How LLVM computes Dominator Tree & Dominance Frontier

Introduction

The dominator tree and the dominance frontier for a function are two pieces of very important informations used in SSA-construction and other compiler optimizations. In this article, I am going to explain in details how the LLVM compiler computes dominator trees and dominance frontiers. We will also take a look at some other LLVM optimization passes that rely on these information.

Version: LLVM 7.0.0

Author: Ziang Wan

Dominator Tree: the Pass - domtree

Firstly, let’s take a look at the algorithm to compute the dominator tree for the basic blocks of a function. In LLVM, the dominator tree computation is a function pass through a function. It should already be run once when LLVM is translating high-level languages into LLVM IR. However, you can also specifically enforce opt to run the pass by

```
opt -domtree ...
```

If your pass needs to get the dominator tree information for the current function. Use:

```
// in your getAnalysisUsage(AnalysisUsage &AU)
AU.addRequired<DominatorTreeWrapperPass>();

...

// later in your code
DominatorTree &DT = getAnalysis<DominatorTreeWrapperPass>(
).getDomTree();
```

Source Code Locations

If you prefer to read the source code, you can take a look at these files. However, notice that these files also implement the computation of post-dominator tree and post-dominance relationship.

- /include/llvm/IR/Dominators.h

- /lib/IR/Dominators.cpp

  The above two files define and implement the pass -domtree.

- /include/llvm/Support/GenericDomTree.h

- /include/llvm/Support/GenericDomTreeConstruction.h

  The above two files defines and implements data structures used in both dom and postdom trees. The actual construction algorithm of a dominator tree is in GenericDomTreeConstruction.h.

About the DominatorTree object

The pass will give you a `DominatorTree` object containing the dominator tree information for the current function. There are a handful of useful functions you can call, including a variety of `dominates()` and `isReachableFromEntry()`.

Internally, the DominatorTree is represented using a standard tree data structure. Each tree node represents a single basic block in the function. Each node also maintains a pointer to its immediate dominator and a list of pointers to its children in the dominance tree, which are basic blocks being dominated by this node. A dummy node is created to be the immediate dominator of the entry block.

The algorithm & its Time Complexity

In short, you need to know that the DominatorTreeWrapperPass computes the dominator tree in O((N+M)*(log(N)) time. Updating the dominator tree is faster as the pass does not recompute the whole tree during inserting/deleting.

The pass has a custom algorithm that computes the dominator tree. It firstly does a DFS walk over the CFG of the function and labels each basic block like topology sort. Then, it computes the semi-dominator for each basic block, and, from the semi-dominators, it computes the immediate dominator for each basic block. Finally, a tree structure is built according to the immediate dominator for each basic block.

Throughout the process, a map that maps a tree node to the values associated with that node is used. In tree walks, worklists, which are implemented as vectors, are used in while loops. From my observation, no other complicated data structure is used in the algorithm.

For a more detailed explanation on the algorithm, see Dominator Tree of a Directed Graph

I will show you a brief example of dominator tree construction for a simple c program:

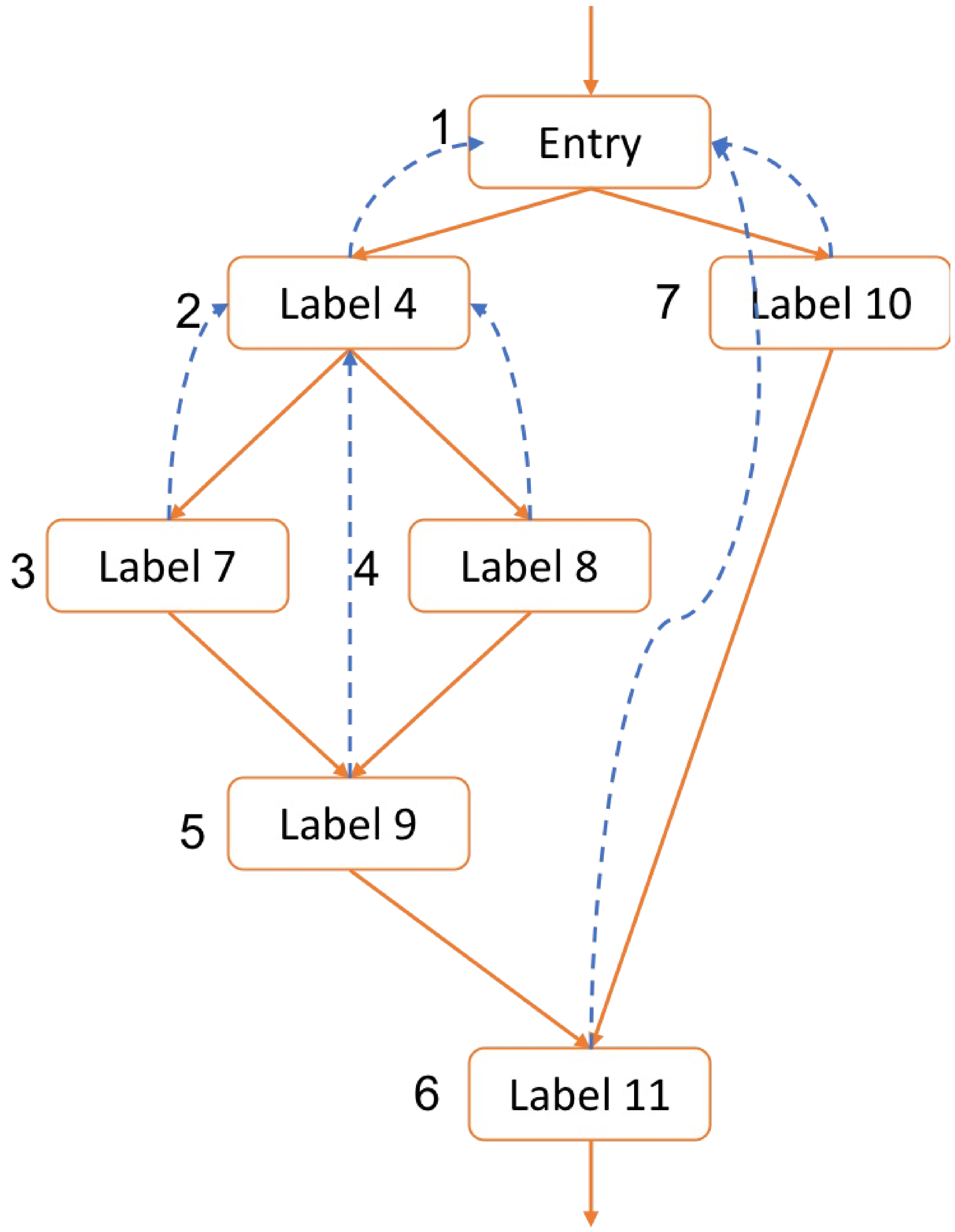
```
int main(int argc, char *argv[]){
    int t1 = 1;
    int t2 = 2;
    if (t1 != t2) {
        if (t1 > 1) {
            t1 = 2;
        } else {
            t1 = 10;
        }
    } else {
        t2 = 3;
    }
    return 0;
}
```

The LLVM IR for the above code with `-sccp` and `-instcombine` optimization passes:

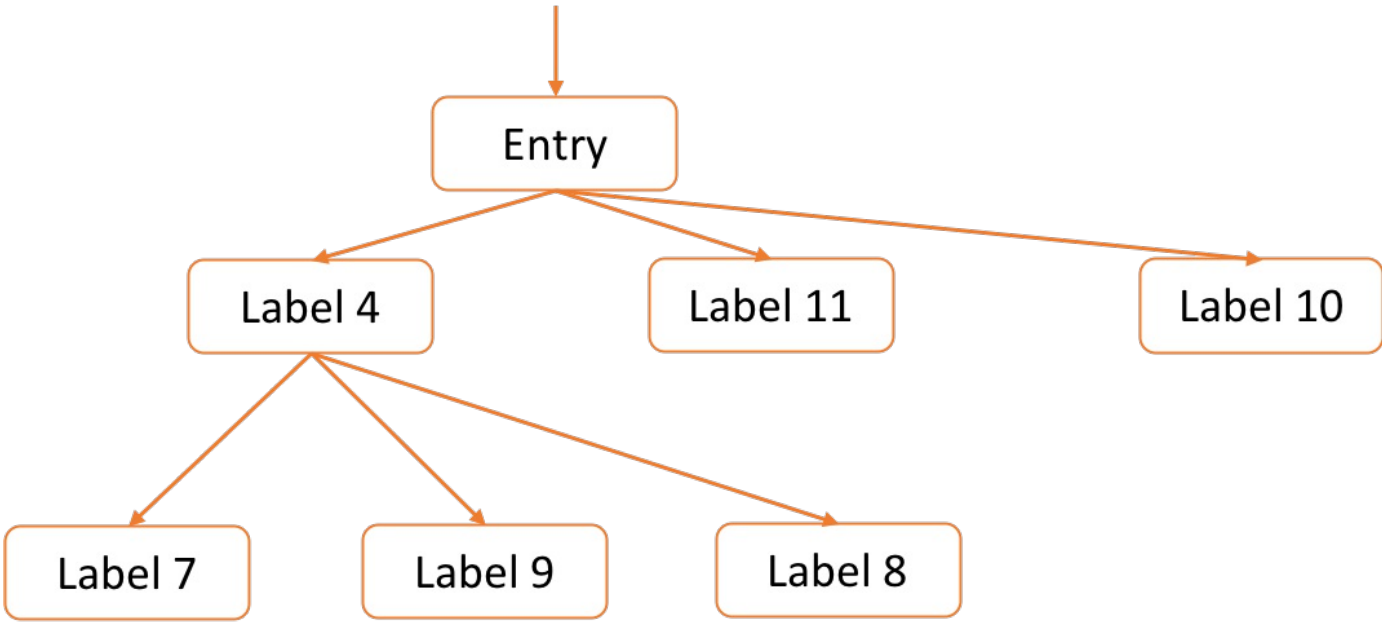
```
; Function Attrs: noinline nounwind uwtable
define i32 @main(i32, i8**) #0 {
  %3 = alloca i32, align 4
  store i32 1, i32* %3, align 4
  br i1 true, label %4, label %10

; <label>:4:                                      ; preds
= %2
  %5 = load i32, i32* %3, align 4
  %6 = icmp sgt i32 %5, 1
  br i1 %6, label %7, label %8

; <label>:7:                                      ; preds
= %4
  br label %9

; <label>:8:                                      ; preds
= %4
```

```
  br label %9

; <label>:9:                                      ; preds
= %8, %7
  %storemerge = phi i32 [ 10, %8 ], [ 2, %7 ]
  store i32 %storemerge, i32* %3, align 4
  br label %11

; <label>:10:                                     ; preds
= %2
  br label %11

; <label>:11:                                     ; preds
= %10, %9
  ret i32 0
}
```

The CFG with DFS-walk labels and semi-dominance relationship (dashed blue arrow):



Finally, the dominator tree. However, the immediate dominator for a basic block won't always be the semi-dominator for that block. For more details, see the above link.



Does the algorithm take advantage of the SSA form?

No, it does not. The algorithm performs at the basic block level. It doesn't matter what is the form of the instructions inside each basic block.

What transformation in LLVM code can take advantage of -domtree?

The mighty dominator tree information is used in many optimization passes. To name just a few:

- /lib/Transforms/Scalar/GVN.cpp (global value numbering)
- /lib/Transforms/Scalar/SROA.cpp (scalar replacement of aggregates)
- /lib/Transforms/Scalar/LoopUnrollPass.cpp (loop unrolling)

- /lib/Transforms/InstCombine/InstructionCombining.cpp (instruction combining)
- /lib/Transforms/Mem2Reg.cpp (memory to register)

A general pattern for these transformations is that they all need def-use information for values.

Dominance Frontier: the Pass -domfrontier

Then, let’s take a look at the algorithm that computes dominance frontier information for a function. LLVM computes the dominance frontier information for a function in a function pass: -domfrontier. The pass needs to know the dominator tree for that function so the pass has to be run after the -domtree pass. Again, to enfoce opt:

```
opt -domfrontier ...
```

In code:

```
// in your getAnalysisUsage(AnalysisUsage &AU)
AU.addRequired<DominanceFrontierWrapperPass>();

...

// later in your code
DominanceFrontier &DF = getAnalysis<DominanceFrontierWrapp
```

```
erPass>().getDominanceFrontier();
```

Source Code Locations

LLVM 7.0.0 still relys on the old version of the algorithm:

- /include/llvm/Analysis/DominanceFrontier.h
- /include/llvm/Analysis/DominanceFrontierImpl.h
- /lib/Analysis/DominanceFrontier.cpp

  The above three files define and implement the pass -domtree.

However, the above three files have already been marked deprecated. No more use of the two should be added.

About the DominanceFrontier object

The pass will give you a `DominanceFrontier` object containing the dominance frontier information for the current function. The object contains a protected member that is a map from a block to a set of blocks in the dominance frontier of that block. In code: `std::map<Block *, std::set<Block *>>`. The only way to access the map is through the iterator of the `DominanceFrontier` object. For instance:

```
for (auto const &it : DF) {
    BasicBlock *B = it.first;
    std::set<BasicBlock *> &DF = it.second;
}
```

One other interesting member function I discover is `getRoot`, which returns the root block of the CFG.

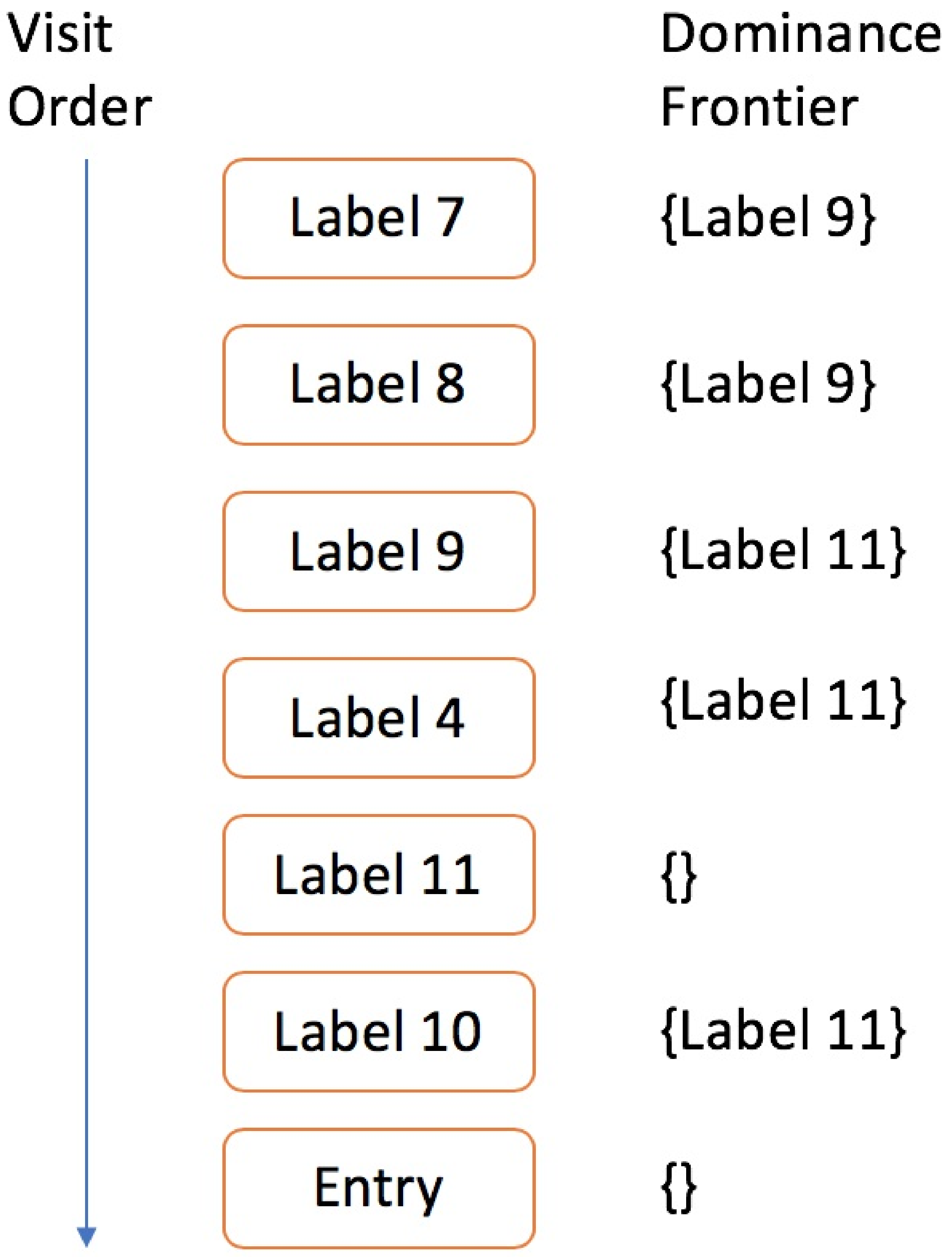
The algorithm & its Time Complexity

The algorithm used to compute the dominance frontier maintains a worklist of basic blocks. Every iteration, it pulls one basic block from the list and computes its dominance frontier.The algorithm keeps going until the worklist is empty.

If worklist starts at the leaf basic blocks in the dominator tree and goes upward until the root, the algorithm can achieve a linear running time O(N). In LLVM 7.0.0, the implementation starts at the root node and does a post-order traversal of the dominator tree.

For a more detailed explanation of the algorithm, see the original paper or the slides at slide 17.

Following the example for -domtree, the domiance frontier information for that c program is:



Visit Order		Dominance Frontier
↓	Label 7	{Label 9}
	Label 8	{Label 9}
	Label 9	{Label 11}
	Label 4	{Label 11}
	Label 11	{}
	Label 10	{Label 11}
	Entry	{}

Does the algorithm take advantage of the SSA form?

No, it does not. The algorithm, like -domtree, performs at the basic block level. It doesn’t matter what is the form of the instructions inside each basic block.

What transformation in LLVM code can take advantage of -domfrontier?

There are currently two such transformations:

- /lib/Transforms/Scalar/ADCE.cpp (aggressive dead code elimination)
- /lib/Transforms/Scalar/GVNHoist.cpp (global value numbering and hoisting)

Other util files that use `DominanceFrontier`:

- /lib/Transforms/Utils/PromoteMemoryToRegister.cpp (-mem2reg util)
- /lib/Transforms/Utils/SSAUpdaterBulk.cpp