LLVM & HPSSA

Hot Path SSA Form in LLVM

Presented By Abhay[1] & Muzzammil[1]

[1]IIT Kanpur
PRAISE Group

Dr. Subhajit Roy, Dr. Awanish Pandey, Mr. Sumit Lahiri

What we modified in LLVM Source?

- New `llvm::intrinsic` signature, `"llvm.tau"` to support addition and removal of τ-functions to the LLVM SSA IR representation.

```
+ //===---------- intrinsic for tau ----------------=====//
+ def int_tau : DefaultAttrsIntrinsic<[llvm_any_ty],
+                                      [llvm_vararg_ty],
+                                      []>;
```

What we modified in LLVM Source?

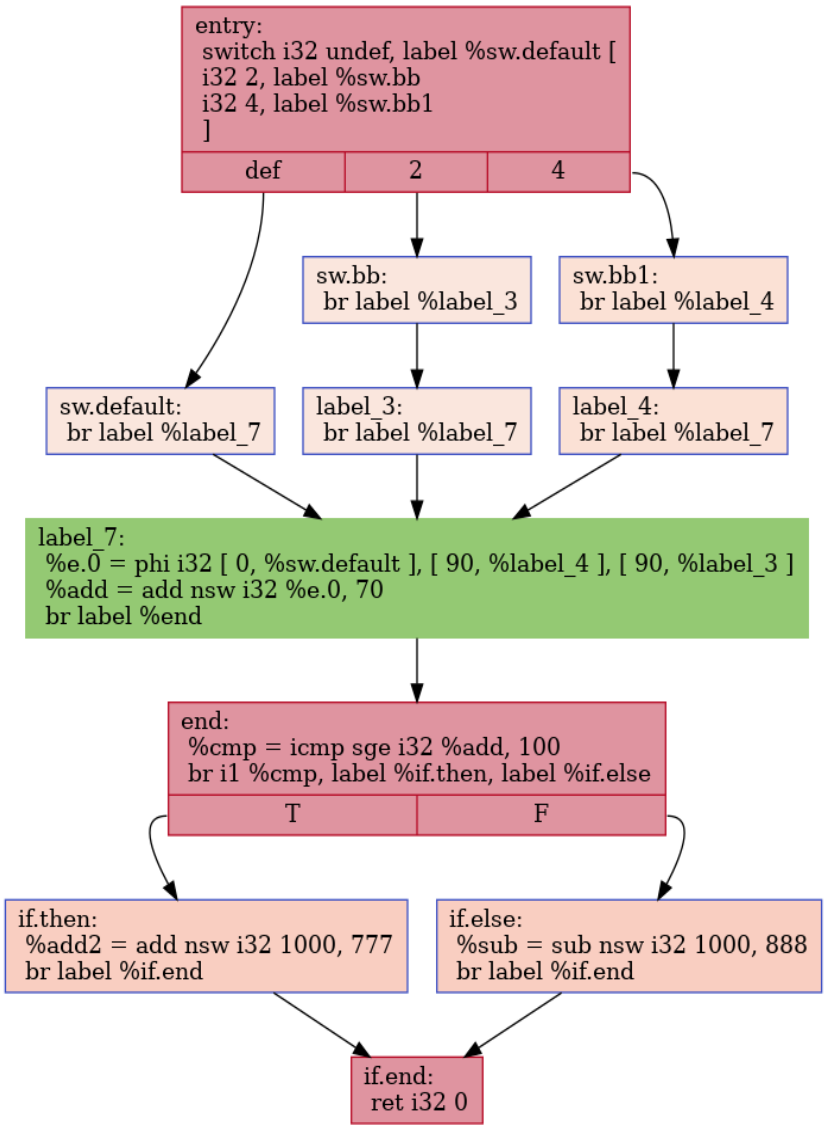
- Modified `Verifier::verifyDominatesUse()` function since we don't want our intrinsic to interfere with `dominators` computation.

```
1  + //========== Changes for tau.intrinsic ================//
2  void Verifier::verifyDominatesUse(Instruction &I, unsigned i) {
3    Instruction *Op = cast<Instruction>(I.getOperand(i));
4    +  if (CallInst *CI = dyn_cast<CallInst>(&I)) {
5    +  Function *CallFunction = CI->getCalledFunction();
6    +  if (CallFunction != NULL && CallFunction->getIntrinsicID()==
7    +    Function::lookupIntrinsicID("llvm.tau")) {
8    +      return;
9    +    }
10   +  }
```

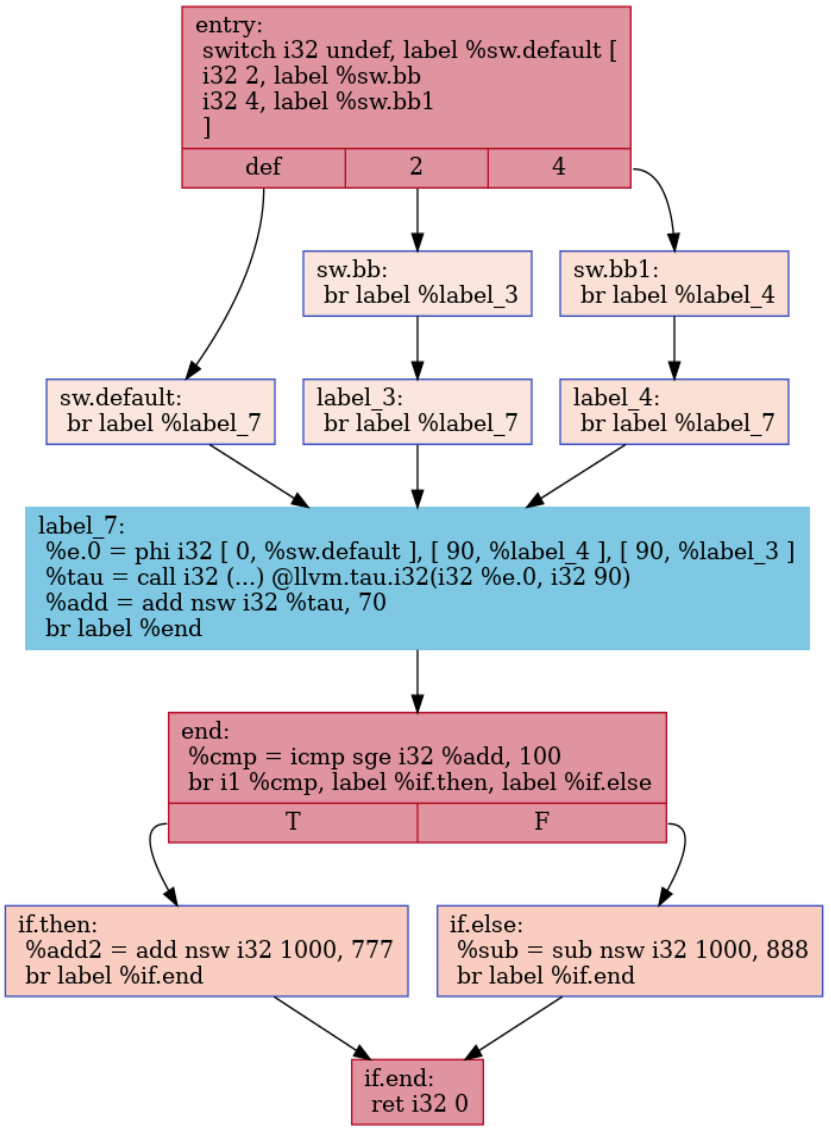
HPSSAPass : Overview

- `class HPSSAPass : public PassInfoMixin<HPSSAPass>`
  - Implemented `llvm::HPSSAPass` pass using the new LLVM Pass Manager.
  - Function `HPSSAPass::run(Function \&F, ...)` runs over a `llvm::Function` and inserts `"llvm.tau"` intrinsic calls with speculative and safe arguments at strategic positions in the LLVM IR and handles argument allocation for `"llvm.tau"` intrinsic calls as described in the previous slides.
- Key HPSSA Data Structures :
  - Hot Path Set using `llvm::BitVector` for maintaining hot paths in the program.
  - Definition Accumalator, `defAccumulator(op, currBB)` function. The argument "op" is a phi argument that reaches basic-block "currBB" via hot path .
  - A stack of map values `std::map<Value*,Value*>` to store the most "recent" tau definition encountered so far corresponding for a tau variable used later in variable renaming.

HPSSA Transformation



CFG for 'main' function before HPSSA Pass



CFG for 'main' function after HPSSA Pass

HPSSAPass : Main Pass

- `HPSSAPass::run(Function \&F, FunctionAnalysisManager \&AM)`
  - Invoke `HPSSAPass::getProfileInfo(Function \&F)` function to get a compact representation of all the hot paths in the program and then call `HPSSAPass::getCaloricConnector(Function \&F)` to get all the caloric connectors from the hot path information. This is a precursor to finding strategic positions to place `"llvm.tau"` intrinsic calls in the LLVM IR.
  - Runs over each basic block in the function "F" in topological order using iterator returned from `llvm::Function::RPOT()` call.
  - Uses the `llvm::dominates()` function from `llvm::DominatorTreeAnalysis` to check for dominance frontier while processing the child nodes of the current basic block. This step is a part of correctly placing `"llvm.tau"` intrinsic calls in the LLVM IR.
  - Uses the renaming stack and `HPSSAPass::Search(BasicBlock \&BB, DomTreeNode \&DTN)` function to search and replace all use of PHI result operand with that returned by the `"llvm.tau"` intrinsic call.

HPSSAPass : Destruction Pass

- Out of HPSSA Form.
  - A seperate pass using the new LLVM Pass Manager.
    `class TDSTRPass : public PassInfoMixin<TDSTRPass>`
  - Using `TDSTRPass::run(Function \&F, ...)`, we replace all use of existing tau operands with first argument of `"llvm.tau"` intrinsic (corresponds to the safe argument) and remove the `"llvm.tau"` intrinsic calll from the LLVM IR.
  - The LLVM IR becomes identical to what it was before running the HPSSA Pass.

HPSSAPass : Usage [It is easy!]

- Include `llvm::HPSSAPass` header file.
- Load shared object using opt tool. `opt -load HPSSA.cpp.so ...`

```
1  #include <HPSSA.h> // import the header.
2  class MyExamplePass : public PassInfoMixin<MyExamplePass> {
3    public: PreservedAnalyses run(Function &F,
4      FunctionAnalysisManager &AM);
5  };
6  ...
7  PreservedAnalyses MyExamplePass::run(Function &F,
8    FunctionAnalysisManager &AM) {
9    if (F.getName() != "main")
10     return PreservedAnalyses::all();
11   HPSSAPass hpssaUtil; // Make a HPSSAPass Object.
12   hpssaUtil.run(F, AM);  // Call the HPSSAPass::run() function.
13   std::vector<Instruction *> TauInsts // Calling utility function.
14     = hpssaUtil.getAllTauInstrunctions(F);
15   ...
16 }
17 /// [output] Total Tau Instructions : 7
```

New Additions to SCCP Pass

- Modified the existing SCCP Pass to add in `SCCPInstVisitor::visitTauNode()` function similar to `SCCPInstVisitor::visitPHINode()`, which handles the special `"llvm.tau"` intrinsic instructions added for τ-functions.
- Added a new lattice element type `"spec_constant"` in `ValueLattice` class supporting operations on speculative constants.
- Added new functions in the `SCCPInstVisitor` and `SCCPSolver` class to handle operations on speculative constants using `markSpeculativeConstant()` function.

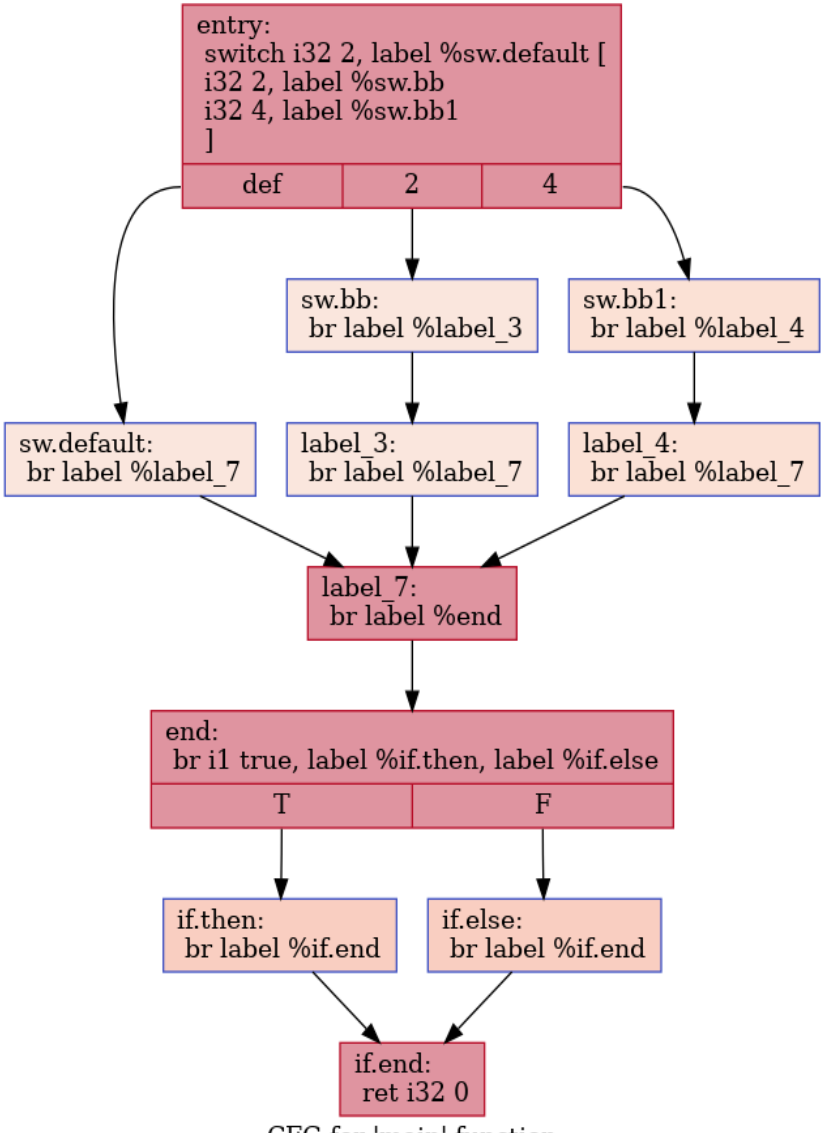
Further Modifications

- Modified the `SCCPInstVisitor::mergeIn()` function to handle lattice "meet" operation for the new speculative constants introduced.
- Since we added the τ-functions as an `"llvm.tau"` intrinsic which is essentially an `llvm:CallInst` type, we modified all appropriate visit and marking functions in `SCCPInstVisitor`, `SCCPSolver` and `SCCPPass` to handle this case separately by calling `visitTauNode()`.
- Modified utility functions in `SCCPInstVisitor` and `SCCPSolver` class to print marking of speculative constants and related operations for debugging purpose.

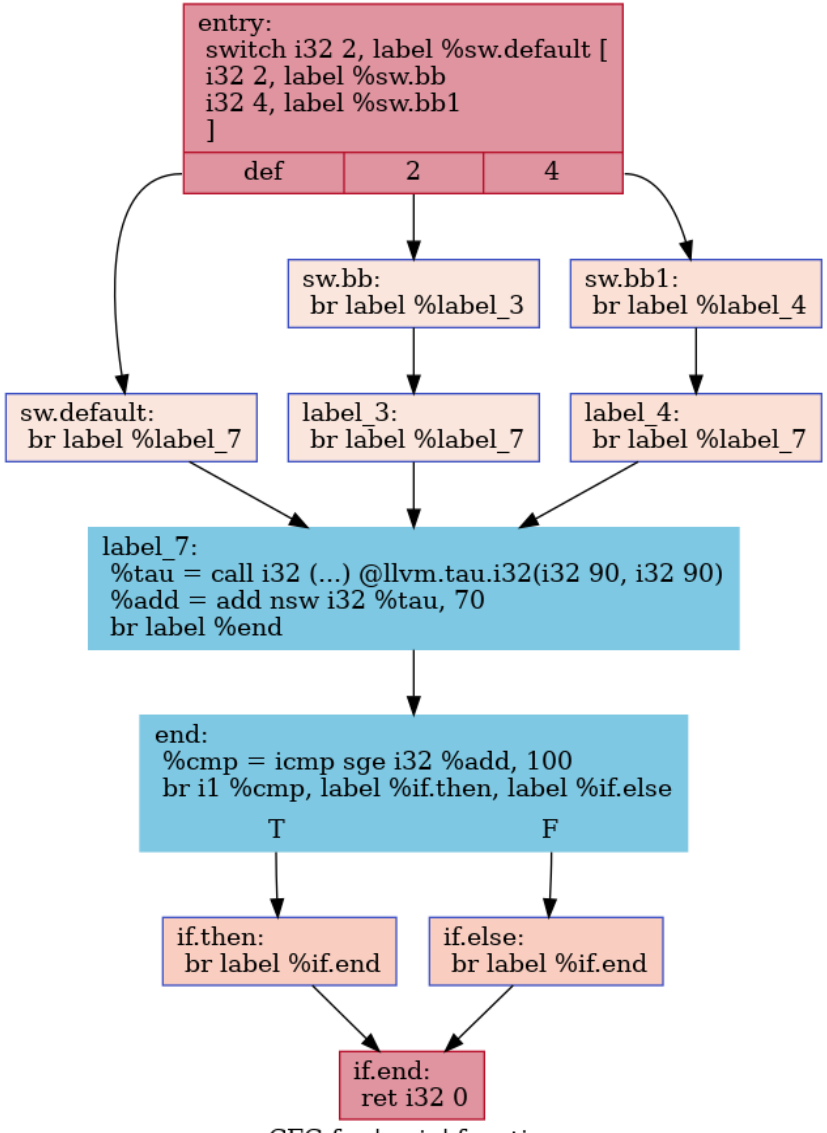
```
1  ... // logs
2  [BBWorkList] Visiting LLVM Instrinsic : llvm.tau (call)
3    Visiting Tau Instruction
4      Speculative Operand : , speculative constant
5      Speculative Operand : llvm.tau.i32, speculative constant
6    Merged speculative constant into   %tau = call i32 (...)
7      @llvm.tau.i32(i32 %e.0, i32 90) : speculative constant
8    ValueLattice (TauState) : speculative constant
9  ...
```

```
1  // Example for SSCCP Pass
2  int main() {
3    int a = 1000, z, c, e = 0;
4    switch(c) {
5      case 2 : goto label_3; break;
6      case 4 : goto label_4; break;
7      default : goto label_7; }
8    label_3:
9      e = 90;
10     goto label_7;
11   label_4:
12     e = 100 - 10;
13     goto label_7;
14   label_7:
15     e = e + 70;  // e in rhs is 90.
16     goto end;
17   end:
18     if (e >= 100) {  // e is greater than 100 always
19       a = a + 777;
20     } else {
21       a = a - 888;
22     } return 0; }
```

SSCCP with an Example



CFG for 'main' function



CFG for 'main' function