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 India Pvt. Ltd.

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.

pg.2

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.

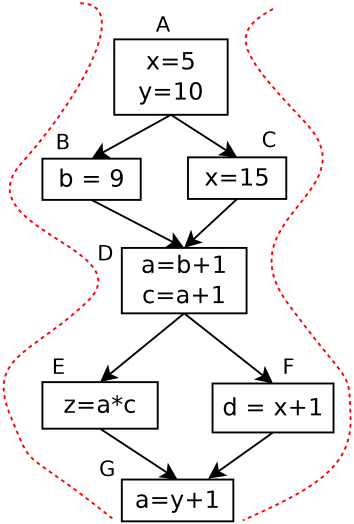
The Hot Path SSA form is based on the following research papers.

- Subhajit Roy and Y.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

Presentation Outline

Presentation Outline

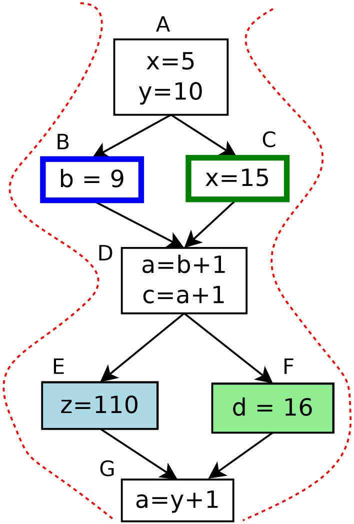
Profile-guided analysis on paths



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**

Why is path-profile-guided analysis hard?

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

Program Representation		Profile Information

`int main(void) {`
`int a = 90;`
`....`
`}`
CFG, AST,
TAC, ASM ...
	⊕	BB1→BB7→BB3→ ... : 7890
		BB1→BB2→BB8→ ... : 9500
		Array,
		HashTable, Map, ...

pg.13

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

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

... and PGO 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);
    NumActiveIncoming++;
    if (beta.isOverdefined())
      break;
  }

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

  beta.mergeInSpec(x0);
  if (x0.isOverdefined())
    TauState.mergeIn(beta);

  ... // futher 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);
    }
  }
  ... // rest of the code.
}
```

pg.19

... and PGO 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){
    SpecV
    beta.
    NumAc
    if (b
      bre
  }

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

  beta.mergeInSpec(x0);
  if (x0.isOverdefined())
    TauState.mergeIn(beta);

  ... // futher 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);
                                         Function();
                                         ("llvm.tau")){
        visitTauNode(I);
      } else {
        visit(I);
      }
    } else {
      visit(I);
    }
  }
  ... // rest of the code.
}
```

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 !

... and PGO 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);
    NumActiveIncomin
    if (beta.isOverd
      break;
  }

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

  beta.mergeInSpec(x0);
  if (x0.isOverdefined())
    TauState.mergeIn(beta);

  ... // futher 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();
                          ID() ==
        Function::lookupIntrinsicID("llvm.tau")){
        visitTauNode(I);
      } else {
        visit(I);
      }
    } else {
      visit(I);
    }
  }
  ... // rest of the code.
}
```

It took us only an afternoon to transform SCCP to SpecSCCP

Presentation Outline

SCCP vs SpecSCCP

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

SCCP

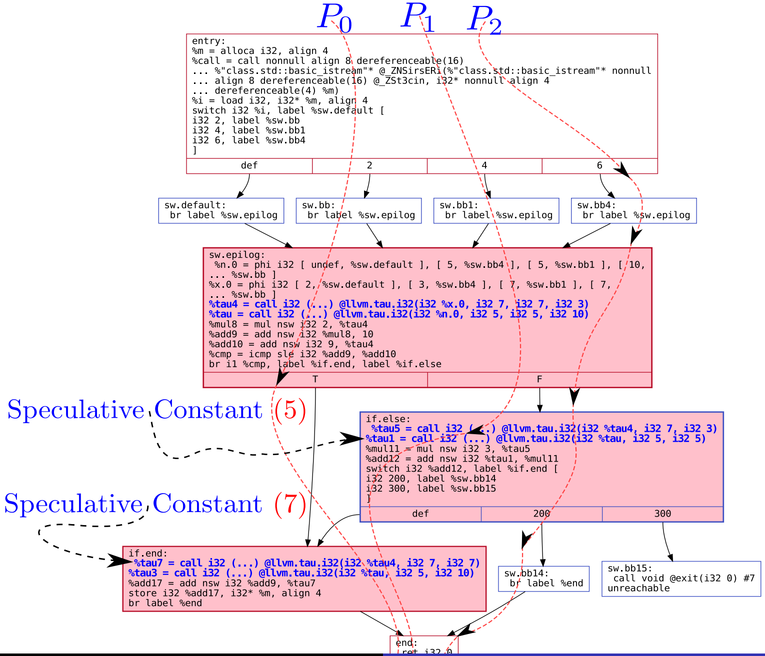
```
int main() {
  int x = 2, m, n, y, z = 9, c = 1;
  std::cin >> m ;
  switch( m ) {
    case 2 : x = 2 * c + 5; n = 10; break;
    case 4 : x = 2 * c + 5; n = x - 2; break;
    case 6 : x = 2 * c + 1; n = x + 2; break;
    default : break;
  }
  y = 2 *
  if ( y <=
    // ..
  } else {
    z = n +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

SpecSCCP

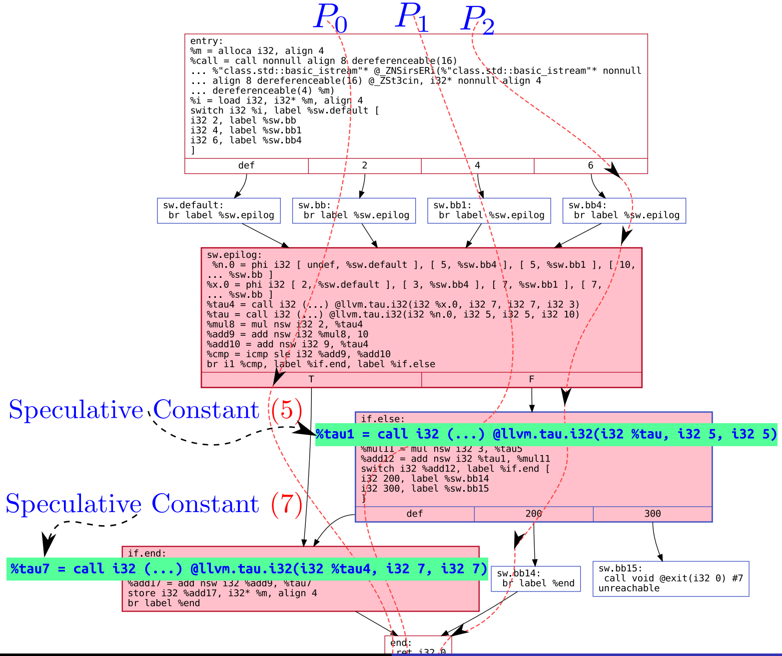
```
int main() {
  int x = 2, m, n, y, z = 9, c = 1;
  std::cin >> m ;
  switch( m ) {
    case 2 : x = 2 * c + 5; n = 10; break;
    case 4 : x = 2 * c + 5; n = x - 2; break;
    case 6 : x = 2 * c + 1; n = x + 2; break;
    default : break;
  }

    // ..
  } 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;
}
```

SpecSCCP discovers n & x as speculative constants.

Legend: Overdefined Real Constants Speculative Constants


P_0 P_1 P_2
entry:
%m = alloca i32, align 4
%call = call nonnull align 8 dereferenceable(16)
... %"class.std::basic_istream"* @_ZNSirsERi(%"class.std::basic_istream"* nonnull
... align 8 dereferenceable(16) @_ZSt3cin, i32* nonnull align 4
... dereferenceable(4) %m)
%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
]
def
2
4
6
sw.default:
br label %sw.epilog
sw.bb:
br label %sw.epilog
sw.bb1:
br label %sw.epilog
sw.bb4:
br label %sw.epilog
sw.epilog:
%n.0 = phi i32 [undef, %sw.default], [5, %sw.bb4], [5, %sw.bb1], [10,
... %sw.bb]
%x.0 = phi i32 [2, %sw.default], [3, %sw.bb4], [7, %sw.bb1], [7,
... %sw.bb]
%tau4 = call i32 (...) @llvm.tau.i32(i32 %x.0, i32 7, i32 7, i32 3)
%tau = call i32 (...) @llvm.tau.i32(i32 %n.0, i32 5, i32 5, i32 10)
%mul8 = mul nsw i32 2, %tau4
%add9 = add nsw i32 %mul8, 10
%add10 = add nsw i32 9, %tau4
%cmp = icmp sle i32 %add9, %add10
br i1 %cmp, label %if.end, label %if.else
T
F
Speculative Constant (5)
if.else:
%tau5 = call i32 (...) @llvm.tau.i32(i32 %tau4, i32 7, i32 3)
%tau1 = call i32 (...) @llvm.tau.i32(i32 %tau, i32 5, i32 5)
%mul11 = mul nsw i32 3, %tau5
%add12 = add nsw i32 %tau1, %mul11
switch i32 %add12, label %if.end [
i32 200, label %sw.bb14
i32 300, label %sw.bb15
]
def
200
300
Speculative Constant (7)
if.end:
%tau7 = call i32 (...) @llvm.tau.i32(i32 %tau4, i32 7, i32 7)
%tau3 = call i32 (...) @llvm.tau.i32(i32 %tau, i32 5, i32 10)
%add17 = add nsw i32 %add9, %tau7
store i32 %add17, i32* %m, align 4
br label %end
sw.bb14:
br label %end
sw.bb15:
call void @exit(i32 0) #7
unreachable
end:
ret i32 0

pg.25


P_0
P_1
P_2
entry:
%m = alloca i32, align 4
%call = call nonnull align 8 dereferenceable(16)
... %"class.std::basic_istream"* @_ZNSirsERi(%"class.std::basic_istream"* nonnull
... align 8 dereferenceable(16) @_ZSt3cin, i32* nonnull align 4
... dereferenceable(4) %m)
%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
]
def
2
4
6
sw.default:
br label %sw.epilog
sw.bb:
br label %sw.epilog
sw.bb1:
br label %sw.epilog
sw.bb4:
br label %sw.epilog
sw.epilog:
%n.0 = phi i32 [undef, %sw.default], [5, %sw.bb4], [5, %sw.bb1], [10,
... %sw.bb]
%x.0 = phi i32 [2, %sw.default], [3, %sw.bb4], [7, %sw.bb1], [7,
... %sw.bb]
%tau4 = call i32 (...) @llvm.tau.i32(i32 %x.0, i32 7, i32 7, i32 3)
%tau = call i32 (...) @llvm.tau.i32(i32 %n.0, i32 5, i32 5, i32 10)
%mul8 = mul nsw i32 2, %tau4
%add9 = add nsw i32 %mul8, 10
%add10 = add nsw i32 9, %tau4
%cmp = icmp sle i32 %add9, %add10
br i1 %cmp, label %if.end, label %if.else
T
F
Speculative Constant (5)
if.else:
%tau1 = call i32 (...) @llvm.tau.i32(i32 %tau, i32 5, i32 5)
%add12 = add nsw i32 %tau1, %mul11
switch i32 %add12, label %if.end [
i32 200, label %sw.bb14
i32 300, label %sw.bb15
]
def
200
300
Speculative Constant (7)
if.end:
%tau7 = call i32 (...) @llvm.tau.i32(i32 %tau4, i32 7, i32 7)
store i32 %add17, i32* %m, align 4
br label %end
sw.bb14:
br label %end
sw.bb15:
call void @exit(i32 0) #7
unreachable
end:

pg.26

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 :
  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)
```

pg.27

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 ...`

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

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

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 ..
  }
```

pg.28

Presentation Outline

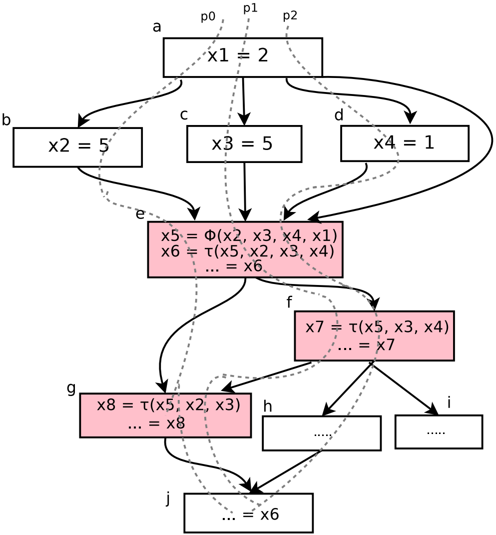
Presentation Outline

Semantics of a ϕ-function

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

Semantics of a τ-function

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



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]

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

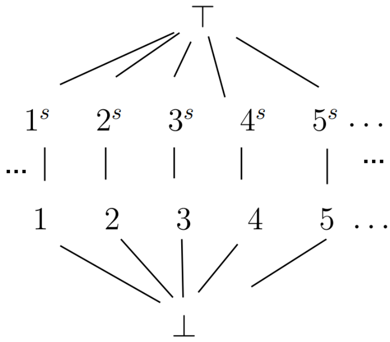
Speculative Sparse Conditional Constant Propagation (SpecSCCP)

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

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



‘

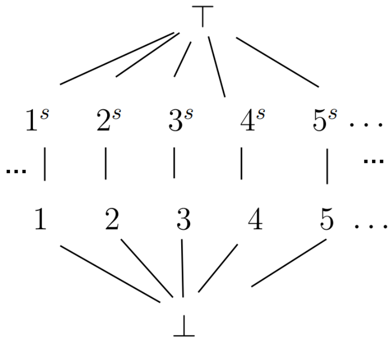
Speculative Sparse Conditional Constant Propagation (SpecSCCP)

- 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 τ-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!

Presentation Outline

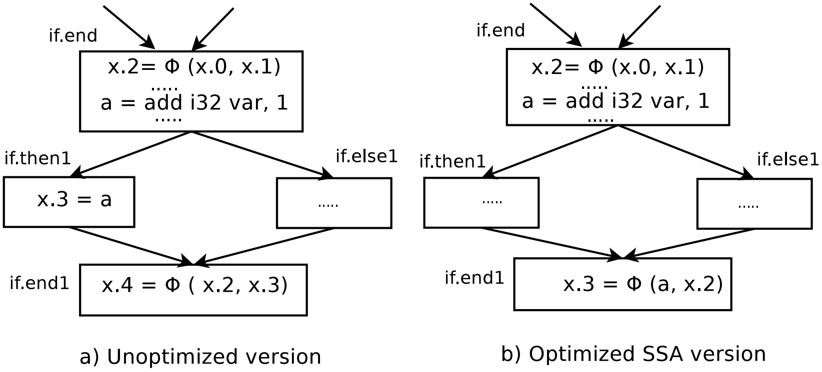
Presentation Outline

pg.40

Brief Algorithm

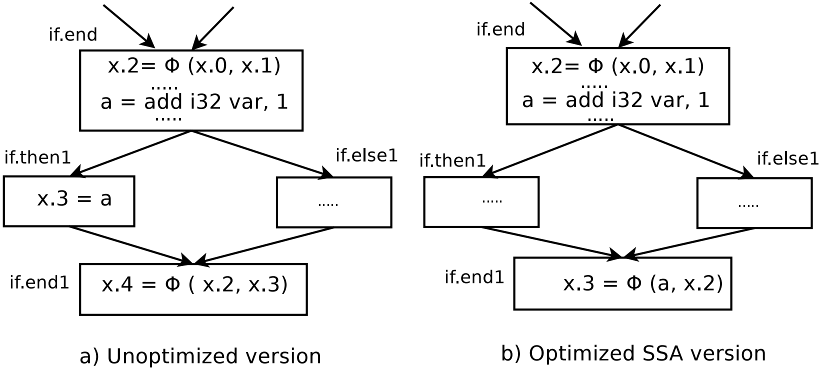
- **Insert τ-functions**
  - Insert at Thermal Frontiers
- **Allocate arguments to τ-functions**
  - path-sensitive traversal through the program to identify definitions that reach τ-functions through hot paths
  - constrains its inspection to only the ϕ-functions and the τ-functions

Optimized SSA forms



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*...

ϕ – *congruence* property

Shreedhar et al. [SAS'99]

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

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

Presentation Outline

What we modified in LLVM Source?

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

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

pg.46

What we modified in LLVM Source?

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

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

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

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

HPSSAPass : Overview

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

pg.48

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 .

pg.51

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.

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.

pg.54

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.

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 τ-functions.[1]

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

HPSSAPass : Details of `run(...)`

- `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.

pg.62

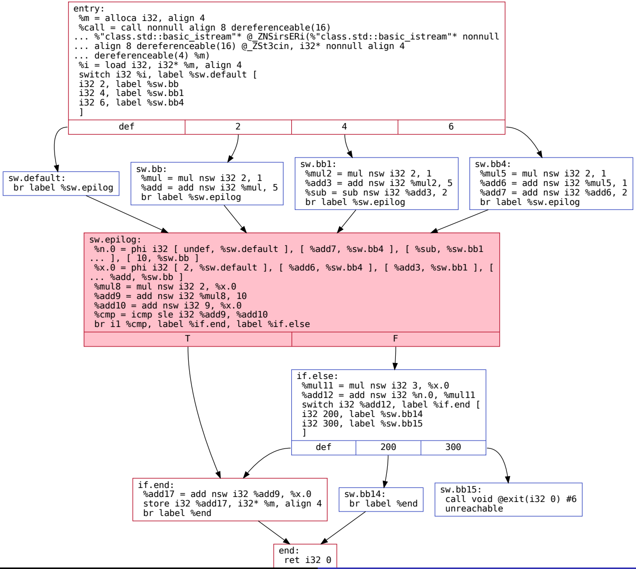
HPSSAPass : Details of `run(...)`

- `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 : Details of `run(...)`

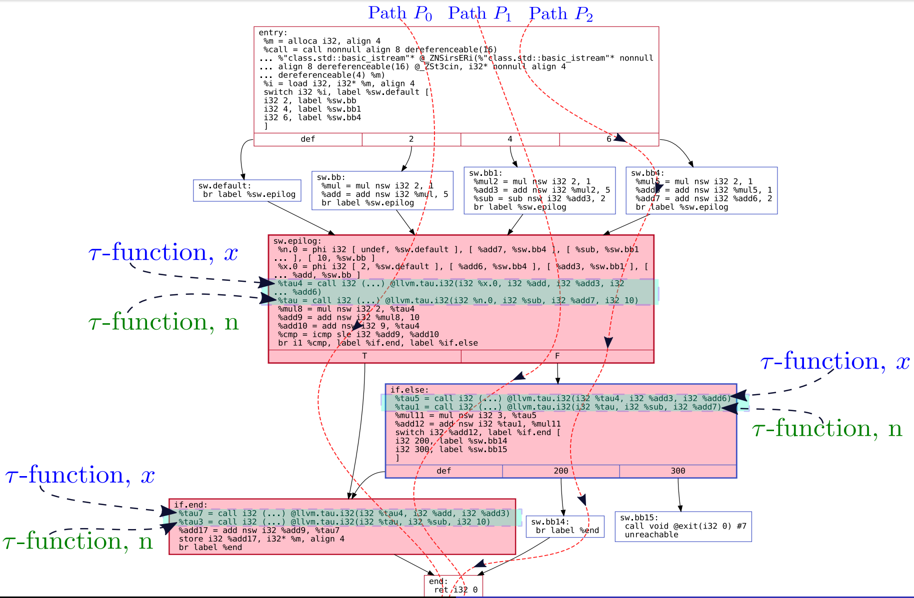
- `HPSSAPass::run(Function &F, FunctionAnalysisManager &AM)`
  - Invokes `HPSSAPass::getProfileInfo()` function to get a compact representation of all the profiled hot paths in the program and then calls `HPSSAPass::getCaloricConnector()` to get all the caloric connectors from the hot path information.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



pg.65

Program in Hot Path SSA Form



pg.66