Generalized Points-to Graph: A New Abstraction of Memory in Presence of Pointers

Pritam Gharat

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

September 2018

Disclaimer

Some of the slides in Introduction are borrowed from CS618 course conducted at IIT Bombay

Outline

- Introduction
- Motivation
- Generalized Points-to Graph (GPG) as a uniform representation for memory and memory transformer
- An Overview of GPG optimizations
- Implementation and Empirical Measurements
- Future Work

Part I

Introduction

Pointer Analysis

- Answers the following questions for indirect accesses:
  - Which data is read? $x = *y$
  - Which data is written? $*x = y$
  - Which procedure is called? $p()$ or $x \rightarrow f()$
- Computationally intensive analyses are ineffective with imprecise points-to analysis, e.g., model checking, interprocedural analyses

Pointer Analysis: Precision versus Scalability

Ideally, an analysis should be

- Sound
- Precise
- Scalable

Pointer Analysis: Precision versus Scalability

Ideally, an analysis should be

- Sound
- Precise
- Scalable

The state of the art points-to analyses say that precision and scalability do not go hand-in-hand

Pointer Analysis: Precision versus Scalability

Ideally, an analysis should be

- Sound
- Precise
- Scalable

The state of the art points-to analyses say that precision and scalability do not go hand-in-hand

Several approximations trade-off precision for scalability

Pointer Analysis: Precision versus Scalability

Ideally, an analysis should be

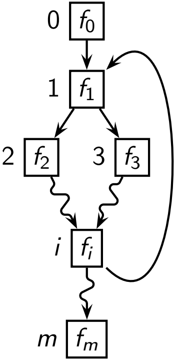
- Sound
- Precise
- Scalable

Main factors enhancing the precision of an analysis

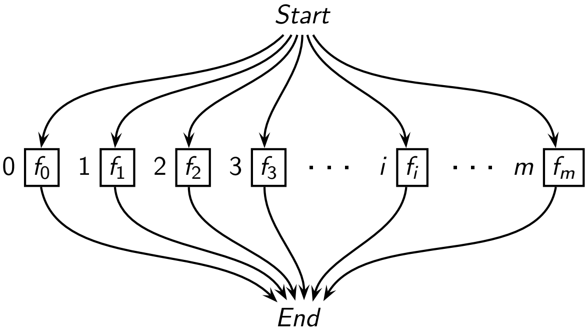
- Flow sensitivity
- Context sensitivity

Flow Sensitivity Vs. Flow Insensitivity

Flow Sensitive

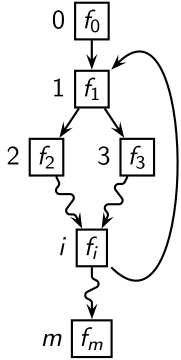


Flow Insensitive

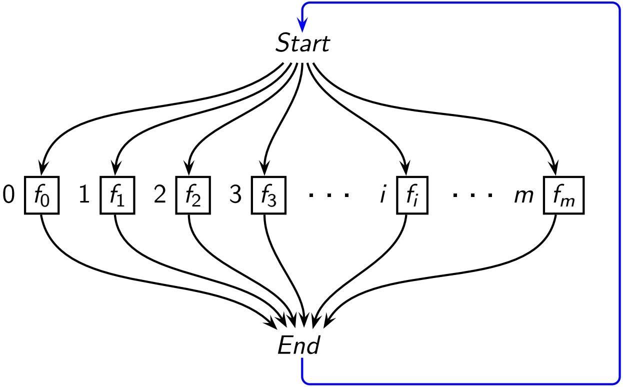


Flow Sensitivity Vs. Flow Insensitivity

Flow Sensitive



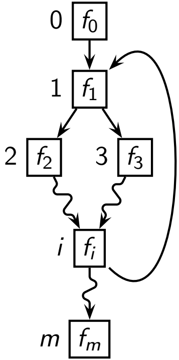
Flow Insensitive



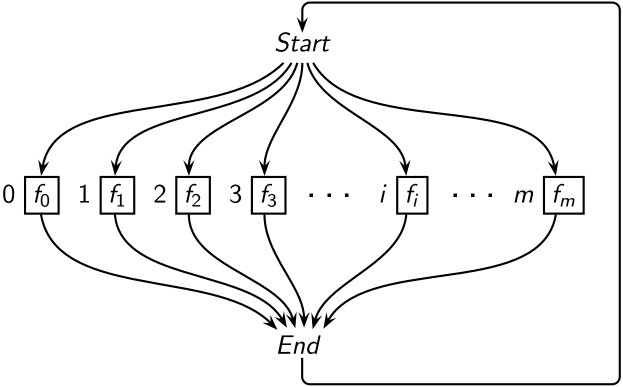
Assumption: Statements can be executed in any order

Flow Sensitivity Vs. Flow Insensitivity

Flow Sensitive

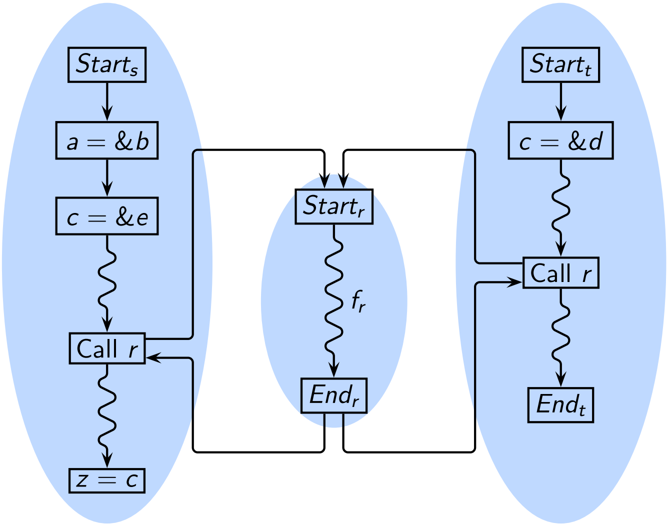


Flow Insensitive

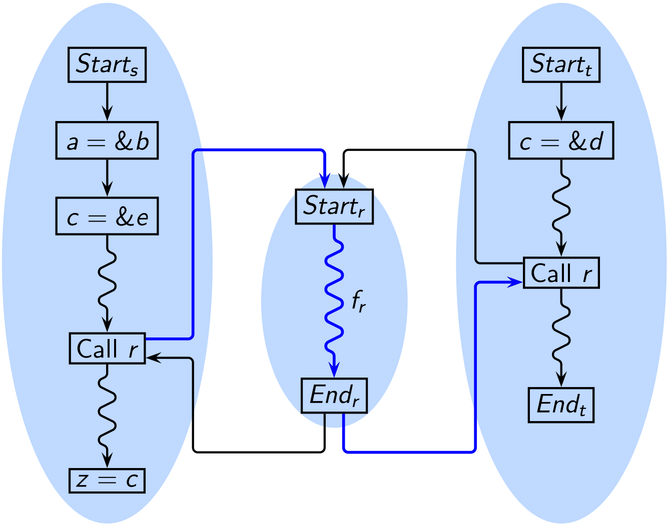


Arbitrary compositions of flow functions in any order
$\Rightarrow$ *Flow insensitivity*

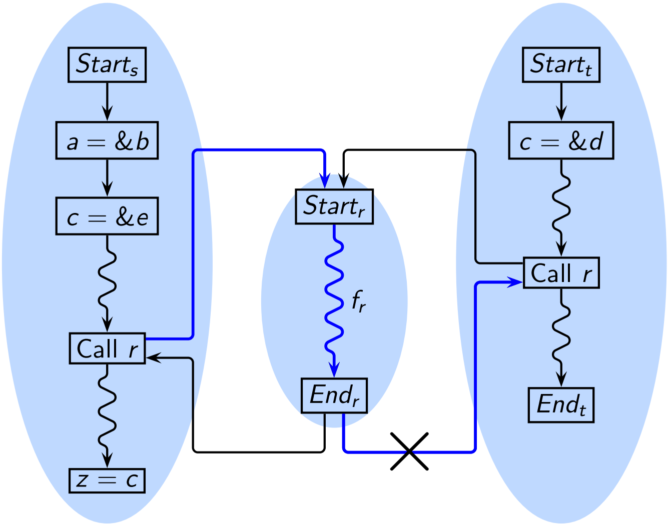
Context Sensitivity Vs. Context Insensitivity



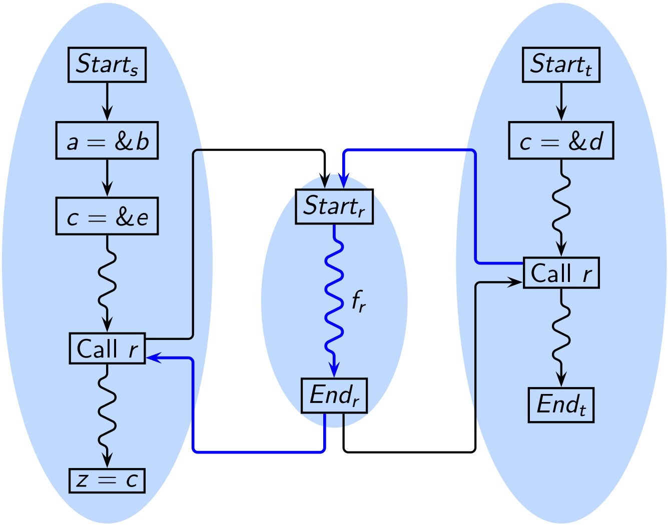
Context Sensitivity Vs. Context Insensitivity



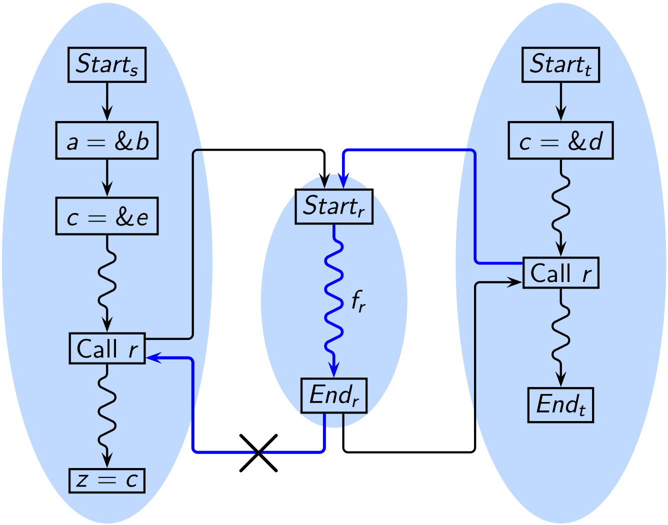
Context Sensitivity Vs. Context Insensitivity



Context Sensitivity Vs. Context Insensitivity



Context Sensitivity Vs. Context Insensitivity



The Goal of My Ph.D. Work

Most approaches begin with a scalable method and try to increase the precision

The Goal of My Ph.D. Work

Most approaches begin with a scalable method and try to increase the precision

My approach begins with a precise method and tries to increase the scalability

The Goal of My Ph.D. Work

Improving the scalability of pointer analysis without losing precision

Most approaches begin with a scalable method and try to increase the precision

My approach begins with a precise method and tries to increase the scalability

The Goal of My Ph.D. Work

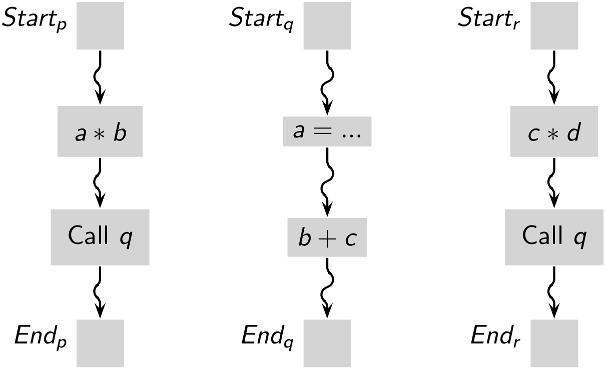
Improving the scalability of pointer analysis without losing precision

GPG-based approach hinges on the following observations:

- Flow- and context-sensitive points-to information is small and sparse even for large programs
- The real killer of scalability in program analysis is not the amount of data that an analysis computes but the amount of control flow that the data may be subjected to in search of precision.
- It is the control flow that has the effect of introducing an exponential multiplier in the size of the data
- If control flow can be minimized carefully, there is a good chance of scaling a program analysis without compromising on precision

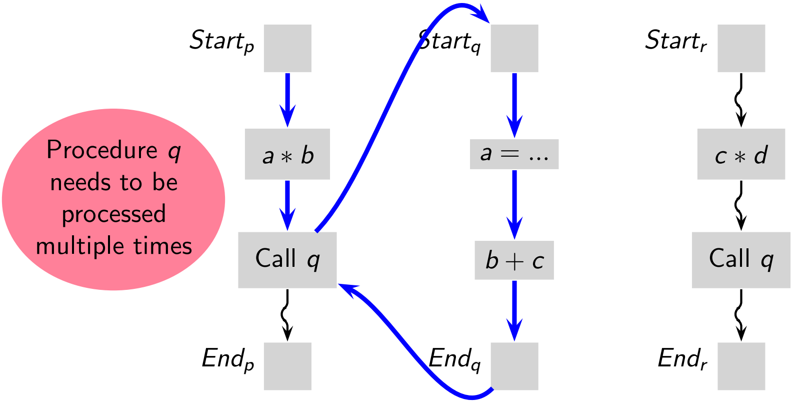
Top-down Vs. Bottom-up Interprocedural Analysis

Top-down Analysis for Available Expressions Analysis



Top-down Vs. Bottom-up Interprocedural Analysis

Top-down Analysis for Available Expressions Analysis



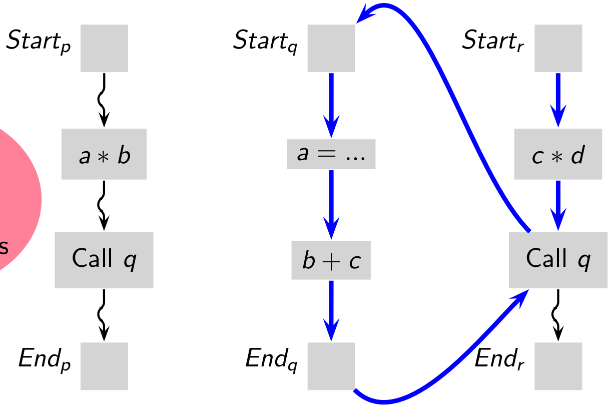
Expression $b + c$ is available in procedure p

Expression $a * b$ is not available in procedure p

Top-down Vs. Bottom-up Interprocedural Analysis

Top-down Analysis for Available Expressions Analysis

Procedure q needs to be processed multiple times



$Start_p$

$a * b$

Call q

End_p

$Start_q$

$a = \dots$

$b + c$

End_q

$Start_r$

$c * d$

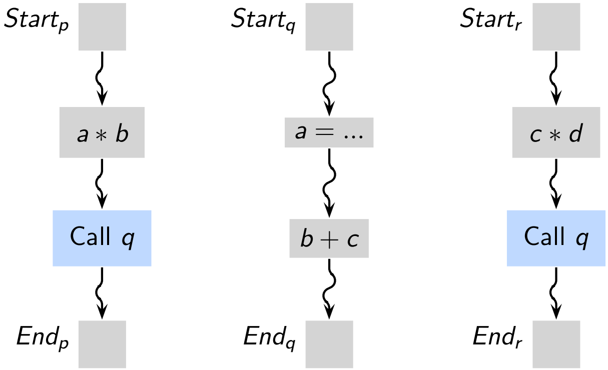
Call q

End_r

Expressions $b + c$ and $c * d$ are available in procedure r

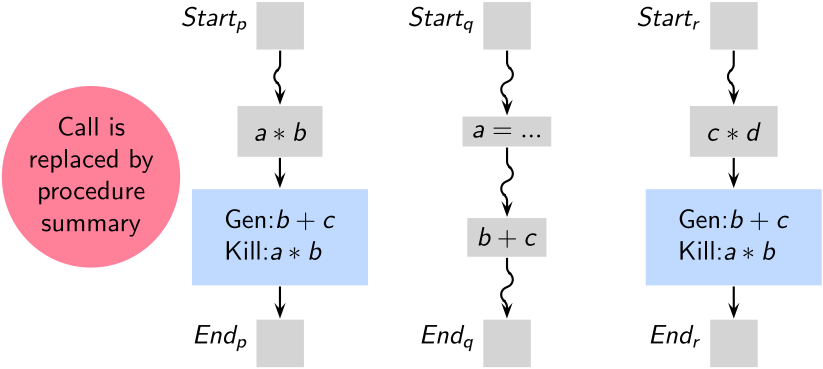
Top-down Vs. Bottom-up Interprocedural Analysis

Bottom-Up Analysis for Available Expressions Analysis



Top-down Vs. Bottom-up Interprocedural Analysis

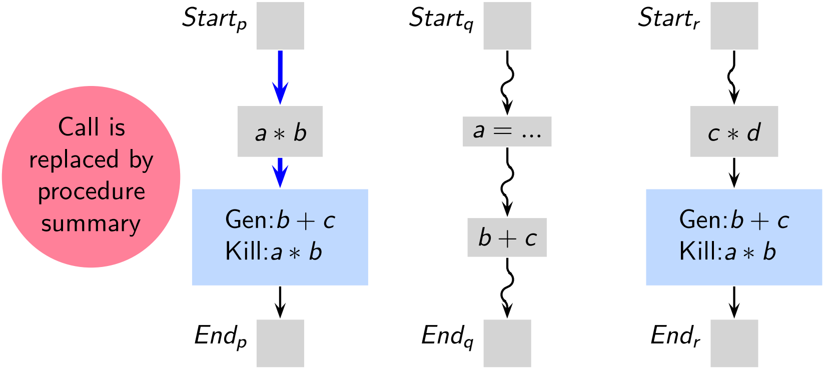
Bottom-Up Analysis for Available Expressions Analysis



Using procedure summary of *g* at call sites

Top-down Vs. Bottom-up Interprocedural Analysis

Bottom-Up Analysis for Available Expressions Analysis

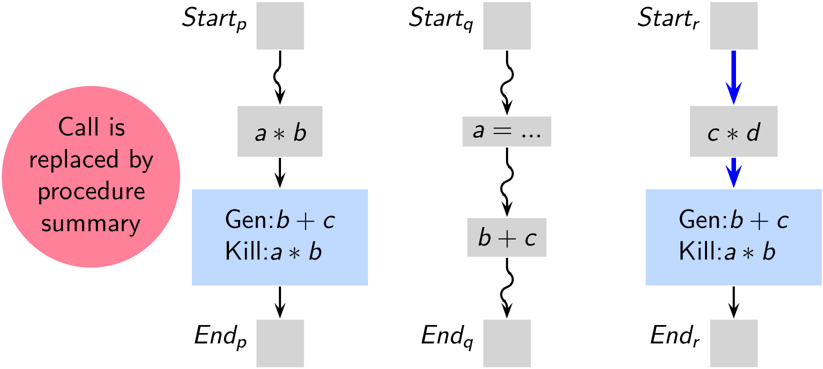


Expression $b + c$ is available in procedure p

Expression $a * b$ is not available in procedure p

Top-down Vs. Bottom-up Interprocedural Analysis

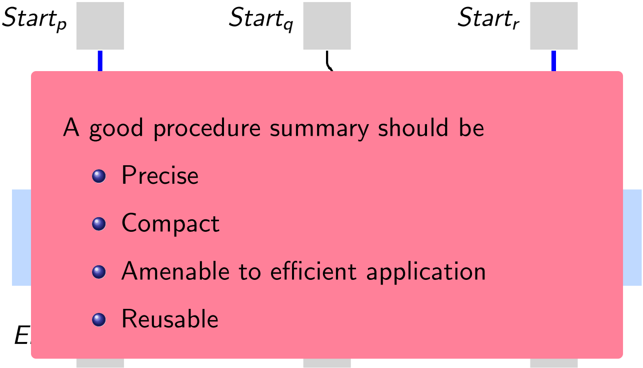
Bottom-Up Analysis for Available Expressions Analysis



Expressions $b + c$ and $c * d$ are available in procedure r

Top-down Vs. Bottom-up Interprocedural Analysis

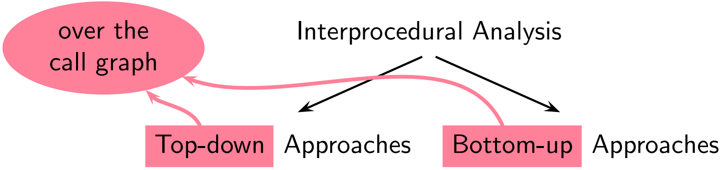
Bottom-Up Analysis for Available Expressions Analysis



A good procedure summary should be

- Precise
- Compact
- Amenable to efficient application
- Reusable

Interprocedural Pointer Analysis


over the call graph

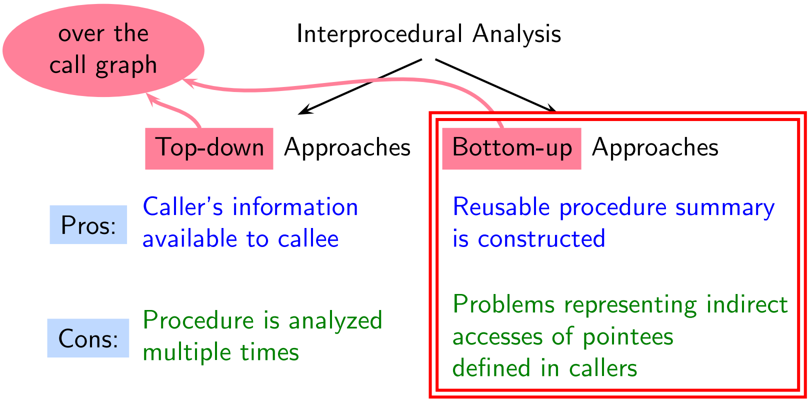
Interprocedural Analysis

Top-down Approaches

Bottom-up Approaches

Pros:	Caller's information available to callee	Reusable procedure summary is constructed
Cons:	Procedure is analyzed multiple times	Problems representing indirect accesses of pointees defined in callers

Interprocedural Pointer Analysis



over the call graph

Interprocedural Analysis

Top-down Approaches

Bottom-up Approaches

Pros: Caller's information available to callee

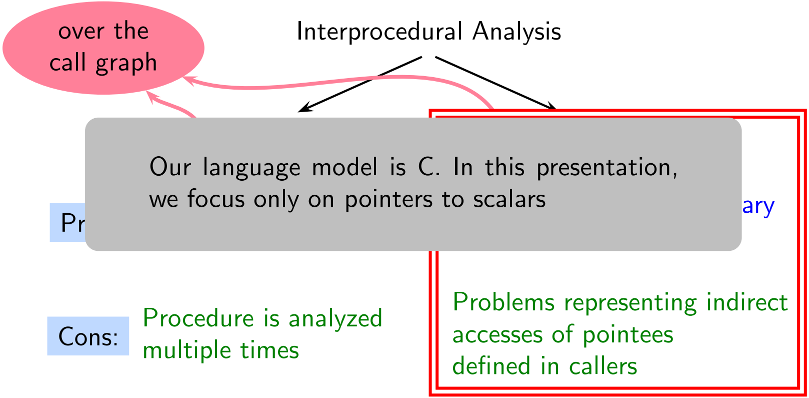
Reusable procedure summary is constructed

Cons: Procedure is analyzed multiple times

Problems representing indirect accesses of pointees defined in callers

We focus on bottom-up approaches and propose a compact representation of procedure summary for pointer analysis

Interprocedural Pointer Analysis



Our language model is C. In this presentation, we focus only on pointers to scalars

Procedure is analyzed multiple times

Problems representing indirect accesses of pointees defined in callers

We focus on bottom-up approaches and propose a compact representation of procedure summary for pointer analysis

Summarizing a Procedure for Points-to Analysis

A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence

Summarizing a Procedure for Points-to Analysis

A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence

- Data dependence exists ⇒

  Can be eliminated and the

  Control flow between the updates would be redundant

```
1. x = &a;
2. y = x;
```

Summarizing a Procedure for Points-to Analysis

A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence

- Data dependence exists ⇒

  Can be eliminated and the

  Control flow between the updates would be redundant

$$\boxed{\begin{array}{l} 1.\ x = \&a; \\ 2.\ y = x; \end{array}}$$

$$\Downarrow$$

$$\boxed{\begin{array}{l} x = \&a; \\ y = \&a; \end{array}}$$

Summarizing a Procedure for Points-to Analysis

A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence

- Data dependence exists ⇒

  Can be eliminated and the

  Control flow between the updates would be redundant

- Data dependence does not exist ⇒

  Redundant memory updates can be eliminated

```
1. x = &a;
2. y = &b;
3. x = &b;
```

Summarizing a Procedure for Points-to Analysis

A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence

- Data dependence exists ⇒

  Can be eliminated and the

  Control flow between the updates would be redundant

- Data dependence does not exist ⇒

  Redundant memory updates can be eliminated

$$\boxed{\begin{array}{l} 1.\ \ x = \&a; \\ 2.\ \ y = \&b; \\ 3.\ \ x = \&b; \end{array}}$$

$$\Downarrow$$

$$\boxed{\begin{array}{l} y = \&b; \\ x = \&b; \end{array}}$$

Summarizing a Procedure for Points-to Analysis

A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence

- Data dependence exists ⇒

  Can be eliminated and the

  Control flow between the updates would be redundant

- Data dependence does not exist ⇒

  Redundant memory updates can be eliminated

- Data dependence is unknown ⇒

  More information is required

  Available when inlined at call sites

```
1. y = &b;
2. *x = &a;
```

Summarizing a Procedure for Points-to Analysis

A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence

- Data dependence exists ⇒

  Can be eliminated and the

  Control flow between the updates would be redundant

- Data dependence does not exist ⇒

  Redundant memory updates can be eliminated

- Data dependence is unknown ⇒

  More information is required

  Available when inlined at call sites

  - Control flow between the updates required

```
1. y = &b;
2. *x = &a;
```

Summarizing a Procedure for Points-to Analysis

A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence

- Data dependence exists ⇒

  Can be eliminated and the

  Control flow between the updates would be redundant

- Data dependence does not exist ⇒

  Redundant memory updates can be eliminated

- Data dependence is unknown ⇒

  More information is required

  Available when inlined at call sites

  - Control flow between the updates required
  - Some accesses of pointees have definitions in the callers

```
1. y = &b;
2. *x = &a;
```

Summarizing a Procedure for Points-to Analysis

A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence

- Data dependence exists ⇒

  Can be eliminated and the

  Control flow between the updates would be redundant

- Data dependence does not exist ⇒

  Redundant memory updates can be eliminated

- Data dependence is unknown ⇒

  More information is required

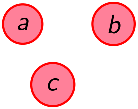
  Available when inlined at call sites

  - Control flow between the updates required
  - Some accesses of pointees have definitions in the callers
  - Some optimizations need to be postponed

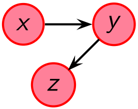
1. $y = \&b;$
2. $*x = \&a;$
3. $z = y;$

Memory and Memory Transformer

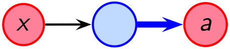
Memory in absence of pointers



Memory in presence of pointers

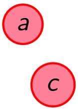


Memory Transformer

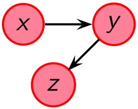


Memory and Memory Transformer

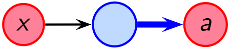
Memory in absence of pointers



Memory in presence of pointers



Memory Transformer

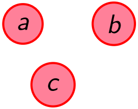


For memory transformer,

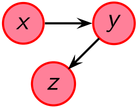
- Blue edges ⇒ information generated
- Black edges ⇒ carried forward input information

Memory and Memory Transformer

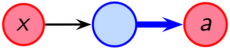
Memory in absence of pointers



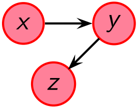
Memory in presence of pointers



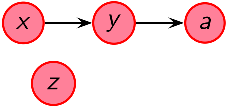
Memory Transformer



Input Memory



Output Memory



Part II

Motivation

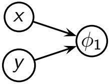
Bottom-up Approaches: The State of the Art

Accesses of pointees that are defined in the callers are represented using placeholders

Bottom-up Approaches: The State of the Art

Accesses of pointees that are defined in the callers are represented using placeholders

e.g., $x = y \Rightarrow$

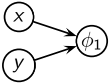


ϕ_1 is a placeholder

Bottom-up Approaches: The State of the Art

Accesses of pointees that are defined in the callers are represented using placeholders

e.g., $x = y \Rightarrow$



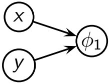
ϕ_1 is a placeholder

- Context based analysis [Zhang-PLDI-14, Wilson-PLDI-95]
  - Use aliases present in the caller
  - Construct a collection of partial transfer functions (PTFs)

Bottom-up Approaches: The State of the Art

Accesses of pointees that are defined in the callers are represented using placeholders

e.g., $x = y \Rightarrow$



ϕ_1 is a placeholder

- Context based analysis [Zhang-PLDI-14, Wilson-PLDI-95]
  - Use aliases present in the caller
  - Construct a collection of partial transfer functions (PTFs)
- Context independent analysis [Sălcianu-VMCAI-05, Madhavan-SAS-12]
  - No aliases assumed in the calling contexts
  - Construct a single procedure summary

Limitation of Placeholders

- Placeholders explicate the pointees defined in callers
  (Low level abstraction of memory)

Limitation of Placeholders

- Placeholders explicate the pointees defined in callers
  (Low level abstraction of memory)
- This results in
  - either multiple call-specific procedure summaries, or

Reuse of a placeholder for a flow sensitive summary flow function depends on the aliases in the calling contexts

Limitation of Placeholders

- Placeholders explicate the pointees defined in callers (Low level abstraction of memory)
- This results in
  - either multiple call-specific procedure summaries, or
  - large number of placeholders

In absence of aliases from the calling contexts, every access is represented by a separate placeholder. Control flow is also required

Reuse of a placeholder for a flow sensitive summary flow function depends on the aliases in the calling contexts

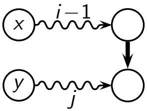
Part III

Generalized Points-to Graphs

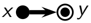
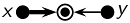
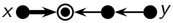
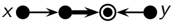
Representing Basic Pointer Assignments using the Generalized Points-to Updates

General Case

GPU $x \xrightarrow[s]{i|j} y$



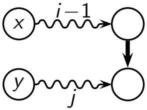
Specific Examples

Pointer assignment	GPU	Relevant memory graph after the assignment
$s: x = \&y$	$x \xrightarrow[s]{1\|0} y$	
$s: x = y$	$x \xrightarrow[s]{1\|1} y$	
$s: x = *y$	$x \xrightarrow[s]{1\|2} y$	
$s: *x = y$	$x \xrightarrow[s]{2\|1} y$	

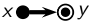
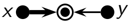
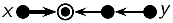
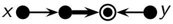
Representing Basic Pointer Assignments using the Generalized Points-to Updates

General Case

GPU $x \xrightarrow[s]{i|j} y$



Specific Examples

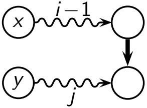
Pointer assignment	GPU	Relevant memory graph after the assignment
$s: x = \&y$	$x \xrightarrow[s]{1\|0} y$	
$s: x = y$	$x \xrightarrow[s]{1\|1} y$	
$s: x = *y$	$x \xrightarrow[s]{1\|2} y$	
$s: *x = y$	$x \xrightarrow[s]{2\|1} y$	

- The direction in a GPU is to distinguish between what is being defined to what is being read
- For pointer analysis, case $i = 0$ does not exist
- classical points-to update is a special case of generalized points-to update with $i = 1$ and $j = 0$

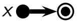
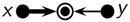
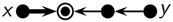
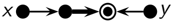
Representing Basic Pointer Assignments using the Generalized Points-to Updates

General Case

GPU $x \xrightarrow[s]{i|j} y$



Specific Examples

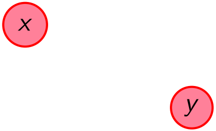
Pointer assignment	GPU	Relevant memory graph after the assignment
$s: x = \&y$	$x \xrightarrow[s]{1\|0} y$	
$s: x = y$	$x \xrightarrow[s]{1\|1} y$	
$s: x = *y$	$x \xrightarrow[s]{1\|2} y$	
$s: *x = y$	$x \xrightarrow[s]{2\|1} y$	

- The direction in a GPU is to distinguish between what is being defined to what is being read
- For pointer analys
- classical points-to upda generalized points-to update with $i = 1$ and $j = 0$

GPU represents both memory and memory transformer

Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```

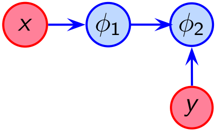


All variables are global

Red nodes are known named locations

Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```



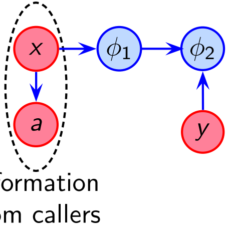
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

```
f()
{
   *x = y
}
```



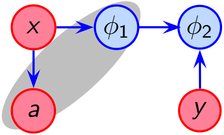
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

```
f()
{
   *x = y
}
```



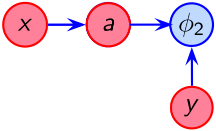
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

```
f()
{
   *x = y
}
```



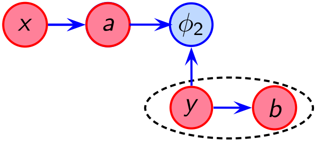
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```



Information from callers

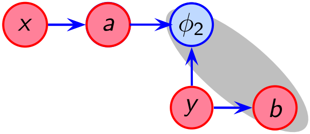
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```



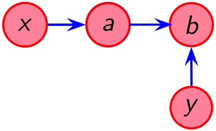
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```



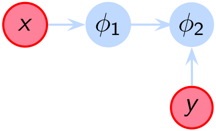
All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

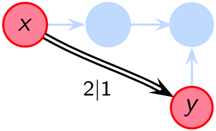
```
f()
{
   *x = y
}
```



Blue arrows are low level view of memory in terms of classical points-to facts

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```

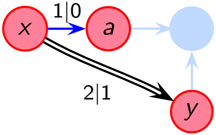


Blue arrows are low level view of memory in terms of classical points-to facts

Black arrows are high level view of memory in terms of generalized points-to facts

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

```
f()
{
    *x = y
}
```

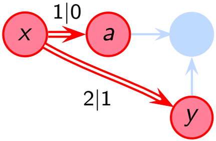


Blue arrows are low level view of memory in terms of classical points-to facts

Black arrows are high level view of memory in terms of generalized points-to facts

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```

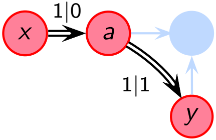


Blue arrows are low level view of memory in terms of classical points-to facts

Black arrows are high level view of memory in terms of generalized points-to facts

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```

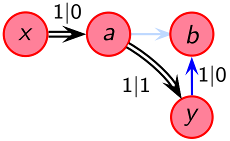


Blue arrows are low level view of memory in terms of classical points-to facts

Black arrows are high level view of memory in terms of generalized points-to facts

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```

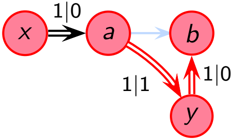


Blue arrows are low level view of memory in terms of classical points-to facts

Black arrows are high level view of memory in terms of generalized points-to facts

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

```
f()
{
   *x = y
}
```

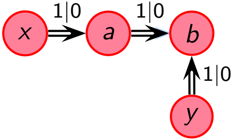


Blue arrows are low level view of memory in terms of classical points-to facts

Black arrows are high level view of memory in terms of generalized points-to facts

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```

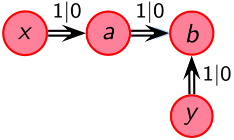


Blue arrows are low level view of memory in terms of classical points-to facts

Black arrows are high level view of memory in terms of generalized points-to facts

Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

```
f()
{
  *x = y
}
```



Blue arrows are low level view of memory in terms of classical points-to facts

Black arrows are high level view of memory in terms of generalized points-to facts

This abstraction does not introduce any imprecision over the classical points-to graph

GPU Composition

- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*

GPU Composition

- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*

$x = \&y;$

$z = *x;$

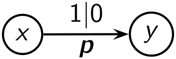
GPU Composition

- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*

$x = \&y;$

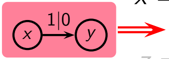
$z = *x;$

GPG



GPU Composition

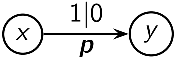
- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*



$x = \&y;$

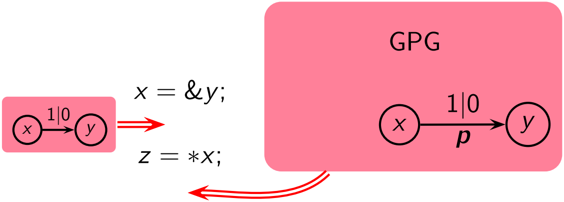
$z = *x;$

GPG



GPU Composition

- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*

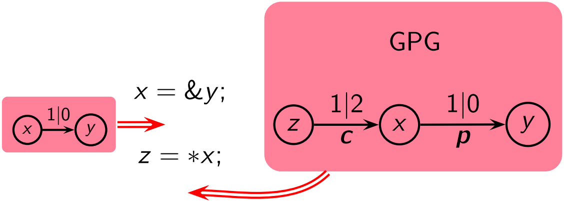


$x = \&y;$

$z = *x;$

GPU Composition

- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*

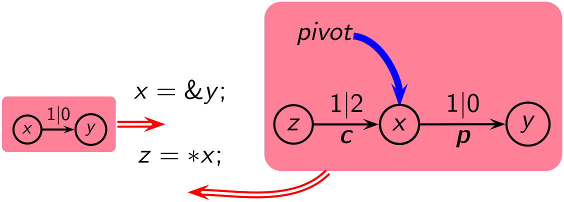


$x = \&y;$

$z = *x;$

GPU Composition

- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*



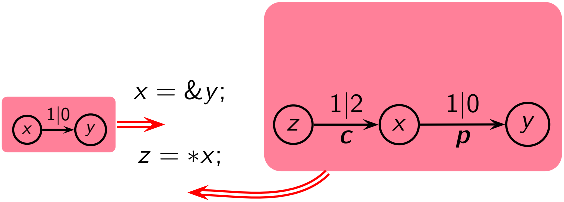
$x = \&y;$

$z = *x;$

GPU Composition

- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*
- Reduces the *indlev* of $\boldsymbol{c}$ by using information from $\boldsymbol{p}$
    - Eliminating pivot and creating a reduced GPU $\boldsymbol{r}$ between other two nodes by using *pivot* as a bridge

$\boldsymbol{c} \Rightarrow$ Consumer GPU, $\boldsymbol{p} \Rightarrow$ Producer GPU



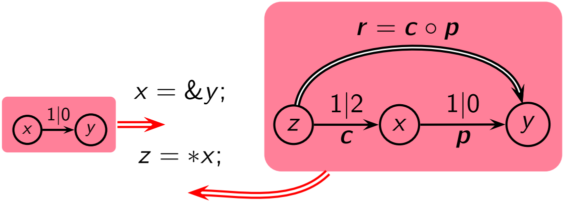
$x = \&y;$

$z = *x;$

GPU Composition

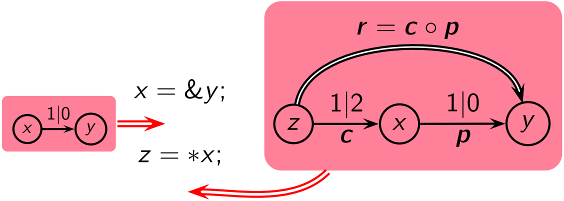
- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*
- Reduces the *indlev* of $\boldsymbol{c}$ by using information from $\boldsymbol{p}$
  - Eliminating pivot and creating a reduced GPU $\boldsymbol{r}$ between other two nodes by using *pivot* as a bridge

$\boldsymbol{c}$ ⇒ Consumer GPU, $\boldsymbol{p}$ ⇒ Producer GPU



GPU Composition

- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*
- Reduces the *indlev* of $\boldsymbol{c}$ by using information from $\boldsymbol{p}$
  - Eliminating pivot and creating a reduced GPU $\boldsymbol{r}$ between other two nodes by using *pivot* as a bridge
- Requires the *indlev*s of the pivot in both the GPUs to be made same



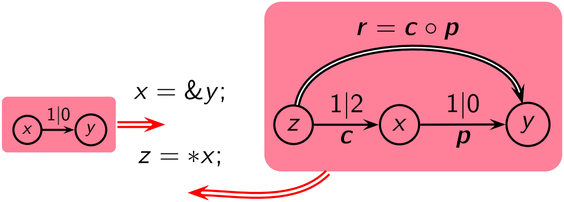
$x = \&y;$

$z = *x;$

$\boldsymbol{r} = \boldsymbol{c} \circ \boldsymbol{p}$

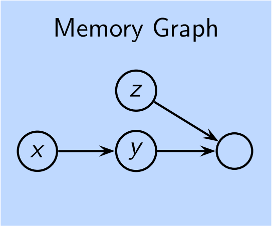
GPU Composition

- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*
- Reduces the *indlev* of $\boldsymbol{c}$ by using information from $\boldsymbol{p}$
  - Eliminating pivot and creating a reduced GPU $\boldsymbol{r}$ between other two nodes by using *pivot* as a bridge
- Requires the *indlev*s of the pivot in both the GPUs to be made same



$x = \&y;$

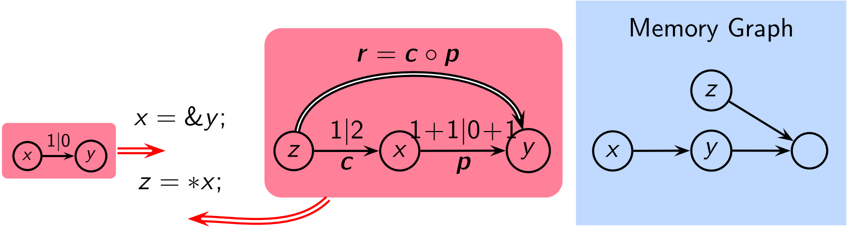
$z = *x;$



Memory Graph

GPU Composition

- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*
- Reduces the *indlev* of $\boldsymbol{c}$ by using information from $\boldsymbol{p}$
  - Eliminating pivot and creating a reduced GPU $\boldsymbol{r}$ between other two nodes by using *pivot* as a bridge
- Requires the *indlev*s of the pivot in both the GPUs to be made same

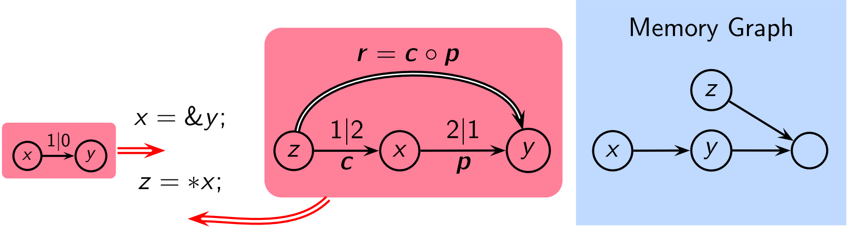


$x = \&y;$

$z = *x;$

GPU Composition

- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*
- Reduces the *indlev* of $\boldsymbol{c}$ by using information from $\boldsymbol{p}$
  - Eliminating pivot and creating a reduced GPU $\boldsymbol{r}$ between other two nodes by using *pivot* as a bridge
- Requires the *indlev*s of the pivot in both the GPUs to be made same

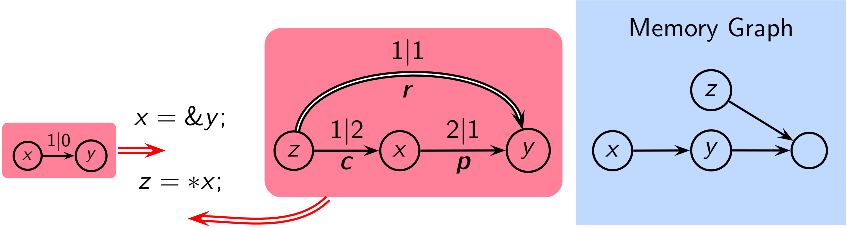


$x = \&y;$

$z = *x;$

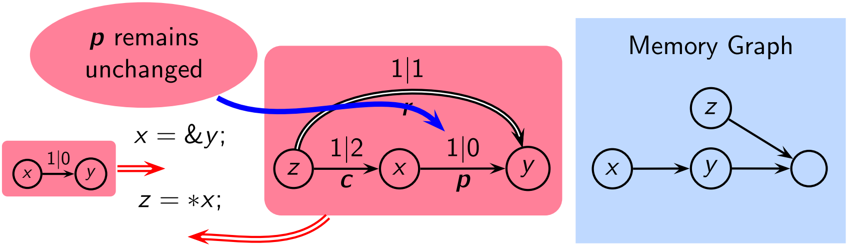
GPU Composition

- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*
- Reduces the *indlev* of $\boldsymbol{c}$ by using information from $\boldsymbol{p}$
  - Eliminating pivot and creating a reduced GPU $\boldsymbol{r}$ between other two nodes by using *pivot* as a bridge
- Requires the *indlev*s of the pivot in both the GPUs to be made same



GPU Composition

- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*
- Reduces the *indlev* of $\boldsymbol{c}$ by using information from $\boldsymbol{p}$
  - Eliminating pivot and creating a reduced GPU $\boldsymbol{r}$ between other two nodes by using *pivot* as a bridge
- Requires the *indlev*s of the pivot in both the GPUs to be made same



p remains unchanged

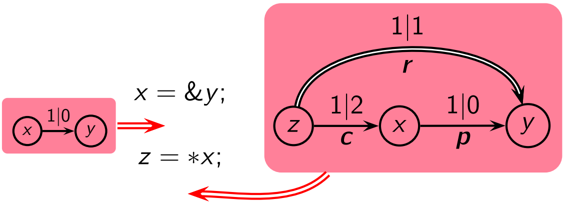
$x = \&y;$

$z = *x;$

Memory Graph

GPU Composition

- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*
- Reduces the *indlev* of $\boldsymbol{c}$ by using information from $\boldsymbol{p}$
  - Eliminating pivot and creating a reduced GPU $\boldsymbol{r}$ between other two nodes by using *pivot* as a bridge
- Requires the *indlev*s of the pivot in both the GPUs to be made same



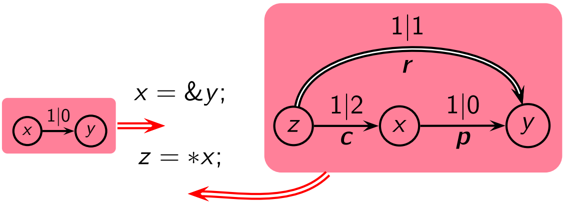
$x = \&y;$

$z = *x;$

Data dependence through x is eliminated. Control flow becomes redundant

GPU Composition

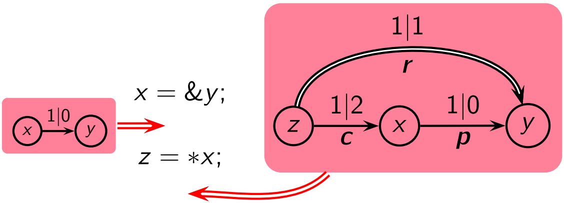
- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*
- Reduces the *indlev* of $\boldsymbol{c}$ by using information from $\boldsymbol{p}$
  - Eliminating pivot and creating a reduced GPU $\boldsymbol{r}$ between other two nodes by using *pivot* as a bridge
- Requires the *indlev*s of the pivot in both the GPUs to be made same



GPUs $\boldsymbol{r}$ and $\boldsymbol{c}$ are equivalent in the context of $\boldsymbol{p}$

GPU Composition

- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*
- Reduces the *indlev* of $\boldsymbol{c}$ by using information from $\boldsymbol{p}$
  - Eliminating pivot and creating a reduced GPU $\boldsymbol{r}$ between other two nodes by using *pivot* as a bridge
- Requires the *indlev*s of the pivot in both the GPUs to be made same



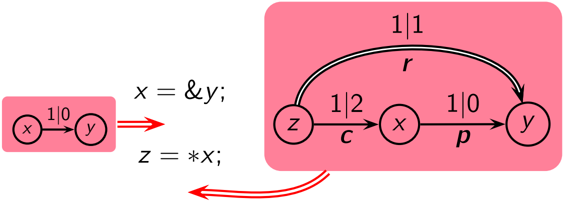
$x = \&y;$

$z = *x;$

Strength reduction optimization replaces $\boldsymbol{c}$ by $\boldsymbol{r}$

GPU Composition

- Represented by $\boldsymbol{c} \circ \boldsymbol{p}$; performed only when they share a common node called the *pivot*
- Reduces the *indlev* of $\boldsymbol{c}$ by using information from $\boldsymbol{p}$
  - Eliminating pivot and creating a reduced GPU $\boldsymbol{r}$ between other two nodes by using *pivot* as a bridge
- Requires the *indlev*s of the pivot in both the GPUs to be made same



$x = \&y;$

$z = *x;$

GPU reduction is a series of GPU compositions

Generalized Points-to Graphs (GPGs) I

A GPG is a graph with

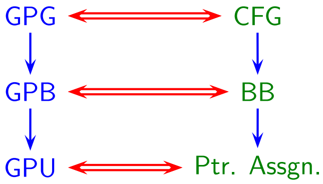
- Nodes called as generalized points-to blocks (GPBs)
    - A GPB contains a set of GPUs
- Edges representing control flow between GPBs

Generalized Points-to Graphs (GPGs) I

A GPG is a graph with

- Nodes called as generalized points-to blocks (GPBs)
  - A GPB contains a set of GPUs
- Edges representing control flow between GPBs

A GPG is analogous to a CFG of a procedure



Generalized Points-to Graphs (GPGs) I

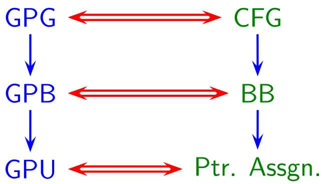
A GPG

- N
- E

First difference:

- GPUs in a GPB represent parallel assignments
- Assignments in a basic block are sequential

A GPG is analogous to a CFG of a procedure



GPG ⟺ CFG

GPB ⟺ BB

GPU ⟺ Ptr. Assgn.

Generalized Points-to Graphs (GPGs) I

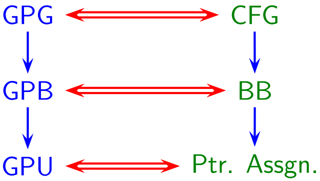
A GPG is a graph with

- N
- Ed

Second difference:

- CFGs contain call basic blocks
- GPGs do not have call GPBs

A GPG is analogous to a CFG of a procedure



Generalized Points-to Graphs (GPGs) II

Construction of Initial GPGs:

- Non-pointer assignments and condition tests are removed
- Each pointer assignment s is transliterated to its GPU (γ_s)
- A separate GPB is created for assignment in the CFG
- GPG edges are induced from the control flow of the CFG
- GPGs contain only variables that are shared across procedures

GPGs then undergo extensive optimizations

The Big Picture View of GPG Construction

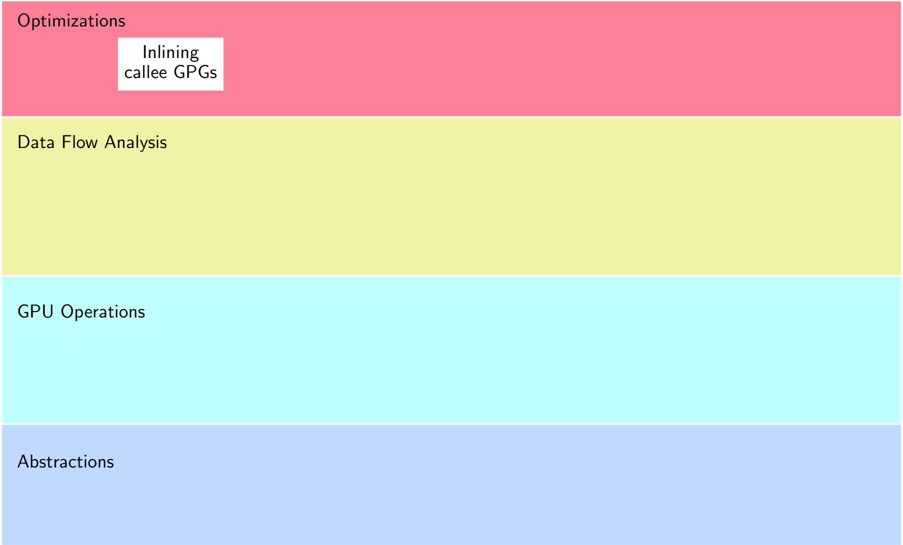
Optimizations

Data Flow Analysis

GPU Operations

Abstractions

The Big Picture View of GPG Construction



Optimizations

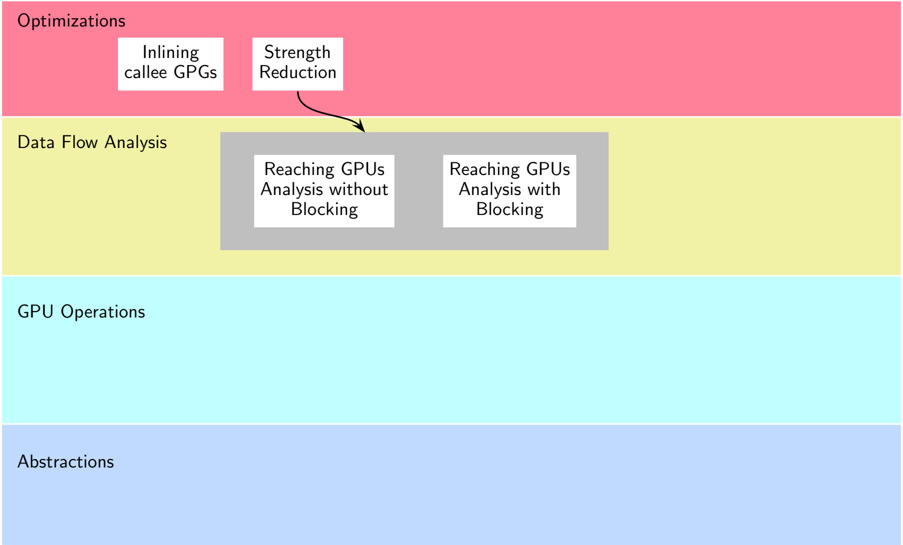
Inlining callee GPGs

Data Flow Analysis

GPU Operations

Abstractions

The Big Picture View of GPG Construction



Optimizations

Inlining callee GPGs

Strength Reduction

Data Flow Analysis

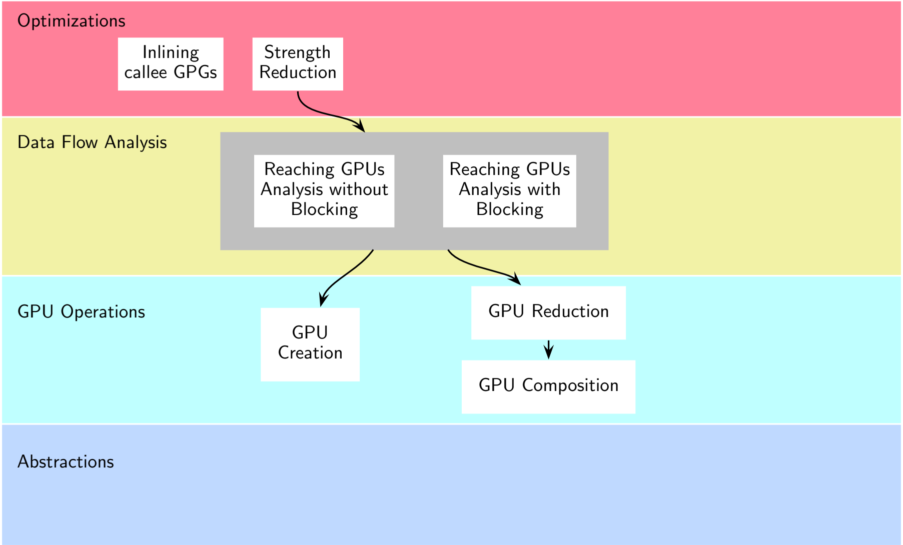
Reaching GPUs Analysis without Blocking

Reaching GPUs Analysis with Blocking

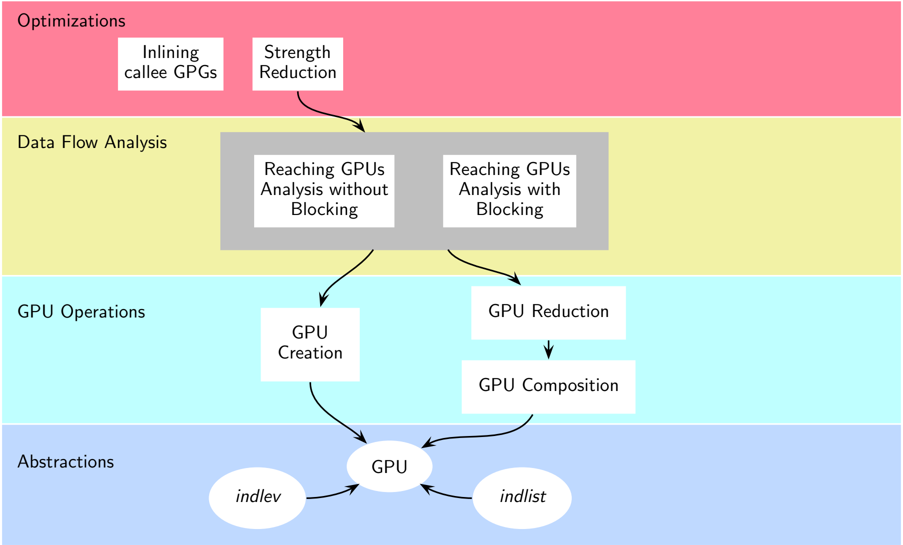
GPU Operations

Abstractions

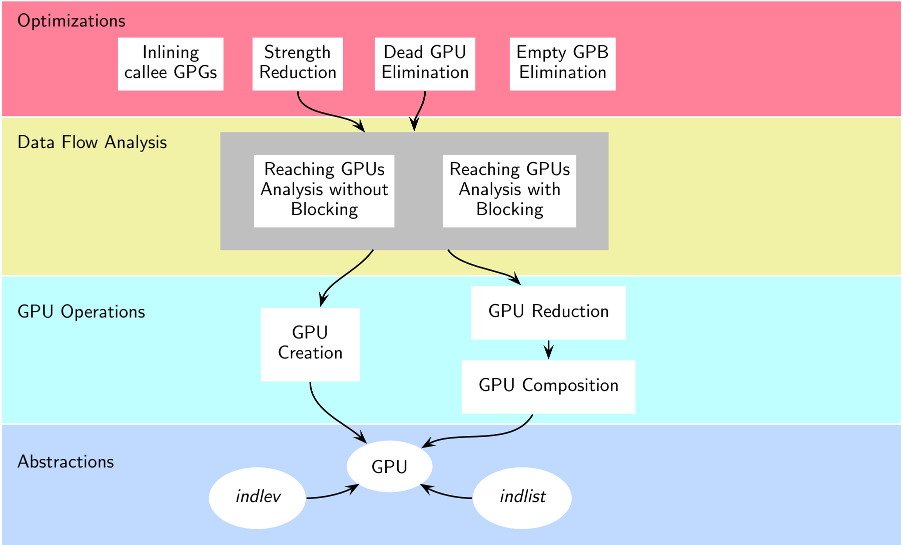
The Big Picture View of GPG Construction



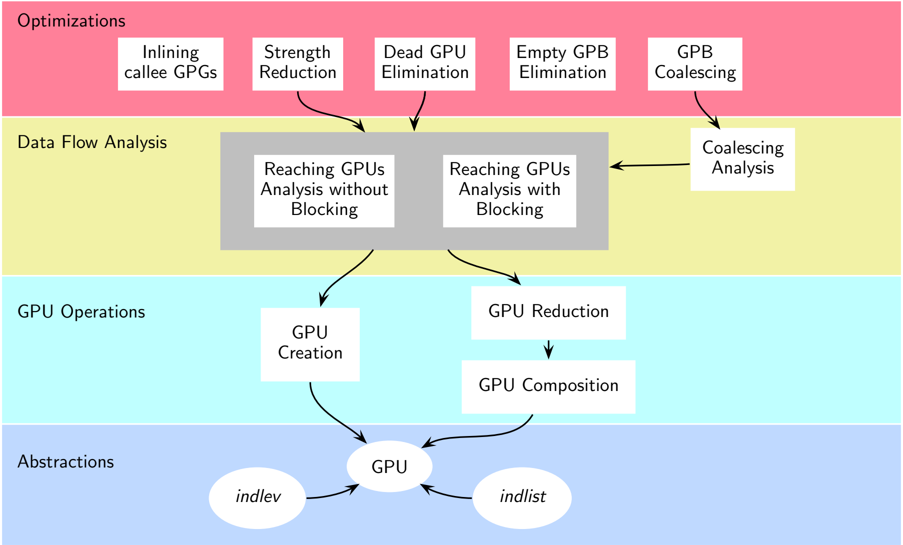
The Big Picture View of GPG Construction



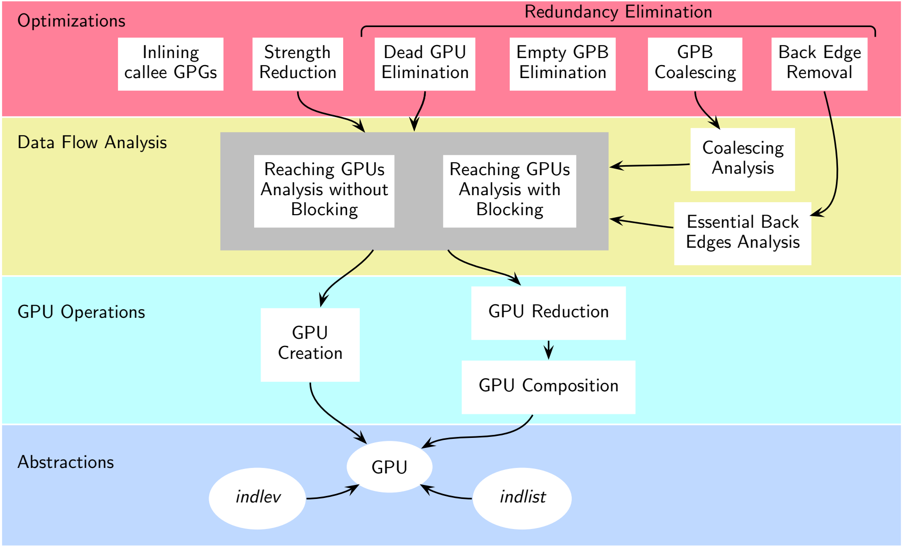
The Big Picture View of GPG Construction



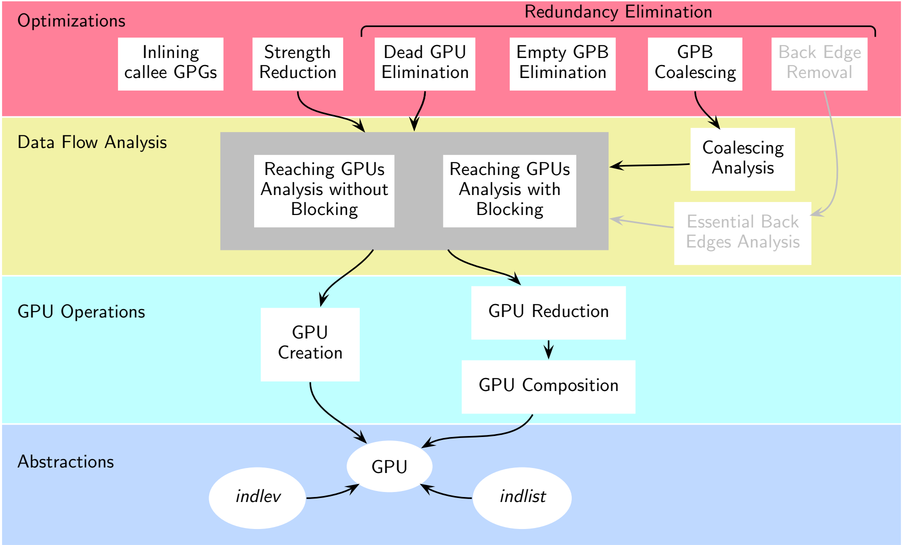
The Big Picture View of GPG Construction



The Big Picture View of GPG Construction

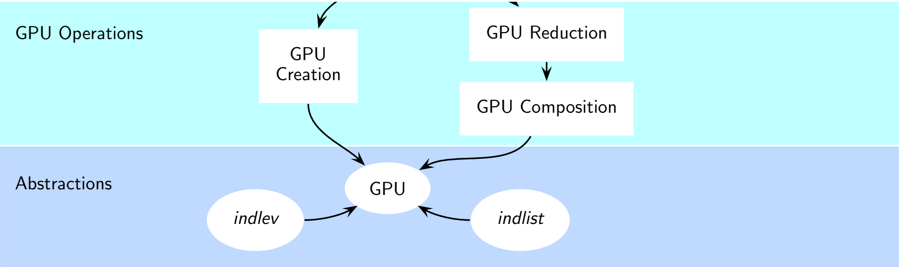


The Big Picture View of GPG Construction

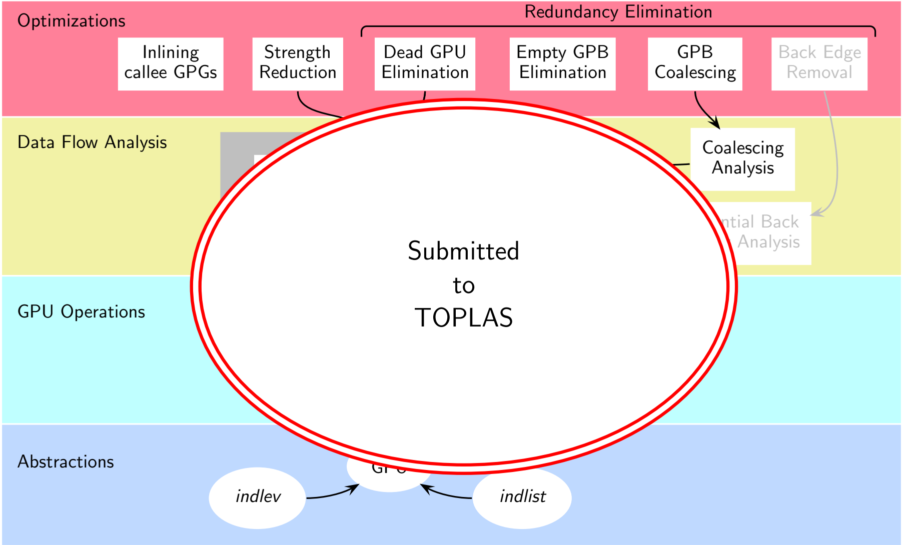


The Big Picture View of GPG Construction

Published in SAS 2016



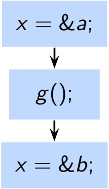
The Big Picture View of GPG Construction



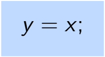
Submitted
to
TOPLAS

GPGs Across Optimizations

CFG of
proc f

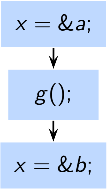


CFG of
proc g



GPGs Across Optimizations

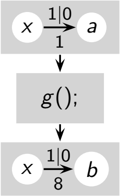
CFG of proc f


$x = \&a;$

$g();$

$x = \&b;$

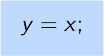
Initial GPG of proc f


$x \xrightarrow[1]{1|0} a$

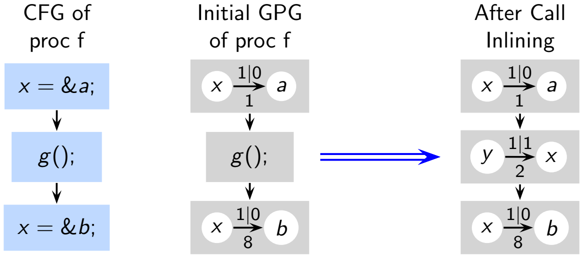
$g();$

$x \xrightarrow[8]{1|0} b$

CFG of proc g


$y = x;$

GPGs Across Optimizations



CFG of proc f

$x = \&a;$

$g();$

$x = \&b;$

Initial GPG of proc f

$x \xrightarrow[1]{1|0} a$

$g();$

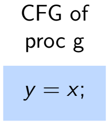
$x \xrightarrow[8]{1|0} b$

After Call Inlining

$x \xrightarrow[1]{1|0} a$

$y \xrightarrow[2]{1|1} x$

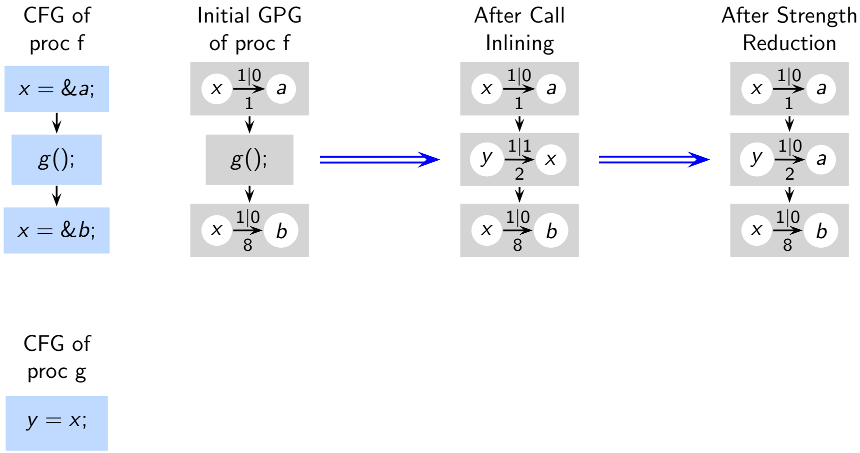
$x \xrightarrow[8]{1|0} b$



CFG of proc g

$y = x;$

GPGs Across Optimizations



CFG of proc f

$x = \&a;$

$g();$

$x = \&b;$

Initial GPG of proc f

$x \xrightarrow[1]{1|0} a$

$g();$

$x \xrightarrow[8]{1|0} b$

After Call Inlining

$x \xrightarrow[1]{1|0} a$

$y \xrightarrow[2]{1|1} x$

$x \xrightarrow[8]{1|0} b$

After Strength Reduction

$x \xrightarrow[1]{1|0} a$

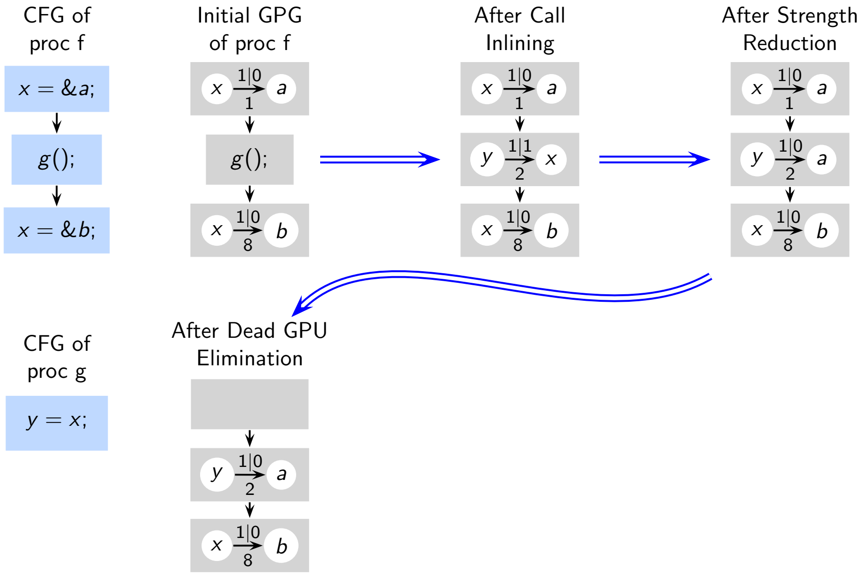
$y \xrightarrow[2]{1|0} a$

$x \xrightarrow[8]{1|0} b$

CFG of proc g

$y = x;$

GPGs Across Optimizations



CFG of proc f

$x = \&a;$

$g();$

$x = \&b;$

Initial GPG of proc f

$x \xrightarrow[1]{1|0} a$

$g();$

$x \xrightarrow[8]{1|0} b$

After Call Inlining

$x \xrightarrow[1]{1|0} a$

$y \xrightarrow[2]{1|1} x$

$x \xrightarrow[8]{1|0} b$

After Strength Reduction

$x \xrightarrow[1]{1|0} a$

$y \xrightarrow[2]{1|0} a$

$x \xrightarrow[8]{1|0} b$

CFG of proc g

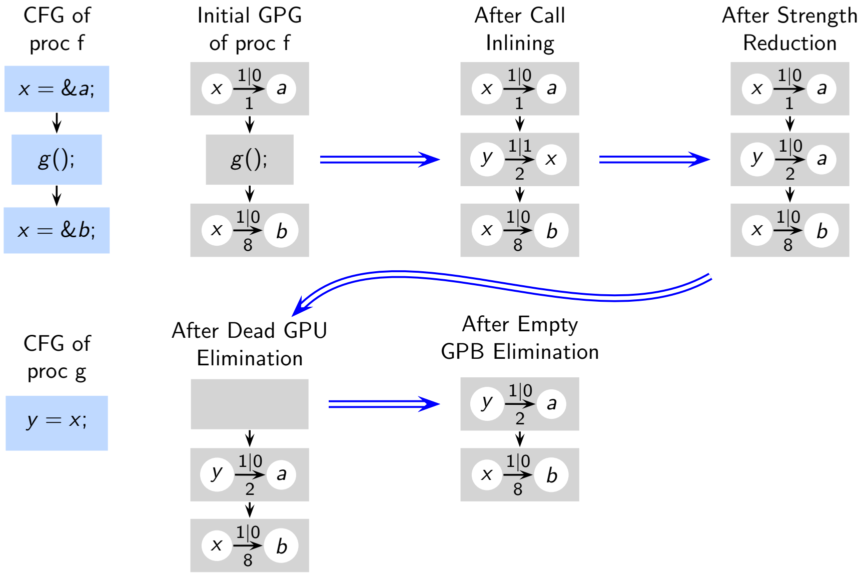
$y = x;$

After Dead GPU Elimination

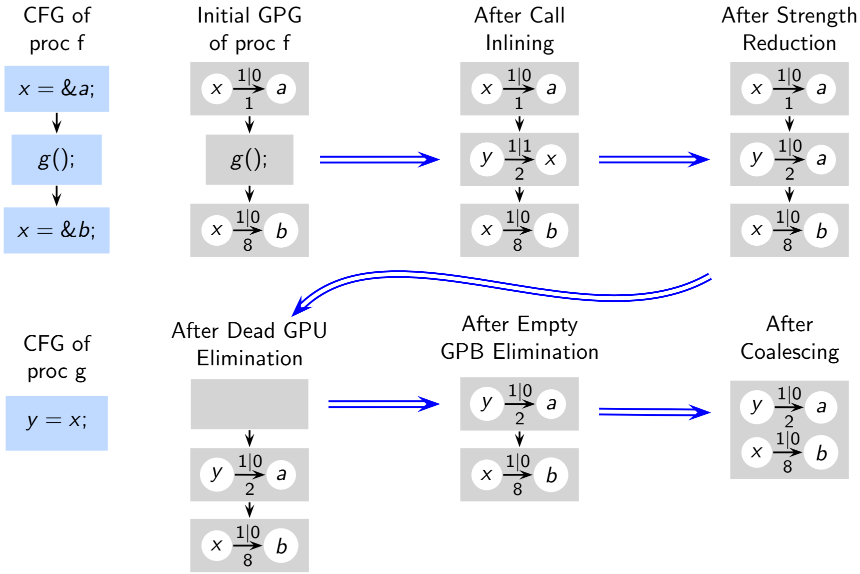
$y \xrightarrow[2]{1|0} a$

$x \xrightarrow[8]{1|0} b$

GPGs Across Optimizations


CFG of proc f
$x = \&a;$
$g();$
$x = \&b;$
Initial GPG of proc f
$x \xrightarrow[1]{1|0} a$
$g();$
$x \xrightarrow[8]{1|0} b$
After Call Inlining
$x \xrightarrow[1]{1|0} a$
$y \xrightarrow[2]{1|1} x$
$x \xrightarrow[8]{1|0} b$
After Strength Reduction
$x \xrightarrow[1]{1|0} a$
$y \xrightarrow[2]{1|0} a$
$x \xrightarrow[8]{1|0} b$
CFG of proc g
$y = x;$
After Dead GPU Elimination
$y \xrightarrow[2]{1|0} a$
$x \xrightarrow[8]{1|0} b$
After Empty GPB Elimination
$y \xrightarrow[2]{1|0} a$
$x \xrightarrow[8]{1|0} b$

GPGs Across Optimizations



CFG of proc f

$x = \&a;$

$g();$

$x = \&b;$

Initial GPG of proc f

$x \xrightarrow[1]{1|0} a$

$g();$

$x \xrightarrow[8]{1|0} b$

After Call Inlining

$x \xrightarrow[1]{1|0} a$

$y \xrightarrow[2]{1|1} x$

$x \xrightarrow[8]{1|0} b$

After Strength Reduction

$x \xrightarrow[1]{1|0} a$

$y \xrightarrow[2]{1|0} a$

$x \xrightarrow[8]{1|0} b$

CFG of proc g

$y = x;$

After Dead GPU Elimination

$y \xrightarrow[2]{1|0} a$

$x \xrightarrow[8]{1|0} b$

After Empty GPB Elimination

$y \xrightarrow[2]{1|0} a$

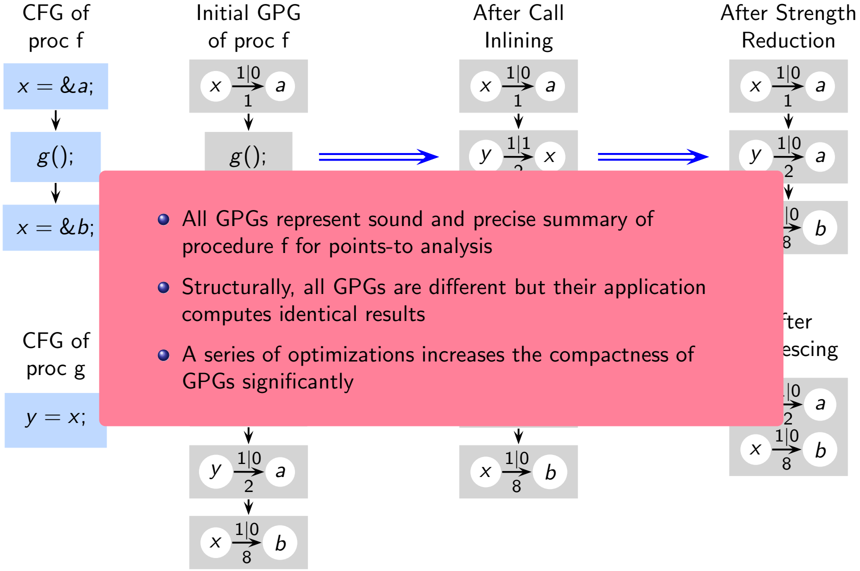
$x \xrightarrow[8]{1|0} b$

After Coalescing

$y \xrightarrow[2]{1|0} a$

$x \xrightarrow[8]{1|0} b$

GPGs Across Optimizations



CFG of proc f

$x = \&a;$

$g();$

$x = \&b;$

CFG of proc g

$y = x;$

Initial GPG of proc f

$x \xrightarrow[1]{1|0} a$

$g();$

$y \xrightarrow[2]{1|0} a$

$x \xrightarrow[8]{1|0} b$

After Call Inlining

$x \xrightarrow[1]{1|0} a$

$y \xrightarrow[2]{1|1} x$

$x \xrightarrow[8]{1|0} b$

After Strength Reduction

$x \xrightarrow[1]{1|0} a$

$y \xrightarrow[2]{1|0} a$

$\xrightarrow[8]{|0} b$

fter escing

$\xrightarrow[2]{|0} a$

$x \xrightarrow[8]{1|0} b$

- All GPGs represent sound and precise summary of procedure f for points-to analysis
- Structurally, all GPGs are different but their application computes identical results
- A series of optimizations increases the compactness of GPGs significantly

Factors affecting Scalability

Three issues that cause non-scalability

- Modelling indirect accesses of pointees that are defined in callers without examining their code
  - GPUs track indirection levels that relate (transitively indirect) pointees of a variable with those of other variables
- Preserving data dependence between memory updates
  - Maintain minimal control flow between memory updates ensuring soundness and precision
- Incorporating the effect of summaries of the callee procedures transitively
  - Series of GPG optimizations gives compactness that mitigate the impact of transitive inlining

Part IV

Implementation and Empirical Measurements

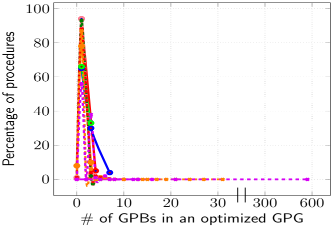
Implementation

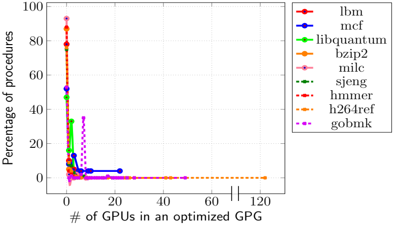
- Implemented in GCC 4.7.2 using the LTO framework
- Measurements carried out on SPEC CPU2006 benchmarks on a machine with 16 GB RAM with eight 64-bit Intel i7-4770 CPUs running at 3.40GHz
- We could scale our analysis on benchmarks upto 158kLoC
- Also implemented flow- and context-insensitive points-to analysis and flow-insensitive and context-sensitive points-to analysis

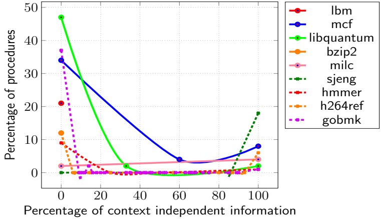
Effectiveness of GPGs

- Compactness of GPGs.
- Percentage of context independent information (CI)
  - A procedure summary is very useful if it contains high percentage of context-independent information (GPUs with *indlev* "1|0").

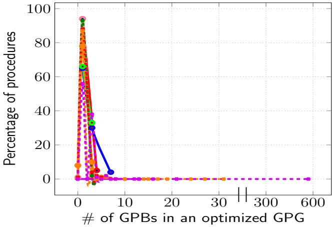
Effectiveness of GPGs

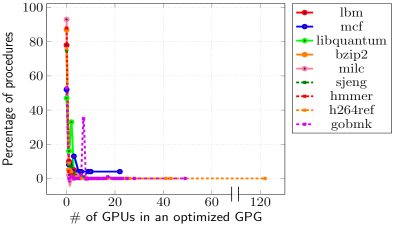




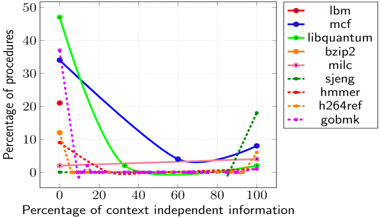


Effectiveness of GPGs



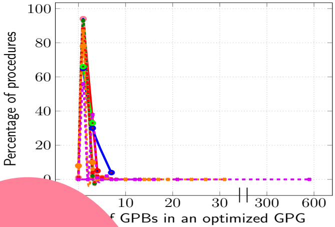


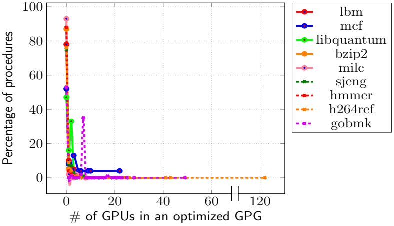
Majority of GPGs have 1 to 3 GPBs



GPGs for a large number of procedures have 0 GPUs

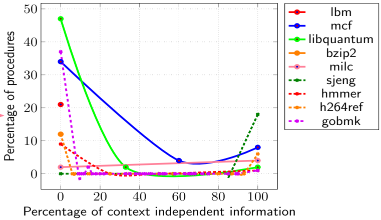
Effectiveness of GPGs



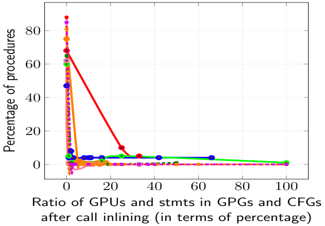


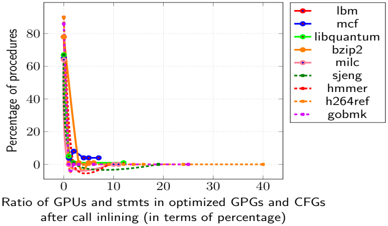
Number of procedures with a high % of context independent information is larger in larger benchmarks

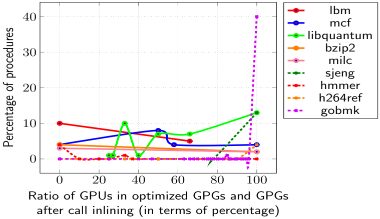
⟹



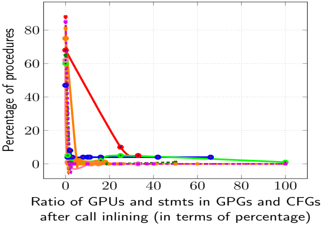
Size of GPGs Relative to the Size of Procedures in terms of GPUs and Pointer Assignments

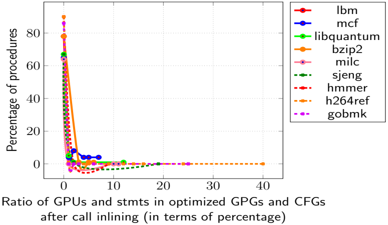




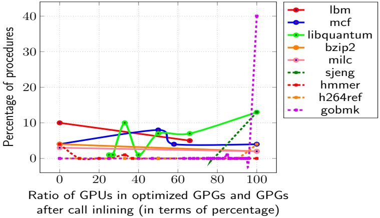


Size of GPGs Relative to the Size of Procedures in terms of GPUs and Pointer Assignments

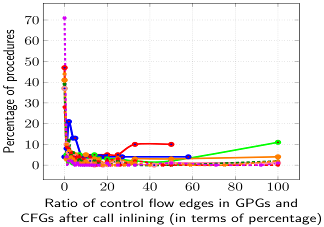


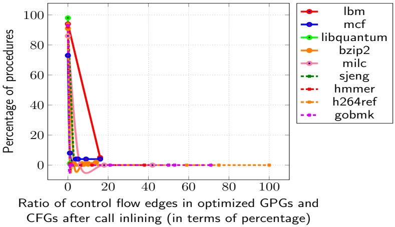


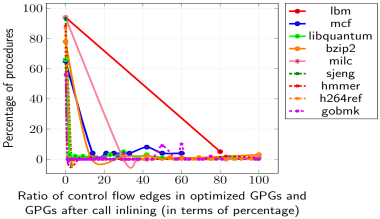
Smaller the ratio, more is the reduction and more compact are the GPGs



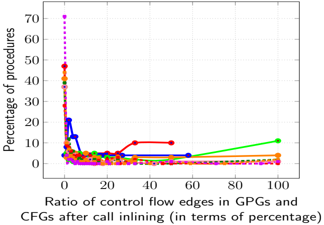
Size of GPGs Relative to the Size of Procedures in terms of control flow edges

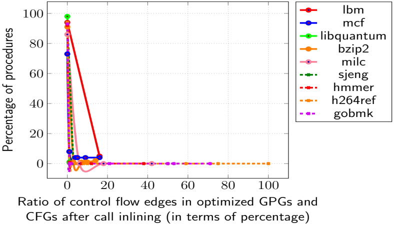




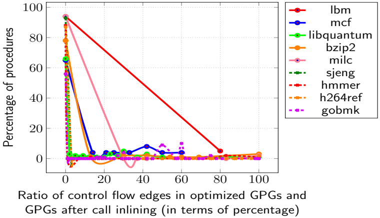


Size of GPGs Relative to the Size of Procedures in terms of control flow edges

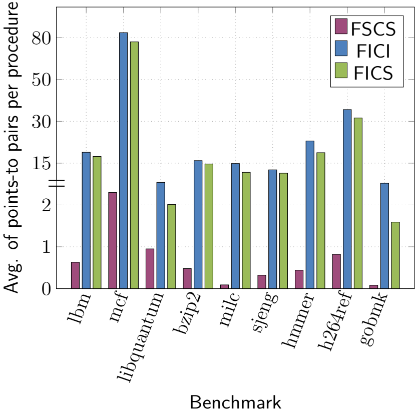




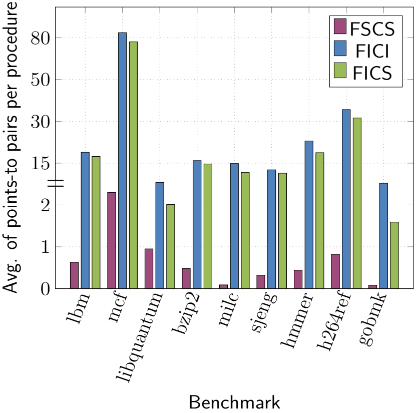
Optimization of
control flow
is more compared
to the optimization
of GPUs



Data Measurements



Data Measurements



Average number of points-to pairs in FSCS is much smaller than FICI and FICS

Part V

Points-to Information Computation

Points-to Information Computation

Traditional bottom-up approach consists of two phases:

- a bottom-up phase for constructing procedure summaries
- a top-down phase for computing points-to information using procedure summaries

Points-to Information Computation

- Interleaving of strength reduction and call inlining ⇒

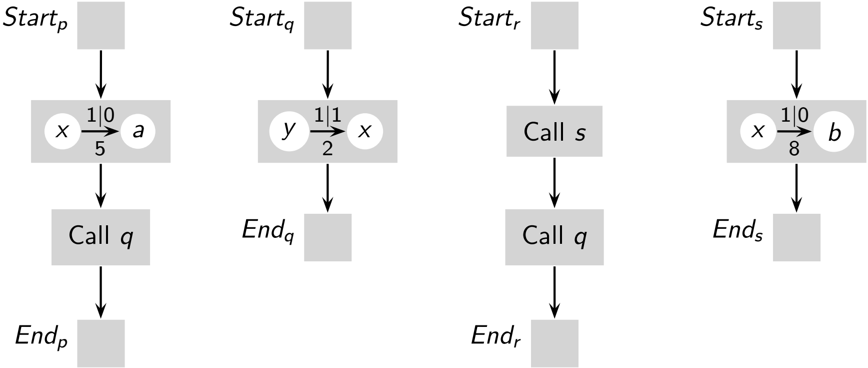
  The top-down phase redundant

- Points-to information is computed by bringing the definitions and uses of a pointer to a common context

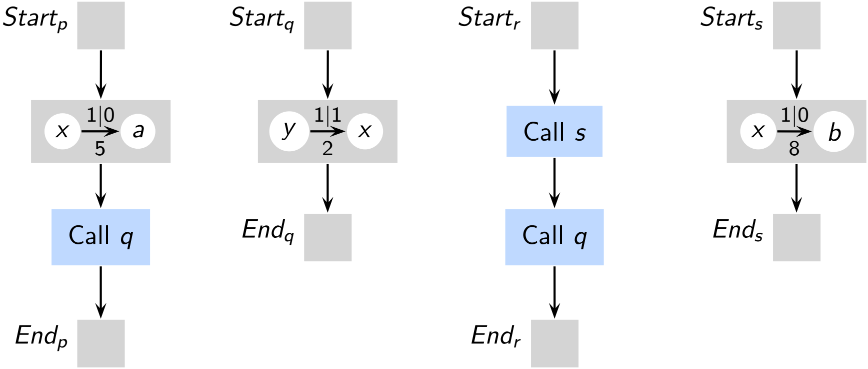
  Can be achieved by pushing

  - a use to a caller
  - a definition to a caller
  - both use and definition to a caller
  - neither (if they are already in the same procedure)

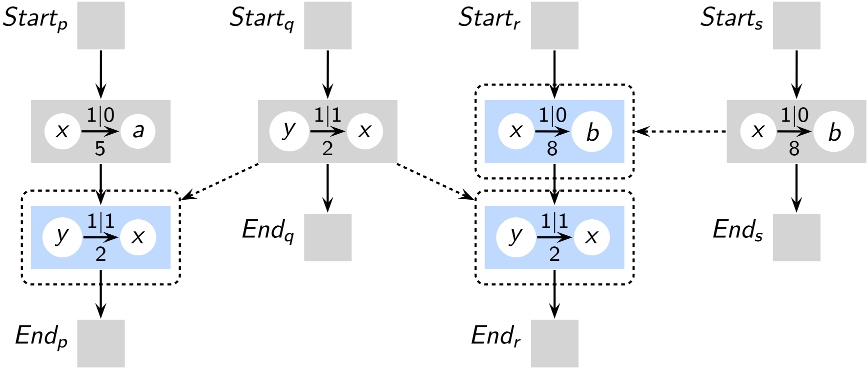
Points-to Information Computation



Points-to Information Computation

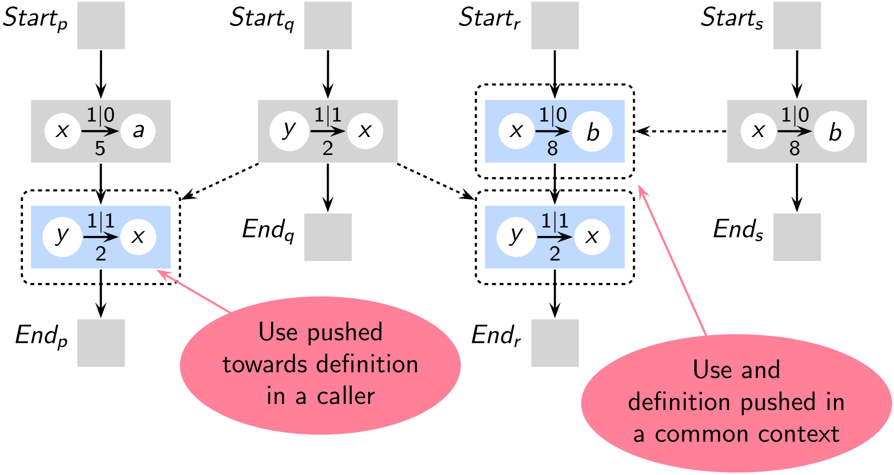


Points-to Information Computation


$Start_p$
$x \xrightarrow[5]{1|0} a$
$y \xrightarrow[2]{1|1} x$
End_p
$Start_q$
$y \xrightarrow[2]{1|1} x$
End_q
$Start_r$
$x \xrightarrow[8]{1|0} b$
$y \xrightarrow[2]{1|1} x$
End_r
$Start_s$
$x \xrightarrow[8]{1|0} b$
End_s

After call inlining

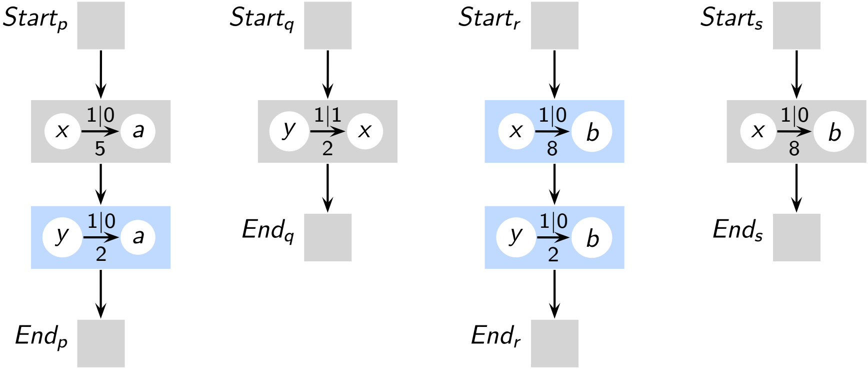
Points-to Information Computation


$Start_p$
$Start_q$
$Start_r$
$Start_s$
$x \xrightarrow[5]{1|0} a$
$y \xrightarrow[2]{1|1} x$
$x \xrightarrow[8]{1|0} b$
$x \xrightarrow[8]{1|0} b$
$y \xrightarrow[2]{1|1} x$
End_q
$y \xrightarrow[2]{1|1} x$
End_s
End_p
End_r

Use pushed towards definition in a caller

Use and definition pushed in a common context

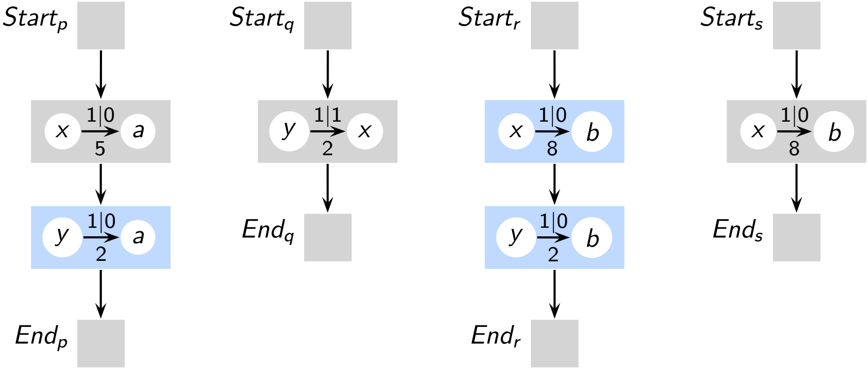
Points-to Information Computation



$Start_p$
$x \xrightarrow[5]{1|0} a$
$y \xrightarrow[2]{1|0} a$
End_p
$Start_q$
$y \xrightarrow[2]{1|1} x$
End_q
$Start_r$
$x \xrightarrow[8]{1|0} b$
$y \xrightarrow[2]{1|0} b$
End_r
$Start_s$
$x \xrightarrow[8]{1|0} b$
End_s

After strength reduction optimization

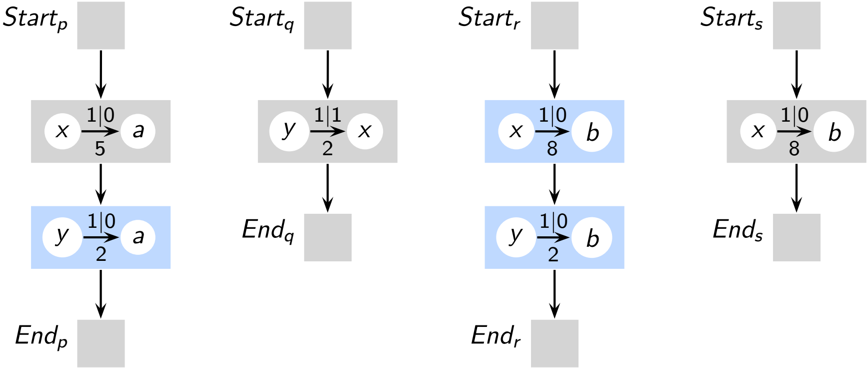
Points-to Information Computation



$Start_p$
$x \xrightarrow[5]{1|0} a$
$y \xrightarrow[2]{1|0} a$
End_p
$Start_q$
$y \xrightarrow[2]{1|1} x$
End_q
$Start_r$
$x \xrightarrow[8]{1|0} b$
$y \xrightarrow[2]{1|0} b$
End_r
$Start_s$
$x \xrightarrow[8]{1|0} b$
End_s

Stmt. id	Points-to Information
2	$\{y \xrightarrow[2]{1\|0} a, y \xrightarrow[2]{1\|0} b\}$

Points-to Information Computation

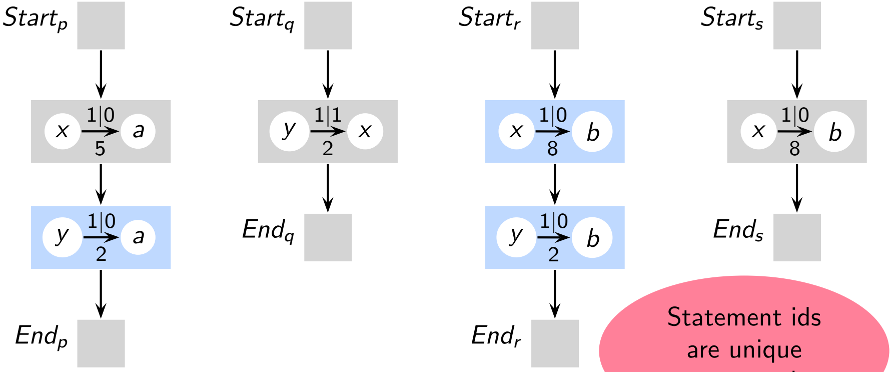


$Start_p$
$x \xrightarrow[5]{1|0} a$
$y \xrightarrow[2]{1|0} a$
End_p
$Start_q$
$y \xrightarrow[2]{1|1} x$
End_q
$Start_r$
$x \xrightarrow[8]{1|0} b$
$y \xrightarrow[2]{1|0} b$
End_r
$Start_s$
$x \xrightarrow[8]{1|0} b$
End_s

Stmt. id	Points-to Information
2	$\{y \xrightarrow[2]{1\|0} a, y \xrightarrow[2]{1\|0} b\}$

Context-sensitive points-to information for statement 2

Points-to Information Computation



$Start_p$
$x \xrightarrow[5]{1|0} a$
$y \xrightarrow[2]{1|0} a$
End_p
$Start_q$
$y \xrightarrow[2]{1|1} x$
End_q
$Start_r$
$x \xrightarrow[8]{1|0} b$
$y \xrightarrow[2]{1|0} b$
End_r
$Start_s$
$x \xrightarrow[8]{1|0} b$
End_s

Statement ids are unique across procedures

Stmt. id	Points-to Information
2	$\{y \xrightarrow[2]{1\|0} a, y \xrightarrow[2]{1\|0} b\}$

Context-sensitive points-to information for statement 2

Part VI

Future Work

Future Work

It would be useful to explore the possibilities:

- Restricting the GPG construction to live pointer variables for scalability.
- Studying the interactions between GPGs and the abstractions of a client analysis, say property proving application for verification.
- Extending the scope of GPG-based points-to analysis to concurrent programs such as Java programs containing threads.

Part VII

Thank You ☺

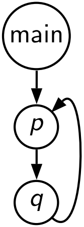
Part VIII

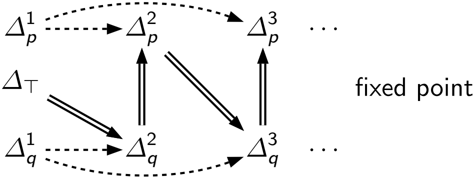
Extra Slides

Part IX

Advanced Features of Languages

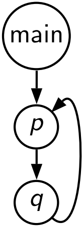
Handling Recursion





- Δ_p^1 contains recursive call to q and Δ_q^1 contains recursive call to p.
- Δ_q^2 is constructed from Δ_q^1 by using $\Delta_\top$ as a summary for call to p.
- Δ_p^2 is constructed from Δ_p^1 by using Δ_q^2 as a summary for call to q.
- Δ_q^3 is constructed from Δ_q^2 by using Δ_p^2 as a summary for call to p.
- Δ_p^3 is constructed from Δ_p^2 by using Δ_q^3 as a summary for call to q.
- $\ldots \Rightarrow$ Fixed point.

Handling Recursion

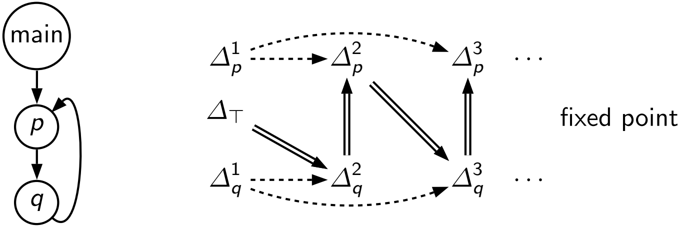


$$\begin{tikzcd}
\Delta_p^1 \arrow[r, dashed] \arrow[rr, dashed, bend left] & \Delta_p^2 \arrow[ddr, Rightarrow] & \Delta_p^3 & \cdots & \\
\Delta_\top \arrow[dr, Rightarrow] & & & & \text{fixed point} \\
\Delta_q^1 \arrow[r, dashed] \arrow[rr, dashed, bend right] & \Delta_q^2 \arrow[uu, Rightarrow] & \Delta_q^3 \arrow[uu, Rightarrow] & \cdots &
\end{tikzcd}$$

- Δ_p^1 contains recursiv Δ^1 contains recursive call to p.
- Δ_q^2 is const ummary for call to p.
- Δ_p^2 is con mary for call to q.
- Δ_q^3 is constr summary for call to p.
- Δ_p^3 is constructed from Δ_p by using Δ_q^3 as a summary for call to q.
- $\ldots \Rightarrow$ Fixed point.

Fixed point is reached when the data flow values converge, not when the resultant GPGs converge

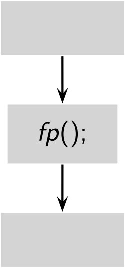
Handling Recursion



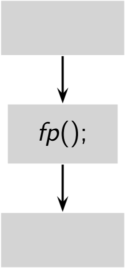
- Δ_p^1 contains recursive Δ_q^1 contains recursive call to p.
- Δ_q^2 is constructed ns a summary for call to p.
- Δ_p^2 is constructe a summary for call to q.
- Δ_q^3 is constructed as a summary for call to p.
- Δ_p^3 is constructed from Δ_p ng Δ_q^3 as a summary for call to q.
- … ⇒ Fixed point.

Fixed point is reached in a finite number of steps because the lattice is finite

Handling Function Pointers

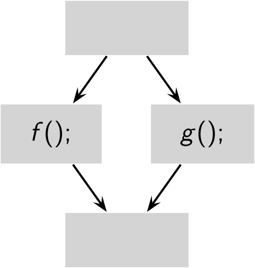


Handling Function Pointers



If pointees of fp are f and g

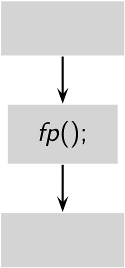
Handling Function Pointers



If pointees of *fp* are *f* and *g*

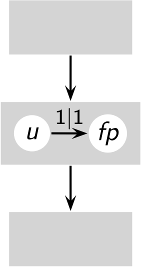
Calls to *f* and *g* could be recursive or non-recursive

Handling Function Pointers



If pointees of *fp* are not available locally

Handling Function Pointers



If pointees of *fp* are not available locally

Model indirect call as a use statement

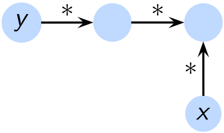
Handling Structures

Statement	Field-sensitive representation	Field-insensitive representation	Our choice
$x = *y$	$x \xrightarrow{[*]\|[*,*]} y$	$x \xrightarrow{1\|2} y$	$x \xrightarrow{1\|2} y$
$x = y \to n$	$x \xrightarrow{[*]\|[*,n]} y$	$x \xrightarrow{1\|2} y$	$x \xrightarrow{[*]\|[*,n]} y$

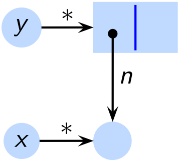
Handling Structures

Statement	Field-sensitive representation	Field-insensitive representation	Our choice
$x = *y$	$x \xrightarrow{[*]\|[*,*]} y$	$x \xrightarrow{1\|2} y$	$x \xrightarrow{1\|2} y$
$x = y \rightarrow n$	$x \xrightarrow{[*]\|[*,n]} y$	$x \xrightarrow{1\|2} y$	$x \xrightarrow{[*]\|[*,n]} y$

$$x \xrightarrow{[*]|[*,*]} y$$

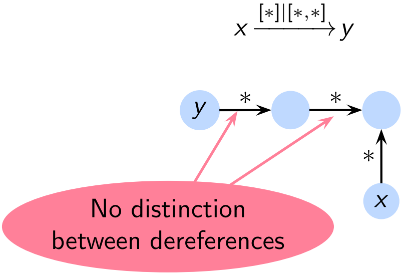


$$x \xrightarrow{[*]|[*,n]} y$$



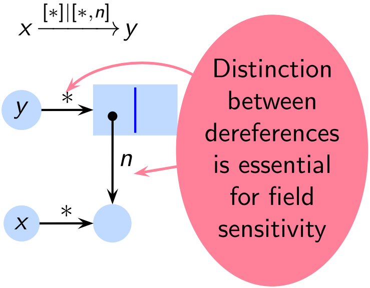
Handling Structures

Statement	Field-sensitive representation	Field-insensitive representation	Our choice
$x = *y$	$x \xrightarrow{[*]\mid[*,*]} y$	$x \xrightarrow{1\mid 2} y$	$x \xrightarrow{1\mid 2} y$
$x = y \rightarrow n$	$x \xrightarrow{[*]\mid[*,n]} y$	$x \xrightarrow{1\mid 2} y$	$x \xrightarrow{[*]\mid[*,n]} y$



$x \xrightarrow{[*]\mid[*,*]} y$

No distinction between dereferences



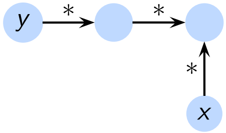
$x \xrightarrow{[*]\mid[*,n]} y$

Distinction between dereferences is essential for field sensitivity

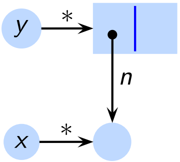
Handling Structures

Statement	Field-sensitive representation	Field-insensitive representation	Our choice
$x = *y$	$x \xrightarrow{[*]\vert[*,*]} y$	$x \xrightarrow{1\vert 2} y$	$x \xrightarrow{1\vert 2} y$
$x = y \rightarrow n$	$x \xrightarrow{[*]\vert[*,n]} y$	$x \xrightarrow{1\vert 2} y$	$x \xrightarrow{[*]\vert[*,n]} y$

$x \xrightarrow{1|2} y$



$x \xrightarrow{[*]|[*,n]} y$



Handling Structures

Statement	Field-sensitive representation	Field-insensitive representation	Our choice
$x = *y$	$x \xrightarrow{[*]\mid[*,*]} y$	$x \xrightarrow{1\mid 2} y$	$x \xrightarrow{1\mid 2} y$
$x = y \rightarrow n$	$x \xrightarrow{[*]\mid[*,n]} y$	$x \xrightarrow{1\mid 2} y$	$x \xrightarrow{[*]\mid[*,n]} y$

Imprecise representation

Handling Structures

Statement	Field-sensitive representation	Field-insensitive representation	Our choice
$x = *y$	$x \xrightarrow{[*]\|[*,*]} y$	$x \xrightarrow{1\|2} y$	$x \xrightarrow{1\|2} y$
$x = y \rightarrow n$	$x \xrightarrow{[*]\|[*,n]} y$	$x \xrightarrow{1\|2} y$	$x \xrightarrow{[*]\|[*,n]} y$

List operations are similar to the arithmetic operations performed on indirection levels for GPU composition

Miscellaneous Features

- Our heap abstraction consists of:
  - allocation-site-based-abstraction
  - *k*-limited indirection lists
- Arrays, pointer arithmetic, address escaped variables undergo weak updates. Hence their effect is over-approximated

Is Flow and Context Sensitivity Important? (I)

- Articles [Hind and Pioli 1998;2000; Hind 2001] claim that the better precision is not worth the price one has to pay for flow sensitivity
    - This claim is criticized because [Staiger-Stöhr 2013]:
        - Study performed on relatively small programs
        - Indirect strong updates not supported
        - Field-insensitive analyses
- Work by Hardekopf and Lin [2009, 2011] with very good results for flow-sensitive pointer analysis supports Staiger-Stöhr's theory

Is Flow and Context Sensitivity Important? (II)

- Lack of flow sensitivity in race detection algorithm [Naik-Aiken 2006] affects the synchronization idioms that the approach can handle precisely
- The pointer-flow used for taint analysis is ineffective without context sensitivity [Tripp-Pistoia 2009]
- A context sensitive call graph is more precise [Grove-Chambers 2001]

Is Flow and Context Sensitivity Important? (III)

- Jens Palsberg in his key note talk [SAS 2012] says that context-sensitive analysis improved the precision of “May Happen in Parallel Analysis”
- Object sensitivity [Milanova-Ryder 2005] shows significant improvement in the precision of side-effect analysis and call graph construction compared to a context-insensitive analysis

Context Based Bottom-up Approach

The need of multiple partial transfer functions (PTFs)

Example:

1. $x = *y$;
2. $*z = q$;
3. $p = *y$;

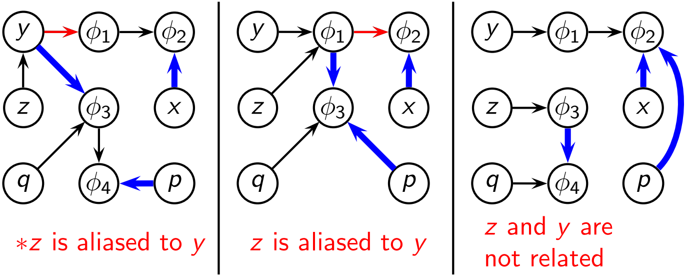
Two dereferences of y are separated by a possibly side-effect causing statement through z

Context Based Bottom-up Approach

The need of multiple partial transfer functions (PTFs)

Example:

1. $x = *y;$
2. $*z = q;$
3. $p = *y;$



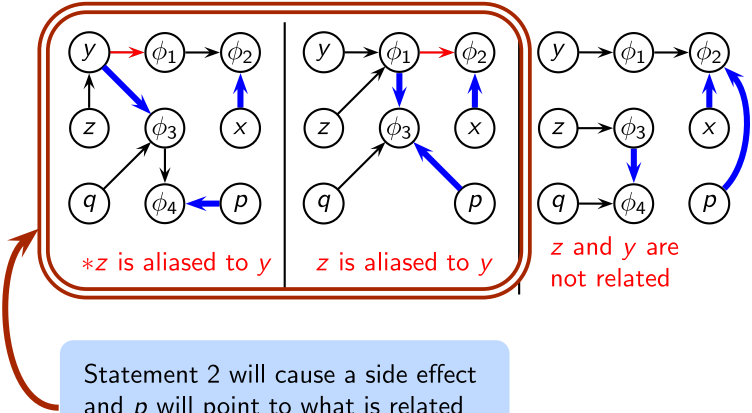
Red edges	$\Rightarrow$	killed information
Blue edges	$\Rightarrow$	information generated
Black edges	$\Rightarrow$	carried forward input information

Context Based Bottom-up Approach

The need of multiple partial transfer functions (PTFs)

Example:

1. $x = *y;$
2. $*z = q;$
3. $p = *y;$



$*z$ is aliased to y

z is aliased to y

z and y are not related

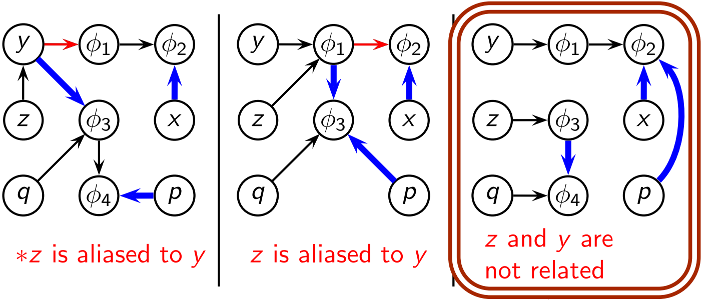
Statement 2 will cause a side effect and p will point to what is related to q and not what is related to x

Context Based Bottom-up Approach

The need of multiple partial transfer functions (PTFs)

Example:

1. $x = *y;$
2. $*z = q;$
3. $p = *y;$



$*z$ is aliased to y

z is aliased to y

z and y are not related

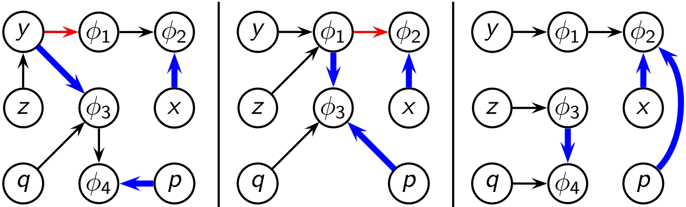
Statement 2 will NOT cause a side effect and p will point to what is related to x and not what is related to q

Context Based Bottom-up Approach

The need of multiple partial transfer functions (PTFs)

Example:

1. $x = *y;$
2. $*z = q;$
3. $p = *y;$



$*z$ is aliased to y

z is aliased to y

z and y are not related

Alias information eliminates data dependence, hence no control flow required

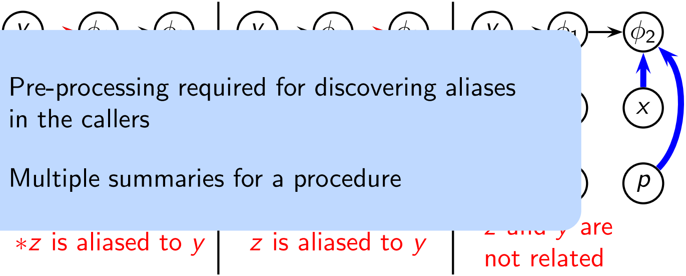
Only relevant aliases are considered

Context Based Bottom-up Approach

The need of multiple partial transfer functions (PTFs)

Examp

1. $x =$
2. $*z =$
3. $p =$



- Pre-processing required for discovering aliases in the callers
- Multiple summaries for a procedure

$*z$ is aliased to y | z is aliased to y | z and y are not related

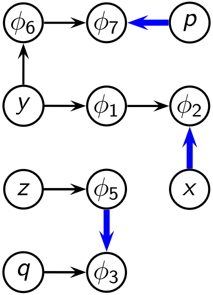
Only relevant aliases are considered

Context Independent Bottom-up Approach

Construction of a single flow-sensitive procedure summary

Example:

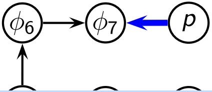
1. $x = *y;$
2. $*z = q;$
3. $p = *y;$



Different accesses of the same variable may require different placeholders

Context Independent Bottom-up Approach

Construction of a single flow-sensitive procedure summary



- Large number of placeholders
  $\Rightarrow$ size of procedure summary may be proportional to the # of statements
- A flow-insensitive version may require fewer placeholders $\Rightarrow$ affects precision

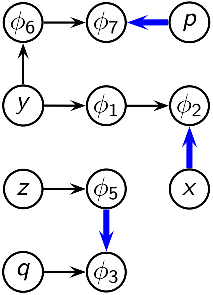
Different accesses of the same variable may require different placeholders

Context Independent Bottom-up Approach

Construction of a single flow-sensitive procedure summary

Example:

1. $x = *y;$
2. $*z = q;$
3. $p = *y;$



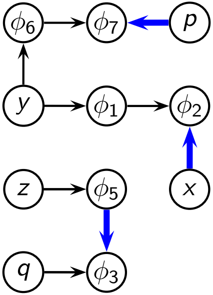
Ordering of generated edges is important

Context Independent Bottom-up Approach

Construction of a single flow-sensitive procedure summary

Example:

1. $x = *y;$
2. $*z = q;$
3. $p = *y;$



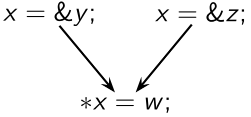
Ordering of generated edges is important

If $\phi_5 \rightarrow \phi_3$ is considered before $x \rightarrow \phi_2$, it will amount to swapping statements 1 and 2
Hence, x and p will be always be aliased ignoring the possible side-effect of statement 2

Strong and Weak Updates in Strength Reduction Optimization

- Kill occurs only when a single pointer is defined
- We call it a strong update

Strong and Weak Updates in Strength Reduction Optimization



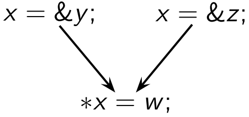
$x = \&y;$

$x = \&z;$

$*x = w;$

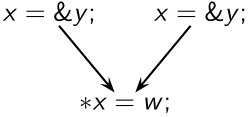
Weak Update

Strong and Weak Updates in Strength Reduction Optimization


$x = \&y;$ $x = \&z;$

$*x = w;$

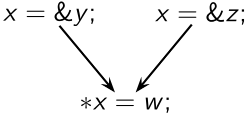
Weak Update


$x = \&y;$ $x = \&y;$

$*x = w;$

Strong Update

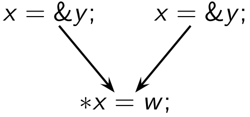
Strong and Weak Updates in Strength Reduction Optimization



$x = \&y;$ $x = \&z;$

$*x = w;$

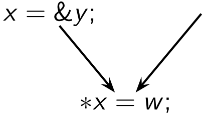
Weak Update



$x = \&y;$ $x = \&y;$

$*x = w;$

Strong Update

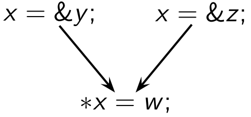


$x = \&y;$

$*x = w;$

?

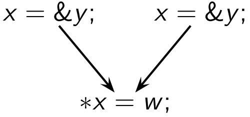
Strong and Weak Updates in Strength Reduction Optimization



$x = \&y;$ $x = \&z;$

$*x = w;$

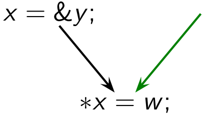
Weak Update



$x = \&y;$ $x = \&y;$

$*x = w;$

Strong Update



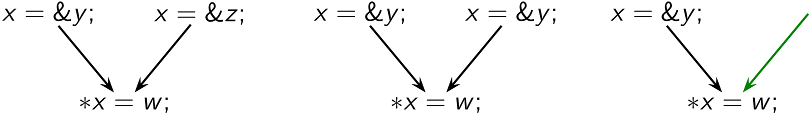
$x = \&y;$

$*x = w;$

Possibly weak update

Definition-free path for x

Strong and Weak Updates in Strength Reduction Optimization



$x = \&y;$ $x = \&z;$ → $*x = w;$

Weak Update

$x = \&y;$ $x = \&y;$ → $*x = w;$

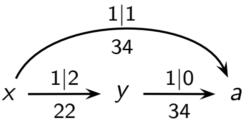
Strong Update

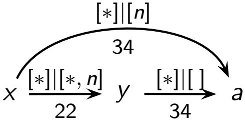
$x = \&y;$ → $*x = w;$

Possibly weak update

Definition-free path distinguishes between strong and weak updates

GPU Composition for Structures





- Difference of *indlev* of *y* (2 − 1) is computed.
- Difference (2 − 1) is positive.
- Add the difference to *indlev* of *a*.

- Remainder of *indlist* of *y* (*remainder*([∗], [∗, *n*])) is computed.
- [∗] is prefix of [∗, *n*].
- Append the remainder to *indlist* of *a*.