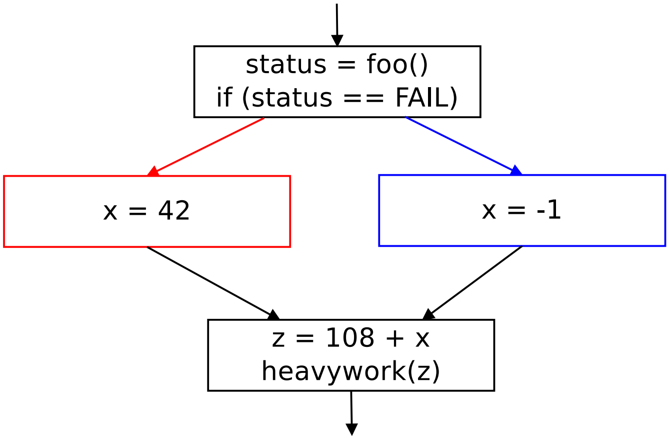
The Hot Path SSA Form: Algorithms and Applications

Subhajit Roy

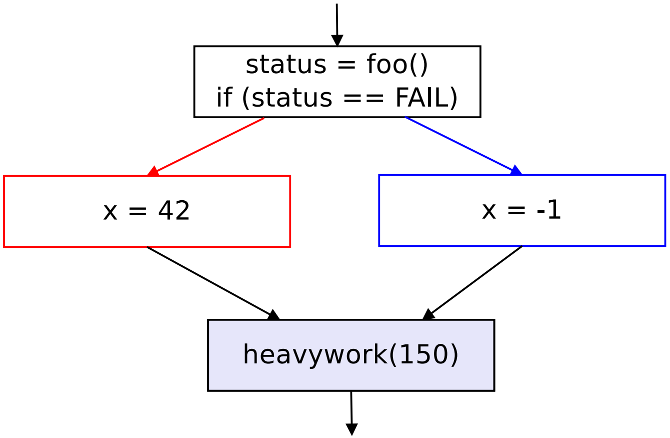
Department of Computer Science and Engineering,
Indian Institute of Technology Kanpur

Outline

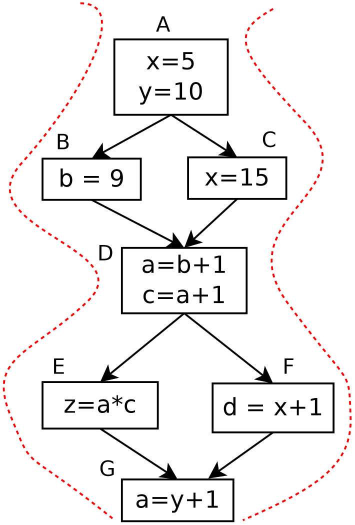
A case for profile-guided optimizations (PGO)



A case for profile-guided optimizations (PGO)



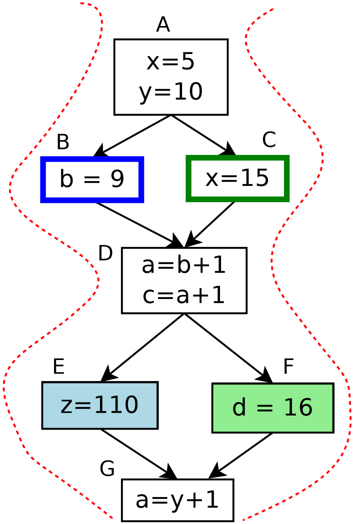
Profile-guided analysis on paths



Summary

- Profile-guided analysis across paths is stronger—can capture correlations between control-flow of basic-blocks
- Collecting path-profiles seems challenging—requires "recording" of a sequence of basic-blocks

Profile-guided analysis on paths



Summary

- Profile-guided analysis across paths is stronger—can capture correlations between control-flow of basic-blocks
- Collecting path-profiles seems challenging—requires "recording" of a sequence of basic-blocks

Profiling acyclic paths

Ball-Larus Acyclic Profiling [Ball & Larus, MICRO'96]

- Core idea: assign an identifier to each path, that can be calculated efficiently at runtime
- Record frequencies against these identifiers (instead of a sequence of node identifiers)

Profiling acyclic paths

Ball-Larus Acyclic Profiling [Ball & Larus, MICRO'96]

- Core idea: assign an identifier to each path, that can be calculated efficiently at runtime
- Record frequencies against these identifiers (instead of a sequence of node identifiers)

Capturing still longer paths (k-iteration paths)

- Allows capturing correlations across loop iterations [Roy & Srikant, CGO'09]; a generalization of the Ball-Larus algorithm
- Subsequent work by other groups [D'Elia & Demetrescu, OOPSLA'13]; uses a prefix forest to record BL paths

Profile-guided analyses

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

Profile-guided analyses

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

Profile-guided analyses

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

Profile-guided analyses to optimizations

- **Speculation**: *check and retry*
  - eg. value speculation
  - Can impact code size (significant impact w/o speculation support in hardware)

Profile-guided analyses to optimizations

- **Speculation**: *check and retry*
  - eg. value speculation
  - Can impact code size (significant impact w/o speculation support in hardware)
- **Compensation code**
  - eg. superblock scheduling
  - Can impact code size

Profile-guided analyses to optimizations

- **Speculation**: *check and retry*
  - eg. value speculation
  - Can impact code size (significant impact w/o speculation support in hardware)
- **Compensation code**
  - eg. superblock scheduling
  - Can impact code size
- **Error resilient modules**
  - modules for statistical summarization on samples, generate data for ML models
  - No impact on code size

Profile-guided analyses to optimizations

- **Speculation**: *check and retry*
  - eg. value speculation
  - Can impact code size (significant impact w/o speculation support in hardware)
- **Compensation code**
  - eg. superblock scheduling
  - Can impact code size
- **Error resilient modules**
  - modules for statistical summarization on samples, generate data for ML models
  - No impact on code size
- **Optimizations that don't impact correctness**
  - eg. register allocation
  - No impact on code size

Outline

Why is path-profile-guided analysis hard?

disparate data-structures: program + profile

```
int main ( )
{
    int a[][4]= {{8,2,4}, {1,2,4}, {7,5,4}, {2,5,1}};
    for (int i=0; i<4; 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 ) ; // bug
printf("f(%d,%d,%d,%d,%d,%d) = %d\n", x,y,z,t1,t2,t3,(t3 && !t2));
if ( 1 ) max = z;
//if ( t3 && t2 ) max = z;
if ( t1 && !t3 ) 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.
  - ...

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

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

...providing a **common representation** for both "traditional" as well as profile-guided analysis and optimizations

The Hot Path SSA Form (HPSSA)

Semantics of a ϕ-function

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

The Hot Path SSA Form (HPSSA)

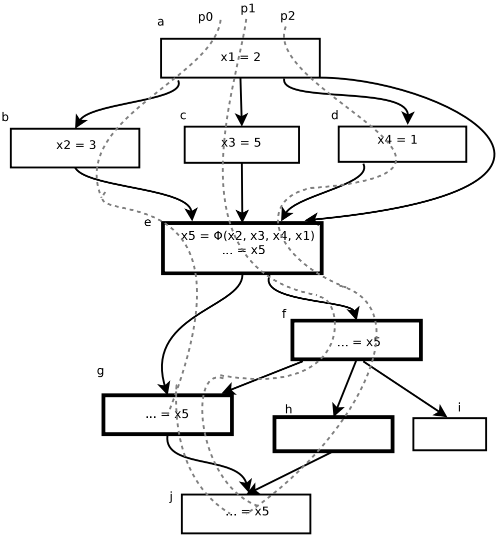
Semantics of a ϕ-function

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

Semantics of a τ-function

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

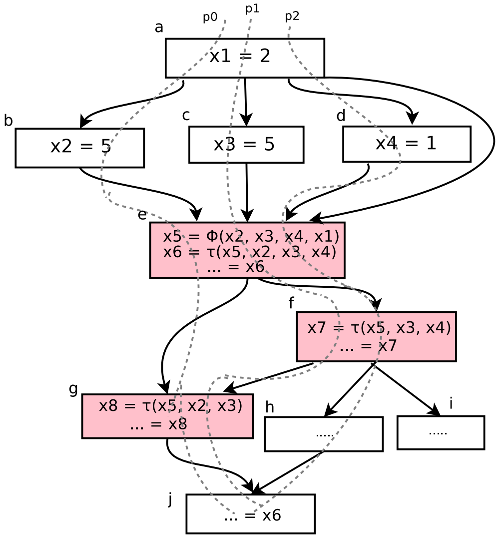
The SSA form



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

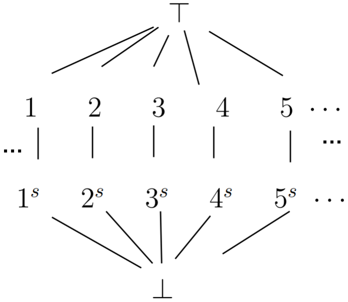
Speculative Sparse Conditional Constant Propagation (SSCCP)

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

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

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

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



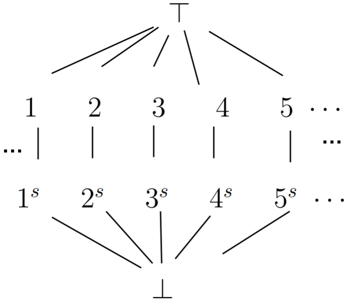
Speculative Sparse Conditional Constant Propagation (SSCCP)

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

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

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

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



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

Speculative Sparse Conditional Constant Propagation (SSCCP)

Table 1. Speculative Constants discovered by the SSCP algorithm. ('~' indicates *almost*; **grp**, **prg**, & **src** refer to inputs **graphic**, **program** & **source** respectively).

Program	Inpt	Variable Uses		Expression Uses		Total		
		Uses	HitRt	Uses	HitRt	Hits	Misses	HitRt
181.mcf	-	33110	100.00	49665	100.00	82775	0	100.00
175.vpr	-	6938074	100.00	8110837	100.00	15048911	0	100.00
164.gzip	grp	26592	100.00	5	100.00	26597	0	100.00
	prg	17412	100.00	5	100.00	17417	0	100.00
	src	4721	99.98	5	100.00	4725	1	99.98
197.parser	-	165970964	~100.00	340	97.94	165970861	443	~100.00
256.bzip2	grp	132106650	~100.00	938	76.97	132107372	216	~100.00
	prg	100819492	~100.00	6576416	15.67	101849942	5545966	94.84
	src	108134316	~100.00	5256006	17.94	109077366	4312956	96.20

Speculative Sparse Conditional Constant Propagation (SSCCP)

Table 1. Speculative Constants discovered by the SSCP algorithm. ('~' indicates *almost*; **grp**, **prg**, & **src** refer to inputs **graphic**, **program** & **source** respectively).

Program	**Inpt**	**Variable Uses**		**Expression Uses**		**Total**		
		Uses	**HitRt**	**Uses**	**HitRt**	**Hits**	**Misses**	**HitRt**
181.mcf	-	33110	100.00	49665	100.00	82775	0	100.00
175.vpr	-	6938074	100.00	8110837	100.00	15048911	0	100.00
164.gzip	grp	26592	100.00	5	100.00	26597	0	100.00
	prg	17412	100.00	5	100.00	17417	0	100.00
	src	4721	99.98	5	100.00	4725	1	99.98
197.parser	-	165970964	~100.00	340	97.94	165970861	443	~100.00
256.bzip2	grp	132106650	~100.00	938	76.97	132107372	216	~100.00
	prg	100819492	~100.00	6576416	15.67	101849942	5545966	94.84
	src	108134316	~100.00	5256006	17.94	109077366	4312956	96.20

Optimized on "train" inputs, results produced on "ref" inputs... Shows the value of profile-guided analysis for code understanding and optimizations...

Profile-Guided Register Allocation [Jain et al., HiPC'16]

```
# BB#4:                              # %if.then
                                     #   in Loop: Header=B
        addl    %edi, 32(%esp)       # 4-byte Folded Spill |
        addl    $20, %esi
        addl    $117, %ebp
        jmp     .LBB0_6
        .align  16, 0x90
.LBB0_5:                             # %if.else
                                     #   in Loop: Header=B
        addl    $101, 40(%esp)       # 4-byte Folded Spill
                                     >
        subl    40(%esp), %ecx       # 4-byte Folded Reloa |
                                     >
        addl    $33, %edi
        addl    $-11, %ebp
```

```
# BB#4:                              # %if.then
                                     #   in Loop: Header=B
        addl    %edi, %ecx
        addl    $20, %esi
        addl    $117, %ebp
        jmp     .LBB0_6
        .align  16, 0x90
.LBB0_5:                             # %if.else
                                     #   in Loop: Header=B
        addl    $101, 40(%esp)       # 4-byte Folded Spill
        movl    36(%esp), %edx       # 4-byte Reload
        subl    40(%esp), %edx       # 4-byte Folded Reloa
        movl    %edx, 36(%esp)       # 4-byte Spill
        addl    $33, %edi
        addl    $-11, %ebp
```

.LBB0_5 is hot: 40(%esp) is cached in %ecx	BB#4 is hot: 32(%esp) is cached in %ecx

ILP based register allocator

- Implements integer linear programming based profile-guided register allocation
- Base allocator (without profile information) is simplified version of [Goodwin and Wilken, 1996]

Base Allocator

$$\mathbf{min} \sum_{v \in V, l \in L} S_{v,l} \, . \, x_{v,l,\sigma}, \quad \text{(minimize spills)} \quad \textit{such that}$$

$$\forall_{v \in V} \forall_{l \in L} \sum_{r \in R} x_{v,l,r} = 1 \quad \text{(a variable to at least one location)}$$

$$\forall_{r \in R - \sigma} \forall_{l \in L} \sum_{v \in V} x_{v,l,r} \leq 1 \quad \text{(a register holds at most one variable)}$$

Cost for τ-functions

Modelling τ-functions

The cost for $t = \tau(safe, t_1, t_2, ..., t_n)$:

$$TauCost = \sum_{i=1}^{i=n} (loc(t, t_i) * freq(t_i))$$

where $loc(t, t_i)$ is 1 for the locations where t and t_i are different, else 0.

The objective function

The objective function for register allocation over HPSSA is modified to include the *TauCost* discussed above.

The objective fuction

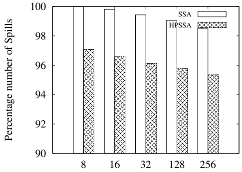
$$\textbf{min} \sum_{v \in V, l \in L} S_{v,l} \cdot x_{v,l,\sigma} + \sum_{i=1}^{i=m} TauCost_i \qquad (1)$$

where, $TauCost_i$ is the cost for the i^{th} τ-function (and m is the total number of τ-functions in the program), and

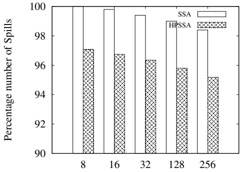
$$S_{v,l} = \begin{cases} 100, & \text{if v has further uses along hot path;} \\ 15, & \text{if v has further uses only along cold path;} \\ 0, & \text{if v has no further use.} \end{cases}$$

Almost a trivial modification to the objective function...

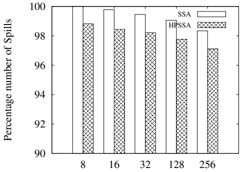
Percentage of spills for different register sizes



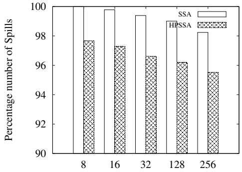
(a) gzip



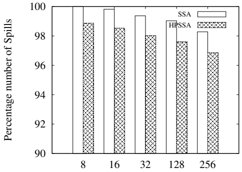
(b) bzip2



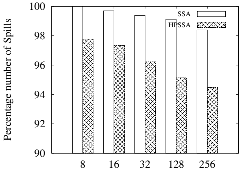
(c) mcf



(d) Crafty



(e) GAP



(f) Parser

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 (θ) 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 (θ) 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 δ 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 δ is not killed along that path. A definition δ 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 δ is not killed at least along one cold path from n_1 to n_2.

Inserting τ — *functions*

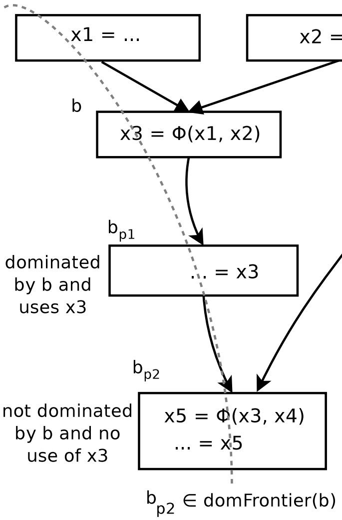
Necessary condition for τ-functions

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

Inserting τ-functions



Inserting τ − *functions*

Definition 4. Thermal Frontier (TF)

For definition d defined at a node u reaching v, v is in the Thermal Frontier of (u,d), $v \in TF(u, d)$, iff

1. the node v is also exposed to a reaching definition d' defined at a node $u \notin Dom(w)$ (w not dominated by u)
2. $\theta(u \rightsquigarrow v) \neq \theta(w \rightsquigarrow v)$
3. v is the first node on the paths $u \rightsquigarrow v$ and $w \rightsquigarrow v$ that satisfies the above properties.

Inserting τ — *functions*

Definition 4. Thermal Frontier (TF)

For definition d defined at a node u reaching v, v is in the Thermal Frontier of (u,d), $v \in TF(u, d)$, iff

1. the node v is also exposed to a reaching definition d' defined at a node $u \notin Dom(w)$ (w not dominated by u)
2. $\theta(u \rightsquigarrow v) \neq \theta(w \rightsquigarrow v)$
3. v is the first node on the paths $u \rightsquigarrow v$ and $w \rightsquigarrow v$ that satisfies the above properties.

Theorem

For a set of visible definitions of a variable x at a set of nodes κ, τ-statements are only required at the Iterated Thermal Frontier ITF^x for variable x.

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

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

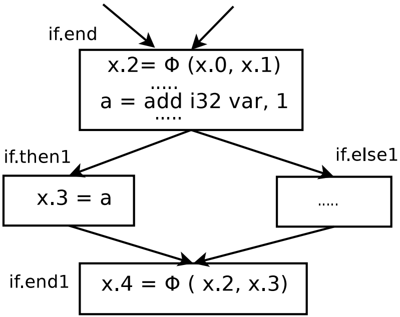
HPSSA over SSA

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

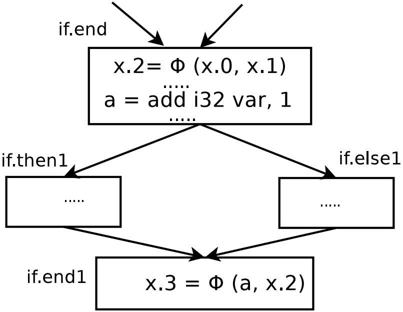
A naïve attempt: Mimic [Roy et al.]

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

Optimized SSA forms



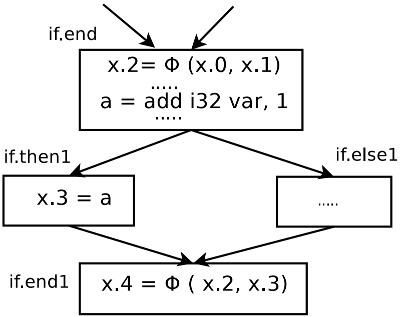
a) Unoptimized version



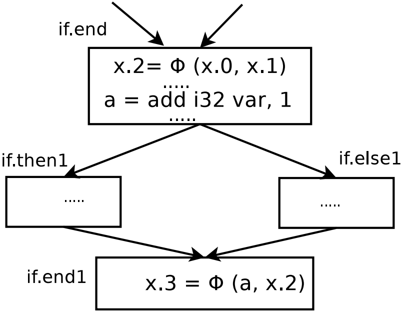
b) Optimized SSA version

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*...

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!**

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

Allocating τ-function arguments

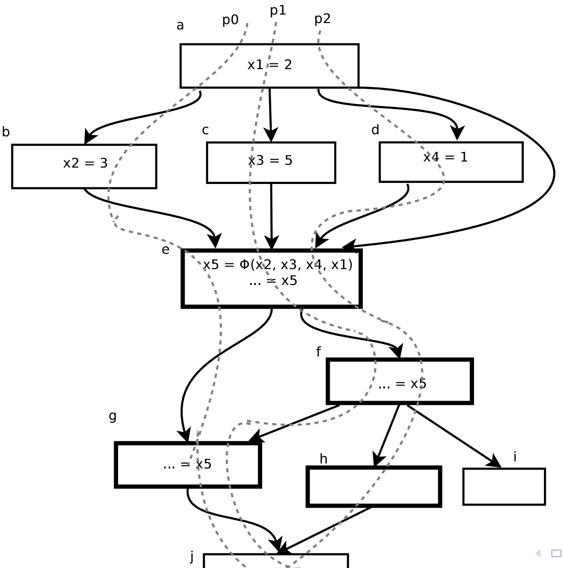
Uses a path-sensitive stack: **phiStack**

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

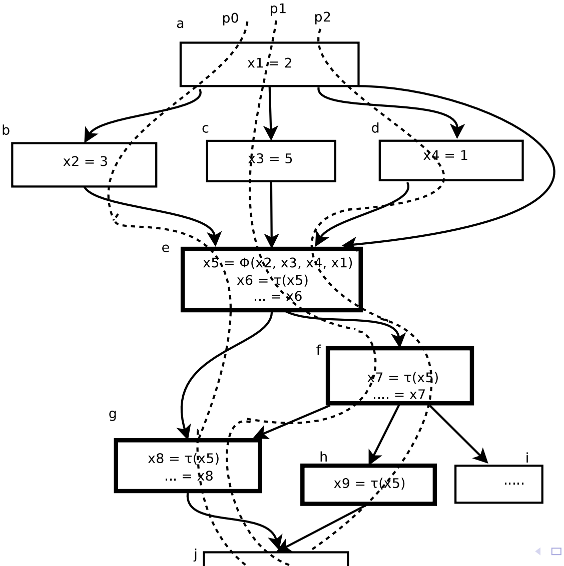
High-level algorithm

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

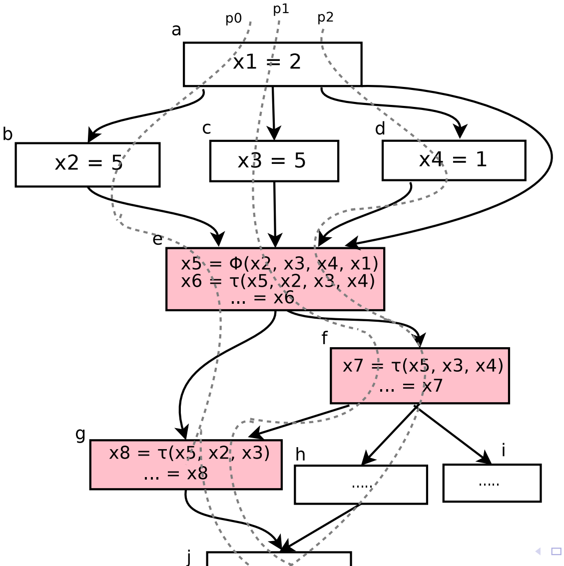
SSA program



τ-functions inserted



τ-arguments allocated



Outline

Conclusions

- We propose a new algorithm that converts SSA programs to HPSSA. Being built over the SSA form, our new algorithm is much more suited for compiler frameworks that offer an SSA-based intermediate representation. This algorithm is also simpler and more efficient than the existing algorithm that builds HPSSA programs from non-SSA programs.
- Our algorithm is capable of operating on optimized SSA forms (resulting from compiler optimizations on the SSA form) that do not satisfy the *Phi Congruence Property*.
- We design an ILP-based path-profile guided register allocation. We provide experimental evaluation of the performance of our approach on eight benchmarks of SPEC CINT2000 benchmark suite.

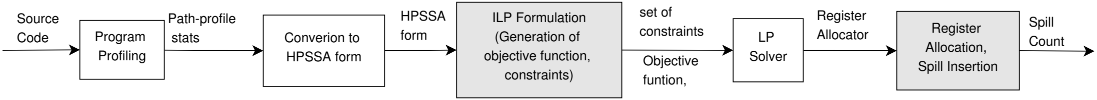
Questions

Questions?

dasd

- 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.

dasd


Source Code
Program Profiling
Path-profile stats
Converion to HPSSA form
HPSSA form
ILP Formulation (Generation of objective function, constraints)
set of constraints
Objective funtion,
LP Solver
Register Allocator
Register Allocation, Spill Insertion
Spill Count

dasd

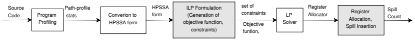
Calculate all caloric connectors in the CFG of the program.
isInserted(I,b)=**false** forall instruction I and basic-block b
Traverse CFG in topological order, performing the following:
for all ϕ-function instruction $I \in$ basic-block b **do**
    for all hot path p passing through b **do**
        for b_p= next basic-blocks on p **till** $b_p \notin$ DomFront(b) **do**
            if $b_p \in$ CaloricConnectors $\wedge \neg$*isInserted*(I, b_p) **then**
                insert a τ-function $x_{new} = \tau(x_{phi})$ corresponding to the ϕ-function I: $x_{phi} = \phi(\dots)$ after all ϕ-function statements in block b_p
                isInserted(I, b_p) = **true**
            end if
        end for
    end for
end for

dasd

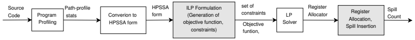
While processing a basic block b,

for all ϕ-function I: $x_o = \phi(x_1, x2, \ldots, x_n) \in b$ **do**
  for all x_i for which $\Omega_{x_i}(b)$ exists **do**
    $phiStack_{x_i}$.**push**($\Omega_{x_i}(b)$, b)
  end for
  if $b \in$ IncubationNode **then**
    frame = [ ]
    for all $(x_i, \xi_i) \in phiStack_{x_i}(b).Top$ **do**
      frame = frame.**add**($x_i, \xi_i \cup IncubPaths(b)$)
    end for
    for all arguments x_i in ϕ-function I **do**
      if x_i is a concrete defn. $\wedge$ $b_p \to b$ an incoming edge **then**
        frame = frame.**add**($x_i, pathSet(b_p \to b) \cup$ IncubPaths(b))
      end if
      if x_i is a ϕ-argument corresponding to a hot backedge **then**
        frame = frame.**add**(x_i,IncubPaths(b))
      end if
    end for
    $phiStack_x$.**push**(frame, b)
  end if
end for
for all τ-function instructions $I \in b$ **do**
  for all $(x_i, \xi_i) \in phiStack_{x_i}(b).Top$ **do**
    if $\xi_i \cap pathSet(b) \neq \emptyset$ **then**
      add (x_i, $\xi_i \cap pathSet(b)$) to speculative arguments of I
    end if
  end for
end for
for all outgoing edge $b \to b_s$ **do**
  if b_s is a join node **then**
    for all $(x_i, \xi_i) \in phiStack_{x_i}.Top$ **do**
      if $\xi_i \cap pathSet(b \to b_s) \neq \emptyset$ **then**
        $\Omega_{x_i}(b_s) = \xi_i \cap pathSet(b \to b_s)$
      end if
    end for
  end if
end for
Make a recursive call on the children of node b in the dominator tree, traversing them in topological of the nodes in the control-flow graph order.

dasd


Source Code
Program Profiling
Path-profile stats
Converion to HPSSA form
HPSSA form
ILP Formulation (Generation of objective function, constraints)
set of constraints
Objective funtion,
LP Solver
Register Allocator
Register Allocation, Spill Insertion
Spill Count

dasd


Source Code
Program Profiling
Path-profile stats
Converion to HPSSA form
HPSSA form
ILP Formulation (Generation of objective function, constraints)
set of constraints
Objective funtion,
LP Solver
Register Allocator
Register Allocation, Spill Insertion
Spill Count

dasd