Department of Computer Science and Engineering

*UGP-II/CS396*

# Implementation of HPSSA Construction Algorithm in the LLVM Compiler Framework

*Authors:*
Abhay Mishra, 190017
Mohd Muzzammil, 190503

*Supervisors:*
Prof. Subhajit Roy
Mr. Awanish Pandey
Mr. Sumit Lahiri

November 2021

## Acknowledgements

We are deeply indebted to **Prof. Subhajit Roy** for allowing us to work with him. We are thankful to him for having regular meetings despite his busy schedule and helping us making our implementation more efficient. We would also like to thank **Mr. Sumit Lahiri(PhD CSE, Y19)** and **Mr. Awanish Pandey(Senior Engineer, Qualcomm)** for their invaluable support whenever we were stuck, helping us out to make steady progress along the way and in achieving our objective.
Last but not the least, we would like to thank our families for their consistent support during the pandemic.

## Abstract

Hot Path Static Single Assignment, HPSSA in short, is a data structure built upon the well recognized Static Single Assignment(SSA) Form of Intermediate Representation(IR) of Programs. HPSSA form weaves Static Program Information with run-time information allowing speculative optimizations by compilers. The HPSSA Construction Algorithm [1] takes Hot Path Profile Information and uses Data Flow Analysis to instrument the CFG of the program in SSA form. The instrumentation is in the form of tau functions which serve as a container of the run time profile information. The current implementation consists of a set of two compiler passes built using modern LLVM compiler framework. Construction pass inserts tau function at suitable places, which can be used later by other passes and destruction pass removes the instrumentation converting the program back to SSA form. The use of suitable data structures and CFG traversal routines makes the current implementation efficient in running time and memory usage.

The project primarily involved writing two compiler passes: one for tau insertion and argument allocation. Another for removing the instrumentation. For automated testing of memory efficiency and robustness of the implementation, a custom path profiler was built.

## Contents

## 1 What is HPSSA form?

Hot Path Static Single Assignment, HPSSA in short, is a data structure built upon SSA-like Intermediate Representation(IR). The core idea is to introduce $\tau$-functions as definition “filters”, similar to the $\phi$-functions of SSA form which are used as definition “mergers". Using this filtering mechanism, we can filter the hot reaching definitions from the cold ones. This structure weaves static program information with run-time path profile information and facilitates speculative optimizations by compilers that were not possible before.

## 2 Terminology

Following terms will be used frequently, so we desribe them below:

- **(Acyclic) Hot path**: For a program, given a set of inputs and a threshold frequency, the acyclic paths that are being visited more frequently than this fixed threshold are called hot paths of that program.
- **Buddy Set**: A Buddy Set is associated with a basic block. A set of hot paths reaching a basic block are called *buddies* at the current basic block if they reach the current basic block through same sequence of basic blocks. A buddy set is therefore the collection of buddies.
- **Caloric Connector**: A caloric connector is associated with a $\phi$-instruction. These are basic blocks where both hot and cold arguments of a $\phi$-instruction reach simultaneously.
- **Dominance and Dominator Tree**: A basic block $BB_2$ is dominated by $BB_1$ if every path reaching $BB_2$ must pass through $BB_1$. A *dominator tree* is a construction on the CFG in which each children is immediately dominated by the parent in the CFG.
- **Structure of LLVM IR**: LLVM IR is structured in the following way($\rightarrow$ denotes *contains*):
  ```
  Module -> {Global Var, Functions, Debug Info, ...}
  Function -> Basic Blocks -> Instructions
  ```

## 3 Implementation Phases

Our Implementation was divided into the following phases, and we will discuss them at length in the subsequent sections:

1. Getting The Hot Path Profile Information
    - Implementation of Path Profiler
2. Tau Insertion Phase
    - Representing $\tau$-instruction in LLVM IR
    - Computing Caloric Connectors
    - Inserting $\tau$-instructions
    - Allocating Arguments To $\tau$-instructions
    - Allocating Arguments To $\tau$-instructions (Implementation)
    - Replacing uses of $\phi$-instruction with corresponding $\tau$-instructions
3. Tau Destruction Phase
    - Deleting the $\tau$-instruction
    - Inserting appropriate instruction at its place

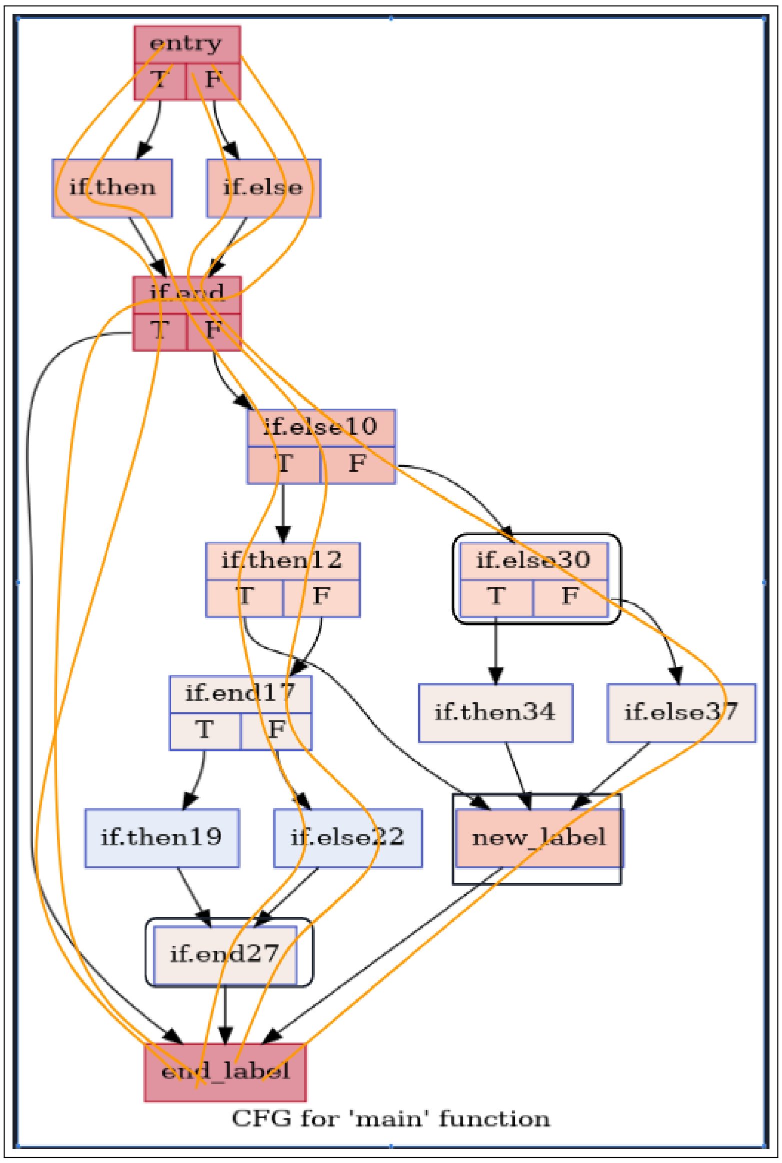


Figure 1: CFG of an example program.Hot paths and Caloric connectors are have been marked.

## 4 Getting The Hot Path Profile Information

The Hot path Profile Information is stored in `HotPathSets`:

```
HotPathSets :: map<BasicBlock*, BitVector>
HotPathSets(BB) = HotPath
HotPath :: BitVector, set bits in HotPath signifies
that the hot path with id=index passes through BB.
```

The routine `HPSSAPass::getProfileInfo(&F)` takes input in the following format:

```
No of Hot paths
<1> No of basic blocks in Hot path
<1> BasicBlock1 BasicBlock2 ...
<2>
<2>
...
...
```

And Outputs `HotPathSets`.

The routine simply reads each line, and for each basic block `BB` present in a hot path `HP`, it sets the bit corresponding to HP in HotPathSets[BB]. Thus, at the end for each basic block `BB`, `HotPathSets[BB]` stores information about the hot paths that passes through BB. Finally we return `HotPathSets`.

## 5 Implementation of Path Profiler

The rudimentary path profiler we implemented processes only main function of the program and skips others. Because we use acyclic path profile information as input to our HPSSA Pass, we need to break paths on loop headers(similar to Ball Larus Path Profiling).

The main steps of instrumentation are described below:

1. Initialize `count` to 0 and open a file `bbMap.txt`
2. Store all the backedges of the main function.

   ```
   llvm::FindFunctionBackedges(F, result)
   result = { <from, to> | a backedge starts from
             "from" and goes to "to" }
   ```

3. For each basic block BB in function
   - (a) Increment the `count`.
   - (b) insert a call to `counter` at the end of BB with `count` as the argument.
     - We Used
       ```
       CallInst::Create(Func,Args,NameStr,InsertBefore)
       InsertBefore = BB.getTerminator()
       // last instruction of basic block
       ```
4. Because the values of `count` uniquely identify a basic block, we store the mapping between Basic Block in `bbMap.txt` The format is:

   ```
   No of basic blocks
   <1> count FunctionName BasicBlockName
   <2>
   ...
   ```

5. For each Backedge, we split and insert a new basic block. This new basic block contains a call to `counter` with "-1" as its argument.

- We use

  ```
  llvm::SplitEdge(from,to)
  "from" and "to" are basic blocks and were
  stored in the beginning
  ```

The `counter` function opens a file "pathList" (or create a new one if its not already there) and start writing from newline. It will keep printing the `count` passed as an argument to this file in a line. However, Whenever "-1" is passed as an argument it start writing to a new line . As clear from this description, each line represents a path.

A simple routine named "bbReader" takes this "pathList" file and "bbMap.txt" file as input and according to the threshold value selects path which are visited more than this threshold value. The "bbMap.txt" file contains the mapping, which will be used to convert the values of `count` to corresponding Basic Block names. This routine will write its output to "pathProfile.txt" in the following format:

```
No of Hot paths
<1> No of basic blocks in Hot path
<1> BasicBlock1 BasicBlock2 ...
<2>
<2>
...
...
```

This "pathProfile.txt" will be passed as an input to our `HPSSAPass::getProfileInfo()`.

## 6 Tau Insertion Pass

Tau Insertion Pass consists of the following functions:

- `map<BasicBlock *, BitVector> getProfileInfo(Function &F)`: Returns the acyclic hot path profile information from the profiler.
- `map<BasicBlock *, bool> getCaloricConnector(Function &F)`: Computes Caloric Connectors from the given hot path information.
- `PreservedAnalyses run(Function &F, FunctionAnalysisManager &AM)`: Iterates over the phi functions and insert corresponding tau functions at suitable places. Allocates arguments to the tau functions.
- `void Search(BasicBlock &BB, DomTreeNode &DTN)`: Replaces the uses of the phi with its appropriate tau counterpart.

## 7 Tau Representation in LLVM

As recommended in LLVM docs on extending LLVM, we have represented the $\tau$-instruction in LLVM as an intrinsic. The signature is as follows:

```
def int_tau : DefaultAttrsIntrinsic
             < [llvm_any_ty],
               [llvm_vararg_ty],
                          [  ] >;
```

The intrinsic takes argument of any type and supports variable arguments. CFG verifier is modified to ignore checks on this new intrinsic as we do not implement how to convert $\tau$-instruction to bitcode and will have to eventually remove them anyway.

## 8 Computing Caloric Connectors

The function `HPSSAPass::getCaloricConnector(Function F)` is the routine that marks the basic blocks as caloric connector. The function returns a map named `isCaloric` with the following signature:

```
isCaloric :: map<BasicBlock*, Bool>
isCaloric(currBB) = True  | if currBB is a caloric connector
                    False | otherwise
```

In order to infer whether or not a basic block is a caloric connector, we need to maintain buddy set information. A buddy set is represent as:

```
BuddySet :: map<BasicBlock *, vector<BitVector>>
BuddySet(BB) = {Buddy | Every hot path in Buddy carry same
set of hot definition}
Buddy : A BitVector, each set bit represent a hot path.
```

In order to keep track of new hot paths that might start from loop headers(Incubation Nodes), we maintain a set storing all the hot paths encountered till now. We use `allPaths` and `IncubationPaths`:

```
allPaths        :: BitVector
A set bit in allPaths while visiting a basic block BB
signifies that the path with id = index, has been
encountered when we reached BB. e.g. All hot paths starting
from entry block will be always in the set.

IncubationPaths :: BitVector
A set bit in IncubationPaths while visiting a basic block
BB signifies that the path with id = index, has started
(Incubated) from BB. e.g. all hot paths that start from
entry basic block will be in this set while starting from
entry basic block, but for all upcoming basic blocks, these
hot paths will not be present in IncubationPaths
```

Maintaining `allPaths` is simple, while visiting basic blocks, we update allPaths as follows:

```
allPaths |= IncubationPaths
```

Having all these data structures, The algorithm proceeds as described below:

1. Traversed the CFG in topological order.
    - `Function::RPOT()` was used for this purpose
    - BB is the basic block under consideration.
2. Calculate IncubationPaths

    -
      ```
      IncubationPath = allPaths;
      IncubationPath &= HotPathSet[BB]; // bitwise AND
      IncubationPath ^= HotPathSet[BB]; // bitwise XOR
      HotPathSet[currBB] provides information about the
      hot paths that pass through current basic blocks
      ```
3. If BB is the entry block then `BuddySet` simply consist of one BitVector, containing all hot paths starting from this entry block. i.e.

    ```
    BuddySet[BB] = { HotPathSet[BB] }
    ```

    A entry block cannot be a caloric connector and we have updated the `BuddySet` so continue to the next basic block.
4. If it is not a entry basic block then iterate over all the predecessors of the current basic blocks.
    - Using `llvm::predecesors(BasicBlock*)`
5. For a predecessor,
    (a) if no hot path passes through it, then a cold definition reaches BB through this predecessor, i.e. A cold path flows through BB.
    (b) Othrewise, A hot path flows through current basic block. Now, Iterate over all buddies(Each carrying a unique hot defn) and do the following:

        i. If buddy and current basic block has no path in common, then a hot defintion is missing, thus a cold defn flows through this predecessor block.
        ii. If there is a common path, then update the buddy set information of BB.
    (c) The update of Buddy set information must maintain the invariant ( or the property of Buddies ) that a set of hot paths are buddies if they reached the current basic block by following same set of basic blocks.

6. All hot definition must be passed through each hot path incubating from current basic block, if any. Implementation wise:

   ```
   for all buddies:
           buddies |= IncubationPaths
   ```

7. If BB contains both a hot path and a cold path it is marked as a caloric connector:

   ```
   isCaloric[BB] = True
   ```

8. Finally after traversing all basic block, return `isCaloric`.

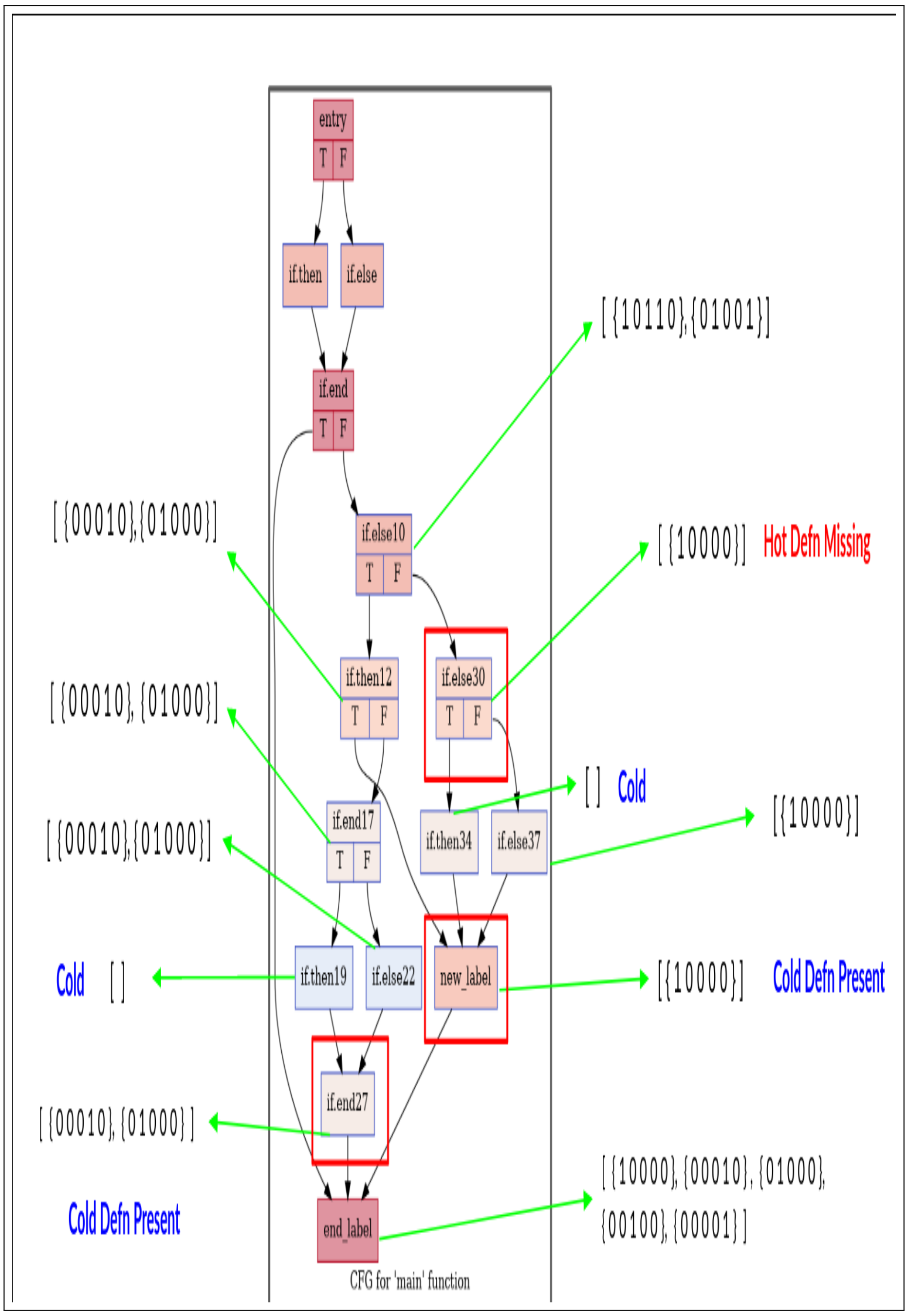


Figure 2: Computing caloric connectors

# 9 Allocating Arguments To Tau Functions

Arguments of tau are a subset of arguments of corresponding phi functions that reach through hot path at the current location. We use a structure defAccumulator for storing the information about hot definitions as suggested in the original paper. The structure of defAccumulator is as shown below:

```
defAccumulator( phi, currBB ) = frame
frame(hotDef) = hotPaths, "hotDef" flows through
                "hotPaths"
phi       := A phi function
currBB    := current basic block
hotDef    := A single hot definition
hotPaths := A set of hotPaths which carried "hotDef"
            to the current basic block.
```

The purpse of defAccumulator is two-fold:

- Provide arguments for taus if needed.
- Store hot paths through which a particular hot definition reaches current basic block.This allows us to infer which definition would be passed to successor along a hot path.

## Implementation Details

The implementation consists of following steps:

1. Traverse the CFG
   - We traversed the basic blocks in topological order using `Function::RPOT()`
   - This ensures that whenever we visit a basic block all of its immediate predecessors have been visited.
   - Thus all the hot definitions needed at the current basic block must be available at the time of visit.
2. Traverse over each phi of current basic block and initialize the `defAccumulator`.

  - The traversal is done using `BasicBlock::phis()`.
  - The LLVM counterpart of different structures used in `defAccumulator` are described below:

    ```
    phi       := PhiNode*
    currBB    := BasicBlock*
    hotDef    := Value*
    hotPaths  := BitVector
    frame := map<{PhiNode*, BitVector}}
    defAccumulator :=
             map<{PhiNode*, BasicBlock*}, frame>
    ```

  - For each argument of phi, check whether it is a hot definition or not. If it is a hot definition then it flows through all the hot paths coming to the current block:

    ```
    defAccumulator(phi,currBB) = frame
    frame[arg] = HotPathSet[currBB]
    ```

3. For each phi, traverse the basic blocks in topological order starting from current basic block
   - We simply use a copy of current `rpo_iterator` (corresponding to current basic block being visited).
4. If a tau is present in the block then allocate the hot definition reaching to the block as an argument to the tau.
5. Pass the Definitions to the successors through hot paths starting from current basic blocks.
   - We use `LLVM::succesor()` and operations on `BitVector` to get the hot definitions which should be passed to the succesors.
6. For a given phi, we stop traversing the CFG when we hit the dominance frontier of `currBB`.
   - We used `LLVM::DominatorAnalysisPass` and `LLVM::dominates()` for this purpose.

After the execution of algorithm, a basic block in which a tau is inserted looks like:

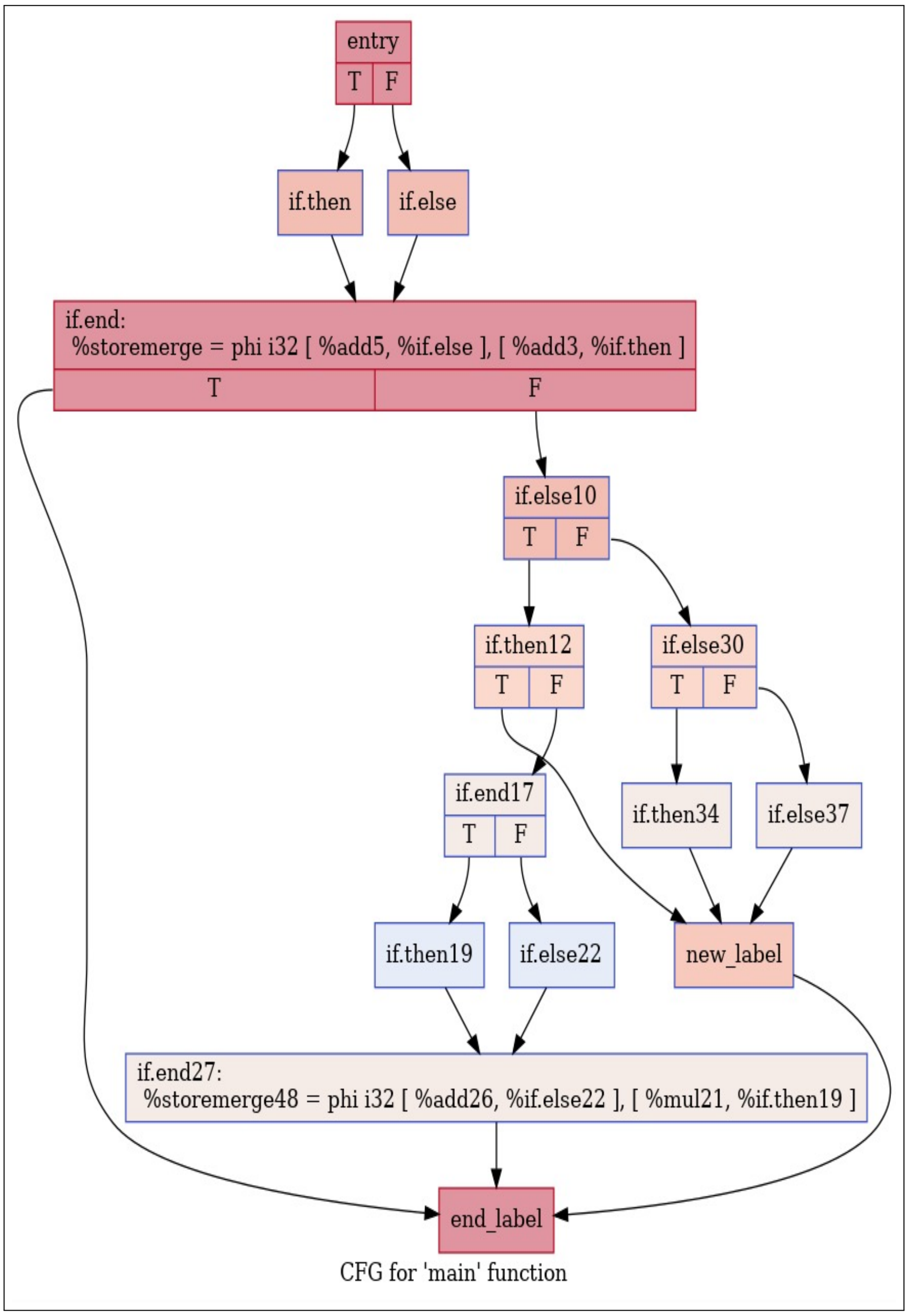


Figure 3: CFG before tau insertion

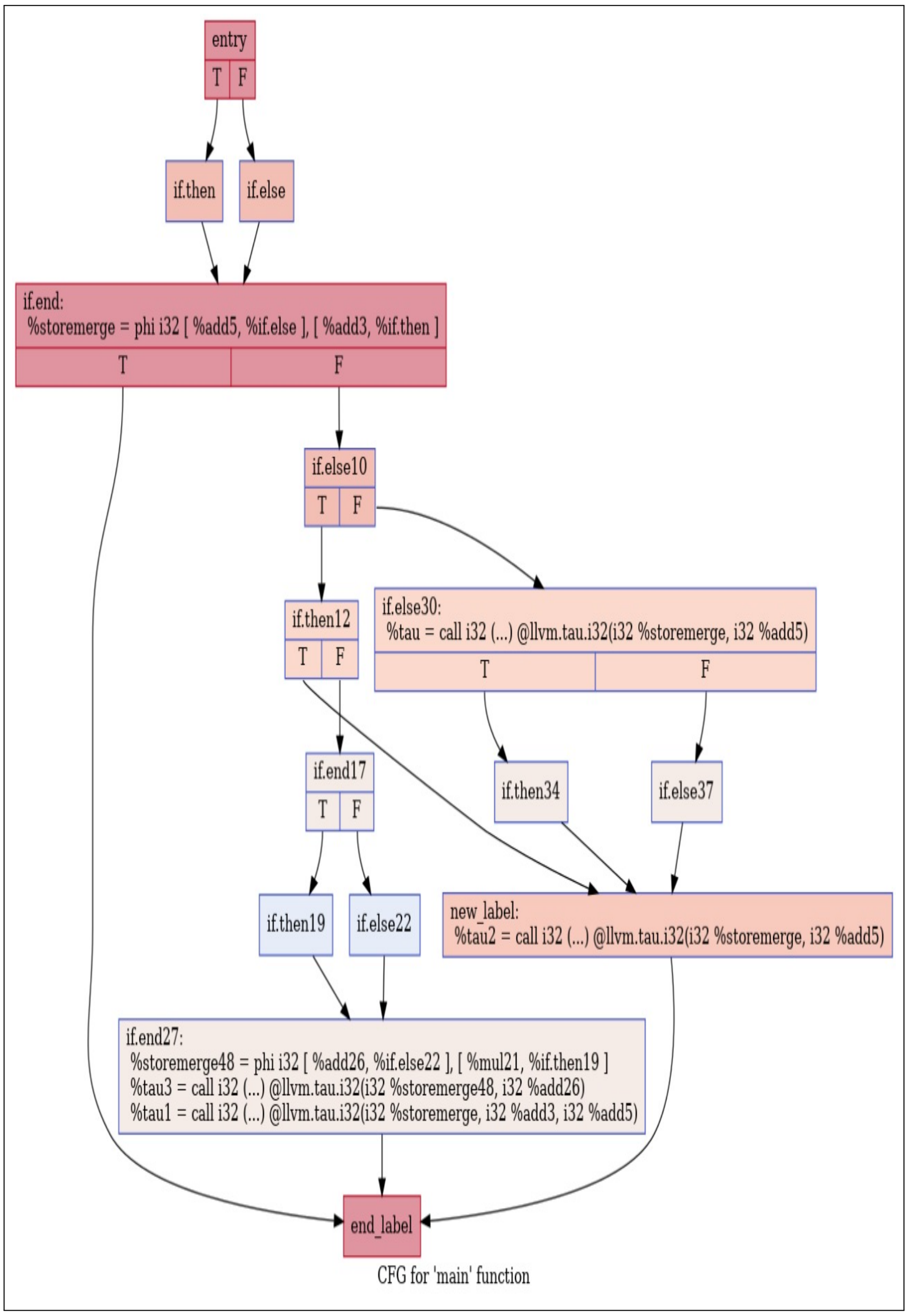
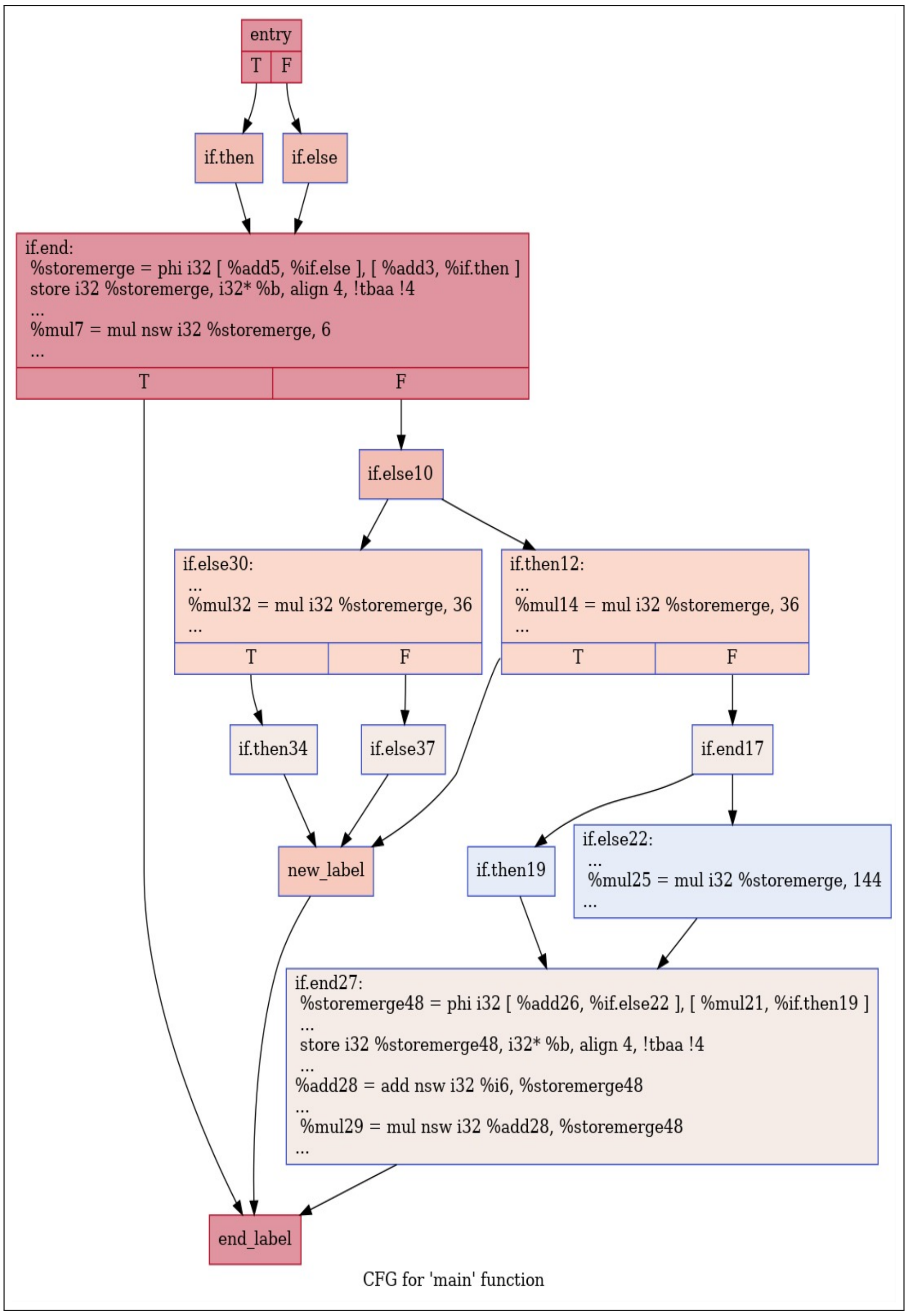

entry
T
F
if.then
if.else
if.end:
%storemerge = phi i32 [ %add5, %if.else ], [ %add3, %if.then ]
T
F
if.else10
T
F
if.then12
T
F
if.else30:
%tau = call i32 (...) @llvm.tau.i32(i32 %storemerge, i32 %add5)
T
F
if.end17
T
F
if.then34
if.else37
if.then19
if.else22
new_label:
%tau2 = call i32 (...) @llvm.tau.i32(i32 %storemerge, i32 %add5)
if.end27:
%storemerge48 = phi i32 [ %add26, %if.else22 ], [ %mul21, %if.then19 ]
%tau3 = call i32 (...) @llvm.tau.i32(i32 %storemerge48, i32 %add26)
%tau1 = call i32 (...) @llvm.tau.i32(i32 %storemerge, i32 %add3, i32 %add5)
end_label
CFG for 'main' function

Figure 4: CFG after tau insertion

## 10 Replacing uses of phi with taus

We have modified the SSA form renaming algorithm [2] to fit our purpose, and the renaming is done in the following manner:

- Maintain a renaming stack:
  - We maintain map that maps a $\phi$-instruction with its corresponding stack.
  - `Renaming_stack(`$\phi$`) = {`$\phi, \tau_1, \tau_2, \cdots$`}`
  - The top element(back) of stack stores the most recent definition.
- `Search(BB, DTN) Procedure`
  - Replace uses of each $\phi$ with its most recent definition.
  - Initialize stack entry for new $\phi$ encountered if any, by pushing the $\phi$ itself.
  - Push the $\tau$-instructions to stacks of its first argument which will be the $\phi$ corresponding to which it was inserted. (We ensured that the first argument is phi node in argument allocation)
  - Recurse through children of current basic block in the dominator tree. We use `llvm :: DominatorTreeNode` to compute the dominator tree.
  - Pop the $\tau$-instructions(and not $\phi$) pushed onto the stacks corresponding to this basic block.



```
if.end:
 %storemerge = phi i32 [ %add5, %if.else ], [ %add3, %if.then ]
 store i32 %storemerge, i32* %b, align 4, !tbaa !4
 ...
 %mul7 = mul nsw i32 %storemerge, 6
 ...
```

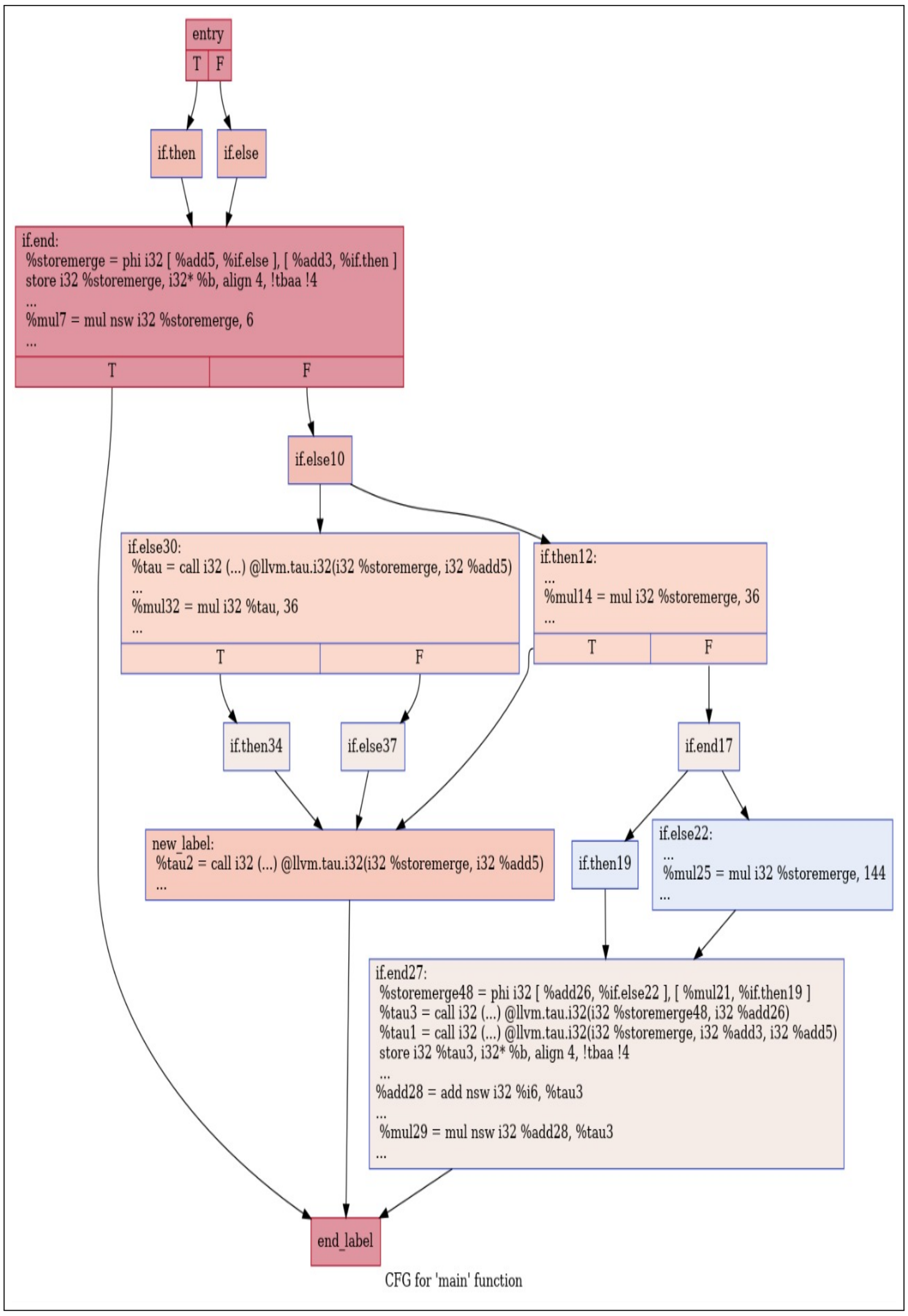
```
if.else30:
 ...
 %mul32 = mul i32 %storemerge, 36
 ...
```

```
if.then12:
 ...
 %mul14 = mul i32 %storemerge, 36
 ...
```

```
if.else22:
 ...
 %mul25 = mul i32 %storemerge, 144
 ...
```

```
if.end27:
 %storemerge48 = phi i32 [ %add26, %if.else22 ], [ %mul21, %if.then19 ]
 ...
 store i32 %storemerge48, i32* %b, align 4, !tbaa !4
 ...
%add28 = add nsw i32 %i6, %storemerge48
...
 %mul29 = mul nsw i32 %add28, %storemerge48
...
```

Figure 5: CFG before replacing uses



```
entry
T | F

if.then

if.else

if.end:
 %storemerge = phi i32 [ %add5, %if.else ], [ %add3, %if.then ]
 store i32 %storemerge, i32* %b, align 4, !tbaa !4
 ...
 %mul7 = mul nsw i32 %storemerge, 6
 ...
T | F

if.else10

if.else30:
 %tau = call i32 (...) @llvm.tau.i32(i32 %storemerge, i32 %add5)
 ...
 %mul32 = mul i32 %tau, 36
 ...
T | F

if.then12:
 ...
 %mul14 = mul i32 %storemerge, 36
 ...
T | F

if.then34

if.else37

if.end17

new_label:
 %tau2 = call i32 (...) @llvm.tau.i32(i32 %storemerge, i32 %add5)
 ...

if.then19

if.else22:
 ...
 %mul25 = mul i32 %storemerge, 144
 ...

if.end27:
 %storemerge48 = phi i32 [ %add26, %if.else22 ], [ %mul21, %if.then19 ]
 %tau3 = call i32 (...) @llvm.tau.i32(i32 %storemerge48, i32 %add26)
 %tau1 = call i32 (...) @llvm.tau.i32(i32 %storemerge, i32 %add3, i32 %add5)
 store i32 %tau3, i32* %b, align 4, !tbaa !4
 ...
 %add28 = add nsw i32 %i6, %tau3
 ...
 %mul29 = mul nsw i32 %add28, %tau3
 ...

end_label
```

Figure 6: CFG after replacing uses

# 11 Tau Destruction Phase

Destroying tau after its use is straightforward. The first argument of the tau is the phi corresponding to which it was inserted. We simply assign the value of first argument to the tau. This is equivalent to removing tau and replacing all uses of tau with orignal phi. The final program is again in its original form ( which was in SSA form ).

## 11.1 Implementation Details

Since LLVM IR does not support replacing RHS of an instructing with a value of different type (`tau:: CallInst` and `first arg:: PhiNode*`, we accomplished the same in indirect manner :

1. Created an alloca instruction and store the original phi

   ```
   Builder:: IRBuilder<>
   origPhi = tau.getOperand(0)
   newTau = Builder.CreateAlloca()
   ```

2. Created a store instruction, to store the value of the phi in the newly created alloca variable.

   ```
   Builder.CreateStore(origPhi,newTau)
   ```

3. Created a load instruction, loaded the value of the alloca variable.

   ```
   tauLoad = Builder.CreateLoad(newTau,newTau.load)
   ```

4. Replaced all the uses of tau with the new loaded value.

   ```
   tau.replaceAllUsesWith(tauLoad)
   ```

5. Deleted the original tau instruction.

   ```
   tau.eraseFromParent()
   ```

The function used above all present in `LLVM`, however there signature might be different as irrelevant arguments have been ommitted for brevity.

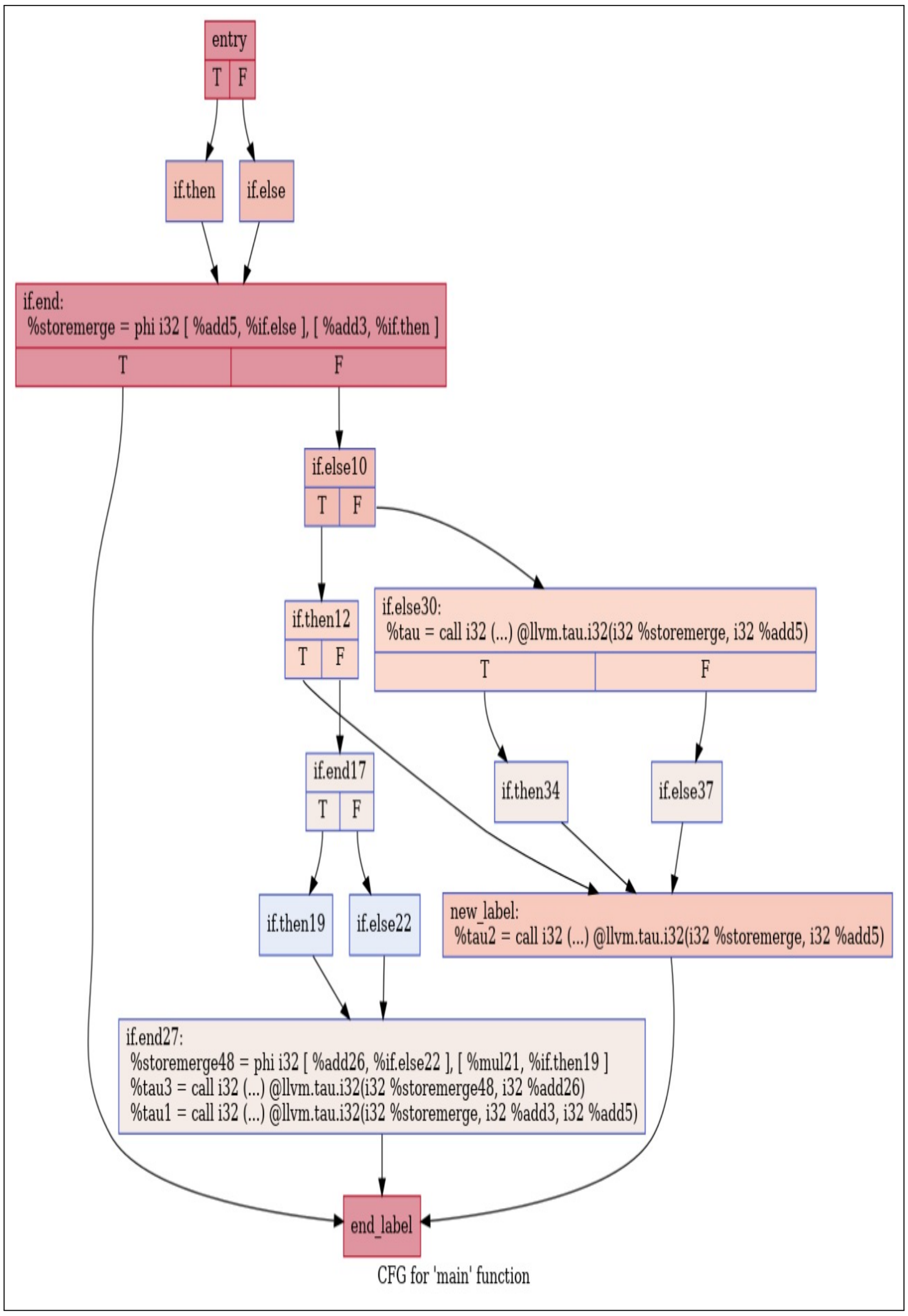
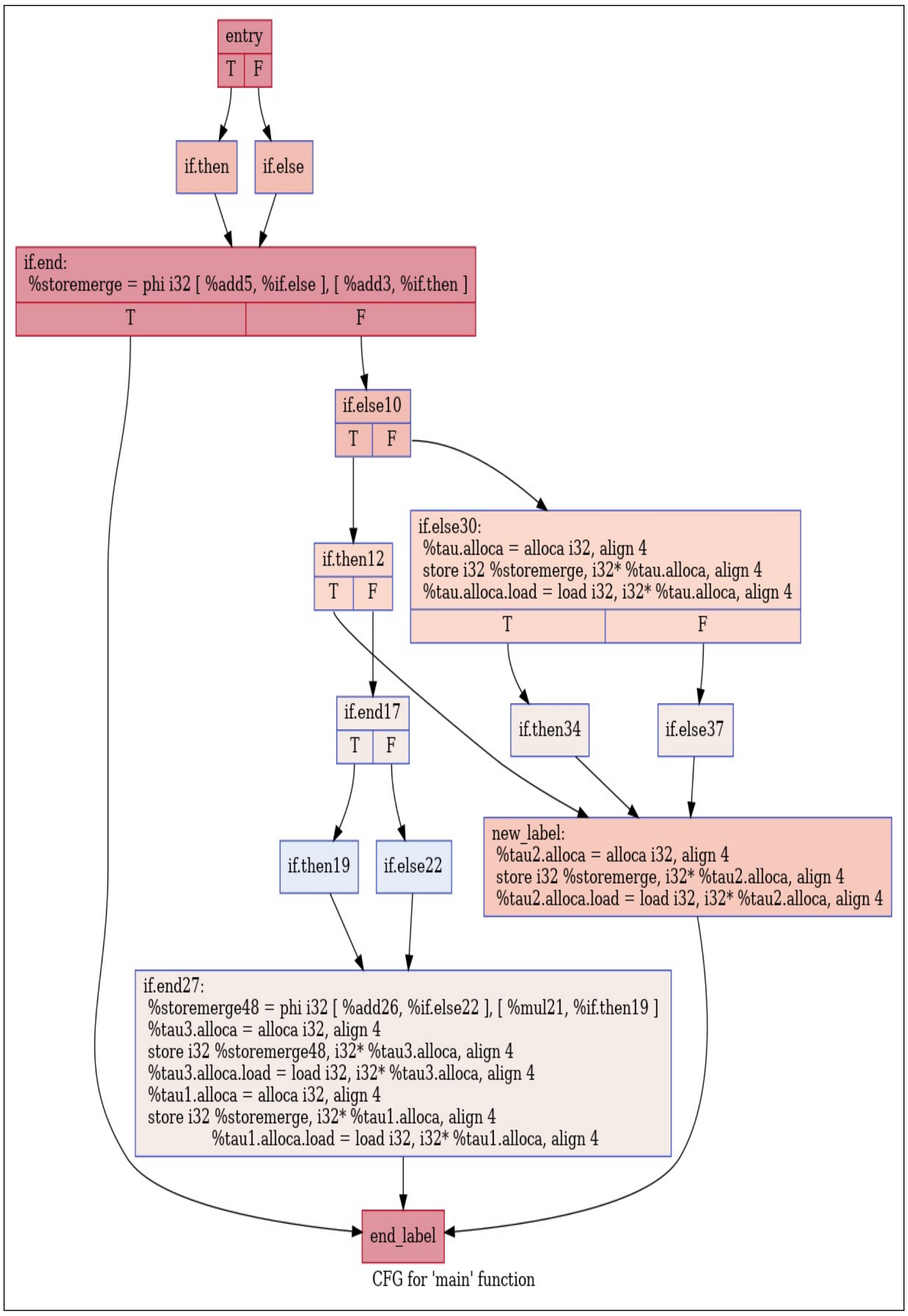

entry
T
F
if.then
if.else
if.end:
%storemerge = phi i32 [ %add5, %if.else ], [ %add3, %if.then ]
T
F
if.else10
T
F
if.then12
T
F
if.else30:
%tau = call i32 (...) @llvm.tau.i32(i32 %storemerge, i32 %add5)
T
F
if.end17
T
F
if.then34
if.else37
if.then19
if.else22
new_label:
%tau2 = call i32 (...) @llvm.tau.i32(i32 %storemerge, i32 %add5)
if.end27:
%storemerge48 = phi i32 [ %add26, %if.else22 ], [ %mul21, %if.then19 ]
%tau3 = call i32 (...) @llvm.tau.i32(i32 %storemerge48, i32 %add26)
%tau1 = call i32 (...) @llvm.tau.i32(i32 %storemerge, i32 %add3, i32 %add5)
end_label
CFG for 'main' function

Figure 7: CFG before tau destruction



```
if.end:
 %storemerge = phi i32 [ %add5, %if.else ], [ %add3, %if.then ]
```

```
if.else30:
 %tau.alloca = alloca i32, align 4
 store i32 %storemerge, i32* %tau.alloca, align 4
 %tau.alloca.load = load i32, i32* %tau.alloca, align 4
```

```
new_label:
 %tau2.alloca = alloca i32, align 4
 store i32 %storemerge, i32* %tau2.alloca, align 4
 %tau2.alloca.load = load i32, i32* %tau2.alloca, align 4
```

```
if.end27:
 %storemerge48 = phi i32 [ %add26, %if.else22 ], [ %mul21, %if.then19 ]
 %tau3.alloca = alloca i32, align 4
 store i32 %storemerge48, i32* %tau3.alloca, align 4
 %tau3.alloca.load = load i32, i32* %tau3.alloca, align 4
 %tau1.alloca = alloca i32, align 4
 store i32 %storemerge, i32* %tau1.alloca, align 4
            %tau1.alloca.load = load i32, i32* %tau1.alloca, align 4
```

Figure 8: CFG after tau destruction

## 12 Future Work

1. Implement a efficient path profiler for testing our pass on large programs.
2. Code Cleaning and pushing the code upstream the LLVM github repository.
3. Leverage the additional path profile information for improving fuzzing techniques.
4. Explore the behaviour of NLP models like Code2Vec on this new predictive IR.

## References

[1] Smriti Jaiswal, Praveen Hegde, and Subhajit Roy. Constructing hpssa over ssa. In *Proceedings of the 20th International Workshop on Software and Compilers for Embedded Systems*, SCOPES '17, page 31–40, New York, NY, USA, 2017. Association for Computing Machinery. ISBN 9781450350396. doi: 10.1145/3078659.3078660. URL https://doi.org/10.1145/3078659.3078660.

[2] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, oct 1991. ISSN 0164-0925. doi: 10.1145/115372.115320. URL https://doi.org/10.1145/115372.115320.