The Hot Path SSA Form in LLVM

Algorithms & Applications

IIT Kanpur
Dept. Of Computer Science & Engineering

[1]IIT Kanpur
PRAISE Group

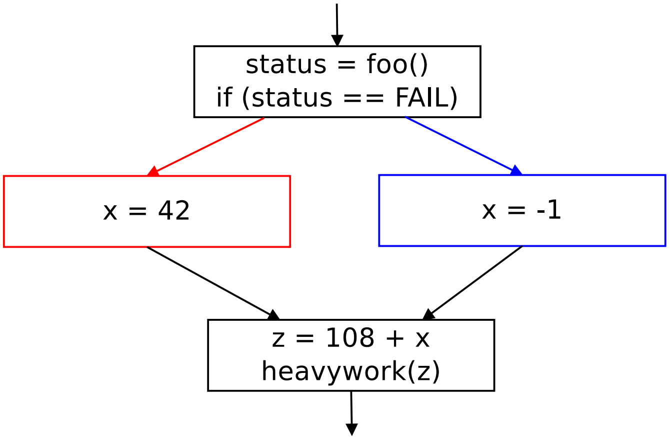
Mohd. Muzzammil, Abhay Kumar, Sumit Lahiri
Dr. Awanish Pandey, Dr. Subhajit Roy

Presentation Outline : Section 1

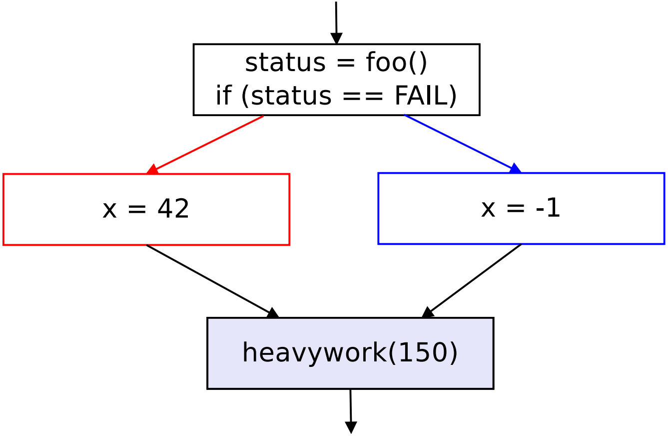
Doing Speculative Analysis is Hard !

Importance of speculative analysis. It is hard to do. Whole research papers dedicated to single speculative analysis. But with HPSSA form it's a breeze.

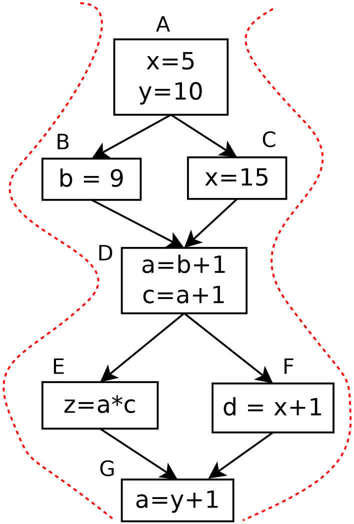
A case for profile-guided optimizations (PGO)



A case for profile-guided optimizations (PGO)



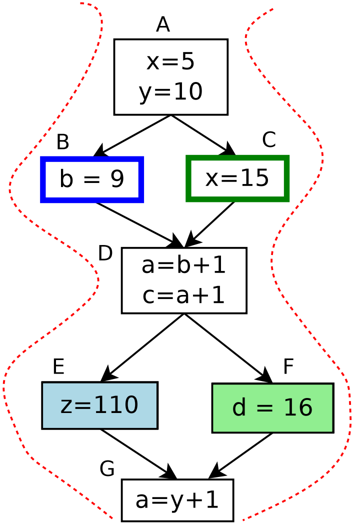
Profile-guided analysis on paths



Summary

- Profile-guided analysis across paths is stronger—can capture correlations between control-flow of basic-blocks
- Collecting path-profiles seems challenging—requires “recording” of a sequence of basic-blocks

Profile-guided analysis on paths



Summary

- Profile-guided analysis across paths is stronger—can capture correlations between control-flow of basic-blocks
- Collecting path-profiles seems challenging—requires "recording" of a sequence of basic-blocks

Profiling acyclic paths

Ball-Larus Acyclic Profiling [Ball & Larus, MICRO'96]

- Core idea: assign an identifier to each path, that can be calculated efficiently at runtime
- Record frequencies against these identifiers (instead of a sequence of node identifiers)

Profiling acyclic paths

Ball-Larus Acyclic Profiling [Ball & Larus, MICRO'96]

- Core idea: assign an identifier to each path, that can be calculated efficiently at runtime
- Record frequencies against these identifiers (instead of a sequence of node identifiers)

Capturing still longer paths (k-iteration paths)

- Allows capturing correlations across loop iterations [Roy & Srikant, CGO'09]; a generalization of the Ball-Larus algorithm
- Subsequent work by other groups [D'Elia & Demetrescu, OOPSLA'13]; uses a prefix forest to record BL paths

Profile-guided analyses

- **Code understanding**
  - Can expose refactoring opportunities

Profile-guided analyses

- **Code understanding**
  - Can expose refactoring opportunities
- **Program testing and verification**
  - Data-driven synthesis of invariants
  - Guided testing for low frequency paths

Profile-guided analyses

- **Code understanding**
  - Can expose refactoring opportunities
- **Program testing and verification**
  - Data-driven synthesis of invariants
  - Guided testing for low frequency paths
- **Profile-guided optimizations**

Profile-guided analyses to optimizations

- **Speculation**: *check and retry*
  - eg. value speculation
  - Can impact code size (significant impact w/o speculation support in hardware)

Profile-guided analyses to optimizations

- **Speculation**: *check and retry*
  - eg. value speculation
  - Can impact code size (significant impact w/o speculation support in hardware)
- **Compensation code**
  - eg. superblock scheduling
  - Can impact code size

Profile-guided analyses to optimizations

- **Speculation**: *check and retry*
  - eg. value speculation
  - Can impact code size (significant impact w/o speculation support in hardware)
- **Compensation code**
  - eg. superblock scheduling
  - Can impact code size
- **Error resilient modules**
  - modules for statistical summarization on samples, generate data for ML models
  - No impact on code size

Profile-guided analyses to optimizations

- **Speculation**: *check and retry*
  - eg. value speculation
  - Can impact code size (significant impact w/o speculation support in hardware)
- **Compensation code**
  - eg. superblock scheduling
  - Can impact code size
- **Error resilient modules**
  - modules for statistical summarization on samples, generate data for ML models
  - No impact on code size
- **Optimizations that don't impact correctness**
  - eg. register allocation
  - No impact on code size

Why is path-profile-guided analysis hard?

- There has been enough interest in path-profile-guided analysis and optimizations....
- ...however, designing path-profile-guided variants of traditional optimizations remained hard
- ...hard enough to justify *publications per optimization*
  - Gupta, Benson, Fang. Path profile guided partial dead code elimination using predication. PACT '97.
  - Gupta, Benson, Fang. Path profile guided partial redundancy elimination using speculation. ICCL '98.
  - ...

Profile Guided Optimization is easy with HPSSA Form

```
visitTauNode() {
    ...
  SpecValueLatticeElement TauState = getValueState(&Tau),
    beta = getValueState(Tau.getOperand(1)),
    x0 = getValueState(Tau.getOperand(0));
  TauState.markUnknown();
  for (unsigned i = 1, e = (Tau.getNumOperands() - 1); i != e; ++i) {
    SpecValueLatticeElement IV = getValueState(Tau.getOperand(i));
    beta.mergeIn(IV);
    NumActiveIncoming++;
    if (beta.isOverdefined())
      break;
  }

  if (x0.isConstantRange())
    TauState.markConstantRange(x0.getConstantRange());
  if (x0.isConstant())
      TauState.markConstant(x0.getConstant());
  if (beta.isConstant() && x0.isConstant()
      && (beta.getConstant() == x0.getConstant()))
      TauState.markSpeculativeConstant(x0.getConstant());
  if (beta.isConstantRange() && x0.isConstantRange()
      && (beta.getConstantRange() == x0.getConstantRange()))
      TauState.markSpeculativeConstantRange(x0.getConstantRange());
  if (beta.isOverdefined() || x0.isOverdefined())
      TauState.markOverdefined();
  ...
}
```

Using HPSSAPass [It is easy!]

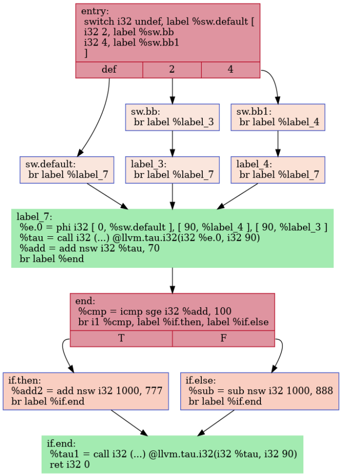
- Include `llvm::HPSSAPass` header file.
- Load shared object using opt tool. `opt -load HPSSA.cpp.so ...`

```
#include <HPSSA.h> // import the header.

class MyExamplePass : public PassInfoMixin<MyExamplePass> {
  public: PreservedAnalyses run(Function &F,
  FunctionAnalysisManager &AM);
};
...

PreservedAnalyses MyExamplePass::run(Function &F,
FunctionAnalysisManager &AM) {
  if (F.getName() != "main")
  return PreservedAnalyses::all();

  HPSSAPass hpssaUtil; // Make a HPSSAPass Object.
  hpssaUtil.run(F, AM);  // Call the HPSSAPass::run() function.

  std::vector<Instruction *> TauInsts
  = hpssaUtil.getAllTauInstrunctions(F); // Calling HPSSA utility function.

  std::cout << "\t\tTotal Tau Instructions : " << TauInsts.size() << "\n";
  ...
}

/// [output] Total Tau Instructions : 7
```

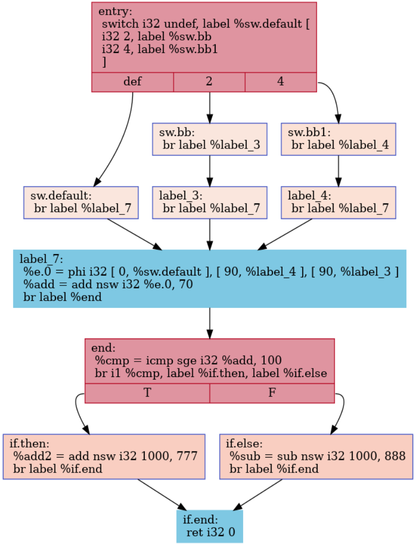
Speculative Pass using HPSSA : SSCCP

We run the speculative SCCP on the example below. Mark the different types of variables. Make the CFG look pretty and add hot path. .

```
int main() {
  int a = 1000, z, c, e = 0;
  std::cin >> c;
  switch(c) {
    case 2 : goto label_3; break;
    case 4 : goto label_4; break;
    default : goto label_7;
  }
  label_3:
    e = 90;
    goto label_7;
  label_4:
    e = 100 - 10;
    goto label_7;
  label_7:
    // e in rhs is 90.
    e = e + 70;
    goto end;
  end:
    // e is greater than 100 always
    if (e >= 100) {
      a = a + 777;
    } else {
      a = a - 888;
    }
  return 0;
```

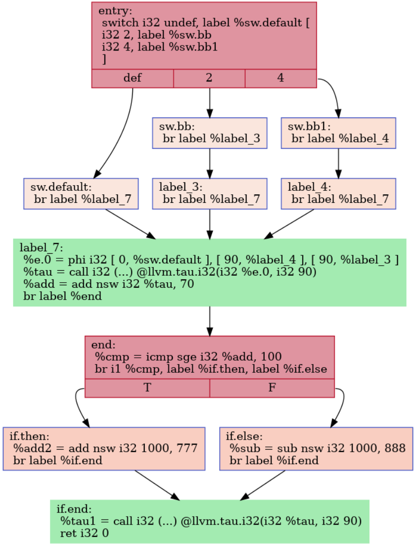


Standard SCCP Pass running on example.



CFG for 'main' function before HPSSA Pass

Speculative SCCP pass running on Example.



CFG for 'main' function after HPSSA Pass

Presentation Outline : Section 2

The Hot Path SSA Form (HPSSA)

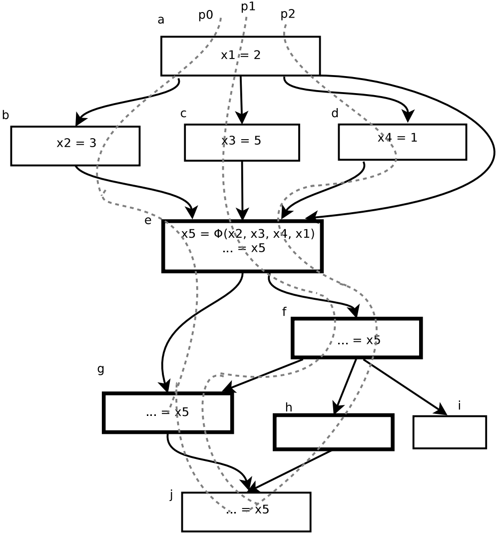
Semantics of a ϕ-function

$y = \phi(x_1, x_2, \ldots, x_n)$

Semantics of a τ-function

$$\tau(x_0, x_1, x_2, \ldots, x_n) = \begin{cases} x_0 & \text{safe interp.} \\ \phi(x_1, x_2, \ldots, x_n) & \text{speculative interp.} \end{cases}$$

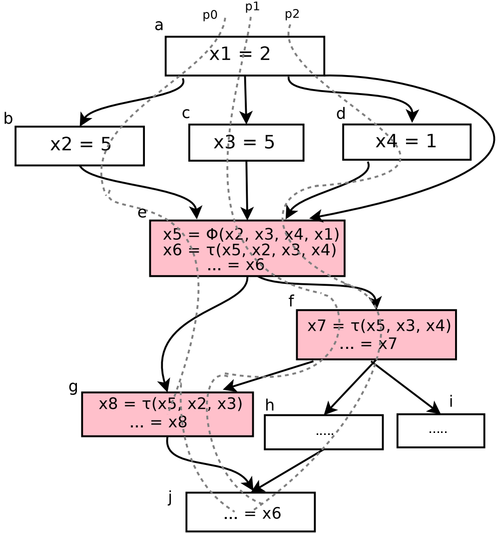
The SSA form



No frequent path carrying:

- def $x_2 = 3$ to use at block **f**
- def $x_4 = 1$ to use at block **g**
- def $x_1 = 2$ to either **f** or **g**

The Hot Path SSA Form



No frequent path carrying:

- def $x_2 = 3$ to use at block **f**
- def $x_4 = 1$ to use at block **g**

The Hot Path SSA Form

Properties

If a program is in the Hot Path SSA form, then,

- each use of a variable is reachable by a single definition; [SSA-like form]

The Hot Path SSA Form

Properties

If a program is in the Hot Path SSA form, then,

- each use of a variable is reachable by a single definition; [SSA-like form]
- **safe interpretation**: [supports traditional analysis]
    - each use of a variable is reachable by the *meet-over-all-paths* reaching definition chains;

The Hot Path SSA Form

Properties

If a program is in the Hot Path SSA form, then,

- each use of a variable is reachable by a single definition; [SSA-like form]
- **safe interpretation**: [supports traditional analysis]
  - each use of a variable is reachable by the *meet-over-all-paths* reaching definition chains;
- **speculative interpretation**: [supports profile-guided analysis]
  - each use of a variable in a basic-block is reachable by the *meet-over-frequent-paths* reaching definitions. [a]

---

[a] or the meet-over-all-paths reaching definition chains, if the use is not reachable from any meet-over-hot-paths reaching definition chain

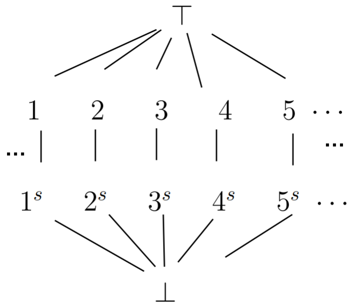
Speculative Sparse Conditional Constant Propagation (SSCCP)

- Introduce new speculative values $\{\ldots, 1^s, 2^s, \ldots\} \in C^S$
- Operation with *speculative* values result in *speculative* results (with same semantics as base operator)

$$\alpha^s \ \langle op \rangle \ \beta = (\alpha \ \langle op \rangle \ \beta)^s$$

- Transfer function for τ-functions ($\beta = x_1 \sqcup x_2 \sqcup \ldots$)

$$\tau(x_0, x_1, \ldots, x_n) \sqcup \begin{cases} x_0 \sqcup \top & \text{if } x_0 \sqcup \beta \neq \top \\ \beta & \text{if } x_0 \sqcup \beta = \top \wedge \beta \in C^s \\ \beta^s & \text{otherwise} \end{cases}$$



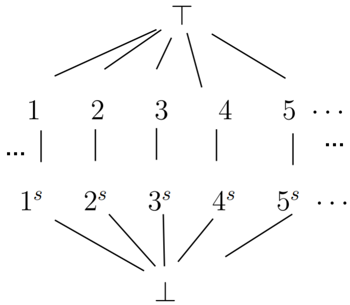
Speculative Sparse Conditional Constant Propagation (SSCCP)

- Introduce new speculative values $\{\ldots, 1^s, 2^s, \ldots\} \in C^S$
- Operation with *speculative* values result in *speculative* results (with same semantics as base operator)

$$\alpha^s \ \langle op \rangle \ \beta = (\alpha \ \langle op \rangle \ \beta)^s$$

- Transfer function for τ-functions ($\beta = x_1 \sqcup x_2 \sqcup \ldots$)

$$\tau(x_0, x_1, \ldots, x_n) \sqcup \begin{cases} x_0 \sqcup \top & \text{if } x_0 \sqcup \beta \neq \top \\ \beta & \text{if } x_0 \sqcup \beta = \top \wedge \beta \in C^s \\ \beta^s & \text{otherwise} \end{cases}$$



Almost trivial to generate profile-guided variants of standard analyses—it took us an afternoon to "port" SCCP to SSCCP!

Speculative SCCP Pass

CFG, Hotpath, show only speculative part. Const green, spec pink on the CFG.

LLVM Implementation : Speculative SCCP Pass

- We implement a speculative version of the SCCP to demonstrate the usefulness of the HPSSA Form.
- Modified the existing SCCP Pass to add in a function which handles the special `"llvm.tau"` intrinsic instructions used 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. Eg. Operands can be marked speculative using `markSpeculativeConstant()` function.

LLVM Implementation : Speculative SCCP Pass

- 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.

```
... // logs
[BBWorkList] Visiting LLVM Instrinsic : llvm.tau (call)
Visiting Tau Instruction
Speculative Operand : , speculative constant
Speculative Operand : llvm.tau.i32, speculative constant
Merged speculative constant into   %tau = call i32 (...)
@llvm.tau.i32(i32 %e.0, i32 90) : speculative constant
ValueLattice (TauState) : speculative constant
...
```

Presentation Outline : Section 3

Outline

Hot/Cold Paths

Definition 1. Hot/Cold Paths

A program path $p : n_1 \rightsquigarrow n_2$ is said to be hot (cold) if the sequence of edges from node n_1 to n_2 appears (does not appear) in any profiled path that occurs frequently in the program profile.

Hot/Cold Paths

Definition 1. Hot/Cold Paths

A program path $p : n_1 \rightsquigarrow n_2$ is said to be hot (cold) if the sequence of edges from node n_1 to n_2 appears (does not appear) in any profiled path that occurs frequently in the program profile.

Definition 2. Temperature (θ) of a node (edge)

- hot: if the node (edge) is present on a hot path;
- cold: if the node (edge) is not present on any hot path.

Hot/Cold Paths

Definition 1. Hot/Cold Paths

A program path $p : n_1 \rightsquigarrow n_2$ is said to be hot (cold) if the sequence of edges from node n_1 to n_2 appears (does not appear) in any profiled path that occurs frequently in the program profile.

Definition 2. Temperature (θ) of a node (edge)

- hot: if the node (edge) is present on a hot path;
- cold: if the node (edge) is not present on any hot path.

Definition 3. Hot/Cold Reaching Definitions and Definition Chains

A definition δ at a basic-block n_1 is said to reach a respective use at a basic-block n_2 hot if there exists a hot path from n_1 to n_2, and δ is not killed along that path. A definition δ at a basic-block n_1 is said to reach a respective use at a basic-block n_2 cold if there does not exist a hot path from n_1 to n_2, and δ is not killed at least along one cold path from n_1 to n_2.

Inserting τ – *functions*

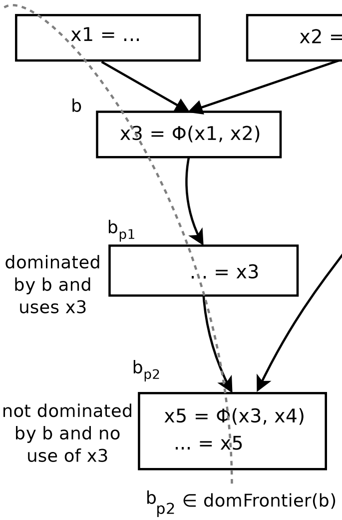
Necessary condition for τ-functions

Lemma 1. A node n requires a τ-function for variable x due to a definition d^x (of a variable x) if

1. n is the junction of a hot and a cold path, i.e., paths at different temperatures meet at this node;
2. n is reachable by at least two different definitions of the variable x.

Proof. If condition I fails, a τ -function is unnecessary as n can then be reachable by only hot or only cold definitions of x. If condition II fails, a τ -function is again unnecessary as the node is then dominated by a definition of x.

Inserting τ-functions



Inserting τ – *functions*

Definition 4. Thermal Frontier (TF)

For definition d defined at a node u reaching v, v is in the Thermal Frontier of (u,d), $v \in TF(u, d)$, iff

1. the node v is also exposed to a reaching definition d' defined at a node $u \notin Dom(w)$ (w not dominated by u)
2. $\theta(u \rightsquigarrow v) \neq \theta(w \rightsquigarrow v)$
3. v is the first node on the paths $u \rightsquigarrow v$ and $w \rightsquigarrow v$ that satisfies the above properties.

Inserting τ − *functions*

Definition 4. Thermal Frontier (TF)

For definition d defined at a node u reaching v, v is in the Thermal Frontier of (u,d), $v \in TF(u, d)$, iff

1. the node v is also exposed to a reaching definition d' defined at a node $u \notin Dom(w)$ (w not dominated by u)
2. $\theta(u \rightsquigarrow v) \neq \theta(w \rightsquigarrow v)$
3. v is the first node on the paths $u \rightsquigarrow v$ and $w \rightsquigarrow v$ that satisfies the above properties.

Theorem

For a set of visible definitions of a variable x at a set of nodes κ, τ-statements are only required at the Iterated Thermal Frontier ITF^x for variable x.

HPSSA construction [Roy et al., CC'10]

- Insert ϕ-functions:
  - insert ϕ-functions at the *iterated dominance frontiers*
- **Insert τ-functions**
  - insert τ-functions at the *iterated thermal frontiers*
- Allocate arguments to ϕ-functions
  - use a variable stack to allocate the ϕ-function arguments
- **Allocate arguments to τ-functions**
  - maintain path-sensitive reaching definitions for each program variable corresponding to each hot path on a path-sensitive stack;
  - for each instruction in the program, the algorithm update the respective stack to record the change in the path-sensitive reaching definitions due to the instruction;
  - when a τ-function is encountered, the current set of reaching definitions on the stack is used to allocate the speculative arguments for the τ-function.

HPSSA construction over SSA [Jaiswal et. al., SCOPES'17]

Difficulties

- Needs the SSA construction identified, broken into and retrofitted with the HPSSA construction phases.
- The algorithm is quite complex!

HPSSA construction over SSA [Jaiswal et. al., SCOPES'17]

Difficulties

- Needs the SSA construction identified, broken into and retrofitted with the HPSSA construction phases.
- The algorithm is quite complex!

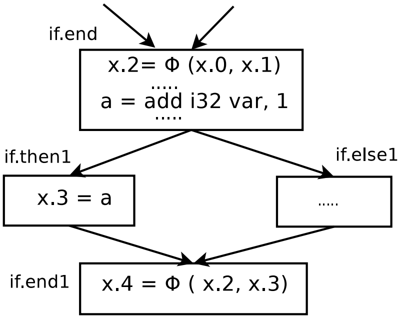
HPSSA over SSA

- Easily incorporated within existing compilers: Construction over the SSA form
- Efficient: Lesser instructions have to be traversed
- Simpler: many constructs are eliminated

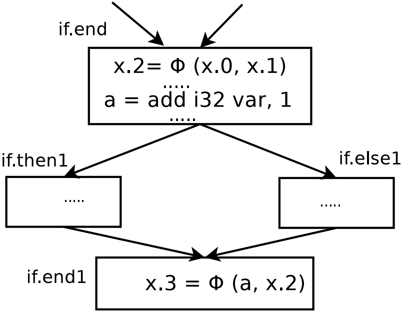
A naïve attempt: Mimic [Roy et al.]

- attempt to "recover" the renamed versions of each base variable that is merged by the ϕ-functions;
- then, allocate a single path-sensitive stack for all versions of the same base variable.

Optimized SSA forms



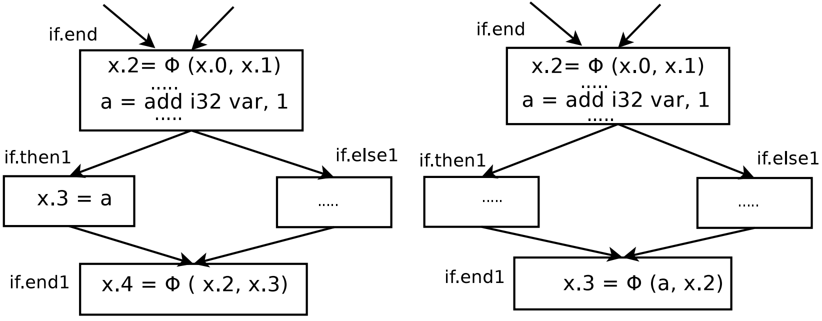
a) Unoptimized version



b) Optimized SSA version

a and x.2 are live simultaneously — hence, cannot be different versions of the same variable

Optimized SSA forms



a) Unoptimized version

b) Optimized SSA version

a and x.2 are live simultaneously — hence, cannot be different versions of the same variable

in the above example, copy propagation breaks the *phi congruence property*...

ϕ – *congruence* property

Shreedhar et al. [SAS'99]

"The occurrences of all resources which belong to the same phi congruence class in a program can be replaced by a representative resource. After the replacement, the phi instruction can be eliminated without violating the semantics of the original program."

- Sreedhar et al. circumvent the problem by translating the optimized SSA form to the conventional SSA form (that satisfies the phi congruence property) before translating out of SSA.
- **We directly build the HPSSA form over the optimized SSA form!**

Brief Algorithm

- **Insert τ-functions**
  - Insert at Thermal Frontiers
- **Allocate arguments to τ-functions**
  - path-sensitive traversal through the program to identify definitions that reach τ-functions through hot paths
  - constrains its inspection to only the ϕ-functions and the τ-functions

Allocating τ-function arguments

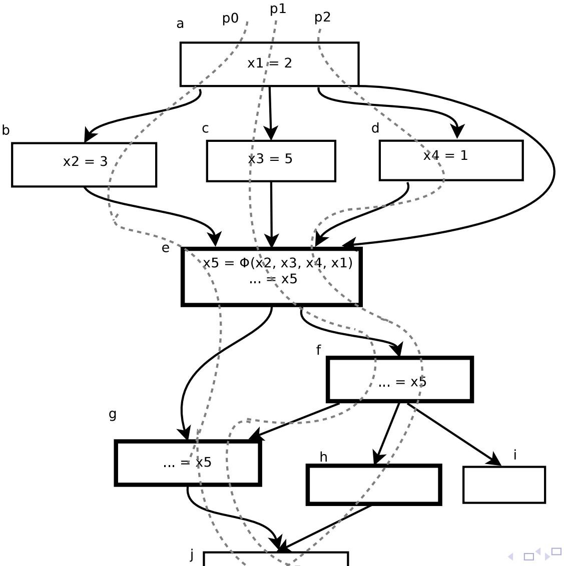
Uses a path-sensitive stack: **phiStack**

- **phiStack** is a stack of **frames**
- each frame $\langle d_i, \xi_i \rangle$ where $\xi_i = \{p_1, p_2, \ldots\}$
- support operations:
  - push(frame f, block b)
  - pop(block b)

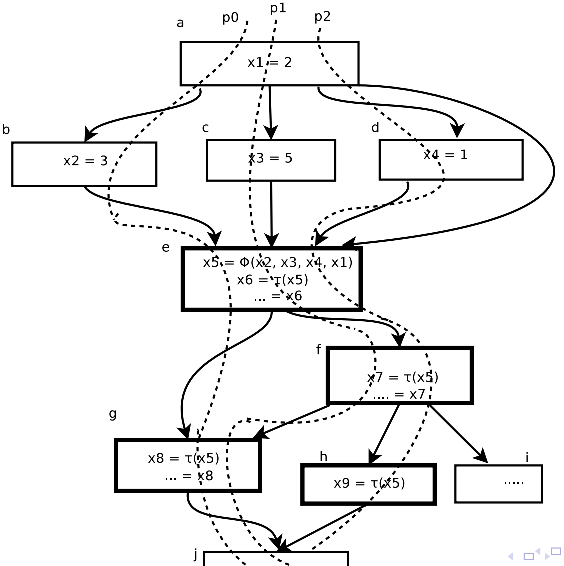
High-level algorithm

- Load arguments from ϕ- and τ-functions along with their hot path sets on **phiStack**
- Assign the definition from the topmost frame of **phiStack** to any τ-function encountered

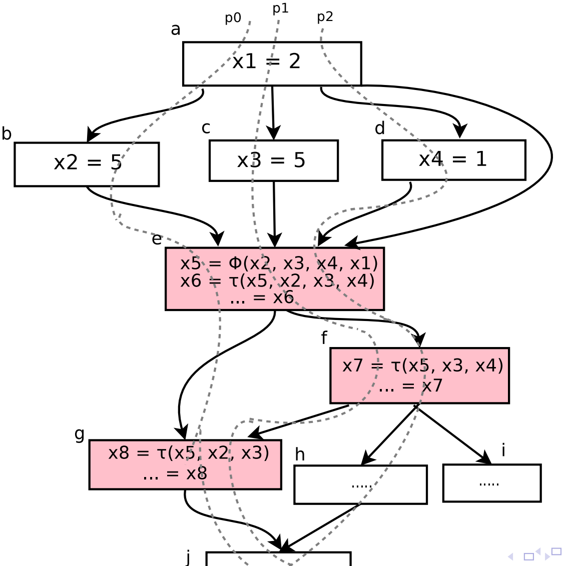
SSA program



τ-functions inserted



τ-arguments allocated



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],
+                         []>;
```

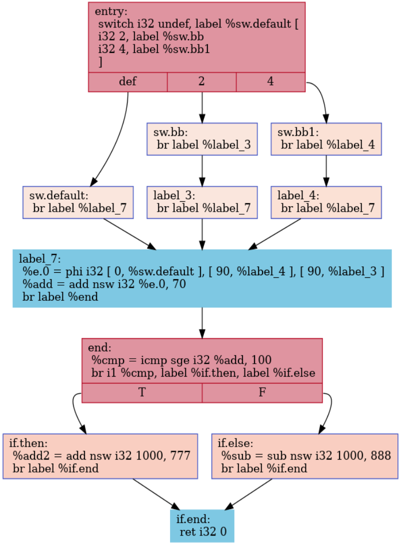
- Modified `Verifier::verifyDominatesUse()` function since we don't want our intrinsic to interfere with `dominators` computation.

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

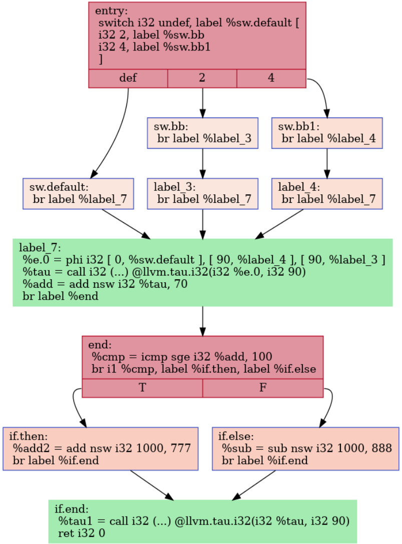
HPSSAPass : Overview

- `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 to get a compact representation of all the profiled hot paths in the program and then call `HPSSAPass::getCaloricConnector()` 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()` 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.

Presentation Outline : Section 4

Conclusion