# The Hot Path SSA Form in LLVM

## Algorithms & Applications

Mohd. Muzzammil[1], Abhay Mishra[1], Sumit Lahiri[1]
Awanish Pandey[2], and Subhajit Roy[1]

[1]Dept. Of Computer Science & Engineering, IIT Kanpur
[2]Qualcomm

# References

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

# References

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. 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

# Presentation Outline

# Presentation Outline

# 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

# Our Objective

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

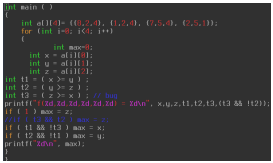
....into a **single, consistent** data-structure

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

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

# Why is path-profile-guided analysis hard?

disparate data-structures: program + profile



```
int main ( )
{
    int a[3][3]= {{0,2,0}, {1,2,0}, {7,5,4}, {2,5,1}};
    for (int i=0; i<3; i++)
    {
        int max=0;
      int x = a[i][0];
      int y = a[i][1];
      int z = a[i][2];
int t1 = ( x >= y ) ;
int t2 = ( y >= z ) ;
int t3 = ( z >= x ) ; // new
printf("(%d,%d,%d,%d,%d,%d) - %d\n", x,y,z,t1,t2,t3,(t3 && !t2));
if ( t ) max = z;
// if ( t && !z ) max = z;
if ( t1 && !t2 ) max = x;
if ( t2 && !t1 ) max = y;
printf("%d\n", max);
}
}
```

| | |
|---|---|
| 1-2-4-5 | 30 |
| 21-2-5-5-1 | 25 |
| 1-2-4-5 | 30 |
| 21-2-5-5-1 | 25 |
| 1-2-4-5 | 30 |
| 21-2-5-5-1 | 25 |
| 1-2-4-5 | 30 |
| 21-2-5-5-1 | 25 |

# 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.
  - ...

# ... but it 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.isConstant())
20     beta.markSpeculativeConstant(beta.getConstant());
21   if (beta.isConstantRange())
22     beta.markSpeculativeConstantRange(beta.getConstantRange());
23
24   if (beta.isOverdefined() || x0.isOverdefined())
25     TauState.markOverdefined();
26
27   beta.mergeInSpec(x0);
28   TauState.mergeIn(beta);
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 }
```

# ... but it 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     SpecV
13     beta.
14     NumAc
15     if (b
16       bre
17   }
18
19   if (beta.isConstant())
20     beta.markSpeculativeConstant(beta.getConstant());
21   if (beta.isConstantRange())
22     beta.markSpeculativeConstantRange(beta.getConstantRange());
23
24   if (beta.isOverdefined() || x0.isOverdefined())
25     TauState.markOverdefined();
26
27   beta.mergeInSpec(x0);
28   TauState.mergeIn(beta);
29   ... // futher processing similar to visitPHINode();
30 }
```

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
9
10
11
12       visitTauNode(I);
13     } else {
14       visit(I);
15     }
16   } else {
17     visit(I);
18   }
19 }
20 ... // rest of the code.
21 }
```

# ... but it is easy with the Hot Path SSA (HPSSA) Form!

```
// Function to process "llvm.tau" function intrinsic.
void SpecSCCPInstVisitor::visitTauNode(Instruction &Tau) {
  ...
  SpecValueLatticeElement TauState = getValueState(&Tau),
    beta = getValueState(Tau.getOperand(1)),
    x0 = getValueState(Tau.getOperand(0));

  if (TauState.isOverdefined())
    return (void)markOverdefined(&Tau);

  for (unsigned i = 1, e = (Tau.getNumOperands() - 1); i != e; ++i){
    SpecValueLatticeElement IV = getValueState(Tau.getOperand(i));
    beta.mergeIn(IV);
    NumActiveIncomi
    if (beta.isOverd
      break;
  }

  if (beta.isConstant())
    beta.markSpeculativeConstant(beta.getConstant());
  if (beta.isConstantRange())
    beta.markSpeculativeConstantRange(beta.getConstantRange());

  if (beta.isOverdefined() || x0.isOverdefined())
    TauState.markOverdefined();

  beta.mergeInSpec(x0);
  TauState.mergeIn(beta);
  ... // futher processing similar to visitPHINode();
}
```

**It took us only an afternoon to transform SCCP to SpecSCCP**

```
// Omit handling of "llvm.tau" intrinsic
// as a regular Instruction.
void SpecSCCPInstVisitor::solve() {
  ...
  ...
  for (auto& I : *&(*(BB))) {
    CallInst* CI = dyn_cast<CallInst>(&I);
    if (CI != NULL) {
      Function* CF = CI->getCalledFunction();
                                    ID() ==
        Function::lookupIntrinsicID("llvm.tau")){
        visitTauNode(I);
      } else {
        visit(I);
      }
    } else {
      visit(I);
    }
  }
  ... // rest of the code.
}
```

# Presentation Outline

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

**Legend:** Overdefined Discovered Constants 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 * x + 10;
11   if ( y <= z
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 }
```

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
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 }
```

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

**Legend:** Overdefined | Discovered Constants | Real Constants | Speculative Constants

# SCCP vs SpecSCCP

Standard SCCP VS. Speculative SCCP Pass.

```
# Running Regular SCCP Pass on Program.
$ opt -sccp -time-passes -debug-only=sccp \
  IR/LL/test.ll -S -o \
  IR/LL/test_sccp_onbaseline.ll \
  -f 2> output/custom_sccp_onbaseline.log

...
Output:
  ...
  Constant: i32 2 =   %mul = mul nsw i32 2, 1
  Constant: i32 7 =   %add = add nsw i32 2, 5
  Constant: i32 2 =   %mul2 = mul nsw i32 2, 1
  Constant: i32 7 =   %add3 = add nsw i32 2, 5
  Constant: i32 5 =   %sub = sub nsw i32 7, 2
  Constant: i32 2 =   %mul5 = mul nsw i32 2, 1
  Constant: i32 3 =   %add6 = add nsw i32 2, 1
  Constant: i32 5 =   %add7 = add nsw i32 3, 2
```

```
# Running HPSSA Transformation followed by Speculative SCCP Pass.
$ opt -load build/SCCPSolverTau.cpp.so
  -load build/HPSSA.cpp.so \
  -load-pass-plugin=build/SpecSCCP.cpp.so \
  -passes="specsccp" \
  -time-passes -debug-only=specsccp \
  IR/LL/test.ll -S -o IR/LL/test_spec_sccp.ll \
  -f 2> output/custom_speculative_sccp.log

...
Output :
  ...
  (TauState) tau1 : speculative constantrange<5, 6>
  (TauState) tau7 : speculative constantrange<7, 8>
  ...
  Constant: i32 2 =   %mul = mul nsw i32 2, 1
  Constant: i32 7 =   %add = add nsw i32 2, 5
  Constant: i32 2 =   %mul2 = mul nsw i32 2, 1
  Constant: i32 7 =   %add3 = add nsw i32 2, 5
  Constant: i32 5 =   %sub = sub nsw i32 7, 2
  Constant: i32 2 =   %mul5 = mul nsw i32 2, 1
  Constant: i32 3 =   %add6 = add nsw i32 2, 1
  Constant: i32 5 =   %add7 = add nsw i32 3, 2
```

# 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 ...`

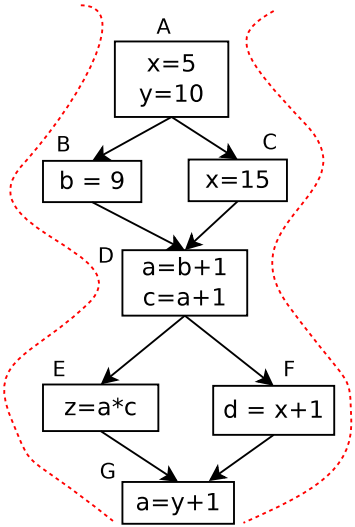
```
1  #include <HPSSA.h> // import the header.
2  #include <SCCP.h>
3
4  class SpecSCCPPass : public PassInfoMixin<SpecSCCPPass> {
5    public: PreservedAnalyses run(Function &F,
6    FunctionAnalysisManager &AM);
7  };
8  ...
9
10 PreservedAnalyses SpecSCCPPass::run(Function &F,
11   FunctionAnalysisManager &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.getAllTauInstrunctions(F); // Calling HPSSA utility function.
20
21     std::cout << "\t\tTotal Tau Instructions : " << TauInsts.size() << ",\n";
22     ...
23   }
```

Output : `Total Tau Instructions : 7, ...rest of the logs`

# Presentation Outline

# Presentation Outline

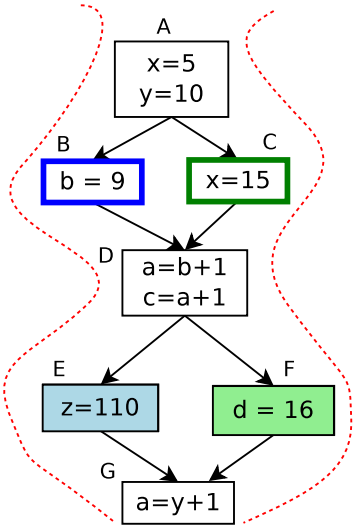
# Profile-guided analysis on paths




## 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

# Presentation Outline

# The Hot Path SSA Form (HPSSA)

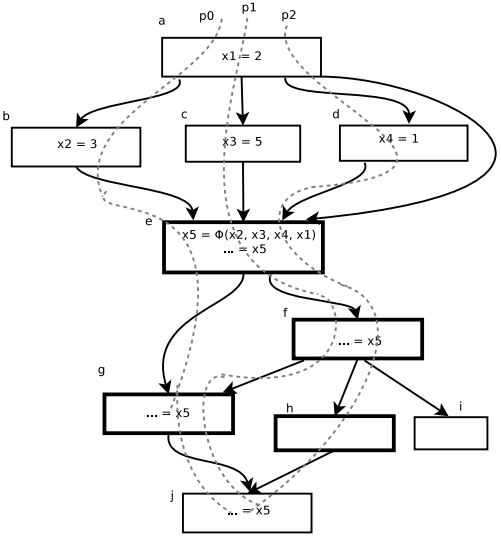
## Semantics of a $\phi$-function

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

## Semantics of a $\tau$-function

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

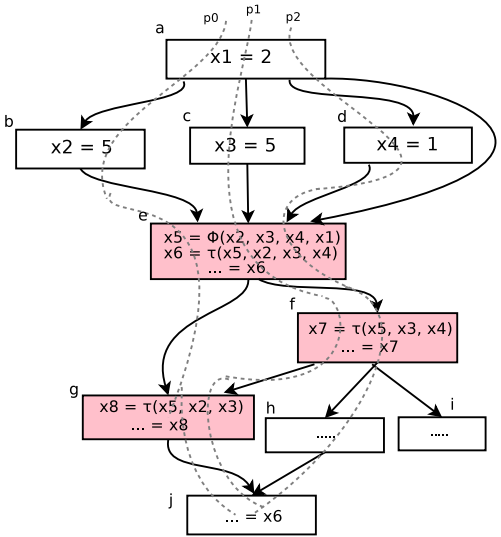
# The SSA form




**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

# Presentation Outline

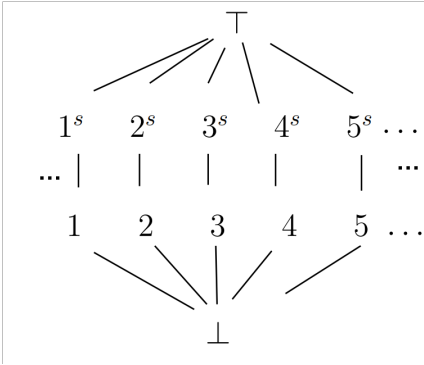
# Speculative Sparse Conditional Constant Propagation (SpecSCCP)

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

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



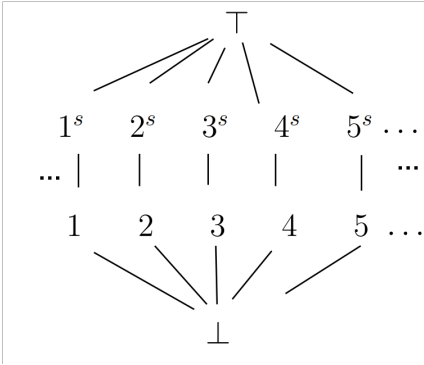
# Speculative Sparse Conditional Constant Propagation (SpecSCCP)

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

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

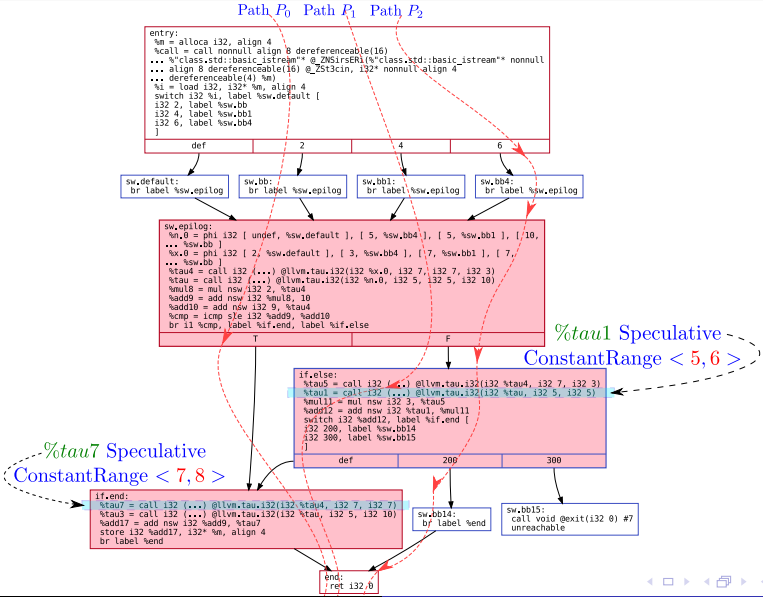
- Transfer function for $\tau$-functions
  ($\beta = x_1 \sqcup x_2 \sqcup \cdots \sqcup x_n$, i.e. join of speculative args.)

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




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

# Profile Guided SpecSCCP Pass in LLVM



# 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.[1]

[1] 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.[1]
- 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.

[1] 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.[1]
- 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()`.

[1] 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.[1]
- 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.

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

# Presentation Outline

# Presentation 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 ($\theta$) 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 $n_2$.

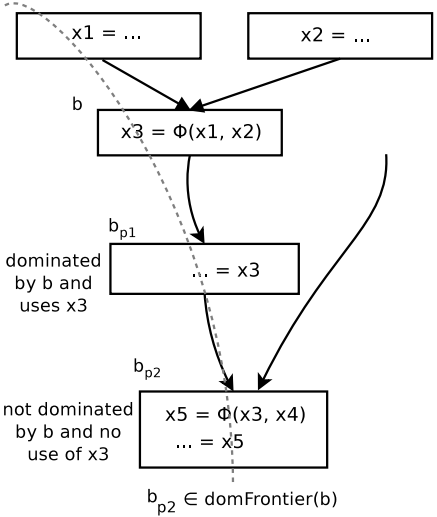
**Necessary condition for $\tau$-functions**

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



**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.

**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!

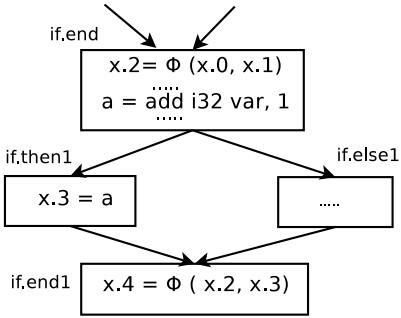
## HPSSA over SSA

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

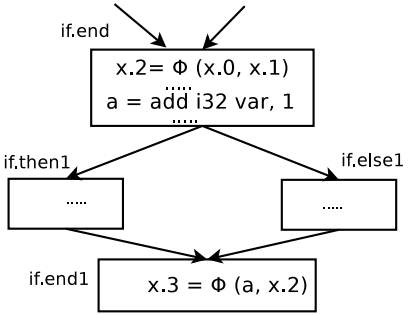
# A naïve attempt: Mimic [Roy et al.]

- 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



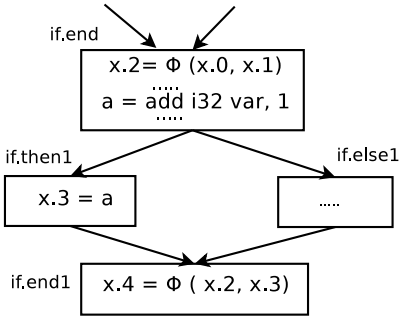

a) Unoptimized version



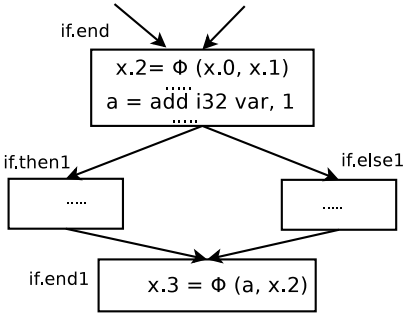

b) Optimized SSA version

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

# Optimized SSA forms




a) 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*...

# $\phi$ – *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

# Presentation Outline

# 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.

```
+ //===---------- 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.

# 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 :

# 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.

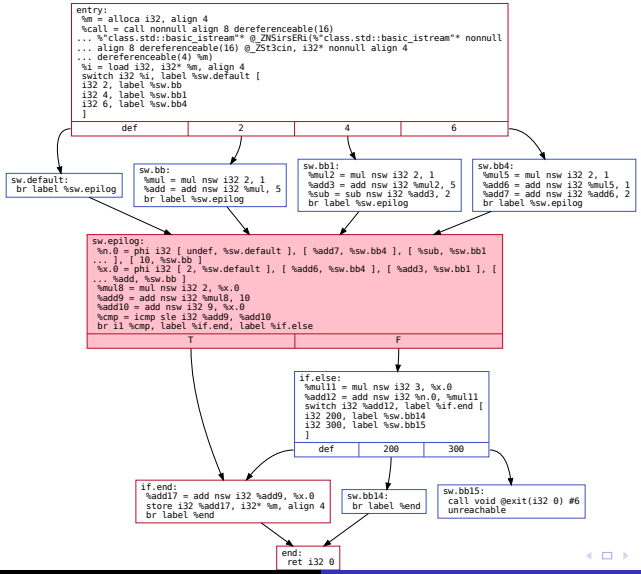
# HPSSAPass : Overview

- `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 .

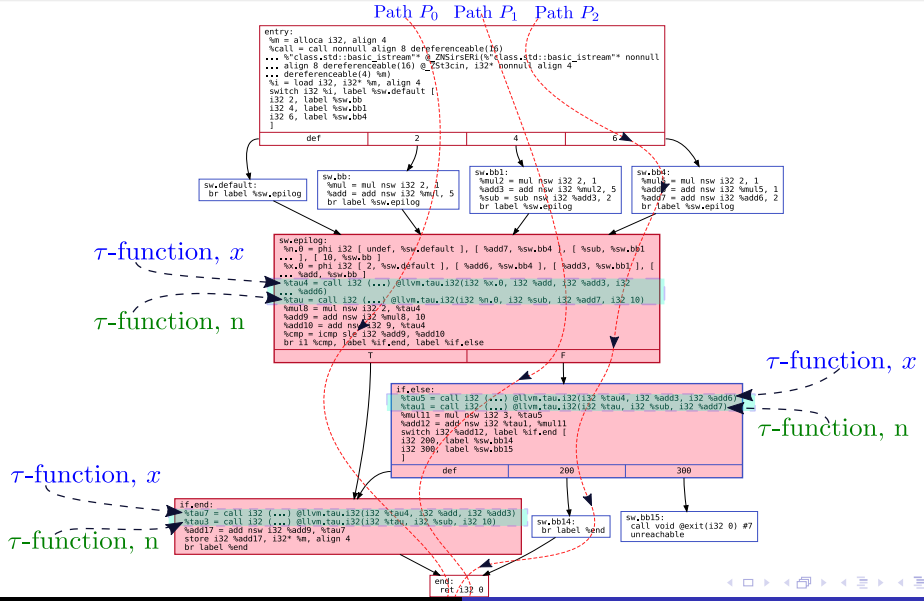
# HPSSAPass : Overview

- `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.

# Program in SSA Form



# Program in Hot Path SSA Form



# HPSSAPass : Main Pass

- `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 : Main Pass

- `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 : Main Pass

- `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.

# HPSSAPass : Destruction Pass

- Out of HPSSA Form.
  - A seperate pass using the new LLVM Pass Manager.
    `class TDSTRPass : public PassInfoMixin<TDSTRPass>`

# 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

# Conclusion

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

# Conclusion

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