The Hot Path SSA Form in LLVM

Algorithms & Applications

Mohd. Muzzammil\textsuperscript{1}, Abhay Mishra\textsuperscript{1}, Sumit Lahiri\textsuperscript{1}\textsuperscript{*}
Awanish Pandey\textsuperscript{2}, and Subhajit Roy\textsuperscript{1}

\textsuperscript{1}Dept. of Computer Science & Engineering, IIT Kanpur
\textsuperscript{2}Qualcomm India Pvt. Ltd.

*Presented by Sumit Lahiri
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.


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.


Presentation Outline

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
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.
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)
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
- **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**
Why is path-profile-guided analysis hard?

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

Program Representation

```
int main(void) {
   int a = 90;
     ....
}
CFG, AST, TAC, ASM ...
```

Profile Information

```
BB1 → BB7 → BB3 → ... : 7890
BB1 → BB2 → BB8 → ... : 9500
Array, HashTable, Map, ...
```
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*
  - ...
Can we **weave** profile information into the program representation
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!

```c
// Function to process "llvm.tau" function intrinsic.
void SpecSCCPInstVisitor::visitTauNode(Instruction &Tau) {
    // Code similar to that in visitPHINode(...).
    if (Tau.getType()->isStructTy())
        return (void)markOverdefined(&Tau);
    if (TauState.isOverdefined())
        return (void)markOverdefined(&Tau);
    // additional code.
    unsigned NumActiveIncoming = 0;
    SpecValueLatticeElement &TauState = getValueState(&Tau),
    beta = getValueState(Tau.getOperand(1)),
    x0 = getValueState(Tau.getOperand(0));
    for (unsigned i = 1, e = (Tau.getNumOperands() - 1); i != e; ++i){
        SpecValueLatticeElement IV = getValueState(Tau.getOperand(i));
        beta.mergeIn(IV);
        NumActiveIncoming++;
        if (beta.isOverdefined())
            break;
    }
    if (beta.isConstantRange() && beta.getConstantRange().isSingleElement())
        beta.markSpeculativeConstantRange(beta.getConstantRange());
    if (beta.isConstant())
        beta.markSpeculativeConstant(beta.getConstant());
    x0.mergeInSpec(beta, TauState);
    ... // further processing similar to visitPHINode();
}
```

// Omit handling of "llvm.tau" intrinsic as a regular Instruction.
```c
void SpecSCCPInstVisitor::solve() {
    ... 
    for (auto &I : *(*(BB))) {
        CallInst* CI = dyn_cast<CallInst>(&I);
        if (CI != NULL) {
            Function* CF = CI->getCalledFunction();
            if (CF != NULL &&
                CF->getIntrinsicID() ==
                Function::lookupIntrinsicID("llvm.tau"){
                visitTauNode(I);
            } else {
                visit(I);
            }
        } else {
            visit(I);
        }
    } else {
        visit(I);
    }
    ... // rest of the code.
}
... and PGO is easy with the Hot Path SSA (HPSSA) Form!

```cpp
// Function to process "llvm.tau" function intrinsic.
void SpecSCCPInstVisitor::visitTauNode(Instruction &Tau) {
    // Code similar to that in visitPHINode(...).
    if (Tau.getType()->isStructTy())
        return (void)markOverdefined(&Tau);
    if (TauState.isOverdefined())
        return (void)markOverdefined(&Tau);
    // additional code.
    unsigned NumActiveIncoming = 0;
    SpecValueLatticeElement &TauState = getValueState(&Tau),
    beta = getValueState(Tau.getOperand(1)),
    x0 = getValueState(Tau.getOperand(0));
    for (unsigned i = 1, e = (Tau.getNumOperands() - 1); i != e; ++i){
        SpecValueLatticeElement IV = getValueState(Tau.getOperand(i));
        beta.mergeIn(IV);
        NumActiveIncoming++;
        if (beta.isOverdefined())
            break;
    }
    if (beta.isConstantRange()
        && beta.getConstantRange().isSingleElement())
        beta.markSpeculativeConstantRange(beta.getConstantRange());
    if (beta.isConstant())
        beta.markSpeculativeConstant(beta.getConstant());
    x0.mergeInSpec(beta, TauState);
    ... // further processing similar to visitPHINode();
}
```

```
// 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();
            if (CF != NULL &&
                CF->getIntrinsicID() == Function::lookupIntrinsicID("llvm.tau")){
                visitTauNode(I);
            } else {
                visit(I);
            } else {
                visit(I);
            } else {
                visit(I);
            } else {
                visit(I);
            }
        } else {
            visit(I);
        }
    ...
```
... and PGO is easy with the Hot Path SSA (HPSSA) Form!

It took us only an afternoon to transform SCCP to SpecSCCP

```
// Function to process "llvm.tau" function intrinsic.
void SpecSCCPInstVisitor::visitTauNode(Instruction &Tau) {
    // Code similar to that in visitPHINode(...).
    if (Tau.getType()->isStructTy())
        return (void)markOverdefined(&Tau);
    if (TauState.isOverdefined())
        return (void)markOverdefined(&Tau);
    // additional code.
    unsigned NumActiveIncoming = 0;
    SpecValueLatticeElement &TauState = getValueState(&Tau),
    beta = getValueState(Tau.getOperand(1)),
    x0 = getValueState(Tau.getOperand(0));
    for (unsigned i = 1, e = (Tau.getNumOperands() - 1); i != e; ++i){
        SpecValueLatticeElement IV = getValueState(Tau.getOperand(i));
        beta.mergeIn(IV);
        NumActiveIncoming++;
        if (beta.isOverdefined())
            break;
    }
    if (beta.isConstantRange() &&
        beta.getConstantRange().isSingleElement())
        beta.markSpeculativeConstantRange(beta.getConstantRange());
    if (beta.isConstant())
        beta.markSpeculativeConstant(beta.getConstant());
    x0.mergeInSpec(beta, TauState);
    ...
}
```
Presentation Outline

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
SCCP

```cpp
int main() {
    int x = 2, m, n, y, z = 9, c = 1;
    std::cin >> m;
    switch(m) {
        case 2 : x = 2 * c + 5; n = 10; break;
        case 4 : x = 2 * c + 5; n = x - 2; break;
        case 6 : x = 2 * c + 1; n = x + 2; break;
        default : break;
    }
    y = 2 * x + 10;
    if (y <= z + x) {
        // ...
    } else {
        z = n + 3 * x;
        switch(z) {
            default : break;
            case 200 : goto end;
            case 300 : exit(0); }
    }
    m = n + x;
    end:
    z = x;
    return 0;
}
```

SpecSCCP

```cpp
int main() {
    int x = 2, m, n, y, z = 9, c = 1;
    std::cin >> m;
    switch(m) {
        case 2 : x = 2 * c + 5; n = 10; break;
        case 4 : x = 2 * c + 5; n = x - 2; break;
        case 6 : x = 2 * c + 1; n = x + 2; break;
        default : break;
    }
    y = 2 * x + 10;
    if (y <= z + x) {
        // ...
    } else {
        z = n + 3 * x; /* n : Speculative Constant 5 */
        switch(z) {
            default : break;
            case 200 : goto end;
            case 300 : exit(0); }
    }
    m = n + x; /* x : Speculative Constant 7 */
    end:
    z = x;
    return 0;
}```

Legend: ■ Overdefined ■ Real Constants ■ Speculative Constants
```cpp
// SCCP
int main() {
    int x = 2, m, n, y, z = 9, c = 1;
    std::cin >> m;
    switch(m) {
        case 2 : x = 2 * c + 5; n = 10; break;
        case 4 : x = 2 * c + 5; n = x - 2; break;
        case 6 : x = 2 * c + 1; n = x + 2; break;
        default : break;
    }
    y = 2 * x + 10;
    if (y <= z + x) {
        //...
    } else {
        z = n + 3 * x;
        switch(z) {
            default : break;
            case 200 : goto end;
            case 300 : exit(0);
        }
        m = n + x;
        end:
        z = x;
        return 0;
    }
}

// SpecSCCP
int main() {
    int x = 2, m, n, y, z = 9, c = 1;
    std::cin >> m;
    switch(m) {
        case 2 : x = 2 * c + 5; n = 10; break;
        case 4 : x = 2 * c + 5; n = x - 2; break;
        case 6 : x = 2 * c + 1; n = x + 2; break;
        default : break;
    }
    y = 2 * x + 10;
    if (y <= z + x) {
        //...
    } else {
        z = n + 3 * x;  // n : Speculative Constant 5
        switch(z) {
            default : break;
            case 200 : goto end;
            case 300 : exit(0);
        }
        m = n + x;  // x : Speculative Constant 7
        end:
        z = x;
        return 0;
    }
}
```

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

SpecSCCP discovers \( n \) & \( x \) as speculative constants.
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 \n4.        IR/LL/test_sccp_onbaseline.ll \n5.        -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

2. # Running HPSSA Transformation followed by Speculative SCCP Pass.
3. $ opt -load build/SCCPSolverTau.so
4.        -load build/HPSSA.so \n5.        -load-pass-plugin=build/SpecSCCP.so \n6.        -passes="specsccp" \n7.        -time-passes -debug-only=specsccp \n8.        IR/LL/test.ll -S -o IR/LL/test_spec_sccp.ll \n9.        -f 2> output/custom_speculative_sccp.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
Speculative Constant: i32 5 = %tau1 = call i32 (...) @llvm.tau.i32(i32 %tau, i32 5, i32 5)
Speculative Constant: i32 7 = %tau7 = call i32 (...) @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.so` ...

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

class YourPGOPass : public PassInfoMixin<YourPGOPass> {
public: PreservedAnalyses run(Function &F,
    FunctionAnalysisManager &AM);
    ... // standard LLVM Pass run() function.
};

PreservedAnalyses YourPGOPass::run(Function &F,
    FunctionAnalysisManager &AM) {
    ...
    HPSSAPass hpssaUtil; // Make a HPSSAPass Object.
    hpssaUtil.run(F, AM); // Call the HPSSAPass::run() function.
    // Rest of the code..
};
```

Muzzammil, Mishra, Lahiri, Pandey, and Roy
Hot Path SSA in LLVM
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

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

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

- each use of a variable is reachable by a single definition; \([\text{SSA-like form}]\)

- **safe interpretation**: \([\text{supports traditional analysis}]\)
  - 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

Muzzammil, Kumar, Lahiri, Pandey, and Roy
Hot Path SSA in LLVM
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
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
  \( (\beta = x_1 \sqcup x_2 \sqcup \cdots \sqcup x_n, \text{i.e. join of speculative args.}) \)

\[
\tau(x_0, x_1, \ldots, x_n) \sqcup \begin{cases} 
T & \text{if } \beta = T \\
\beta & \text{if } \beta \neq T \land x_0 \sqsubseteq \beta \\
\beta^s & \text{otherwise}
\end{cases}
\]

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

Muzzammil, Kumar, Lahiri, Pandey, and Roy

Hot Path SSA in LLVM

pg.38
Introduce new speculative values \( \{..., 1^s, 2^s, ... \} \in \mathcal{C}^S \)

Operation with *speculative* values result in *speculative* results (with same semantics as base operator)

\[
\alpha^s \langle \text{op} \rangle \beta = (\alpha \langle \text{op} \rangle \beta)^s
\]

Transfer function for \( \tau \)-functions

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

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
Presentation Outline

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
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
Optimized SSA forms

\[
x.2 = \Phi(x.0, x.1) \\
a = \text{add i32 var, 1}
\]

... copy propagation breaks the phi congruence property...

Muzzammil, Kumar, Lahiri, Pandey, and Roy Hot Path SSA in LLVM

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

in the above example, copy propagation breaks the \textit{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!**
Presentation Outline

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
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.

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

//===---------- 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;
    }
  }
  ...
}
```
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.

```c++
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;
        }
    }
    ...
```
**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.

```cpp
+ //================= 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.
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.

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

```cpp
+ //===---------- 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;
  +   }
  + }
+ }
```

...
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 argument at strategic positions in the LLVM IR and handles argument allocation for "llvm.tau" intrinsic calls.
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 argument at strategic positions in the LLVM IR and handles argument allocation for " llvm.tau" intrinsic calls.

Key HPSSA Data Structures and Functions:
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 argument at strategic positions in the LLVM IR and handles argument allocation for "llvm.tau" intrinsic calls.

Key HPSSA Data Structures and Functions:

- Hot Path Set using `llvm::BitVector`
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 argument at strategic positions in the LLVM IR and handles argument allocation for "llvm.tau" intrinsic calls.

Key HPSSA Data Structures and Functions:

- **Hot Path Set using `llvm::BitVector`** for maintaining hot paths in the program.
HPSSA Pass: Overview

- `class HPSSA Pass : public PassInfoMixin<HPSSA Pass>`
  - Implemented `llvm::HPSSA Pass` pass using the new LLVM Pass Manager.
  - Function `HPSSA Pass::run(Function &F, ...)` runs over a `llvm::Function` and inserts "llvm.tau" intrinsic calls with speculative and safe argument at strategic positions in the LLVM IR and handles argument allocation for "llvm.tau" intrinsic calls.

Key HPSSA Data Structures and Functions:

- **Hot Path Set using `llvm::BitVector`** for maintaining hot paths in the program.
  
- `std::map<llvm::BasicBlock*, llvm::BitVector> HotPathSet`
class HPSSAPass : public PassInfoMixin<HPSSAPass>

- Implemented \texttt{llvm::HPSSAPass} pass using the new LLVM Pass Manager.
- Function \texttt{HPSSAPass::run(Function \&F, \ldots)} runs over a \texttt{llvm::Function} and inserts "\texttt{llvm.tau}" intrinsic calls with speculative and safe argument at strategic positions in the LLVM IR and handles argument allocation for "\texttt{llvm.tau}" intrinsic calls.

Key HPSSA Data Structures and Functions:

- **Hot Path Set using \texttt{llvm::BitVector}** for maintaining hot paths in the program.
- \texttt{std::map<llvm::BasicBlock*, llvm::BitVector> HotPathSet} used to track hot paths that pass through a given basic block.
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 argument at strategic positions in the LLVM IR and handles argument allocation for "llvm.tau" intrinsic calls.

Key HPSSA Data Structures and Functions:

- Hot Path Set using `llvm::BitVector` for maintaining hot paths in the program.
- `std::map<llvm::BasicBlock*, llvm::BitVector> HotPathSet` used to track hot paths that pass through a given basic block.
- `std::map<std::pair<llvm::BasicBlock *, Value *>, frame> defAcc`
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 argument at strategic positions in the LLVM IR and handles argument allocation for "llvm.tau" intrinsic calls.

Key HPSSA Data Structures and Functions:

- **Hot Path Set using `llvm::BitVector`** for maintaining hot paths in the program.
- `std::map<llvm::BasicBlock*, llvm::BitVector> HotPathSet` used to track hot paths that pass through a given basic block.
- `std::map<std::pair<llvm::BasicBlock *, Value *>, frame> defAcc` keeps track of the hot definitions for a variable that reaches a given basic block.
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 argument at strategic positions in the LLVM IR and handles argument allocation for "llvm.tau" intrinsic calls.

Key HPSSA Data Structures and Functions:

- **Hot Path Set using `llvm::BitVector`** for maintaining hot paths in the program.
- `std::map<llvm::BasicBlock*, llvm::BitVector> HotPathSet` used to track hot paths that pass through a given basic block.
- `std::map<std::pair<llvm::BasicBlock *, Value *>, frame> defAcc` keeps track of the hot definitions for a variable that reaches a given basic block.
- `std::map<llvm::Value *, std::vector<llvm::Value *>> renaming_stack`
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 argument at strategic positions in the LLVM IR and handles argument allocation for "llvm.tau" intrinsic calls.

**Key HPSSA Data Structures and Functions:**

- **Hot Path Set using *llvm::BitVector*** for maintaining hot paths in the program.
  
- **std::map<llvm::BasicBlock*, llvm::BitVector> HotPathSet** used to track hot paths that pass through a given basic block.

- **std::map<std::pair<llvm::BasicBlock *, Value *>, frame> defAcc** keeps track of the hot definitions for a variable that reaches a given basic block.

- **std::map<llvm::Value *, std::vector<llvm::Value *>> renaming_stack** used to store the most “recent” tau definition encountered so far corresponding for a tau variable used later in variable renaming phase in Search(...) function.
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 argument at strategic positions in the LLVM IR and handles argument allocation for "llvm.tau" intrinsic calls.

**Key HPSSA Data Structures and Functions:**

- **Hot Path Set using `llvm::BitVector`** for maintaining hot paths in the program.
- `std::map<llvm::BasicBlock*, llvm::BitVector> HotPathSet` used to track hot paths that pass through a given basic block.
- `std::map<std::pair<llvm::BasicBlock *, Value *>, frame> defAcc` keeps track of the hot definitions for a variable that reaches a given basic block.
- `std::map<llvm::Value *, std::vector<llvm::Value *>> renaming_stack` used to store the most “recent” tau definition encountered so far corresponding for a tau variable used later in variable renaming phase in `Search(...) ` function.
- `HPSSAPass::AllocateArgs(BasicBlock* BB, DomTreeNode& DTN)`
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 argument at strategic positions in the LLVM IR and handles argument allocation for "llvm.tau" intrinsic calls.

Key HPSSA Data Structures and Functions:

- **Hot Path Set using llvm::BitVector** for maintaining hot paths in the program.
- \[\text{std::map<llvm::BasicBlock*, llvm::BitVector> HotPathSet}\] used to track hot paths that pass through a given basic block.
- \[\text{std::map<std::pair<llvm::BasicBlock *, Value *>, frame> defAcc}\] keeps track of the hot definitions for a variable that reaches a given basic block.
- \[\text{std::map<llvm::Value *, std::vector<llvm::Value *>> renaming_stack}\] used to store the most “recent” tau definition encountered so far corresponding for a tau variable used later in variable renaming phase in Search(...) function.
- **HPSSAPass::AllocateArgs(BasicBlock* BB, DomTreeNode& DTN)** handles argument allocation for \(\tau\)-functions inserted.
Out of HPSSA Form.

- A separate pass using the new LLVM Pass Manager.
  
  ```cpp
  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.

\[
x_3 = \tau(x_0, x_1, x_2),
\]

Replace all use of \(x_3\) with \(x_0\).
Out of HPSSA Form.

- A separate pass using the new LLVM Pass Manager.
  
  ```cpp
  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.
Out of HPSSA Form.

A separate pass using the new LLVM Pass Manager.

```cpp
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.
HPSSAPass : Destruction Pass

- Out of HPSSA Form.
  - A separate pass using the new LLVM Pass Manager.
    ```cpp
    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.

\[
x_3 = \tau(x_0, x_1, x_2), \text{ } \tau\text{-function} \quad \quad \quad x_3 = x_0, \text{ } \text{Replace all use of } x_3 \text{ with } x_0.
\]
Presentation Outline

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 open source our work soon and push it to the LLVM “main” branch.
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 open source our work soon and push it to the LLVM “main” branch.

https://github.com/HPSSA-LLVM/HPSSA-LLVM
Thank You!

Questions?
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.
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 class `ValueLattice` 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.
Modified the existing SCCP Pass to add visitTauNode() function which handles the special "llvm.tau" intrinsic instructions used for $\tau$-functions.\textsuperscript{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().

\textsuperscript{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.
Basic blocks from the transformed IR after the SpecSCCP pass with assignSpecValue() calls added.

```llvm
if.else: // Basic Block A ; preds = %sw.epilog
  %tau = call i32 (...) @llvm.tau.i32(i32 %tau8, i32 7, i32 3)
  %tau10 = call i32 (...) @llvm.tau.i32(i32 %tau9, i32 5, i32 5)
  %tau10_spec = call i32 @assgnSpecVal(i32 5) // set speculative value
  %mul11 = mul nsw i32 3, undef
  %add12 = add nsw i32 %tau10_spec, %mul11
  switch i32 %add12, label %sw.default13 [
    i32 200, label %sw.bb14
    i32 300, label %sw.bb15
    ]
if.end: // Basic Block B ; preds = %sw.epilog, %if.else
  %tau11 = call i32 (...) @llvm.tau.i32(i32 %tau8, i32 7, i32 7)
  %tau11_spec = call i32 @assgnSpecVal(i32 7) // set speculative value
  %tau12 = call i32 (...) @llvm.tau.i32(i32 %tau9, i32 5, i32 10)
  %add17 = add nsw i32 undef, %tau11_spec
  store i32 %add17, i32* %m, align 4
  br label %end
```

Muzzammil, Mishra, Lahiri, Pandey, and Roy
Hot Path SSA in LLVM
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

entry:
  \%m = alloca i32, align 4
  \%call = call nonnull align 8 dereferenceable(16)
  ...
  ...
  \%i = load i32, i32* \%m, align 4
  switch i32 \%i, label \%sw.default [
    i32 2, label \%sw.bb
    i32 4, label \%sw.bb1
    i32 6, label \%sw.bb4
  ]

  \%i1 = load i32, i32* \%m, align 4
  switch i32 \%i1, label \%sw.default [
    i32 2, label \%sw.bb
    i32 4, label \%sw.bb1
    i32 6, label \%sw.bb4
  ]

  \%i2 = load i32, i32* \%m, align 4
  switch i32 \%i2, label \%sw.default [
    i32 2, label \%sw.bb
    i32 4, label \%sw.bb1
    i32 6, label \%sw.bb4
  ]

  \%i3 = load i32, i32* \%m, align 4
  switch i32 \%i3, label \%sw.default [
    i32 2, label \%sw.bb
    i32 4, label \%sw.bb1
    i32 6, label \%sw.bb4
  ]
Program in Hot Path SSA Form