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
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
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.
Ideally, an analysis should be

- Sound
- Precise
- Scalable
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.
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.
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

Start

End

Pritam Gharat (IIT Bombay)
Flow Sensitivity Vs. Flow Insensitivity

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*  
⇒ *Flow insensitivity*
Context Sensitivity Vs. Context Insensitivity

\[
\begin{align*}
\text{Start}_s & \\
a &= \& b \\
c &= \& e \\
\text{Call } r & \\
z &= c \\
\end{align*}
\]

\[
\begin{align*}
\text{Start}_r & \\
f_r & \\
\text{End}_r & \\
\end{align*}
\]

\[
\begin{align*}
\text{Start}_t & \\
c &= \& d \\
\text{Call } r & \\
\text{End}_t & \\
\end{align*}
\]
Context Sensitivity Vs. Context Insensitivity

\[ \text{Start}_s \]
\[ a = \& b \]
\[ c = \& e \]
\[ \text{Call } r \]
\[ z = c \]

\[ \text{Start}_r \]
\[ \text{Call } r \]
\[ \text{End}_r \]

\[ \text{Start}_t \]
\[ c = \& d \]
\[ \text{End}_t \]
Context Sensitivity Vs. Context Insensitivity

\begin{align*}
\text{Start}_s \quad a &= \& b \\
\text{Start}_r \quad c &= \& e \\
\text{Call } r \\
\text{End}_r \quad z &= c \\
\text{Start}_t \quad c &= \& d \\
\text{Call } r \\
\text{End}_t
\end{align*}
Context Sensitivity Vs. Context Insensitivity

\[
\begin{align*}
\text{Start}_s & \quad a = \& b \\
& \quad c = \& e \\
& \text{Call } r \\
& \quad z = c \\
\text{Start}_r & \\
& \quad f_r \\
& \text{End}_r \\
\text{Start}_t & \quad c = \& d \\
& \text{Call } r \\
& \quad \text{End}_t
\end{align*}
\]
Context Sensitivity Vs. Context Insensitivity

\[ a = \& b \]
\[ c = \& e \]
Call \( r \)
\[ z = c \]

\[ \text{Start}_s \]
\[ \text{End}_s \]

\[ \text{Start}_r \]
\[ f_r \]
Call \( r \)
\[ \text{End}_r \]

\[ \text{Start}_t \]
\[ c = \& d \]
\[ \text{End}_t \]
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

\[ Start_p \quad a \ast b \quad Call \ q \quad End_p \]

\[ Start_q \quad a = \ldots \quad b + c \quad End_q \]

\[ Start_r \quad c \ast d \quad Call \ q \quad End_r \]
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

$p$

$a = ...$

$b + c$

End

$q$

Start

$r$

$c * d$

Call $q$

End

$r$

Expression $b + c$ is available in procedure $p$

Expression $a * b$ is not available in procedure $p$
Top-down Analysis for Available Expressions Analysis

Procedure $q$ needs to be processed multiple times

Expressions $b + c$ and $c \ast d$ are available in procedure $r$
Top-down Vs. Bottom-up Interprocedural Analysis

Bottom-Up Analysis for Available Expressions Analysis

\[ \text{Start}_p \]
\[ a \ast b \]
\[ \text{Call } q \]
\[ \text{End}_p \]

\[ \text{Start}_q \]
\[ a = \ldots \]
\[ b + c \]
\[ \text{End}_q \]

\[ \text{Start}_r \]
\[ c \ast d \]
\[ \text{Call } q \]
\[ \text{End}_r \]
Top-down Vs. Bottom-up Interprocedural Analysis

Bottom-Up Analysis for Available Expressions Analysis

Call is replaced by procedure summary

Using procedure summary of $g$ at call sites
Top-down Vs. Bottom-up Interprocedural Analysis

Bottom-Up Analysis for Available Expressions Analysis

- **Start\(_p\)**
  - Call is replaced by procedure summary
  - \(a \times b\)
  - Gen: \(b + c\)
  - Kill: \(a \times b\)
  - End\(_p\)

- **Start\(_q\)**
  - \(a = \ldots\)
  - \(b + c\)
  - End\(_q\)

- **Start\(_r\)**
  - \(c \times d\)
  - Gen: \(b + c\)
  - Kill: \(a \times b\)
  - End\(_r\)

Expression \(b + c\) is available in procedure \(p\)
Expression \(a \times b\) is not available in procedure \(p\)
Top-down Vs. Bottom-up Interprocedural Analysis

Bottom-Up Analysis for Available Expressions Analysis

Call is replaced by procedure summary

Expressions $b + c$ and $c \ast d$ are available in procedure $r$
A good procedure summary should be:
- Precise
- Compact
- Amenable to efficient application
- Reusable
Interprocedural Pointer Analysis

Interprocedural Analysis

Top-down Approaches

Pros: Caller's information available to callee

Cons: Procedure is analyzed multiple times

Bottom-up Approaches

Pros: Reusable procedure summary is constructed

Cons: Problems representing indirect accesses of pointees defined in callers

over the call graph
Interprocedural Pointer Analysis

Over the call graph

Top-down Approaches
Pros: Caller’s information available to callee
Cons: Procedure is analyzed multiple times

Interprocedural Analysis

Bottom-up Approaches
Pros: Reusable procedure summary is constructed
Cons: 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.

Pros:
- Caller's information available to callee

Cons:
- Procedure is analyzed multiple times

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

Problems representing indirect accesses of pointees defined in callers
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 $\Rightarrow$
  - 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 $\Rightarrow$
  - Can be eliminated and the
  - Control flow between the updates would be redundant

1. \( x = &a; \)
2. \( y = x; \)

\[ \downarrow \]

\[ x = &a; \]
\[ y = &a; \]
A flow-sensitive analysis requires control flow to be recorded between memory updates that share data dependence

- Data dependence exists $\Rightarrow$
  - Can be eliminated and the control flow between the updates would be redundant

- Data dependence does not exist $\Rightarrow$
  - Redundant memory updates can be eliminated

1. $x = &a$
2. $y = &b$
3. $x = &b$
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

\[
\begin{align*}
1. & \quad x = \& a; \\
2. & \quad y = \& b; \\
3. & \quad x = \& b; \\
\Rightarrow & \\
1. & \quad y = \& b; \\
2. & \quad x = \& b;
\end{align*}
\]
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. \( *x = \& a; \)
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
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
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 $\Rightarrow$
  Can be eliminated and the
  Control flow between the updates would be redundant

- Data dependence does not exist $\Rightarrow$
  Redundant memory updates can be eliminated

- Data dependence is unknown $\Rightarrow$
  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. $\ast x = \&a$
3. $z = y$
Memory and Memory Transformer

Memory in absence of pointers

\[ a \quad b \quad c \]

Memory in presence of pointers

\[ x \rightarrow y \rightarrow z \]

Memory Transformer

\[ x \rightarrow a \]
Memory and Memory Transformer

Memory in absence of pointers

\[ a \quad b \quad c \]

Memory in presence of pointers

\[ x \rightarrow y \quad z \]

Memory Transformer

\[ x \rightarrow \text{blue edge} \rightarrow a \]

For memory transformer,

- Blue edges ⇒ information generated
- Black edges ⇒ carried forward input information
Memory and Memory Transformer

Memory in absence of pointers

\[ a \quad b \quad c \]

Memory in presence of pointers

\[ x \quad y \quad z \]

Memory Transformer

\[ x \rightarrow \quad \text{Input Memory} \quad \rightarrow a \]

\[ x \rightarrow y \rightarrow z \rightarrow \text{Output Memory} \]

Pritam Gharat (IIT Bombay)

Generalized Points-to Graphs

September 2018
Part II

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

\[ x = y \Rightarrow \phi_1 \]

\( \phi_1 \) is a placeholder.
Accesses of pointees that are defined in the callers are represented using placeholders e.g., $x = y \Rightarrow \phi_1$

- $\phi_1$ 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., $x = y \Rightarrow \phi_1$

- $\phi_1$ 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

Re-use of a placeholder for a flow sensitive summary flow function depends on the aliases in the calling contexts.

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

Generalized Points-to Graphs
Representing Basic Pointer Assignments using the Generalized Points-to Updates

<table>
<thead>
<tr>
<th>General Case</th>
<th>Specific Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU $x \frac{i\mid j}{s} y$</td>
<td><strong>Pointer assignment</strong></td>
</tr>
<tr>
<td>$s: x = &amp;y$</td>
<td>$x \frac{1\mid 0}{s} y$</td>
</tr>
<tr>
<td>$s: x = y$</td>
<td>$x \frac{1\mid 1}{s} y$</td>
</tr>
<tr>
<td>$s: x = *y$</td>
<td>$x \frac{1\mid 2}{s} y$</td>
</tr>
<tr>
<td>$s: *x = y$</td>
<td>$x \frac{2\mid 1}{s} y$</td>
</tr>
</tbody>
</table>
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

<table>
<thead>
<tr>
<th>General Case</th>
<th>Specific Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td>GPU (x \xrightarrow{i</td>
<td>j}{s} y)</td>
</tr>
<tr>
<td>(s: x = &amp;y)</td>
<td>(x \xrightarrow{1</td>
</tr>
<tr>
<td>(s: x = y)</td>
<td>(x \xrightarrow{1</td>
</tr>
<tr>
<td>(s: x = *y)</td>
<td>(x \xrightarrow{1</td>
</tr>
<tr>
<td>(s: *x = y)</td>
<td>(x \xrightarrow{2</td>
</tr>
</tbody>
</table>

- The direction in a GPU is to distinguish between what is being defined to what is being read.
- For pointer analysis, \(i = 0\) does not exist.
- Classical points-to updates are a special case of generalized points-to update with \(i = 1\) and \(j = 0\).

The GPU represents both memory and memory transformer.
All variables are global

Red nodes are known named locations
All variables are global

Red nodes are known named locations
Blue nodes are placeholders denoting unknown locations
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

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

All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

Pritam Gharat (IIT Bombay)
Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

All variables are global
Red nodes are known named locations
Blue nodes are placeholders denoting unknown locations

```
f()
{    *x = y
    }
```
Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

All variables are global

Red nodes are known named locations

Blue nodes are placeholders denoting unknown locations

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

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

All variables are global
Red nodes are known named locations
Blue nodes are placeholders denoting unknown locations

```
f()
{
    *x = y
}
```
Classical Points-to Updates: A Low Level Abstraction of Memory for Points-to Analysis

All variables are global

Red nodes are known named locations
Blue nodes are placeholders denoting unknown locations

```c
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

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

```c
f()
{
    *x = y
}
```
**Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis**

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

```plaintext
f()
{
    *x = y
}
```
Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

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.

```plaintext
f()
{
    *x = y
}
```
Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

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

```c
f()
{
    *x = y
}
```
Generalized Points-to Updates: A High Level Abstraction of Memory for Points-to Analysis

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

\[
f() \{
    *x = y
\}
\]

\[
\phi_1 \rightarrow a \rightarrow b \rightarrow y
\]
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

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

f()
{
    \*x = y
}

Pritam Gharat (IIT Bombay)
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 $c \circ p$; performed only when they share a common node called the *pivot*.
**GPU Composition**

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

  \[
  x = \& y; \\
  z = *x;
  \]
GPU Composition

- Represented by \( c \circ p \); performed only when they share a common node called the pivot

\[
x = \& y; \\
z = \ast x;
\]

GPG

\[
\begin{align*}
x & \quad 1|0 \\
\xrightarrow{p} & \quad y
\end{align*}
\]
GPU Composition

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

\[
x = \& y; \\
\text{and} \\
\begin{align*}
1 & \\
\rightarrow & \\
\text{and} \\
\end{align*} \\
\begin{align*}
z & = *x; \\
& \\
\end{align*}
\]
GPU Composition

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

\[
x = \& y; \\
z = \ast x;
\]
Represented by $c \circ p$; performed only when they share a common node called the *pivot*.

$x = \& y$; 
$z = *x$;
GPU Composition

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

\[ x = \& y; \]
\[ z = *x; \]
GPU Composition

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

- Reduces the $\text{indlev}$ of $c$ by using information from $p$.
  - Eliminating pivot and creating a reduced GPU $r$ between other two nodes by using pivot as a bridge.

$c \Rightarrow$ Consumer GPU, $p \Rightarrow$ Producer GPU

\[ x = \& y; \]
\[ z = \ast x; \]
GPU Composition

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

- Reduces the indlev of $c$ by using information from $p$
  - Eliminating pivot and creating a reduced GPU $r$ between other two nodes by using pivot as a bridge

$c \Rightarrow$ Consumer GPU, $p \Rightarrow$ Producer GPU
GPU Composition

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

\[ r = c \circ p \]

\[
\begin{align*}
x &= \& y; \\
z &= \* x;
\end{align*}
\]
GPU Composition

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

- Reduces the $\text{indlev}$ of $c$ by using information from $p$
  - Eliminating pivot and creating a reduced GPU $r$ between other two nodes by using pivot as a bridge

- Requires the $\text{indlevs}$ of the pivot in both the GPUs to be made same

$x = \& y$

$z = *x$
GPU Composition

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

- Reduces the indlev of $c$ by using information from $p$.
  - Eliminating pivot and creating a reduced GPU $r$ between other two nodes by using pivot as a bridge.

- Requires the indlevs of the pivot in both the GPUs to be made same.

\[ x = \& y; \]
\[ z = \ast x; \]
\[ r = c \circ p \]
Represented by $c \circ p$; performed only when they share a common node called the **pivot**

Reduces the **indlev** of $c$ by using information from $p$

- Eliminating pivot and creating a reduced GPU $r$ between other two nodes by using **pivot** as a bridge

Requires the **indlevs** of the pivot in both the GPUs to be made same

$x = \& y$;

$z = *x$;
GPU Composition

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

Memory Graph

- $x = \& y$;
- $z = *x$;
GPU Composition

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

```
\begin{align*}
x &= \& y; \\
z &= *x;
\end{align*}
```
GPU Composition

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

\[ x = \& y; \]
\[ z = \ast x; \]

GPUs $r$ and $c$ are equivalent in the context of $p$. 
GPU Composition

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

\[
\begin{align*}
x &= \& y; \\
z &= *x;
\end{align*}
\]
GPU Composition

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

\[ x = \& y; \]
\[ z = \ast x; \]

GPU reduction is a series of GPU compositions
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 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)

A GPG is a graph with:
- **Nodes** called as generalized points-to blocks (GPBs)
- **Edges** representing control flow between GPBs

A GPG is analogous to a CFG of a procedure

**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...
Generalized Points-to Graphs (GPGs) I

A GPG is a graph with:
- **Nodes:** called as generalized points-to blocks (GPBs)
- **Edges:** representing control flow between GPBs

A GPG is analogous to a CFG of a procedure:

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

Second difference:

A GPG is analogous to a CFG of a procedure.

```
GPG  <->  CFG  
|      |      |
| GPB  +  BB |
|         |      |
| GPU  <->  Ptr. Assgn. |
```
Construction of Initial GPGs:

- Non-pointer assignments and condition tests are removed
- Each pointer assignment $s$ is transliterated to its GPU ($\gamma_s$)
- 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
<table>
<thead>
<tr>
<th>Optimizations</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Flow Analysis</td>
</tr>
<tr>
<td>GPU Operations</td>
</tr>
<tr>
<td>Abstractions</td>
</tr>
</tbody>
</table>
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

Optimizations
- Inlining callee GPGs
- Strength Reduction

Data Flow Analysis
- Reaching GPUs Analysis without Blocking
- Reaching GPUs Analysis with Blocking

GPU Operations
- GPU Creation
- GPU Reduction
- GPU Composition

Abstractions

Pritam Gharat (IIT Bombay)
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
- GPU Creation
- GPU Reduction
- GPU Composition

Abstractions
- indlev
- indlist
The Big Picture View of GPG Construction

Optimizations
- Inlining callee GPGs
- Strength Reduction
- Dead GPU Elimination
- Empty GPB Elimination

Data Flow Analysis
- Reaching GPUs Analysis without Blocking
- Reaching GPUs Analysis with Blocking

GPU Operations
- GPU Creation
- GPU Reduction
- GPU Composition

Abstractions
- indlev
- indlist
The Big Picture View of GPG Construction

Optimizations
- Inlining callee GPGs
- Strength Reduction
- Dead GPU Elimination
- Empty GPB Elimination
- GPB Coalescing

Data Flow Analysis
- Reaching GPUs Analysis without Blocking
- Reaching GPUs Analysis with Blocking
- Coalescing Analysis

GPU Operations
- GPU Creation
- GPU Reduction
- GPU Composition

Abstractions
- indlev
- indlist

Pritam Gharat (IIT Bombay)
## The Big Picture View of GPG Construction

### Optimizations
- Inlining callee GPGs
- Strength Reduction
- Dead GPU Elimination
- Empty GPB Elimination
- GPB Coalescing
- Back Edge Removal

### Redundancy Elimination
- Coalescing Analysis
- Essential Back Edges Analysis

### Data Flow Analysis
- Reaching GPUs Analysis without Blocking
- Reaching GPUs Analysis with Blocking

### GPU Operations
- GPU Creation
- GPU Reduction
- GPU Composition

### Abstractions
- indlev
- indlist
The Big Picture View of GPG Construction

- **Optimizations**
  - Inlining callee GPGs
  - Strength Reduction
  - Dead GPU Elimination
  - Empty GPB Elimination
  - GPB Coalescing
  - Back Edge Removal

- **Data Flow Analysis**
  - Reaching GPUs Analysis without Blocking
  - Reaching GPUs Analysis with Blocking
  - Coalescing Analysis
  - Essential Back Edges Analysis

- **GPU Operations**
  - GPU Creation
  - GPU Reduction
  - GPU Composition

- **Abstractions**
  - `indlev`
  - `indlist`
The Big Picture View of GPG Construction

Published in SAS 2016

GPU Operations

GPU Creation

GPU Reduction

GPU Composition

Abstractions

indlev

indlist
The Big Picture View of GPG Construction

Optimizations
- Inlining callee GPGs
- Strength Reduction
- Dead GPU Elimination
- Empty GPB Elimination
- GPB Coalescing
- Back Edge Removal

Data Flow Analysis
- Reaching GPUs Analysis without Blocking
- Reaching GPUs Analysis with Blocking
- Coalescing Analysis
- Essential Back Edge Analysis

GPU Operations
- GPU Creation
- GPU Reduction
- GPU Composition

Abstractions
- indlev
- indlist

Submitted to TOPLAS

Pritam Gharat (IIT Bombay)

Generalized Points-to Graphs
September 2018
GPGs Across Optimizations

CFG of proc f

\[ x = \&a; \]

\[ g(); \]

\[ x = \&b; \]

CFG of proc g

\[ y = x; \]
GPGs Across Optimizations

CFG of proc f

\[
x = \& a;
\]
\[
g();
\]
\[
x = \& b;
\]

Initial GPG of proc f

\[
x \xrightarrow{1|0 \ 1} a
\]
\[
g();
\]
\[
x \xrightarrow{1|0 \ 8} b
\]

CFG of proc g

\[
y = x;
\]
GPGs Across Optimizations

CFG of proc f

\[ x = \&a; \]
\[ g(); \]
\[ x = \&b; \]

Initial GPG of proc f

\[ x \rightarrow \]
\[ a \]
\[ x \rightarrow \]
\[ g(); \]
\[ x \rightarrow \]
\[ b \]

After Call Inlining

\[ x \rightarrow \]
\[ a \]
\[ y \rightarrow \]
\[ x \]
\[ x \rightarrow \]
\[ b \]

CFG of proc g

\[ y = x; \]
GPGs Across Optimizations

CFG of proc f

\[ x = \& a; \]
\[ g(); \]
\[ x = \& b; \]

Initial GPG of proc f

\[ x \xrightarrow{1} a \]
\[ g(); \]
\[ x \xrightarrow{1} b \]
\[ x \xrightarrow{0} 8 \]

After Call Inlining

\[ x \xrightarrow{1} a \]
\[ y \xrightarrow{1} x \]
\[ x \xrightarrow{1} b \]
\[ x \xrightarrow{0} 8 \]

After Strength Reduction

\[ x \xrightarrow{1} a \]
\[ y \xrightarrow{1} a \]
\[ x \xrightarrow{1} b \]
\[ x \xrightarrow{0} 8 \]
GPGs Across Optimizations

**CFG of proc f**

```
x = &a;
g();
x = &b;
```

**Initial GPG of proc f**

```
\[
\begin{array}{c}
x \xleftarrow{1|0} a \\
x \xleftarrow{1|0} b \\
y \xleftarrow{1|1} x \\
x \xleftarrow{1|0} b \\
y \xleftarrow{1|0} a
\end{array}
\]
```

**After Call Inlining**

```
\[
\begin{array}{c}
x \xleftarrow{1|0} a \\
y \xleftarrow{1|1} x \\
x \xleftarrow{1|0} b \\
x \xleftarrow{1|0} b
\end{array}
\]
```

**After Strength Reduction**

```
\[
\begin{array}{c}
x \xleftarrow{1|0} a \\
y \xleftarrow{1|0} a \\
x \xleftarrow{1|0} b \\
x \xleftarrow{1|0} b
\end{array}
\]
```

**CFG of proc g**

```
y = x;
```

**After Dead GPU Elimination**

```
\[
\begin{array}{c}
y \xleftarrow{1|0} a \\
x \xleftarrow{1|0} b
\end{array}
\]
```

Pritam Gharat ( IIT Bombay)

Generalized Points-to Graphs

September 2018 25 / 50
GPGs Across Optimizations

**CFG of proc f**
- \( x = &a; \)
- \( g(); \)
- \( x = &b; \)

**Initial GPG of proc f**
- \( x \xrightarrow{1|0\,1} a \)
- \( g(); \)
- \( x \xrightarrow{1|0\,8} b \)

**After Call Inlining**
- \( x \xrightarrow{1|0\,1} a \)
- \( y \xrightarrow{1|1\,2} x \)
- \( x \xrightarrow{1|0\,8} b \)

**After Strength Reduction**
- \( x \xrightarrow{1|0\,1} a \)
- \( y \xrightarrow{1|0\,2} a \)
- \( x \xrightarrow{1|0\,8} b \)

**CFG of proc g**
- \( y = x; \)

**After Dead GPU Elimination**
- \( y \xrightarrow{1|0\,2} a \)
- \( x \xrightarrow{1|0\,8} b \)

**After Empty GPB Elimination**
- \( y \xrightarrow{1|0\,2} a \)
- \( x \xrightarrow{1|0\,8} b \)
GPGs Across Optimizations

CFG of proc f

\( x = \& a; \)

\( g(); \)

\( x = \& b; \)

Initial GPG of proc f

\[
\begin{align*}
  x & \xrightarrow{1} a \\
  y & \xrightarrow{1} x \\
  x & \xrightarrow{8} b
\end{align*}
\]

After Call Inlining

\[
\begin{align*}
  y & \xrightarrow{2} a \\
  y & \xrightarrow{1} x \\
  x & \xrightarrow{8} b
\end{align*}
\]

After Strength Reduction

\[
\begin{align*}
  y & \xrightarrow{2} a \\
  y & \xrightarrow{1} x \\
  x & \xrightarrow{8} b
\end{align*}
\]

After Dead GPU Elimination

\[
\begin{align*}
  y & \xrightarrow{2} a \\
  x & \xrightarrow{8} b
\end{align*}
\]

After Empty GPB Elimination

\[
\begin{align*}
  y & \xrightarrow{2} a \\
  x & \xrightarrow{8} b
\end{align*}
\]

After Coalescing

\[
\begin{align*}
  y & \xrightarrow{2} a \\
  x & \xrightarrow{8} b
\end{align*}
\]
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 \textit{indlev} “1|0”).
Effectiveness of GPGs

- Effectiveness measured in terms of optimized GPGs:
  - Percentage of procedures vs. # of GPBs
  - Percentage of procedures vs. # of GPUs

- Context independent information:
  - Percentage of context independent information vs. GPG percentage

- Benchmarks:
  - lbm, mcf, libquantum, bzip2, milc, sjeng, hmmer, h264ref, gobmk

Pritam Gharat (IIT Bombay)
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

**Graphs:**
- **Graph 1:** Ratio of GPUs and stmts in GPGs and CFGs after call inlining (in terms of percentage).
- **Graph 2:** Ratio of GPUs and stmts in optimized GPGs and CFGs after call inlining (in terms of percentage).
- **Graph 3:** Ratio of GPUs in optimized GPGs and GPGs after call inlining (in terms of percentage).

**Legend:**
- lbm
- mcf
- libquantum
- bzip2
- milc
- sjeng
- hmmmer
- h264ref
- gobmk

**Text:** 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

- Ratio of control flow edges in GPGs and CFGs after call inlining (in terms of percentage)
- Ratio of control flow edges in optimized GPGs and CFGs after call inlining (in terms of percentage)
- Ratio of control flow edges in optimized GPGs and GPGs after call inlining (in terms of percentage)
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

![Bar Chart]

- **Avg. of points-to pairs per procedure**
- Benchmarks: lbm, mcf, libquantum, bzip2, milc, sjeng, hmmmer, h264ref, gobmk
- Algorithms: FSCS, FICI, FICS

Pritam Gharat (IIT Bombay)
Data Measurements

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

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
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

\[ \text{Start}_p \quad \text{Start}_q \quad \text{Start}_r \quad \text{Start}_s \]

\[ x \xrightarrow{1|0} a \quad y \xrightarrow{1|1} x \quad \text{Call } s \quad x \xrightarrow{1|0} b \]

\[ \text{Call } q \quad \text{End}_q \quad \text{Call } q \quad \text{End}_s \]

\[ \text{End}_p \quad \text{End}_r \]

Pritam Gharat (IIT Bombay)
Points-to Information Computation

Start $p$

$\begin{align*}
&x \xleftarrow{1} 0 \\
&\text{Call } q
\end{align*}$

End $p$

$\begin{align*}
&\text{Call } q
\end{align*}$

Start $q$

$\begin{align*}
&y \xleftarrow{1} 1 \\
&\text{Call } q
\end{align*}$

End $q$

$\begin{align*}
&\text{Call } q
\end{align*}$

Start $r$

$\begin{align*}
&\text{Call } s
\end{align*}$

Start $s$

$\begin{align*}
&x \xleftarrow{1} 0 \\
&\text{Call } q
\end{align*}$

End $s$

$\begin{align*}
&\text{Call } q
\end{align*}$

$\begin{align*}
&\text{Call } s
\end{align*}$

Pritam Gharat (IIT Bombay)
Points-to Information Computation

After call inlining
Use pushed towards definition in a caller

Use and definition pushed in a common context
Points-to Information Computation

After strength reduction optimization
Points-to Information Computation

\[
\begin{align*}
\text{Start}_p & \quad \text{Start}_q & \quad \text{Start}_r & \quad \text{Start}_s \\
\text{End}_p & \quad \text{End}_q & \quad \text{End}_r & \quad \text{End}_s
\end{align*}
\]

Stmt. id | Points-to Information
---|---
2 | \{y \xrightarrow{\frac{1}{2}} a, y \xrightarrow{\frac{1}{2}} b\}
Points-to Information Computation

Context-sensitive points-to information for statement 2

Stmt. id | Points-to Information
--- | ---
2 | \( \{ y \xrightarrow{1|0}{2} a, y \xrightarrow{1|0}{2} b \} \)
Points-to Information Computation

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

Context-sensitive points-to information for statement 2

Statement ids are unique across procedures
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

- $\Delta_p^1$ contains recursive call to $q$ and $\Delta_q^1$ contains recursive call to $p$.
- $\Delta_q^2$ is constructed from $\Delta_q^1$ by using $\Delta_T$ as a summary for call to $p$.
- $\Delta_p^2$ is constructed from $\Delta_p^1$ by using $\Delta_q^2$ as a summary for call to $q$.
- $\Delta_q^3$ is constructed from $\Delta_q^2$ by using $\Delta_p^2$ as a summary for call to $p$.
- $\Delta_p^3$ is constructed from $\Delta_p^2$ by using $\Delta_q^3$ as a summary for call to $q$.
- $\ldots \Rightarrow$ Fixed point.
Handling Recursion

- $\Delta_1^p$ contains recursive call to $q$, and $\Delta_1^q$ contains recursive call to $p$.
- $\Delta_2^q$ is constructed from $\Delta_1^q$ by using $\Delta_2^p$ as a summary for call to $p$.
- $\Delta_3^q$ is constructed from $\Delta_2^q$ by using $\Delta_3^p$ as a summary for call to $p$.
- $\Delta_2^p$ is constructed from $\Delta_1^p$ by using $\Delta_2^q$ as a summary for call to $q$.
- $\Delta_3^p$ is constructed from $\Delta_2^p$ by using $\Delta_3^q$ as a summary for call to $q$.
- ... $\Rightarrow$ Fixed point.

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

- $\Delta_1^p$ contains recursive call to $p$.
- $\Delta_1^q$ contains recursive call to $q$.
- $\Delta_2^q$ is constructed from $\Delta_1^q$ using $\Delta_2^p$ as a summary for call to $p$.
- $\Delta_2^p$ is constructed from $\Delta_1^p$ using $\Delta_2^q$ as a summary for call to $q$.
- $\Delta_3^q$ is constructed from $\Delta_2^q$ using $\Delta_3^p$ as a summary for call to $p$.
- $\Delta_3^p$ is constructed from $\Delta_2^p$ using $\Delta_3^q$ as a summary for call to $q$.
- ... $\Rightarrow$ Fixed point.

Fixed point is reached in a finite number of steps because the lattice is finite.
Handling Function Pointers

```
x = &a;  
a fp();  
```

Pritam Gharat (IIT Bombay)
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

```
x = &a;
a fp();
```

If pointees of `fp` are not available locally
Handling Function Pointers

If pointees of \( fp \) are not available locally:

\[ u \xrightarrow{1|1} fp \]

Model indirect call as a use statement.
## Handling Structures

<table>
<thead>
<tr>
<th>Statement</th>
<th>Field-sensitive representation</th>
<th>Field-insensitive representation</th>
<th>Our choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = \ast y$</td>
<td>$x \xrightarrow{[*],[\ast,\ast]} y$</td>
<td>$x \xrightarrow{1</td>
<td>2} y$</td>
</tr>
<tr>
<td>$x = y \rightarrow n$</td>
<td>$x \xrightarrow{[*],[\ast,n]} y$</td>
<td>$x \xrightarrow{1</td>
<td>2} y$</td>
</tr>
</tbody>
</table>
### Handling Structures

<table>
<thead>
<tr>
<th>Statement</th>
<th>Field-sensitive representation</th>
<th>Field-insensitive representation</th>
<th>Our choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = *y$</td>
<td>$x \xrightarrow{[<em>][</em>,*]} y$</td>
<td>$x \xrightarrow{\frac{1}{2}} y$</td>
<td>$x \xrightarrow{\frac{1}{2}} y$</td>
</tr>
<tr>
<td>$x = y \rightarrow n$</td>
<td>$x \xrightarrow{[<em>][</em>,n]} y$</td>
<td>$x \xrightarrow{\frac{1}{2}} y$</td>
<td>$x \xrightarrow{[<em>][</em>,n]} y$</td>
</tr>
</tbody>
</table>

![Diagram](image_url)
## Handling Structures

<table>
<thead>
<tr>
<th>Statement</th>
<th>Field-sensitive representation</th>
<th>Field-insensitive representation</th>
<th>Our choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = *y$</td>
<td>$x \mathcal{F} { [<em>,</em>,*] } \rightarrow y$</td>
<td>$x \mathcal{F} { [1,2] } \rightarrow y$</td>
<td>$x \mathcal{F} { [1,2] } \rightarrow y$</td>
</tr>
<tr>
<td>$x = y \rightarrow n$</td>
<td>$x \mathcal{F} { [*,n] } \rightarrow y$</td>
<td>$x \mathcal{F} { [1,2] } \rightarrow y$</td>
<td>$x \mathcal{F} { [*,n] } \rightarrow y$</td>
</tr>
</tbody>
</table>

No distinction between dereferences

Distinction between dereferences is essential for field sensitivity

- No distinction between dereferences
- Distinction between dereferences is essential for field sensitivity
### Handling Structures

<table>
<thead>
<tr>
<th>Statement</th>
<th>Field-sensitive representation</th>
<th>Field-insensitive representation</th>
<th>Our choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = \ast y$</td>
<td>$x \xrightarrow{[\ast]{\ast,\ast}} y$</td>
<td>$x \xrightarrow{1</td>
<td>2} y$</td>
</tr>
<tr>
<td>$x = y \rightarrow n$</td>
<td>$x \xrightarrow{[\ast]{\ast,n}} y$</td>
<td>$x \xrightarrow{1</td>
<td>2} y$</td>
</tr>
</tbody>
</table>

**Diagram:**

- **Field-sensitive:**
  - $x \xrightarrow{1|2} y$
  - $y \xrightarrow{\ast} \ast \xrightarrow{\ast} \ast$
  - $x \xrightarrow{\ast}$

- **Field-insensitive:**
  - $x \xrightarrow{[\ast]\{\ast,n\}} y$
  - $y \xrightarrow{\ast} \ast \xrightarrow{\ast} \ast$
  - $x \xrightarrow{\ast}$

- **Our choice:**
  - $x \xrightarrow{1|2} y$
  - $y \xrightarrow{\ast} \ast \xrightarrow{\ast} \ast$
  - $x \xrightarrow{\ast}$

*Illustrations of the diagrams are shown.*
## Handling Structures

<table>
<thead>
<tr>
<th>Statement</th>
<th>Field-sensitive representation</th>
<th>Field-insensitive representation</th>
<th>Our choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x = *y )</td>
<td>( x \xrightarrow{[<em>][</em>,*]} y )</td>
<td>( x \xrightarrow{1</td>
<td>2} y )</td>
</tr>
<tr>
<td>( x = y \rightarrow n )</td>
<td>( x \xrightarrow{[<em>][</em>,n]} y )</td>
<td>( x \xrightarrow{1</td>
<td>2} y )</td>
</tr>
</tbody>
</table>

**Imprecise representation**
### Handling Structures

<table>
<thead>
<tr>
<th>Statement</th>
<th>Field-sensitive representation</th>
<th>Field-insensitive representation</th>
<th>Our choice</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x = *y$</td>
<td>$x \xrightarrow{[<em>][</em>,*]} y$</td>
<td>$x \xrightarrow{1</td>
<td>2} y$</td>
</tr>
<tr>
<td>$x = y \rightarrow n$</td>
<td>$x \xrightarrow{[<em>][</em>,n]} y$</td>
<td>$x \xrightarrow{1</td>
<td>2} y$</td>
</tr>
</tbody>
</table>

List operations are similar to the arithmetic operations performed on indirection levels for GPU composition.
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.
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.
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].
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. $x = *y$; 
   Two dereferences of $y$ are separated by a possibly 
   side-effect causing statement through $z$
2. $*z = q$; 
3. $p = *y$;
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

Red edges \( \Rightarrow \) killed information
Blue edges \( \Rightarrow \) information generated
Black edges \( \Rightarrow \) carried forward input information
The need of multiple partial transfer functions (PTFs)

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

* $z$ 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 $x$
The need of multiple partial transfer functions (PTFs)

Example:
1. \( x = \ast y \);
2. \( \ast z = q \);
3. \( p = \ast y \);

\( \ast z \) is aliased to \( y \)
\( z \) is aliased to \( y \)

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

\( z \) and \( y \) are not related.
The need of multiple partial transfer functions (PTFs)

Example:
1. $x = \ast y$;
2. $\ast z = q$;
3. $p = \ast y$;

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

Only relevant aliases are considered

Alias information eliminates data dependence, hence no control flow required
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 \( y \) and \( z \) is aliased to \( y \)

Only relevant aliases are considered

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

\( z \) and \( y \) are not related
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
Construction of a single flow-sensitive procedure summary

Different accesses of the same variable may require different placeholders

- Large number of placeholders
  ⇒ size of procedure summary may be proportional to the # of statements

- A flow-insensitive version may require fewer placeholders ⇒ affects precision
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. \( p = *y; \)

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.

Ordering of generated edges is important.
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

\[
x = \& y; \quad x = \& z;
\]

\[\ast x = w;\]

Weak Update
Strong and Weak Updates in Strength Reduction Optimization

\[
x = \& y; \quad x = \& z; \quad x = \& y; \quad x = \& y;
\]

\[
* x = w; \quad * x = w;
\]

Weak Update

Strong Update
Strong and Weak Updates in Strength Reduction Optimization

\[ x = \& y; \quad x = \& z; \]
\[ *x = w; \]
\[ x = \& y; \]
\[ *x = w; \]
\[ x = \& y; \]
\[ *x = w; \]

Weak Update

Strong Update

?
Strong and Weak Updates in Strength Reduction Optimization

Weak Update

$\text{Weak Update: } x = \& y; \quad x = \& z; \quad!*x = w;$

Strong Update

$\text{Strong Update: } x = \& y; \quad x = \& y; \quad!*x = w;$

Possibly weak update

$\text{Possibly weak update: } x = \& y; \quad!*x = w;$

Definition-free path for $x$
**Strong and Weak Updates in Strength Reduction Optimization**

Weak Update

\[ x = \& y; \quad x = \& z; \quad \ast x = w; \]

Strong Update

\[ x = \& y; \quad x = \& y; \quad \ast x = w; \]

Possibly weak update

\[ x = \& y; \quad \ast x = w; \]

Definition-free path distinguishes between strong and weak updates
Difference of \textit{indlev} of $y$ (2 – 1) is computed.

Difference (2 – 1) is positive.

Add the difference to \textit{indlev} of $a$.

Remainder of \textit{indlist} of $y$ ($\text{remainder}([*],[*, n])$) is computed.

[*] is prefix of [*, n].

Append the remainder to \textit{indlist} of $a$. 