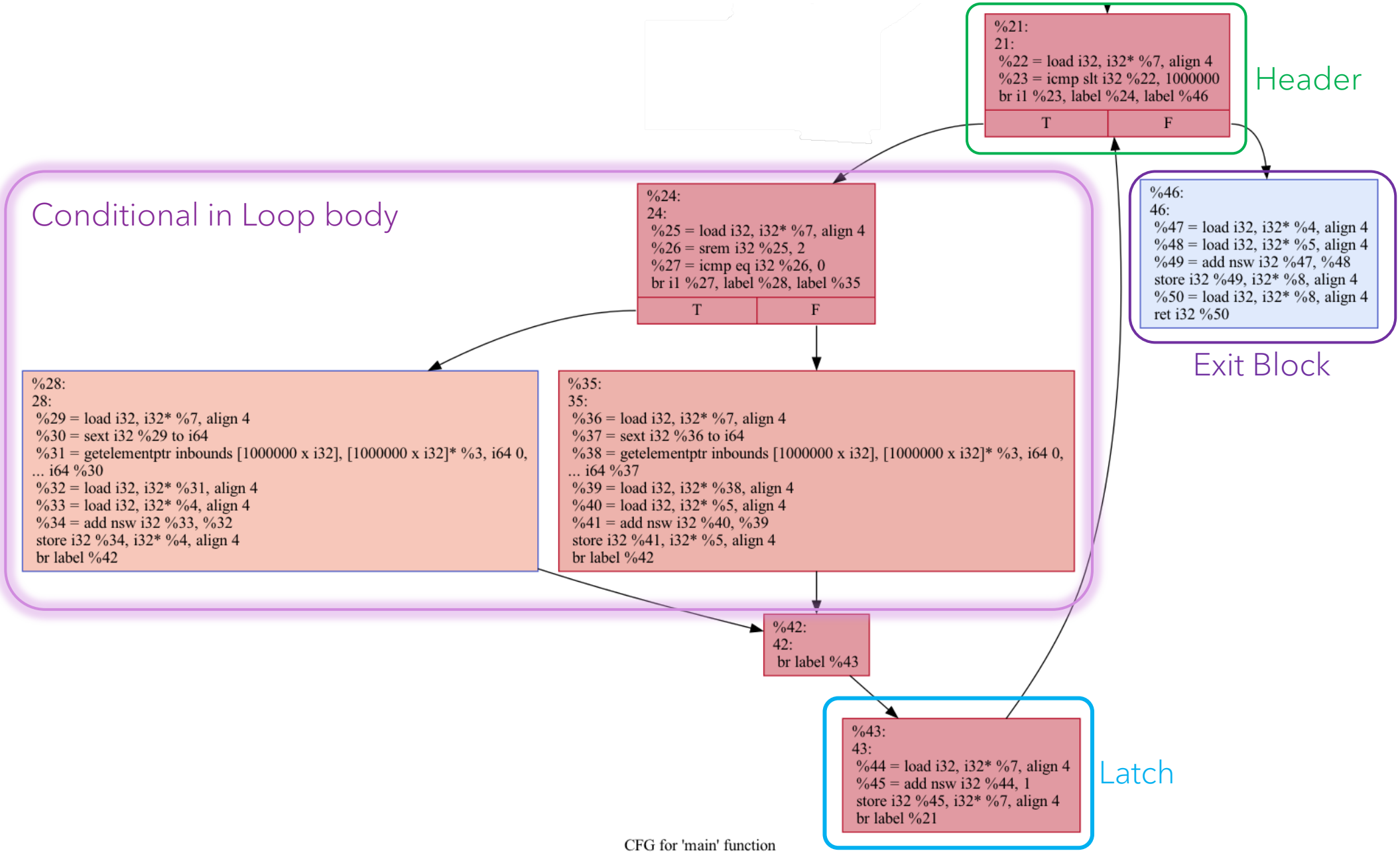
```
//LLVM BC
clang++ -O1 -c -emit-llvm main.cpp -o main03.bc

//Apply LoopUnswitch
opt -enable-new-pm=0 -mem2reg -gvn -instcombine -dce -loop-simplifycfg -licm
-loop-unroll -simple-loop-unswitch main03.bc -o mainUS.bc

//Create CFG
opt -enable-new-pm=0 -dot-cfg -disable-output mainUS.bc

//Convert to PNG
dot -Tpng -o mainUSopt.png .main.dot
```

Before OPT CFG

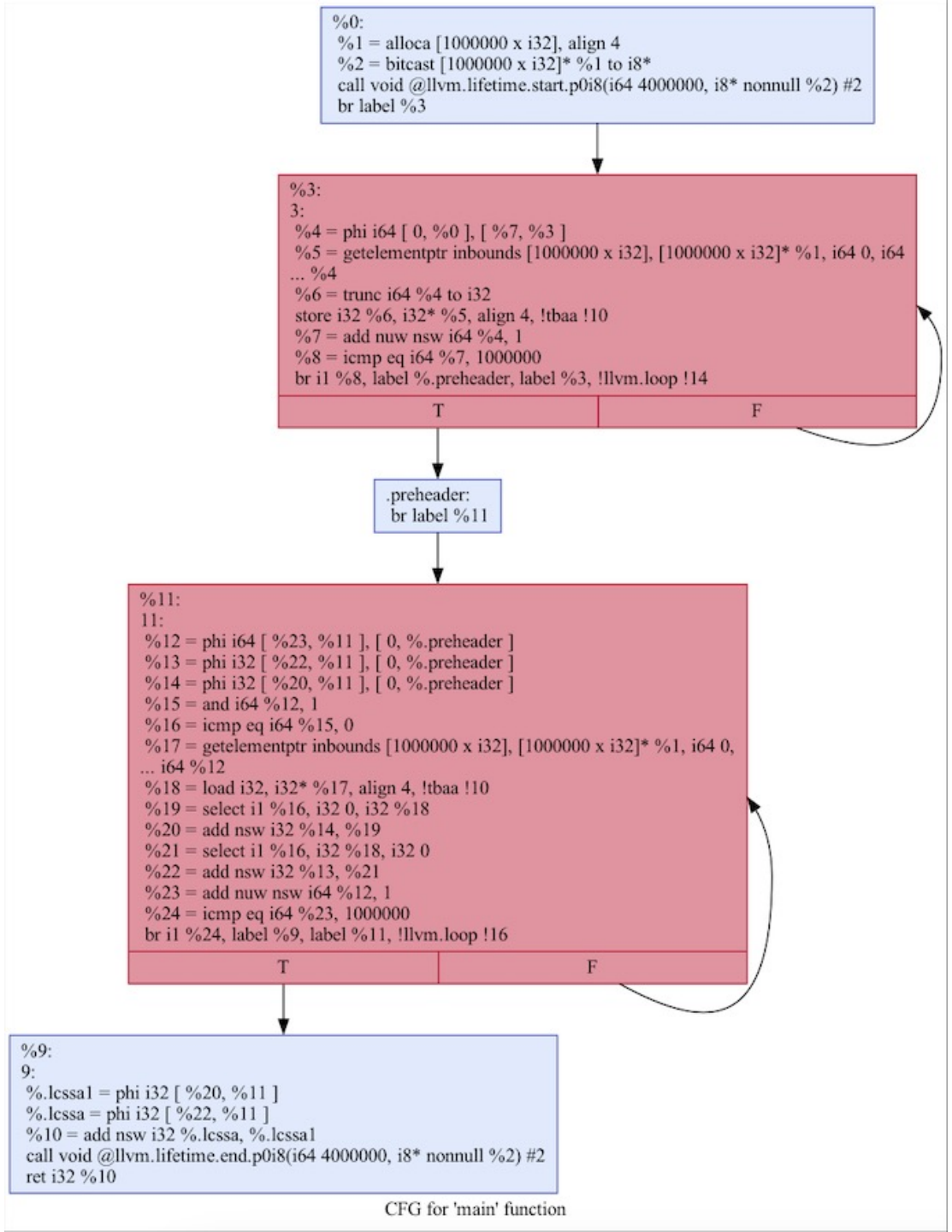


CFG for 'main' function

Before OPT CFG - Main Loop



After OPT CFG



After OPT LLVM IR

(Cleaned & Simplified for readability.)

```
define i32 @main() local_unnamed_addr #0 {
%1 = alloca [1000000 x i32], align 4
%2 = bitcast [1000000 x i32]* %1 to i8*
call void @llvm.lifetime.start.p0i8(i64 4000000, i8* nonnull %2) #2
br label %INITLOOP

INITLOOP: ; preds = %0, %INITLOOP
%init_itr_copy = phi i64 [ 0, %0 ], [ %initIterator, %INITLOOP ]
%array_itr_element = getelementptr inbounds [1000000 x i32], [1000000 x i32]* %1, i64 0, i64 %init_itr_copy
%init_itr_copy_32bits = trunc i64 %init_itr_copy to i32
store i32 %init_itr_copy_32bits, i32* %array_itr_element, align 4, !tbaa !10
%initIterator = add nuw nsw i64 %init_itr_copy, 1
%initloopindexbool = icmp eq i64 %initIterator, 1000000
br i1 %initloopindexbool, label %.preheader, label %INITLOOP, !llvm.loop !14

.preheader: ; preds = %INITLOOP
br label %MAINLOOP

EXITBLOCK: ; preds = %MAINLOOP
%.lcssa1 = phi i32 [ %SUM1, %MAINLOOP ]
%.lcssa = phi i32 [ %SUM2, %MAINLOOP ]
%return_value = add nsw i32 %.lcssa, %.lcssa1
ret i32 %return_value

MAINLOOP: ; preds = %.preheader, %MAINLOOP
%main_itr_copy = phi i64 [ %iterator, %MAINLOOP ], [ 0, %.preheader ]
%SUM2_cpy = phi i32 [ %SUM2, %MAINLOOP ], [ 0, %.preheader ]
%SUM1_cpy = phi i32 [ %SUM1, %MAINLOOP ], [ 0, %.preheader ]
;checking (i%2 == 0)
%AND_itr_1 = and i64 %main_itr_copy, 1 ; doing AND 1 with iterator for 01b (1d) = true; for 10b (2d) = false
%bool_AND_itr_1 = icmp eq i64 %AND_itr_1, 0 ; check if result of AND was 2
;check end
%main_itr_array_element = getelementptr inbounds [1000000 x i32], [1000000 x i32]* %1, i64 0, i64 %main_itr_copy
%Value_from_Array = load i32, i32* %main_itr_array_element, align 4, !tbaa !10
%sum_1_selector = select i1 %bool_AND_itr_1, i32 0, i32 %Value_from_Array
%SUM1 = add nsw i32 %SUM1_cpy, %sum_1_selector
%sum_2_selector = select i1 %bool_AND_itr_1, i32 %Value_from_Array, i32 0
%SUM2 = add nsw i32 %SUM2_cpy, %sum_2_selector
%iterator = add nuw nsw i64 %main_itr_copy, 1
%iteratorcheckBoolean = icmp eq i64 %iterator, 1000000
br i1 %iteratorcheckBoolean, label %EXITBLOCK, label %MAINLOOP, !llvm.loop !16
}
```

After OPT LLVM IR

(Cleaned & Simplified for readability.)

```
MAINLOOP: ; preds = %.preheader, %MAINLOOP
%2E = phi i64 [ %iterator, %MAINLOOP ], [ 0, %.preheader ]
%SUM2_cpy = phi i32 [ %SUM2, %MAINLOOP ], [ 0, %.preheader ]
%SUM1_cpy = phi i32 [ %SUM1, %MAINLOOP ], [ 0, %.preheader ]
;checking (i%2 == 0)
%AND_itr_1 = and i64 %2E, 1
; doing AND 1 with iterator for 01b (1d) = true; for 10b (2d) = false
%bool_AND_itr_1 = icmp eq i64 %AND_itr_1, 0
; check if result of AND was 0; i.e. itr was % 2.
. . .
%Value_from_Array = load i32, i32* % array_element, align 4, !tbaa !10
%sum_1_selector = select i1 %bool_AND_itr_1, i32 0, i32 %Value_from_Array
%SUM1 = add nsw i32 %SUM1_cpy, %sum_1_selector
%sum_2_selector = select i1 %bool_AND_itr_1, i32 %Value_from_Array, i32 0
%SUM2 = add nsw i32 %SUM2_cpy, %sum_2_selector
%iterator = add nuw nsw i64 %main_itr_copy, 1
%iteratorcheckBoolean = icmp eq i64 %iterator, 1000000
br i1 %iteratorcheckBoolean, label %EXITBLOCK, label %MAINLOOP, !llvm.loop !16
}
```

Stripped-down unswitched loop logic

```
;checking (i%2 == 0)
%AND_itr_1 = and i64 %main_itr_copy, 1 ; doing AND 1 with iterator for 01b (1d) = true; for 10b
(2d) = false
%bool_AND_itr_1 = icmp eq i64 %AND_itr_1, 0 ; check if result of AND was 2
; check end

. . .

%Value_from_Array = load i32, i32* %main_itr_array_element, align 4, !tbaa !10

%sum_1_selector = select i1 %bool_AND_itr_1, i32 0, i32 %Value_from_Array
%SUM1 = add nsw i32 %SUM1_cpy, %sum_1_selector

%sum_2_selector = select i1 %bool_AND_itr_1, i32 %Value_from_Array, i32 0
%SUM2 = add nsw i32 %SUM2_cpy, %sum_2_selector
```

When NOT to use unswitching?

- **When the loop contains few iterations**

  Loop unswitching can be useful when there are a lot of iterations in a loop, but if the loop has only a few iterations, the benefit of unswitching may be negligible.

- **When the loop contains complex conditional logic**

  If the loop contains complex conditional logic, unswitching may not be effective or may even increase the code size and runtime.

- **When the loop has side effects**

  If the loop has side effects, such as modifying global variables or performing I/O operations, unswitching may not be safe and can introduce unintended behavior.

- **When the loop has dependencies**

  If the loop contains dependencies, such as loop-carried dependencies, unswitching may not be effective or may even introduce new dependencies.

- **When the loop has non-uniform iterations**

  If the loop has non-uniform iterations, such as if the iterations depend on input data or user input, unswitching may not be effective or may even lead to incorrect behavior.

Thankyou.

Digitally signed by Saket Upadhyay 7
Date: 2023.03.22 17:07:42 -04'00'