

# The Hot Path SSA Form in LLVM

## Algorithms & Applications

IIT Kanpur  
Dept. Of Computer Science & Engineering

<sup>1</sup>IIT Kanpur  
PRAISE Group

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

# Presentation Outline : Section 1

## 1 HPSSA : Why another SSA Form?

- Profile Guided Optimization
- Doing Profile Guided Optimizations is Hard!
- Profile Guided SSCCP Analysis using HPSSA Form

## 2 What is HPSSA form?

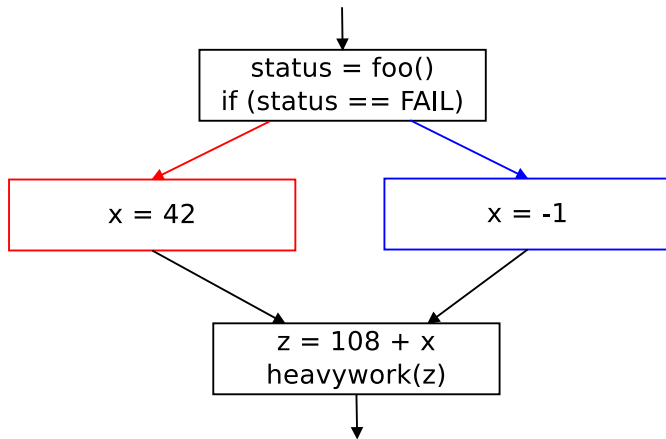
- Hot Path SSA Form
- Profile Guided SSCCP Pass

## 3 How is HPSSA Implemented?

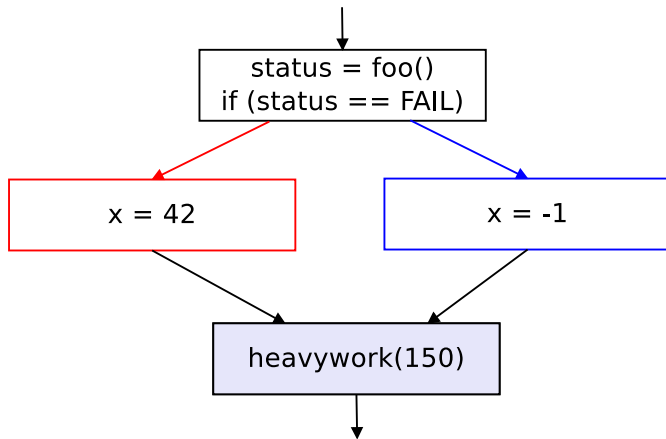
- Constructing HPSSA Form
- Implementing HPSSA Form in LLVM

## 4 Conclusion

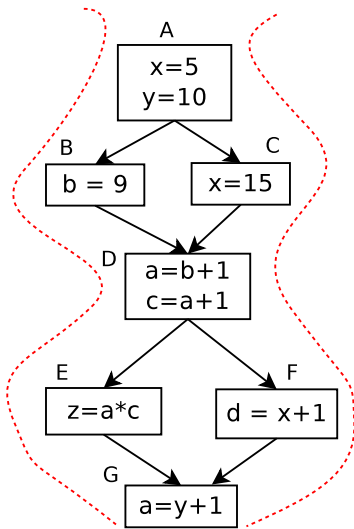
# A case for profile-guided optimizations (PGO)



# A case for profile-guided optimizations (PGO)



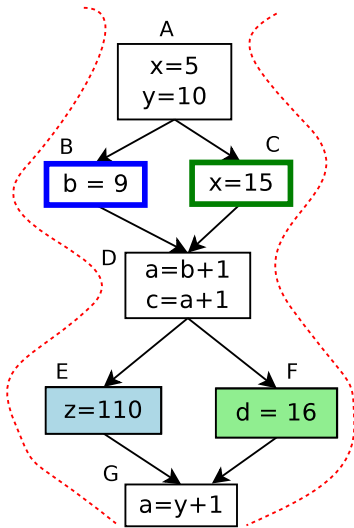
# Profile-guided analysis on paths



## 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

```
1 // Function to process "llvm.tau" function intrinsic.
2 visitTauNode() {
3     ...
4     SpecValueLatticeElement TauState = getValueState(&Tau),
5     beta = getValueState(Tau.getOperand(1)),
6     x0 = getValueState(Tau.getOperand(0));
7     if (TauState.isOverdefined())
8         return (void)markOverdefined(&Tau);
9     for (unsigned i = 1, e = (Tau.getNumOperands() - 1); i != e; ++i){
10         SpecValueLatticeElement IV = getValueState(Tau.getOperand(i));
11         beta.mergeIn(IV);
12         NumActiveIncoming++;
13         if (beta.isOverdefined())
14             break;
15     }
16     beta.mergeIn(x0);
17     if (beta.isConstant())
18         TauState.markSpeculativeConstant(beta.getConstant());
19
20     if (beta.isConstantRange())
21         TauState.markSpeculativeConstantRange(beta.getConstantRange());
22
23     if (beta.isOverdefined() && x0.isOverdefined())
24         TauState.markOverdefined();
25     ...
26 }
27 }
```

```
1 // Omit handling of "llvm.tau" intrinsic
2 // as a regular Instruction.
3 for (auto& I : *(&(BB))) {
4     CallInst* CI = dyn_cast<CallInst>(&I);
5     if (CI != NULL) {
6         Function* CF = CI->getCalledFunction();
7         if (CF != NULL &&
8             CF->getIntrinsicID() ==
9             Function::lookupIntrinsicID("llvm.tau")){
10             visitTauNode(I);
11         } else {
12             visit(I);
13         }
14     } else {
15         visit(I);
16     }
17 }
```

# Using HPSSAPass [It is easy!]

- Include the header file `HPSSA.h` to use `llvm::HPSSAPass` class.
- Load shared object using `opt` tool. `opt -load HPSSA.cpp.so ...`

```
1  #include <HPSSA.h> // import the header.
2  #include <SCCP.h>
3
4  class SpecSCCPass : public PassInfoMixin<SpecSCCPass> {
5      public: PreservedAnalyses run(Function &F,
6          FunctionAnalysisManager &AM);
7  };
8  ...
9
10 PreservedAnalyses SpecSCCPass::run(Function &F,
11 FunctionAnalysisManager &AM) {
12     if (F.getName() != "main")
13         return PreservedAnalyses::all();
14
15     HPSSAPass hpssaUtil; // Make a HPSSAPass Object.
16     hpssaUtil.run(F, AM); // Call the HPSSAPass::run() function.
17
18     std::vector<Instruction *> TauInsts
19     = hpssaUtil.getAllTauInstructions(F); // Calling HPSSA utility function.
20
21     std::cout << "\t\tTotal Tau Instructions : " << TauInsts.size() << "\n";
22     ...
23 }
24
25 /// [output] Total Tau Instructions : 7
```

# Speculative Pass using HPSSA : SSCCP

```
1 int main() {
2   int x = 2, m, n, y, z = 9, c = 1;
3   std::cin >> m;
4   switch(m) {
5     case 2 : x = 2 * c + 5; n = 10; break;
6     case 4 : x = 2 * c + 5; n = x - 2; break;
7     case 6 : x = 2 * c + 1; n = x + 2; break;
8     default : break;
9   }
10  y = 2 * x + 10;
11  if (y <= z + x) {
12    // ..
13  } else {
14    z = n + 3 * x;
15    switch (z) {
16      default : break;
17      case 200 : goto end;
18      case 300 : exit(0); }
19  }
20  m = y + x;
21  end:
22  z = x;
23  return 0;
24 }
```

```
1 int main() {
2   int x = 2, m, n, y, z = 9, c = 1;
3   std::cin >> m;
4   switch(m) {
5     case 2 : x = 2 * c + 5; n = 10; break;
6     case 4 : x = 2 * c + 5; n = x - 2; break;
7     case 6 : x = 2 * c + 1; n = x + 2; break;
8     default : break;
9   }
10  y = 2 * x + 10;
11  if (y <= z + x) {
12    // ..
13  } else {
14    z = n + 3 * x; // n : Speculative Constant 5
15    switch (z) {
16      default : break;
17      case 200 : goto end;
18      case 300 : exit(0); }
19  }
20  m = y + x; // x : Speculative Constant 7
21  end:
22  z = x;
23  return 0;
24 }
```

Legends ■ Overdefined ■ Discovered Constants ■ Real Constants ■ Speculative Constants

# Speculative Pass using HPSSA : SSCCP

## Standard SCCP VS. Speculative SCCP Pass.

```
1  # Running Regular SCCP Pass on Program.
2  $ opt -sccp -time-passes -debug-only=sccp \
3    IR/LL/test.ll -S -o \
4    IR/LL/test_sccp_onbaseline.ll \
5    -f 2> output/custom_sccp_onbaseline.log
6
7  ...
8  Output:
9    ...
10   Constant: i32 2 = %mul = mul nsw i32 2, 1
11   Constant: i32 7 = %add = add nsw i32 2, 5
12   Constant: i32 2 = %mul2 = mul nsw i32 2, 1
13   Constant: i32 7 = %add3 = add nsw i32 2, 5
14   Constant: i32 5 = %sub = sub nsw i32 7, 2
15   Constant: i32 2 = %mul5 = mul nsw i32 2, 1
16   Constant: i32 3 = %add6 = add nsw i32 2, 1
17   Constant: i32 5 = %add7 = add nsw i32 3, 2
```

```
1  # Running HPSSA Transformation followed by Speculative SCCP Pass.
2  $ opt -load build/SCCPSolverTau.cpp.so
3    -load build/HPSSA.cpp.so \
4    -load-pass-plugin=build/SpecSCCP.cpp.so \
5    -passes="specsccp" \
6    -time-passes -debug-only=specsccp \
7    IR/LL/test.ll -S -o IR/LL/test_spec_sccp.ll \
8    -f 2> output/custom_speculative_sccp.log
9
10  ...
11  Output :
12    ...
13    (TauState) tau1 : speculative constantrange<5, 6>
14    (TauState) tau7 : speculative constantrange<7, 8>
15    ...
16    Constant: i32 2 = %mul = mul nsw i32 2, 1
17    Constant: i32 7 = %add = add nsw i32 2, 5
18    Constant: i32 2 = %mul2 = mul nsw i32 2, 1
19    Constant: i32 7 = %add3 = add nsw i32 2, 5
20    Constant: i32 5 = %sub = sub nsw i32 7, 2
21    Constant: i32 2 = %mul5 = mul nsw i32 2, 1
22    Constant: i32 3 = %add6 = add nsw i32 2, 1
23    Constant: i32 5 = %add7 = add nsw i32 3, 2
```

# Presentation Outline : Section 2

- 1 HPSSA : Why another SSA Form?
  - Profile Guided Optimization
  - Doing Profile Guided Optimizations is Hard!
  - Profile Guided SSCCP Analysis using HPSSA Form
- 2 What is HPSSA form?
  - Hot Path SSA Form
  - Profile Guided SSCCP Pass
- 3 How is HPSSA Implemented?
  - Constructing HPSSA Form
  - Implementing HPSSA Form in LLVM
- 4 Conclusion

# The Hot Path SSA Form (HPSSA)

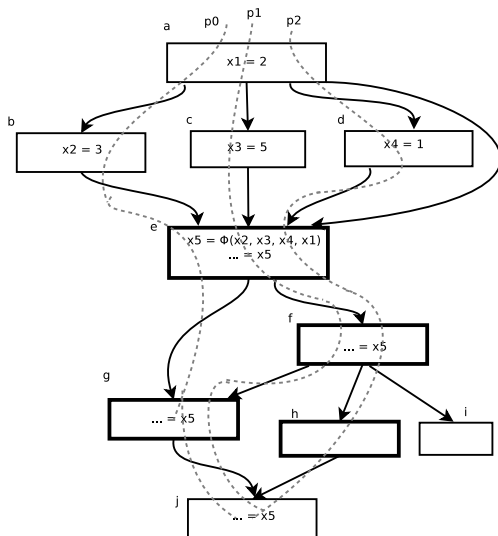
## Semantics of a $\phi$ -function

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

## Semantics of a $\tau$ -function

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

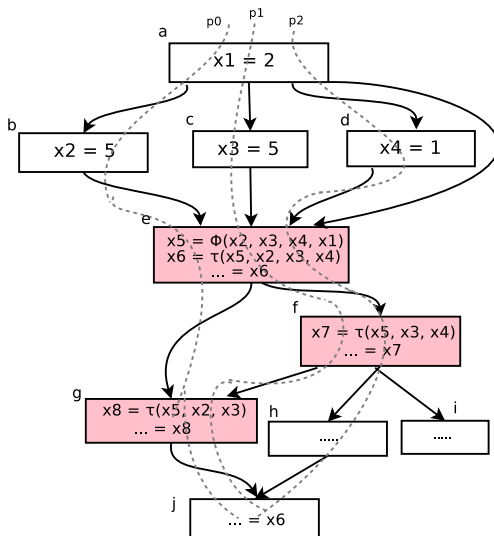
# The SSA form



## 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. <sup>a</sup>

---

<sup>a</sup>or the meet-over-all-paths reaching definition chains, if the use is not reachable from any meet-over-hot-paths reaching definition chain

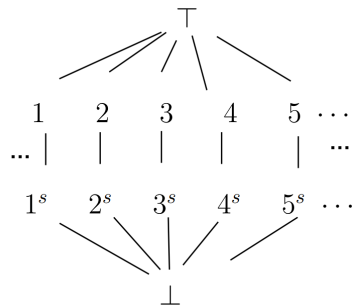
# Speculative Sparse Conditional Constant Propagation (SSCCP)

- Introduce new speculative values  $\{\dots, 1^s, 2^s, \dots\} \in C^s$
- 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  $\tau$ -functions ( $\beta = x_1 \sqcup x_2 \sqcup \dots$ )

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



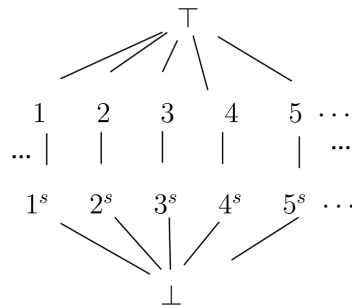
# Speculative Sparse Conditional Constant Propagation (SSCCP)

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

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

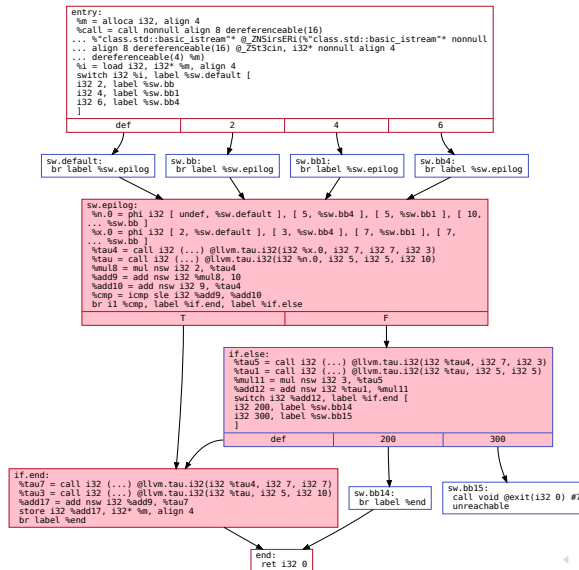
- Transfer function for  $\tau$ -functions ( $\beta = x_1 \sqcup x_2 \sqcup \dots$ )

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



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

# Profile Guided SSCCP Pass in LLVM



# LLVM Implementation : Profile Guided SSCCP Pass

- Modified the existing SSCCP Pass to add `visitTauNode()` function which handles the special `"llvm.tau"` intrinsic instructions used for  $\tau$ -functions.<sup>1</sup>
- Added a new lattice element type `"spec_constant"` in `ValueLattice` class supporting operations on speculative constants. Modified the `SCCPInstVisitor::mergeIn()` function to handle lattice "meet" operation for the new speculative constants introduced.
- Added new functions in the `SCCPInstVisitor` and `SCCPSolver` class to handle operations on speculative constants. Eg. Operands can be marked speculative using as function `markSpeculativeConstant()`.
- Modified the `SCCPInstVisitor::solve()` function to process `"llvm.tau"` intrinsic instructions using `visitTauNode()` instead of the standard `visit()` function.

---

<sup>1</sup>Since we added the  $\tau$ -functions as an `"llvm.tau"` intrinsic, we blocked processing it as a regular LLVM Instruction.

# Presentation Outline : Section 3

- 1 HPSSA : Why another SSA Form?
  - Profile Guided Optimization
  - Doing Profile Guided Optimizations is Hard!
  - Profile Guided SSCCP Analysis using HPSSA Form
- 2 What is HPSSA form?
  - Hot Path SSA Form
  - Profile Guided SSCCP Pass
- 3 How is HPSSA Implemented?
  - Constructing HPSSA Form
  - Implementing HPSSA Form in LLVM
- 4 Conclusion



# 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 ( $\theta$ ) 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 ( $\theta$ ) 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  $\delta$  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  $\delta$  is not killed along that path. A definition  $\delta$  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  $\delta$  is not killed at least along one cold path from  $n_1$  to  $n_2$ .

# Inserting $\tau$ – functions

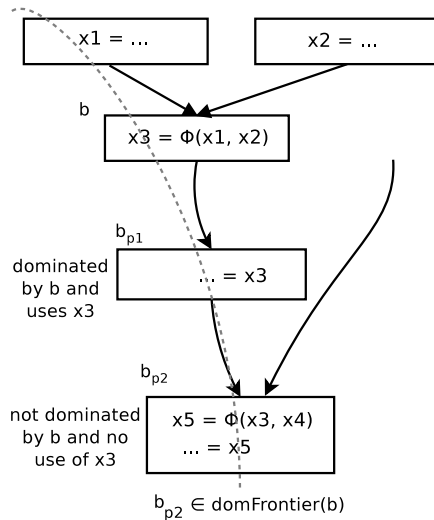
## Necessary condition for $\tau$ -functions

Lemma 1. A node  $n$  requires a  $\tau$ -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  $\tau$ -function is unnecessary as  $n$  can then be reachable by only hot or only cold definitions of  $x$ . If condition II fails, a  $\tau$ -function is again unnecessary as the node is then dominated by a definition of  $x$ .

# Inserting $\tau$ -functions



# Inserting $\tau$ – *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 $\tau$ – 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  $\kappa$ ,  $\tau$ -statements are only required at the Iterated Thermal Frontier  $ITF^x$  for variable  $x$ .

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

- **Insert  $\phi$ -functions:**
  - insert  $\phi$ -functions at the *iterated dominance frontiers*
- **Insert  $\tau$ -functions**
  - insert  $\tau$ -functions at the *iterated thermal frontiers*
- **Allocate arguments to  $\phi$ -functions**
  - use a variable stack to allocate the  $\phi$ -function arguments
- **Allocate arguments to  $\tau$ -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  $\tau$ -function is encountered, the current set of reaching definitions on the stack is used to allocate the speculative arguments for the  $\tau$ -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!

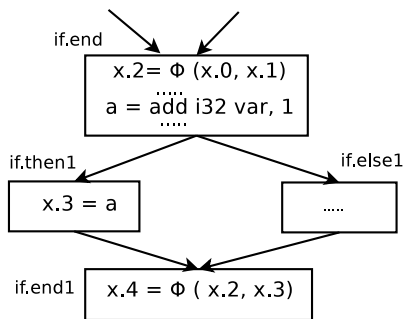
## HPSSA over SSA

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

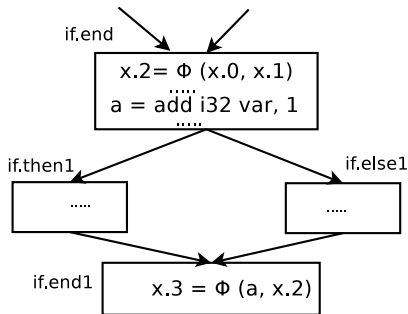
## A naïve attempt: Mimic [Roy et al.]

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

# Optimized SSA forms



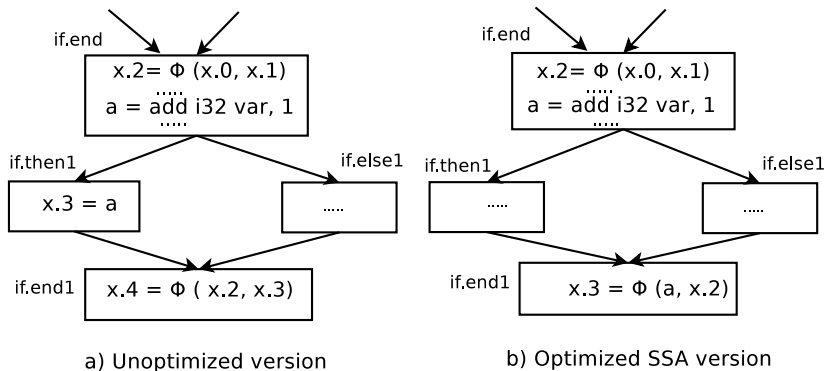
a) Unoptimized version



b) Optimized SSA version

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

# Optimized SSA forms



**a 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*...

## $\phi$ – 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  $\tau$ -functions**
  - Insert at Thermal Frontiers
- **Allocate arguments to  $\tau$ -functions**
  - path-sensitive traversal through the program to identify definitions that reach  $\tau$ -functions through hot paths
  - constrains its inspection to only the  $\phi$ -functions and the  $\tau$ -functions

# Allocating $\tau$ -function arguments

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

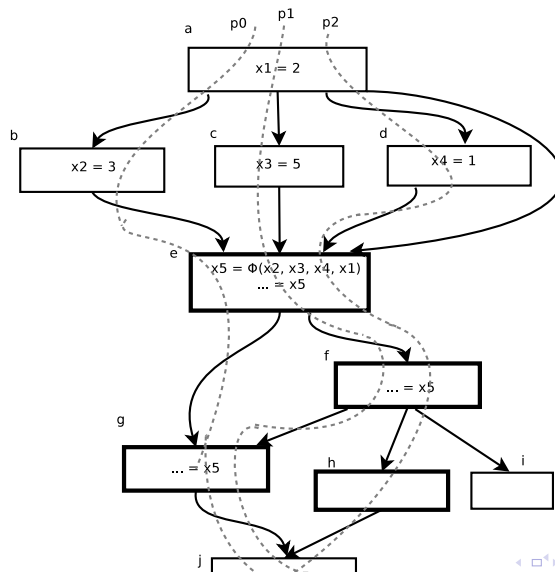
- **phiStack** is a stack of **frames**
- each frame  $\langle d_i, \xi_i \rangle$  where  $\xi_i = \{p_1, p_2, \dots\}$
- support operations:
  - push(frame f, block b)
  - pop(block b)

## High-level algorithm

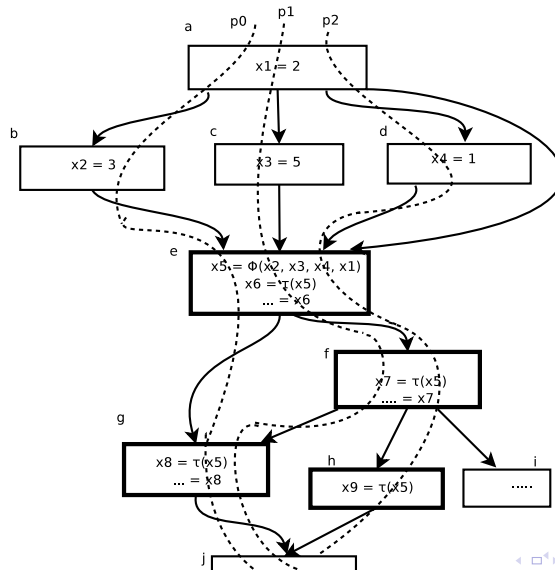
- Load arguments from  $\phi$ - and  $\tau$ -functions along with their hot path sets on **phiStack**
- Assign the definition from the topmost frame of **phiStack** to any  $\tau$ -function encountered



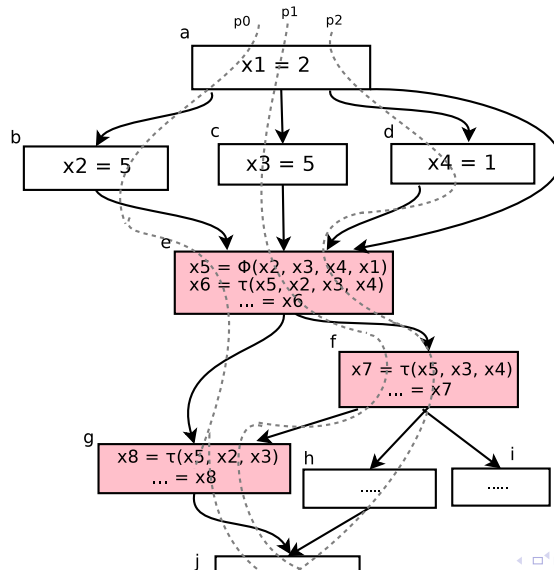
# SSA program



# $\tau$ -functions inserted



# $\tau$ -arguments allocated



# What we modified in LLVM Source?

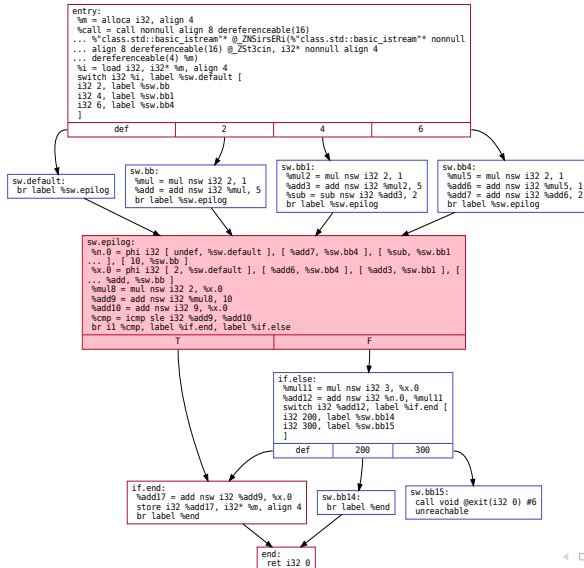
- New `llvm::intrinsic` signature, "`llvm.tau`" to support addition and removal of  $\tau$ -functions to the LLVM SSA IR representation.

```
1 + //----- intrinsic for tau -----//
2 + def int_tau : DefaultAttrsIntrinsic<[llvm_any_ty],
3 +     [llvm_vararg_ty],
4 +     []>;
```

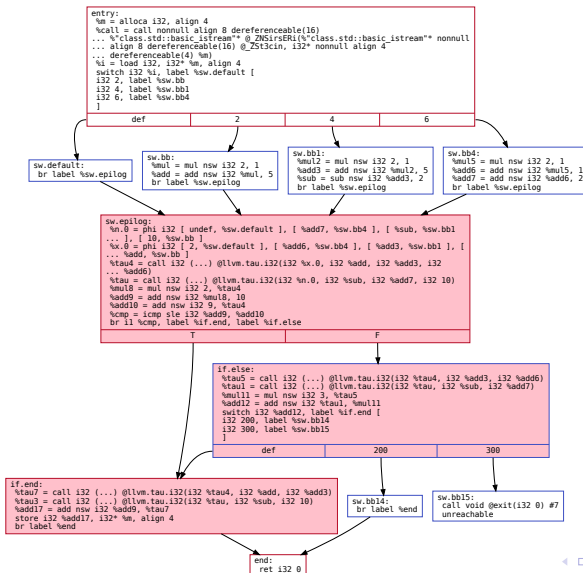
- Modified `Verifier::verifyDominatesUse()` function since we don't want our intrinsic to interfere with `dominators` computation.

```
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    + }
11    ...
```

- `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 Accumulator, `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.



# Hot Path SSA Form



# 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.



- Out of HPSSA Form.
  - A separate 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 call from the LLVM IR.
  - The LLVM IR becomes identical to what it was before running the HPSSA Pass.

# Presentation Outline : Section 4

- 1 HPSSA : Why another SSA Form?
  - Profile Guided Optimization
  - Doing Profile Guided Optimizations is Hard!
  - Profile Guided SSCCP Analysis using HPSSA Form
- 2 What is HPSSA form?
  - Hot Path SSA Form
  - Profile Guided SSCCP Pass
- 3 How is HPSSA Implemented?
  - Constructing HPSSA Form
  - Implementing HPSSA Form in LLVM
- 4 Conclusion

# Conclusion

This work is based on the following research papers.

- Subhajit et. al, The Hot Path SSA Form: Extending the Static Single Assignment Form for Speculative Optimizations. CC 2010:304-323
- Jaiswal et. al, Constructing HPSSA over SSA. SCOPES 2017: 31-40

Other References:

- The LLVM Project. <https://llvm.org/>
- LLVM Language Reference Manual