Implementation of HPSSA construction algorithm in the LLVM compiler framework

Abhay Mishra (190017)
Mohammad Muzzammil (190503)

Supervisor: Prof. Subhajit Roy
Mr Awanish Pandey
Mr. Sumit Lahri

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 τ-functions as definition “filters”, similar to the φ-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.

Objective

- The objective of our project was to implement the HPSSA construction algorithm [Smriti Jaiswal, Praveen Hegde, and Subhajit Roy. 2017. Constructing HPSSA over SSA] in the LLVM compiler framework.
- This Primarily involved writing two compiler passes:
  - One for tau insertion and argument allocation.
  - Another for removing the instrumentation.
- For automated testing, writing a rudimentary path profiler.
- Ensuring memory efficiency and robustness of implementation.

Terminology

- (Acyclic) Hot path : Given a set of inputs, and a threshold frequency, the paths that are being visited more frequently than this fixed threshold.
- Buddy Set: A Buddy set is associated with a basic block. A set of paths are buddies if they reach the current basic block through same sequence of basic blocks.
- Caloric connector : A caloric connector is associated with a phi instruction. These are Basic Blocks where both hot and cold arguments of phi reach simultaneously.
- Domination and Dominator Tree: A basic block BB2 is dominated by BB1 if every path reaching BB2 must pass through BB1. In Dominator Tree, Parent immediately dominates all its Childs.
- IR in LLVM: Modules -> { Global Var, Functions , ... } -> Basic Blocks -> Instructions

Implementation Phases

Representing τ in LLVM IR

- Introduced a new intrinsic with following signature:

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

- Takes arguments of any type
- 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.

Getting The Hot Path Profile Information

- A rudimentary path profiler which insert a “counter” function call at the end of basic blocks.
- “counter” function records the id of basic block.
- When the program exits we get a sequence of basic blocks => A path corresponding to these blocks is registered.
- Run the program on multiple inputs and obtain a list of paths
- A path which is being visited more than a “threshold value” is marked as a “Hot Path”.
- Paths are splitted on back edges (Acyclic Path Profiling). Currently works for programs with single functions.

Implementation of Path Profiler

- Processes only a single function of program (main)
- Starts new paths from the start of loops (“loop headers”)
  - Store all the backedges (from loop end to loop header)

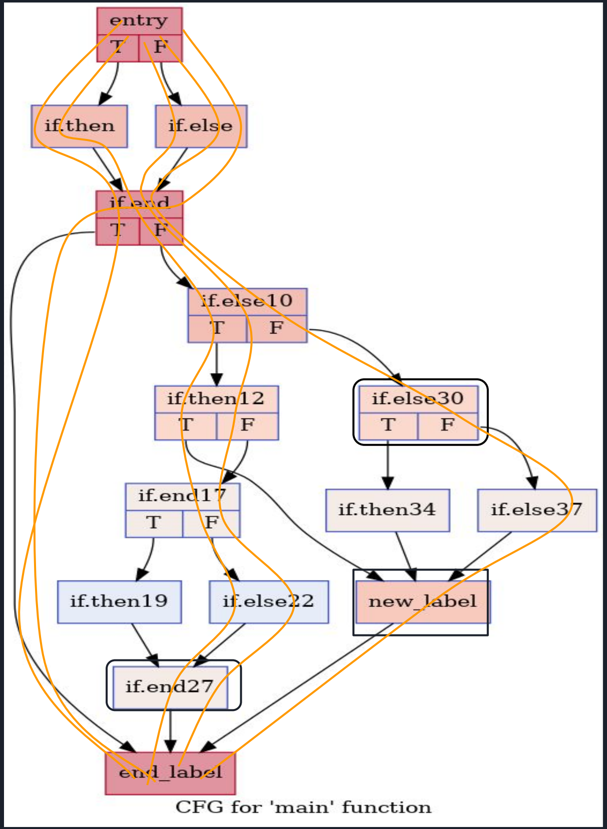
    ```
    llvm :: FindFunctionBackedges()
    ```

  - Create a basic block, inside which a call is made to counter with a special flag (-1)
  - Split the back-edges and insert the basic block in between.

    ```
    splitBB = llvm :: SplitEdge ( from , to )
    ```

  - This ensures that after one iteration a new path will be registered.

An Example CFG...



CFG for 'main' function

- We ran our profiler on an example program to get its hot paths.
- In the CFG on the left, we have marked the hot paths and the corresponding caloric connectors for our example program.
- We will demonstrate our implementation using the same example in the subsequent slides.

Tau Insertion Phase

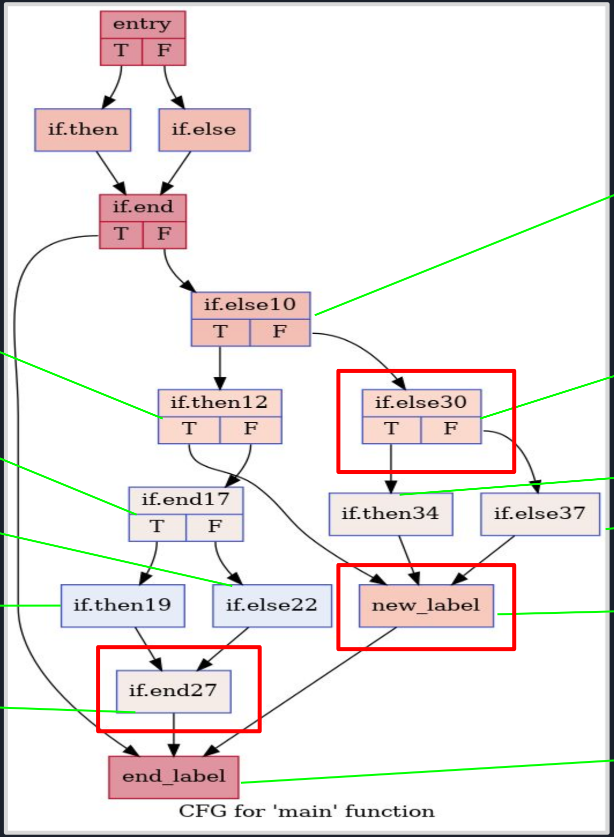
Tau Insertion Pass consists of the following functions:

- `map<BasicBlock *, BitVector> HPSSAPass::getProfileInfo(Function &F)`
  - Returns the acyclic hot path profile information from the profiler.
- `map<BasicBlock *, bool> HPSSAPass::getCaloricConnector(Function &F)`
  - Computes Caloric Connectors from the given hot path information.
- `PreservedAnalyses HPSSAPass::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 HPSSAPass::Search(BasicBlock &BB, DomTreeNode &DTN)`
  - Replaces the uses of the phi with its appropriate tau counterpart.

Computing Caloric Connectors

- Traversed the CFG in topological order.
- Iterated over the predecessors of the current basic block.
- Marked the basic blocks hot and cold, depending on the status of predecessor and current **buddy sets**.
- Used std::map<BasicBlock*, bool> to map the basic block pointers to true if the basic block is a caloric connector, vice-versa.
- To make the implementation **memory efficient,** llvm::BitVector was used to store and manipulate "buddy-paths", with **each index of BitVector denoting a hot path.**
- Using llvm::BitVector simplified the implementation of basic set operations
    - ANDing two BitVectors. (ORing two BitVectors) and intersectionBuddy Set is implemented as a **std::vector of BitVectors.**

Marking Caloric Connectors



[{1 0 1 1 0},{0 1 0 0 1}]

[{0 0 0 1 0}{0 1 0 0 0}]

[{1 0 0 0 0}] Hot Defn Missing

[{0 0 0 1 0}, {0 1 0 0 0}]

[ ] Cold

[{0 0 0 1 0}{0 1 0 0 0}]

[{ 1 0 0 0 0 }]

Cold [ ]

[{ 1 0 0 0 0 }] Cold Defn Present

[{0 0 0 1 0}, {0 1 0 0 0}]

Cold Defn Present

[{1 0 0 0 0}, {0 0 0 1 0}, {0 1 0 0 0}, {0 0 1 0 0}, {0 0 0 0 1}]

Inserting Tau Functions

- We traverse the basic blocks in topological order (using `Function :: RPOT( )`) and then iterate over the phi instructions of each block .
- For a given phi instruction, we need to traverse all the basic blocks reachable from current basic block through a **Hot Path.**
- We traverse the basic blocks again in topological order. We stop going deep in a path whenever we hit a block which dominates current block (or we reach the dominance frontier).
- A tau is inserted if its a caloric connector and a tau is not inserted for current phi already. (Information stored in maps **isCaloric** and **isInserted**).

Allocating Arguments To Tau Functions

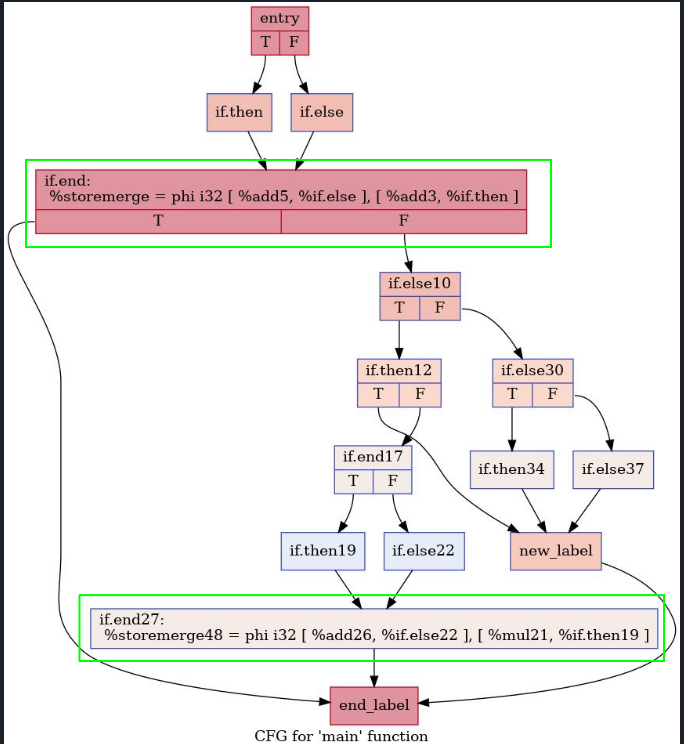
- Arguments of tau are a subset of arguments of corresponding phi functions that reach through hot path at the current location.
- A structure named **defAccumulator** stores information about hot definitions.
    - `defAccumulator( phi, currBB ) = { < hotDef, hotPaths > | hotDef is an argument of phi and reaches hot to currBB through hotPaths }`
- defAccumulator Serve two purpose:
    - Provide arguments for taus if needed.
    - Store hot paths through which the definition reach thus allowing us to infer which definition one should pass to successors.

Allocating Arguments To Tau Functions (Implementation)

For a given phi

- Traverse the basic blocks in topological order (using Function::RPOT()).
- If a basic block is a caloric connector and no tau has been inserted.
- Check if the current basic block has some hot defn stored in defAccumulator.
    - If it doesn’t have any hot defns then continue to next basic block.
- If all the conditions are met insert a tau with its first argument as the corresponding phi , followed by hot defn stored in defAccumulator. Having phi as the **first argument** will be helpful during tau destruction.
- Iterate through all successors
    - If the successor has a common hot path ,which can be efficiently checked by ANDing the two bitvectors. Pass the corresponding hot defs flowing through the common path.
    - Otherwise Ignore.
    - Repeat the procedure on successors (llvm:: successors( )) having common path until we hit the dominance frontier. We use the llvm::DominatorTreeAnalysis Class to get dominance information.

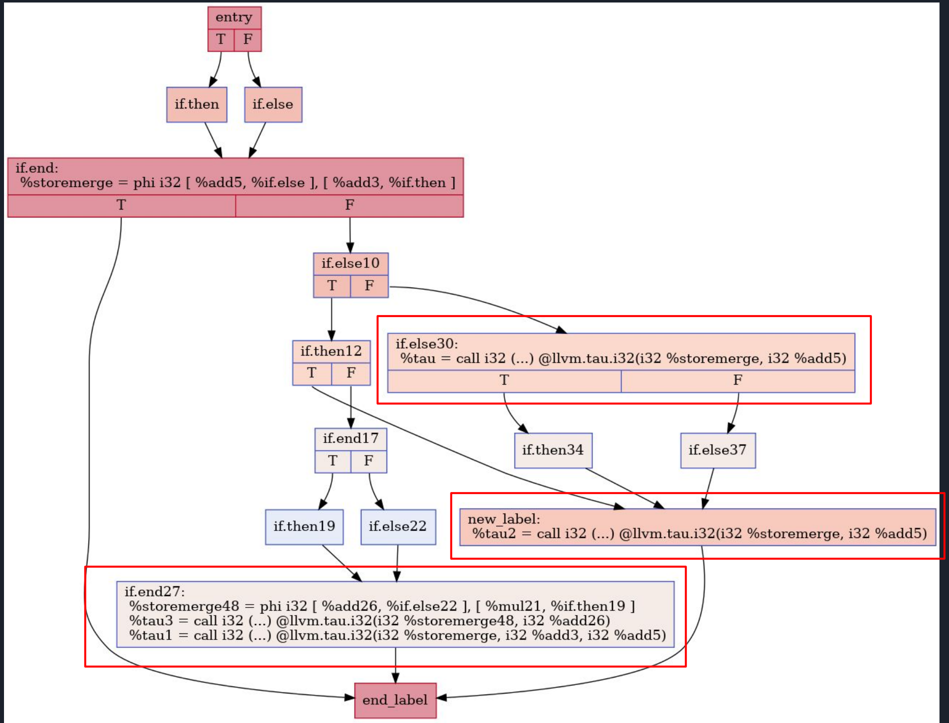
Changes In IR after the Insertion Pass



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

```
if.end27:
 %storemerge48 = phi i32 [ %add26, %if.else22 ], [ %mul21, %if.then19 ]
```

CFG for 'main' function



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

```
if.else30:
 %tau = call i32 (...) @llvm.tau.i32(i32 %storemerge, i32 %add5)
```

```
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)
```

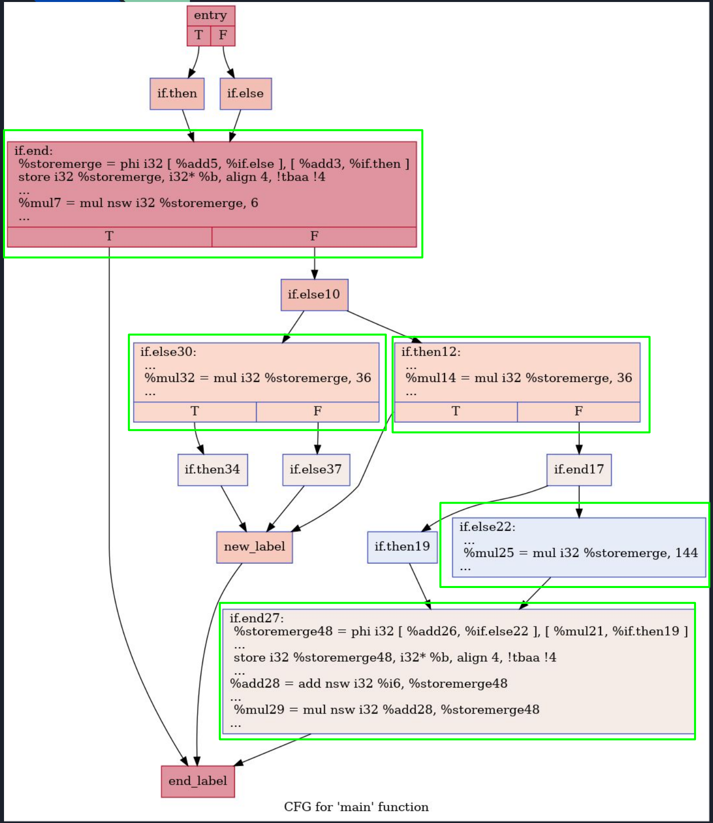
CFG for 'main' function

Replacing uses of phi with taus

We have modified the SSA form renaming algorithm [Efficiently computing static single assignment form and the control dependence graph, Cytron et al.] to fit our purpose, and the renaming is done in the following manner:

- Maintain a renaming stack :
    - Renaming_stack (phi) = { tau1 , tau2, …. }
    - The top element (back) stores the most recent definition.
- Search (BB, DTN)
    - 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 taus 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 childs in Dominator Tree
        - DFS on Dominator Tree using llvm :: DominatorTreeNode facility
    - Pop the taus (and not phi) pushed onto the stacks corresponding to this basic block.

IR after Tau Insertion and Renaming



```
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
 ...
```

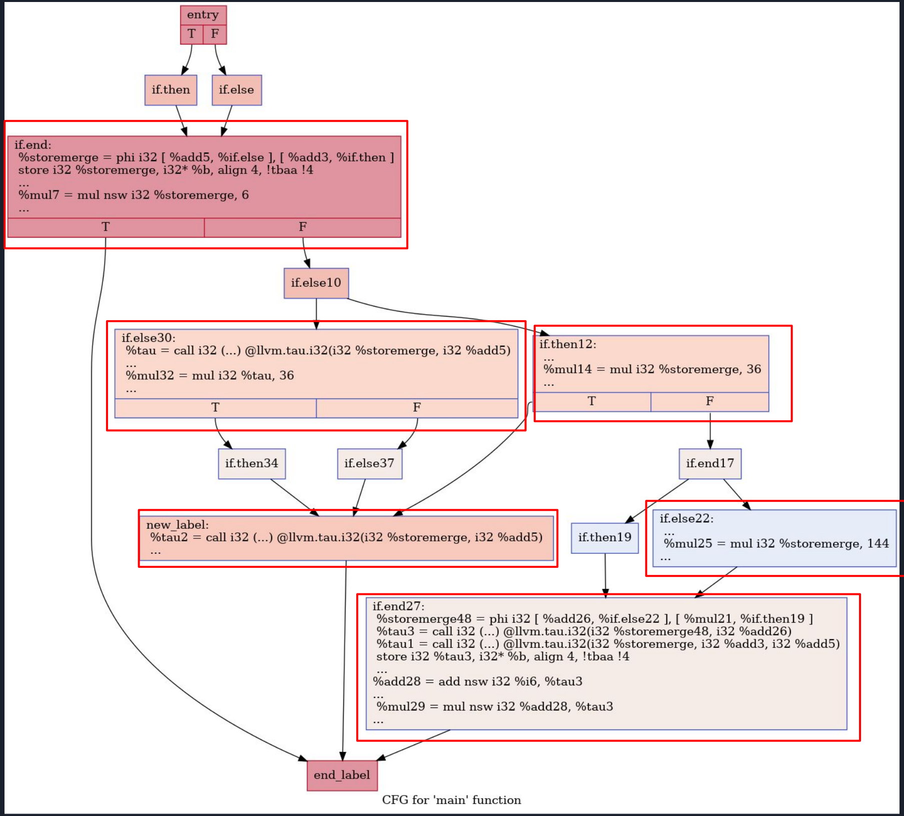
```
if.else30:
 ...
 %mul32 = mul i32 %storemerge, 36
 ...
```

```
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
 ...
```

CFG for 'main' function



```
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:
 %tau = call i32 (...) @llvm.tau.i32(i32 %storemerge, i32 %add5)
 ...
 %mul32 = mul i32 %tau, 36
 ...
```

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

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

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

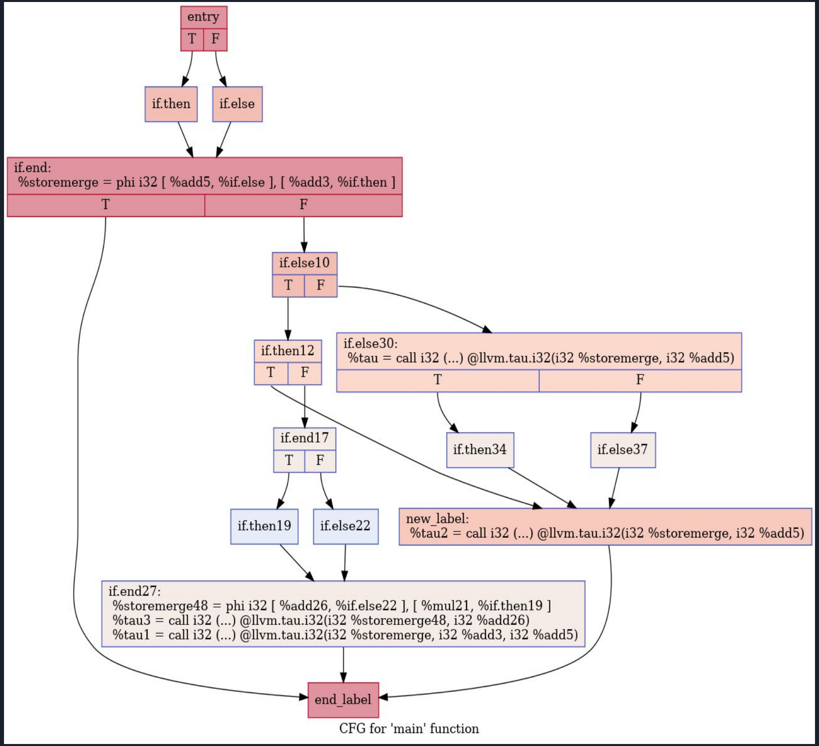
```
if.end27:
 %storemerge48 = phi i32 [ %add26, %if.else22 ], [ %mul21, %if.then19 ]
 %tau3 = call i32 (...) @llvm.tau.i32(i32 %storemerge48, i32 %add26)
 %tau1 = call i32 (...) @llvm.tau.i32(i32 %storemerge, i32 %add3, i32 %add5)
 store i32 %tau3, i32* %b, align 4, !tbaa !4
 ...
%add28 = add nsw i32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

CFG for 'main' function

Tau Destruction Phase

- Destroying tau after its use is straightforward, since the first argument of the tau is the phi for which it was inserted, we can assign tau the value of this phi.
- Since LLVM IR does not support assignment instruction, this was done in replace steps:
    - Created an alloca instruction.
    - Created a store instruction, to store the value of the phi in the newly created alloca variable.
    - Created a load instruction, loaded the value of the alloca variable.
    - Replaced all the uses of tau with the new loaded value.
    - Deleted the original tau instruction.
- Note that we could not have simply replaced all uses of tau with phi, since we only replace those uses of phi that are dominated by the given tau.

IR After Tau Destruction Pass



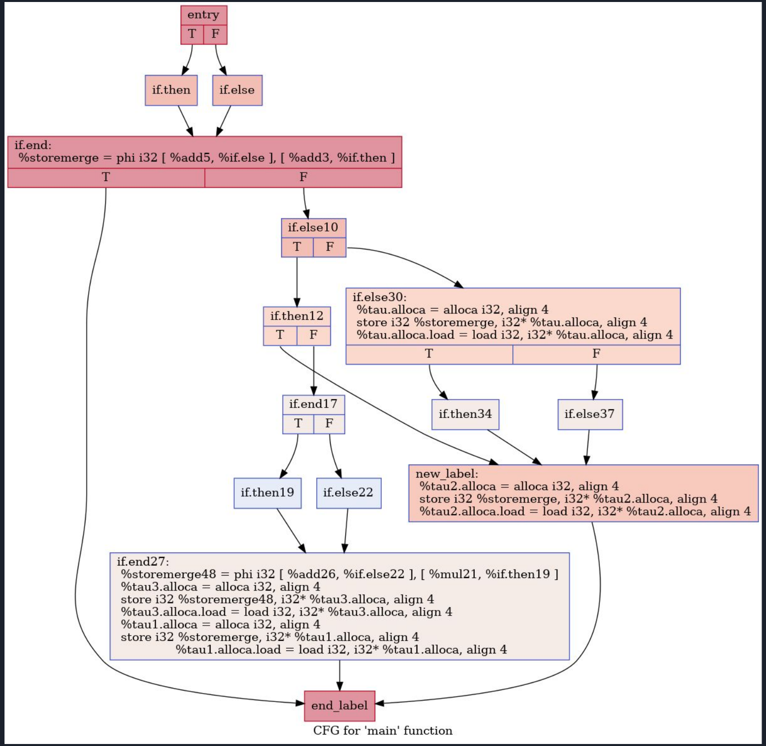
```
if.end:
 %storemerge = phi i32 [ %add5, %if.else ], [ %add3, %if.then ]

if.else30:
 %tau = call i32 (...) @llvm.tau.i32(i32 %storemerge, i32 %add5)

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)
```

CFG for 'main' function



```
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
```

CFG for 'main' function

Future Work

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