

# The Hot Path SSA Form in LLVM

## Algorithms & Applications

Mohd. Muzzammil<sup>1</sup>, Abhay Mishra<sup>1</sup>, Sumit Lahiri<sup>1</sup>  
Awanish Pandey<sup>2</sup>, and Subhajit Roy<sup>1</sup>

<sup>1</sup>Dept. of Computer Science & Engineering, IIT Kanpur

<sup>2</sup>Qualcomm India Pvt. Ltd.

This presentation presents the details of building a robust and efficient implementation of the **Hot Path SSA (HPSSA)** form in the LLVM compiler infrastructure.

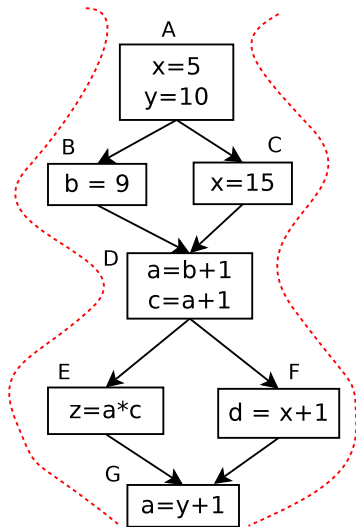
This presentation presents the details of building a robust and efficient implementation of the **Hot Path SSA (HPSSA)** form in the LLVM compiler infrastructure.

The Hot Path SSA form is based on the following research papers.

- Subhajit Roy and Y.N. Srikant. The Hot Path SSA Form: Extending the Static Single Assignment Form for Speculative Optimizations. In CC '10: International Conference on Compiler Construction. 2010. CC 2010:304-323
- 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. 2017. SCOPES 2017: 31-40

- 1 HPSSA : Why another SSA Form?
  - Introduction to Path Profile Guided Optimizations
  - Profile Guided SpecSCCP Analysis using HPSSA Form
- 2 What is HPSSA form?
  - Hot Path SSA Form
  - Profile Guided SpecSCCP Pass
- 3 How is HPSSA Implemented?
  - Constructing HPSSA Form
  - Implementing HPSSA Form in LLVM
- 4 Conclusion

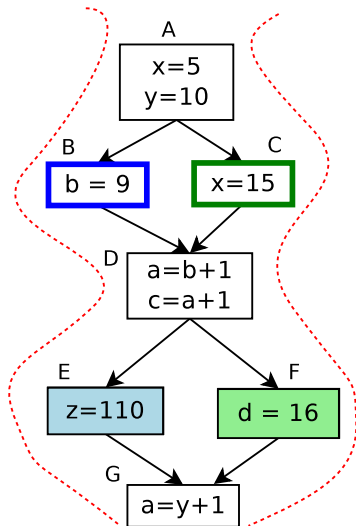
# Profile-guided analysis on paths



## 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

## 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)

## 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



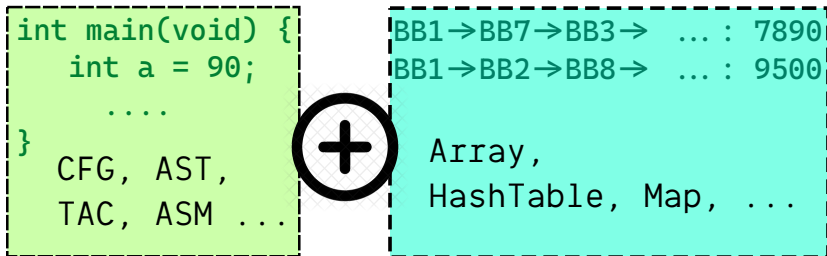
- **Code understanding**
  - Can expose refactoring opportunities

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

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

# Why is path-profile-guided analysis hard?

disparate data-structures, one for **program representation** and other for **profile information**.



# 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.
  - ...

# Our Objective

Can we **weave** profile information into the program representation

# Our Objective

Can we **weave** profile information into the program representation  
....into a **single, consistent** data-structure

# Our Objective

Can we **weave** profile information into the program representation

....into a **single, consistent** data-structure

... that provides the convenience and elegance of an **SSA-like** intermediate form



Can we **weave** profile information into the program representation

....into a **single, consistent** data-structure

... that provides the convenience and elegance of an **SSA-like** intermediate form

...allowing the design of profile-guided versions of “traditional” optimizations with  
**trivial algorithmic modification** of the base algorithms

# ... and PGO is easy with the Hot Path SSA (HPSSA) Form!

```
1 // Function to process "llvm.tau" function intrinsic.
2 void SpecSCCPInstVisitor::visitTauNode(Instruction &Tau) {
3     ...
4     SpecValueLatticeElement &TauState = getValueState(&Tau),
5     beta = getValueState(Tau.getOperand(1)),
6     x0 = getValueState(Tau.getOperand(0));
7
8     if (TauState.isOverdefined())
9         return (void)markOverdefined(&Tau);
10
11     for (unsigned i = 1, e = (Tau.getNumOperands() - 1); i != e; ++i){
12         SpecValueLatticeElement IV = getValueState(Tau.getOperand(i));
13         beta.mergeIn(IV);
14         NumActiveIncoming++;
15         if (beta.isOverdefined())
16             break;
17     }
18
19     if (beta.isConstantRange()
20         && beta.getConstantRange().isSingleElement())
21         beta.markSpeculativeConstantRange(beta.getConstantRange());
22     if (beta.isConstant())
23         beta.markSpeculativeConstant(beta.getConstant());
24
25     beta.mergeInSpec(x0);
26     if (x0.isOverdefined())
27         TauState.mergeIn(beta);
28
29     ... // futher processing similar to visitPHINode();
30 }
```

```
1 // Omit handling of "llvm.tau" intrinsic
2 // as a regular Instruction.
3 void SpecSCCPInstVisitor::solve() {
4     ...
5     ...
6     for (auto& I : *(&(BB))) {
7         CallInst* CI = dyn_cast<CallInst>(&I);
8         if (CI != NULL) {
9             Function* CF = CI->getCalledFunction();
10            if (CF != NULL &&
11                CF->getIntrinsicID() ==
12                Function::lookupIntrinsicID("llvm.tau")){
13                visitTauNode(I);
14            } else {
15                visit(I);
16            }
17        } else {
18            visit(I);
19        }
20    }
21    ... // rest of the code.
22 }
```

# ... and PGO is easy with the Hot Path SSA (HPSSA) Form!

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

Only these few lines were enough to create a new path profile guided analysis, *Speculative Sparse Conditional Constant Propagation (SpecSCCP)* from the currently existing SCCP pass in LLVM !

```
1 // Omit handling of "llvm.tau" intrinsic
2 // as a regular Instruction.
3 void SpecSCCPInstVisitor::solve() {
4     ...
5     ...
6     for (auto& I : *(&(BB))) {
7         CallInst* CI = dyn_cast<CallInst>(&I);
8         if (CI)
9             CI->setDebugLoc(Instruction::DebugLoc::None);
10         CI->setDebugLoc(Instruction::DebugLoc::None);
11         CI->setDebugLoc(Instruction::DebugLoc::None);
12         CI->setDebugLoc(Instruction::DebugLoc::None);
13         visitTauNode(I);
14     } else {
15         visit(I);
16     }
17 } else {
18     visit(I);
19 }
20 }
21 ... // rest of the code.
22 }
```

# ... and PGO is easy with the Hot Path SSA (HPSSA) Form!

```
1 // Function to process "llvm.tau" function intrinsic.
2 void SpecSCCPInstVisitor::visitTauNode(Instruction &Tau) {
3     ...
4     SpecValueLatticeElement &TauState = getValueState(&Tau),
5     beta = getValueState(Tau.getOperand(1)),
6     x0 = getValueState(Tau.getOperand(0));
7
8     if (TauState.isOverdefined())
9         return (void)markOverdefined(&Tau);
10
11     for (unsigned i = 1, e = (Tau.getNumOperands() - 1); i != e; ++i){
12         SpecValueLatticeElement IV = getValueState(Tau.getOperand(i));
13         beta.mergeIn(IV);
14         NumActiveIncomin...
15         if (beta.isOverd...
16             break;
17     }
18
19     if (beta.isConstantRange()
20         && beta.getConstantRange().isSingleElement())
21         beta.markSpeculativeConstantRange(beta.getConstantRange());
22     if (beta.isConstant())
23         beta.markSpeculativeConstant(beta.getConstant());
24
25     beta.mergeInSpec(x0);
26     if (x0.isOverdefined())
27         TauState.mergeIn(beta);
28
29     ... // futher processing similar to visitPHINode();
30 }
```

It took us only an afternoon to transform SCCP to SpecSCCP

```
1 // Omit handling of "llvm.tau" intrinsic
2 // as a regular Instruction.
3 void SpecSCCPInstVisitor::solve() {
4     ...
5     ...
6     for (auto& I : *(&(BB))) {
7         CallInst* CI = dyn_cast<CallInst>(&I);
8         if (CI != NULL) {
9             Function* CF = CI->getCalledFunction();
10             ... ID() ==
11
12             Function::lookupIntrinsicID("llvm.tau")){
13                 visitTauNode(I);
14             } else {
15                 visit(I);
16             }
17         } else {
18             visit(I);
19         }
20     }
21     ... // rest of the code.
22 }
```

# SCCP vs SpecSCCP

## SCCP

```
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 = n + x;
21    end:
22    z = x;
23    return 0;
24 }
```

## SpecSCCP

```
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 = n + x; // x : Speculative Constant 7
21    end:
22    z = x;
23    return 0;
24 }
```

**Legend:** ■ Overdefined ■ Real Constants ■ Speculative Constants

# SCCP vs SpecSCCP

## SCCP

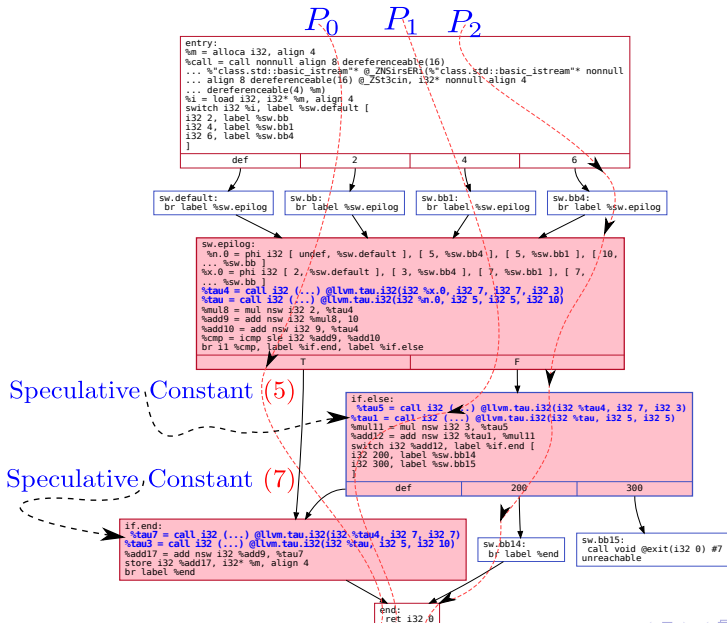
```
1 int main() {
2   int x = 2, m, n, y, z = 9, c = 1;
3   std::cin >> m;
4   switch(m) {
5     case 2 : x = 2 * c + 5; n = 10; break;
6     case 4 : x = 2 * c + 5; n = x - 2; break;
7     case 6 : x = 2 * c + 1; n = x + 2; break;
8     default : break;
9   }
10  y = 2 * m + 10;
11  if (y <= 10) {
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 = n + x;
21  end:
22  z = x;
23  return 0;
24 }
```

## SpecSCCP

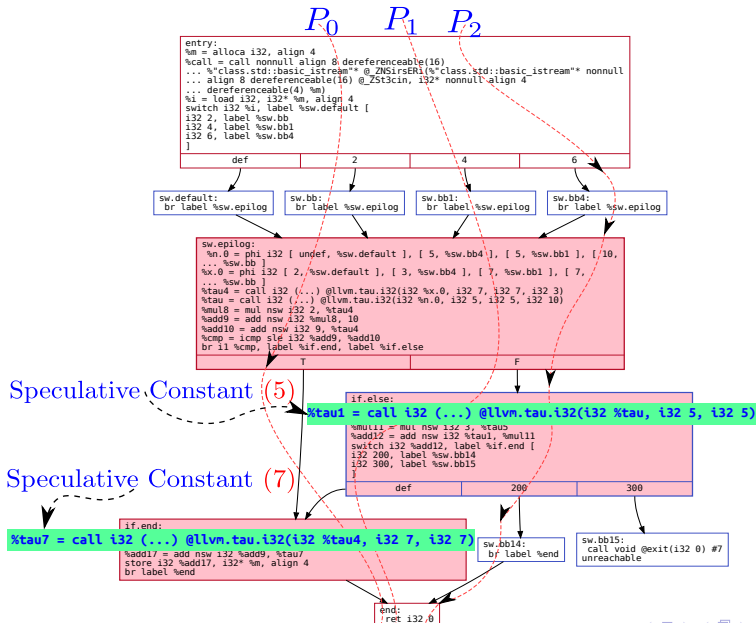
```
1 int main() {
2   int x = 2, m, n, y, z = 9, c = 1;
3   std::cin >> m;
4   switch(m) {
5     case 2 : x = 2 * c + 5; n = 10; break;
6     case 4 : x = 2 * c + 5; n = x - 2; break;
7     case 6 : x = 2 * c + 1; n = x + 2; break;
8     default : break;
9   }
10  y = 2 * m + 10;
11  if (y <= 10) {
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 = n + x; // x : Speculative Constant 7
21  end:
22  z = x;
23  return 0;
24 }
```

SpecSCCP discovers  $n$  &  $x$  as speculative constants.

Legend:   Overdefined   Real Constants   Speculative Constants



Speculative SCCP on 'main' function



Speculative SCCP on 'main' function





# SCCP vs SpecSCCP

## 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
18
19
20
21
22
```

```
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 Constant: i32 2 = %mul = mul nsw i32 2, 1
13 Constant: i32 7 = %add = add nsw i32 2, 5
14 Constant: i32 2 = %mul2 = mul nsw i32 2, 1
15 Constant: i32 7 = %add3 = add nsw i32 2, 5
16 Constant: i32 5 = %sub = sub nsw i32 7, 2
17 Constant: i32 2 = %mul5 = mul nsw i32 2, 1
18 Constant: i32 3 = %add6 = add nsw i32 2, 1
19 Constant: i32 5 = %add7 = add nsw i32 3, 2
20 Speculative Constant: i32 5 = %tau1 = call i32 (...)
21 @llvm.tau.i32(i32 %tau, i32 5, i32 5)
22 Speculative Constant: i32 7 = %tau7 = call i32 (...)
23 @llvm.tau.i32(i32 %tau4, i32 7, i32 7)
```

# Using the HPSSA Form for writing new analyses

- 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 YourPGOPass : public PassInfoMixin<YourPGOPass> {
5     public: PreservedAnalyses run(Function &F,
6         FunctionAnalysisManager &AM);
7 };
8
9 PreservedAnalyses YourPGOPass::run(Function &F,
10     FunctionAnalysisManager &AM) {
11     ...
12     HPSSAPass hpssaUtil; // Make a HPSSAPass Object.
13     hpssaUtil.run(F, AM); // Call the HPSSAPass::run() function.
14     ...
15     // Rest of the code ..
16 }
```

- 1 HPSSA : Why another SSA Form?
  - Introduction to Path Profile Guided Optimizations
  - Profile Guided SpecSCCP Analysis using HPSSA Form
- 2 What is HPSSA form?
  - Hot Path SSA Form
  - Profile Guided SpecSCCP Pass
- 3 How is HPSSA Implemented?
  - Constructing HPSSA Form
  - Implementing HPSSA Form in LLVM
- 4 Conclusion

# The Hot Path SSA Form (HPSSA)

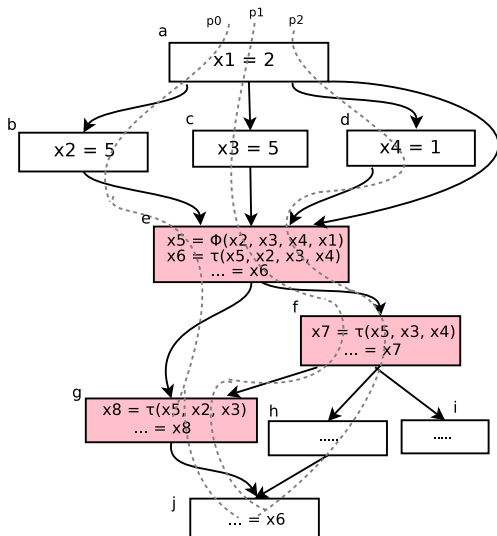
## Semantics of a $\phi$ -function

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

## 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 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**

## 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]

## 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



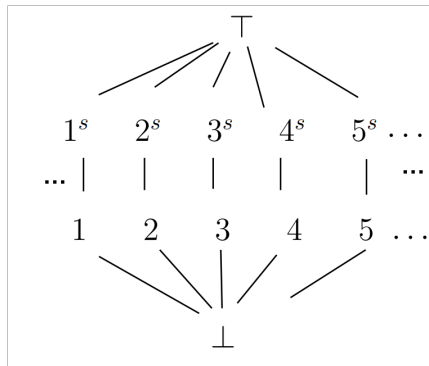
# Speculative Sparse Conditional Constant Propagation (SpecSCCP)

- 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  
( $\beta = x_1 \sqcup x_2 \sqcup \dots \sqcup x_n$ , i.e. join of speculative args.)

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



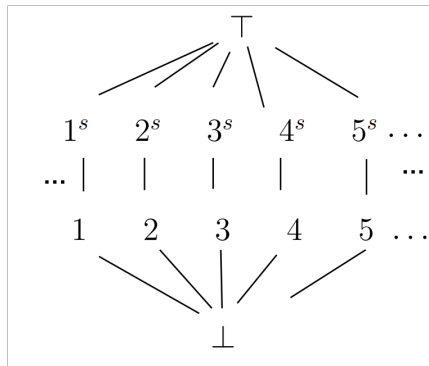
# Speculative Sparse Conditional Constant Propagation (SpecSCCP)

- 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  
( $\beta = x_1 \sqcup x_2 \sqcup \dots \sqcup x_n$ , i.e. join of speculative args.)

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

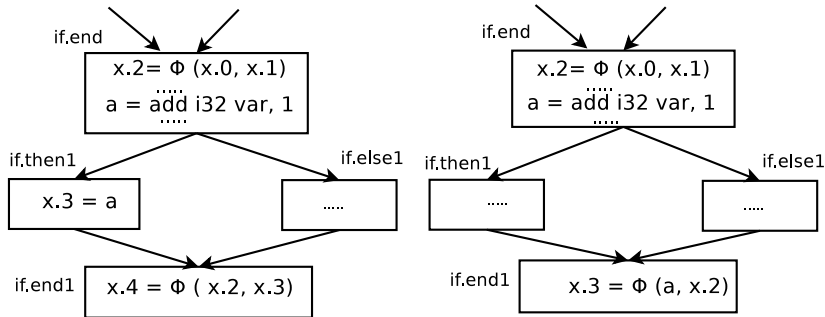


*Almost trivial to generate profile-guided variants of standard analyses—an afternoon to “port” SCCP to SpecSCCP!*

- 1 HPSSA : Why another SSA Form?
  - Introduction to Path Profile Guided Optimizations
  - Profile Guided SpecSCCP Analysis using HPSSA Form
- 2 What is HPSSA form?
  - Hot Path SSA Form
  - Profile Guided SpecSCCP Pass
- 3 How is HPSSA Implemented?
  - Constructing HPSSA Form
  - Implementing HPSSA Form in LLVM
- 4 Conclusion

- **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

# Optimized SSA forms

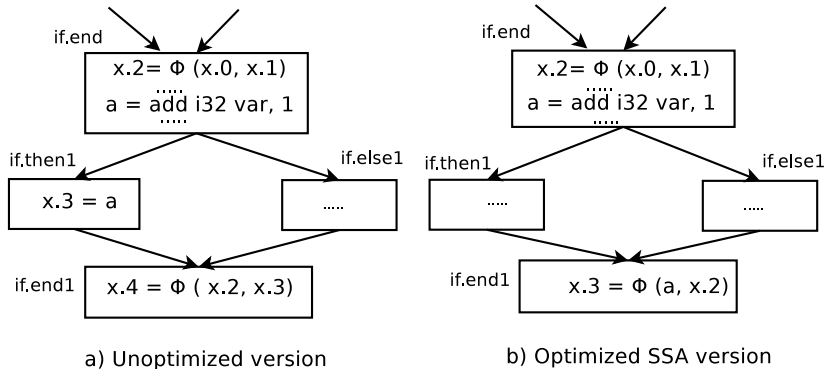


a) Unoptimized version

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*...

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!**

# 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 +           []>;
```



# 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.

- `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 :

- `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.

- `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**.

- `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.

- Out of HPSSA Form.
  - A seperate pass using the new LLVM Pass Manager.

```
class TDSTRPass : public PassInfoMixin<TDSTRPass>
```

- 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.



- 1 HPSSA : Why another SSA Form?
  - Introduction to Path Profile Guided Optimizations
  - Profile Guided SpecSCCP Analysis using HPSSA Form
- 2 What is HPSSA form?
  - Hot Path SSA Form
  - Profile Guided SpecSCCP Pass
- 3 How is HPSSA Implemented?
  - Constructing HPSSA Form
  - Implementing HPSSA Form in LLVM
- 4 Conclusion

- The Hot Path SSA form opens up an exciting opportunity for compiler writers to “port” existing standard analyses to their profile guided variants.

- The Hot Path SSA form opens up an exciting opportunity for compiler writers to “port” existing standard analyses to their profile guided variants.
- We plan to push this work to LLVM main branch soon.

# LLVM Implementation : Profile Guided SpecSCCP Pass

- Modified the existing SCCP Pass to add `visitTauNode()` function which handles the special `"llvm.tau"` intrinsic instructions used for  $\tau$ -functions.<sup>1</sup>

---

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

# LLVM Implementation : Profile Guided SpecSCCP Pass

- Modified the existing SCCP 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"` and `mergeInSpec()` function in `ValueLattice` class supporting operations on speculative constants. Modified the existing `mergeIn()` function to handle lattice "meet" operation for the new speculative constants introduced.

---

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

# LLVM Implementation : Profile Guided SpecSCCP Pass

- Modified the existing SCCP 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"` and `mergeInSpec()` function in `ValueLattice` class supporting operations on speculative constants. Modified the existing `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()`.

---

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

# LLVM Implementation : Profile Guided SpecSCCP Pass

- Modified the existing SCCP 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"` and `mergeInSpec()` function in `ValueLattice` class supporting operations on speculative constants. Modified the existing `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.

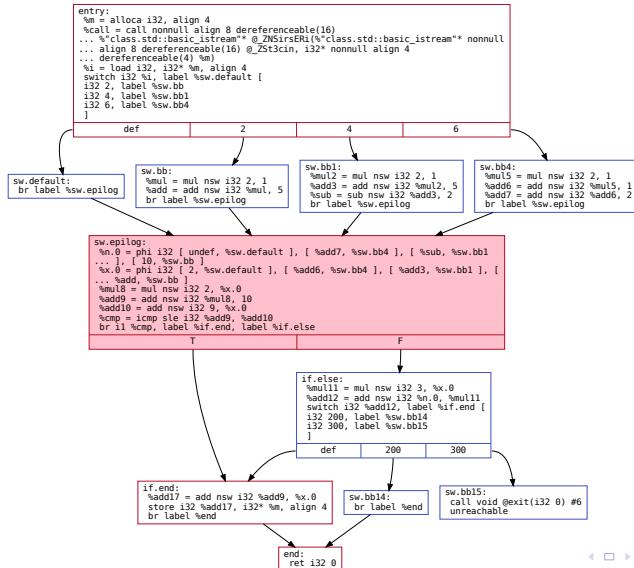
- `HPSSAPass::run(Function &F, FunctionAnalysisManager &AM)`
  - Invokes `HPSSAPass::getProfileInfo()` function to get a compact representation of all the profiled **hot paths** in the program and then calls `HPSSAPass::getCaloricConnector()` to get all the caloric connectors from the **hot path** information. This is a precursor to finding strategic positions to place `"llvm.tau"` intrinsic calls in the LLVM IR.



- `HPSSAPass::run(Function &F, FunctionAnalysisManager &AM)`
  - Invokes `HPSSAPass::getProfileInfo()` function to get a compact representation of all the profiled **hot paths** in the program and then calls `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.

- `HPSSAPass::run(Function &F, FunctionAnalysisManager &AM)`
  - Invokes `HPSSAPass::getProfileInfo()` function to get a compact representation of all the profiled **hot paths** in the program and then calls `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.

# Program in SSA Form



# Program in Hot Path SSA Form

