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

A thesis submitted in partial fulfillment of
the requirements for the degree of

Doctor of Philosophy

by

Pritam M. Gharat
(Roll No. 113050036)

Under the guidance of
Prof. Uday P. Khedker

DEPARTMENT OF COMPUTER SCIENCE & ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
July 2018

Approval Sheet

Dissertation entitled **'Generalized Points-to Graph: A New Abstraction of Memory in Presence of Pointers'** by Ms. Pritam M. Gharat is approved for the degree of Doctor of Philosophy.

Examiners

____

____

Supervisor

____

____

Chairman

____

Date:________

Place: _______

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY, INDIA

A Declaration of Academic Honesty and Integrity

I declare that this written submission represents my ideas in my own words and where others' ideas or words have been included, I have adequately cited and referenced the original sources. I also declare that I have adhered to all principles of academic honesty and integrity and have not misrepresented or fabricated or falsified any idea/data/fact/source in my submission. I understand that any violation of the above will be a cause for disciplinary action by the Institute and can also evoke penal action from the sources which have thus not been properly cited or from whom proper permission has not been taken when needed.

Date:

Pritam M. Gharat
(Roll No. 113050036)

Acknowledgements

A journey through Ph.D. can be best described as a roller coaster ride with many ups and downs. I would like to acknowledge the people who supported and encouraged me in this journey.

First and foremost, I would like to express my gratitude towards my advisor Prof. Uday P. Khedker. I am immensely thankful for his availability whenever I needed a discussion, his ability to foresee potential ideas and challenges, his aspiration for perfection, and a lot of patience. I learnt (and am still learning) how to take an idea to its logical conclusion and how to describe it precisely. He allowed me to work independently on problems that interested me and showed full trust in my ability. When he took over as the head of the department, we were a little apprehensive about his availability. However, he managed to find sufficient time for us. Without his guidance and support, this dissertation would not have been possible.

The first paper in Ph.D. life is always special. I am thankful to Prof. Alan Mycroft who helped me to improve its accessibility significantly. His insights and mathematical perspective of intuitions helped me to improve my formulations. I hope to master this skill in the near future. My collaboration with him set the tone for the rest of the research in this dissertation. I had the privilege of working with him through the rest of the Ph.D. and I thank him with deep gratitude.

I have been fortunate to have Prof. Amitabha Sanyal, Prof. Supratik Chakraborty, and Prof. Supratim Biswas on my Ph.D. committee. Their critiques and insightful comments helped me to refine the dissertation. Prof. Amey Karkare's profiling tool helped to monitor the performance of my implementation. Prashant Singh Rawat's tool in GCC provided a base implementation for my work.

Discussions with lab members Anshuman Dhuliya, Komal Pathade, Swati Jaiswal, and Vini Kanvar and masters student Binapani Beria gave me more clarity in my work. Anshuman Dhuliya, Anushri Jana, Advaita Datar, Tukaram Muske, and Vini Kanvar gave valuable feedback on my papers. My interns Shachi Deshpande, Nishant Sahani, Mugdha Khedkar, and Pushpinder Singh helped me learn how to present my ideas in a simple manner.

I thank all my friends Kavya Alse, Mukulika Maity, Pradnya Kiri Taksande, Radhika

B S, Soumya Narayana, and Swati Jaiswal for just being there. The crazy talks on the mess table with them were like an oasis amidst a dessert of stress and deadlines. I found a dependable friend in Nisha Biju, aka, Nisha di. My worries disappeared by sharing my problems with her. I will miss you all.

I also take this opportunity to express my gratitude to all the faculty members of CSE department. The administrative staff too, particularly Vijay Ambre, Sunanda Ghadge, Harish Solanki, and Trupti Tamhankar have been extremely helpful. Their help and co-operation simplified many things which would have been difficult otherwise.

I thank Tata Consultancy Services for awarding me TCS Research fellowship for my Ph.D.

Last but not least, I would like to thank my parents for their constant source of encouragement and offering me complete freedom to pursue my interests.

Thank you all for being a part of my journey, for your support, love and care, and for sharing moments that turned into some extraordinary memories. I am now ready to turn the next chapter of my life with all the wonderful memories that I made at IIT Bombay.

Date: Pritam M. Gharat

Abstract

Prior approaches to flow- and context-sensitive points-to analysis (FCPA) have not scaled; for top-down approaches, the problem centers on repeated analysis of the same procedure; for bottom-up approaches, the abstractions used to represent procedure summaries have not scaled while preserving precision. Bottom-up approaches for points-to analysis require modelling unknown pointees accessed indirectly through pointers that may be defined in the callers.

We propose a novel abstraction called the Generalized Points-to Graph (GPG) which views points-to relations as memory updates and generalizes them using the counts of indirection levels leaving the unknown pointees implicit. This allows us to construct GPGs as compact representations of bottom-up procedure summaries in terms of memory updates and control flow between them. Their compactness is ensured by the following optimizations: strength reduction reduces the indirection levels, redundancy elimination removes redundant memory updates and minimizes control flow (without over-approximating data dependence between memory updates), and call inlining enhances the opportunities of these optimizations.

Our quest for scalability of points-to analysis leads to the following insight: The real killer of scalability in program analysis is not the amount of data but the amount of control flow that it may be subjected to in search of precision. The effectiveness of GPGs lies in the fact that they discard as much control flow as possible without losing precision (i.e., by preserving data dependence without over-approximation). This is the reason why the GPGs are very small even for the main procedures that contain the effect of entire programs. This allows our implementation to scale to 158kLoC for C programs.

At a more general level, GPGs provide a convenient abstraction of memory and memory transformers in the presence of pointers. Future investigations can try to combine it with other abstractions for static analyses that can benefit from points-to information.

Publications based on this Thesis

Published

Pritam M. Gharat, Uday P. Khedker, and Alan Mycroft. Flow- and context-sensitive points-to analysis using generalized points-to graphs. In Proceedings of the 23rd Static Analysis Symposium, SAS'16, Berlin, Heidelberg, 2016. Springer-Verlag.

Under Review

Pritam M. Gharat, Uday P. Khedker, and Alan Mycroft. Generalized points-to graphs: A new abstraction of memory in presence of pointers. (Submitted to TOPLAS).

Contents

List of Tables

List of Figures

Chapter 1

Introduction

This chapter sets the context of our work in points-to analysis in many ways. We first describe the role of points-to analysis on other program analyses and optimizations (Section 1.1). Section 1.2 describes various design features of points-to analysis and lists our choices. Section 1.3 reviews some basic concepts and then describes different types of procedure summaries. It then describes the challenges in constructing procedure summaries for flow- and context-sensitive points-to analysis and limitations of past approaches. Section 1.4 describes the key ideas of our approach that overcome the limitations and the contributions of our work. Section 1.5 provides the organization of the rest of the thesis.

1.1 The Influence of Pointers on Program Analysis

Given the ubiquity of computer software, its correctness and efficiency are becoming more and more critical. These concerns can be addressed by using *program analysis* which discovers the information representing relevant behaviors of programs. This process is almost always automated and the required information is computed statically (i.e., without executing the program) or dynamically (i.e., during program execution).

Common programming languages used for software development employ pointers which hold addresses of data allowing indirect accesses. This increases both convenience and efficiency but at a price: reasoning about programs becomes difficult making it harder to optimize or verify them. The indirect accesses through pointers can be resolved through a *points-to analysis*. The precision of this resolution influences the precision and scalability of other program analyses and their applications. As illustrated in Figure 1.1, in

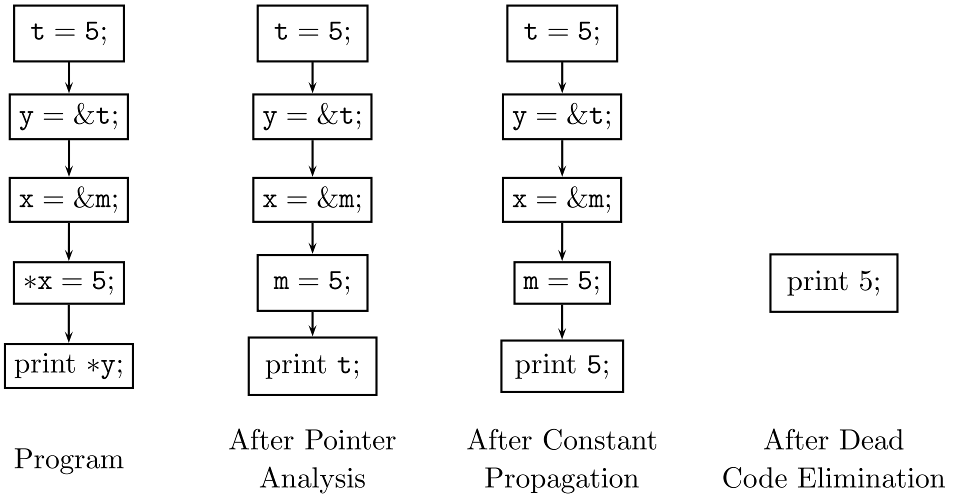


Figure 1.1: The role of points-to analysis in other analyses and optimizations.

the presence of pointers, points-to analysis is a pre-requisite for other analyses and optimizations. Computationally intensive analyses such as model checking are noted as being ineffective on programs containing pointers, partly because of imprecision of points-to analysis [3, 5, 13, 24, 45, 46].

Points-to analysis is, in general, undecidable [44, 55, 56, 79, 81]. Hence, all points-to analyses compute a suitable approximation of actual runtime points-to information. A large number of points-to analysis methods have been published that make a trade-off between the efficiency of the analysis and the precision of the information computed. Many investigations reported in the literature have described the popular points-to analysis methods and have presented a comparative study of the methods with respect to scalability and precision [40, 42, 43, 48, 60, 87, 95, 99, 101]. We have described related work in Chapter 2.

1.2 The Context of the Proposed Points-to Analysis Work

For practical usefulness, a points-to analysis should be interprocedural (i.e., it should handle the effects of procedure calls). It is a challenge to scale points-to analysis to

Nature	Intraprocedural Analysis
Path sensitivity	×
Flow sensitivity	✓
Field sensitivity	✓

Nature	Interprocedural Analysis
Context sensitivity	✓
Top-down traversal	×
Bottom-up traversal	✓

Figure 1.2: Design characteristics of our GPG-based points-to analysis.

programs containing hundreds of procedures. It is commonly believed that a points-to analysis cannot be precise and scalable at the same time. This leads to the tyranny of OR forcing a trade-off between precision and scalability. Hence, most researchers and practitioners have focussed on engineering approximations thereby trading precision for scalability or vice-versa. My work seeks the genius of AND to achieve precision and scalability simultaneously (relative to abstractions for handling undecidability). Figure 1.2 lists all the design features that affect the precision and scalability of a points-to analysis and specifies the features adopted by our approach. The rest of the section briefly describes each design feature. We begin with a distinction between exhaustive and demand-driven analysis which does not fit in the Figure 1.2.

1.2.1 Exhaustive versus Demand-driven Analysis

A demand-driven points-to analysis [17, 38, 97, 100] computes points-to information that is relevant to a query raised by a client analysis; for a different query, the points-to analysis needs to be repeated. An exhaustive analysis, on the other hand, computes all points-to information which can be queried later by a client analysis; multiple queries do not require points-to analysis to be repeated. Achieving the scalability of an exhaustive approach is much harder than that of a demand-driven approach.

Example 1. For Figure 1.3, one query could be to identify available expressions at End_p. This would require analysis of only two procedures p and q and not procedure r. If another query is raised such as availability of expressions at End_r, then procedures q and r are analyzed. For an exhaustive analysis, all the procedures are analyzed and the information is queried after the complete analysis.

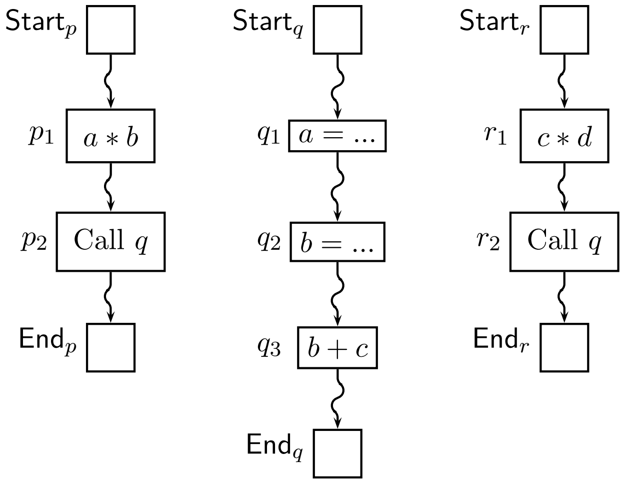


Figure 1.3: An example demonstrating the various design characteristics of an analysis. All variables are global.

1.2.2 Design Features for Intraprocedural Level

This section describes various design features of an analysis at the intraprocedural level.

Path Sensitivity

A path-sensitive analysis [17, 27, 104] distinguishes between the data flow values reaching a program point along different control flow paths. It is practically infeasible because of exponentially large number of paths in a program. In the presence of loops and recursion, this number is infinite.

Flow Sensitivity

A flow-sensitive analysis [11, 53, 62, 64, 88, 107] respects the control flow and computes separate data flow information at each program point. This matters because a pointer could have different pointees at different program points owing to redefinitions. A flow-insensitive analysis [2, 21, 31, 39, 69, 102], on the other hand, does not respect the control flow and computes a single data flow information that holds conservatively for the entire program. Hence, a flow-sensitive analysis provides more precise results than a flow-insensitive analysis but can become inefficient at the interprocedural level.

Example 2. Consider procedure q in Figure 1.3 for available expressions analysis. In a flow-sensitive analysis, an expression is marked as not available at a program point, if there exists a path reaching the program point, along which an operand of the expression is defined and the expression is not computed after it. In a flow-insensitive analysis, since the control flow order is not taken into account, the expression is not available in the procedure if there is a definition of an operand of the expression anywhere in the procedure. Thus, the expression $b + c$ is not available at the exit of procedure q in a flow-insensitive analysis in spite of the fact that the assignment to b is followed by a computation of expression $b + c$. A flow-sensitive analysis precisely computes the availability of expression $b + c$ at the exit of procedure q.

Field Sensitivity

A field-sensitive analysis [76, 77, 114] treats each field of a structure variable as a separate variable. A field-insensitive analysis over-approximates by treating every field of a structure variable as a single variable. Thus, a field-sensitive analysis is more precise than a field-insensitive analysis. For more details, see Chapter 8.

1.2.3 Design Features for Interprocedural Level

This section describes various design features of an analysis at the interprocedural level.

Context Sensitivity

A context-sensitive analysis [22, 50, 53, 61, 62] respects the semantics of procedure calls by analyzing each distinct procedure separately, whereas a context-insensitive [2, 21, 31, 32, 33, 34, 39, 69, 102] analysis merges contexts together. A context-sensitive analysis distinguishes between different calling contexts of procedures and restricts the analysis to interprocedurally valid control flow paths (i.e. control flow paths from program entry to program exit in which every return from a procedure is matched with a call to the procedure such that all call-return matchings are properly nested). A context-insensitive analysis propagates data flow information along all the paths (including the invalid paths) thereby introducing imprecision.

Example 3. In Figure 1.3, interprocedural paths $\mathsf{Start}_p - p_1 - p_2 - \mathsf{Start}_q - \ldots - \mathsf{End}_q - \mathsf{End}_r$ and $\mathsf{Start}_r - r_1 - r_2 - \mathsf{Start}_q - \ldots - \mathsf{End}_q - \mathsf{End}_p$ are invalid paths as they do not adhere to call-return matching. In case of context-insensitive analysis, information is also propagated along interprocedurally invalid paths thereby losing precision and hence expressions $c * d$ and $b + c$ are available after the calls to procedure q in both procedures p and r.

In this example, the valid interprocedural paths are $\mathsf{Start}_p - p_1 - p_2 - \mathsf{Start}_q - \ldots - \mathsf{End}_q - \mathsf{End}_p$ and $\mathsf{Start}_r - r_1 - r_2 - \mathsf{Start}_q - \ldots - \mathsf{End}_q - \mathsf{End}_r$ and information should propagate along only these two paths for a context-sensitive analysis. In case of context-sensitive analysis, expression $b + c$ is available after the call to procedure q in procedure p and expressions $c * d$ and $b + c$ are available after the call to procedure q in procedure r.

A fully context-sensitive analysis does not lose precision even in the presence of recursion. A context-sensitive interprocedural analysis could be classified into two broad categories: top-down approach and bottom-up approach.

Top-down Approach to Context-Sensitive Analysis

A top-down approach to interprocedural analysis propagates information from callers to callees [119] effectively traversing the call graph top-down. In the process, it analyzes a procedure each time a new data flow value reaches it from some call. Several popular approaches fall in this category: the call-strings method [94], its value-based variants [52, 75] and the tabulation-based functional method [82, 94].

Example 4. In Figure 1.3, procedure q is analyzed multiple times in a top-down approach: firstly because of the call from procedure p and secondly because of the call from procedure r.

Bottom-up Approach to Context-Sensitive Analysis

In contrast to top-down approach, bottom-up approaches [9, 23, 30, 47, 65, 94, 103, 109, 110, 112, 115, 116, 119] avoid analyzing a procedure multiple times by constructing its

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

Figure 1.4: Interprocedural points-to analysis.

procedure summary which is used to incorporate the effect of calls to the procedure. Effectively, this approach traverses the call graph bottom-up.

Example 5. In Figure 1.3, procedure q is analyzed first. A procedure summary is created for procedure q as Gen : $b+c$ and Kill : $a*b$. This procedure summary is used at the call sites p_2 and r_2 in procedures p and r respectively.

Figure 1.4 summarizes the two approaches to interprocedural context-sensitive points-to analysis. A context-sensitive bottom-up interprocedural analysis using procedure summaries is performed in two phases: the first phase constructs the procedure summaries and the second phase applies them at the call sites to compute the desired information, which, in our case, is the classical points-to information.

A top-down approach to interprocedural analysis may not scale well but can be more precise than a bottom-up approach, if the latter's procedure summaries fail to capture all relevant details of a procedure valid for all possible calls to it.

Traversals Over Call Graph and Control Flow Graph are Orthogonal

We use the terms top-down and bottom-up for traversals over a call graph; traversals over a control flow graph are termed forward and backward. The two categories of traversals are orthogonal. We have already seen a top-down and bottom-up approach for forward data flow problem (e.g. available expressions analysis) in this section. We now illustrate top-down approach and bottom-up approach for a backward data flow problem such as live variables analysis.

Example 6. In Figure 1.3, a top-down backward live variables analysis also analyzes procedure q multiple times. However, the processing starts from the exit of the procedure rather than the start of the procedure. So the analysis of procedure p begins from End_p, then a call is encountered to procedure q at p_2, as a result, control now goes to $\mathsf{End}_q - q_3 - q_2 - q_1 - \mathsf{Start}_q$ in the backward flow. For a context-sensitive analysis, the control flow returns back to p_1 and then Start_p. Similarly, procedure r is analyzed with re-analysis of procedure q.

For a bottom-up backward live variables analysis, a procedure summary for procedure q is constructed by traversing its control flow graph backwards: $\text{Gen} = \{c\}$ and $\text{Kill} = \{a, b\}$. This procedure summary is inlined at p_2 and r_2 when procedures p and r are analyzed with a backward traversal over their control flow graph. Points-to analysis is a forward data flow problem.

Hence the terms top-down and bottom-up and forward and backward are orthogonal. Thus, both a forward data flow analysis (e.g. available expressions analysis) and a backward data flow analysis (e.g. live variables analysis) could be a top-down or a bottom-up analysis at the interprocedural level.

1.2.4 Our Choice

This work focuses on flow-, field-, and context-sensitive points-to analysis. These design features enhance precision of an analysis and we aim to achieve it without compromising efficiency/scalability through a bottom-up interprocedural approach. Our GPG-based points-to analysis is exhaustive as against demand-driven points-to analysis.

1.3 Background

This section begins by reviewing some basic concepts and then describes different types of procedure summaries. It then describes the challenges in constructing procedure summaries for flow- and context-sensitive points-to analysis. It concludes by describing the limitations of the past approaches. For further details of related work, see Chapter 2.

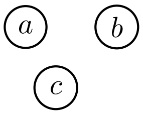
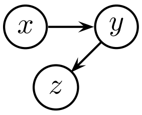
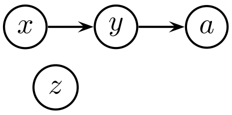
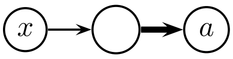
Memory in absence of pointers	Memory in presence of pointers		Memory transformer
M	M_1	M_2	Δ
			

Figure 1.5: Pictorial view of memory and memory transformer. Thick edges in a memory transformer represent the points-to edges *generated* by it, other edges are carried forward from the input memory.

1.3.1 Basic Concepts

This section describes the nature of memory, memory updates, and memory transformers.

1.3.1.1 Abstract and Concrete Memory

Memory and operations on it can be viewed in two ways. Firstly we have the concrete memory view (or semantic view) corresponding to run-time operations where a memory location always points to exactly one memory location or NULL (which is a distinguished memory location). Unfortunately this is, in general, statically uncomputable. Secondly, as is traditional in program analysis, we can consider an abstract view of memory where an abstract location represents one or more concrete locations; this conflation and the uncertainty of conditional branches means that abstract memory locations can point to multiple other locations—as in the classical points-to graph. These views are not independent and abstract operations must over-approximate concrete operations to ensure soundness. The abstract memory associated with a statement s is an over-approximation of the concrete memory associated with every occurrence of s in the same or different control flow paths.

Formally, let L and $\mathsf{P} \subseteq \mathsf{L}$ denote the sets of locations and pointers respectively. The *concrete memory* after a pointer assignment is a function $M : \mathsf{P} \to \mathsf{L}$. The *abstract memory* after a pointer assignment is a relation $M \subseteq \mathsf{P} \times \mathsf{L}$. In either case, we view M as a graph with L as the set of nodes. An edge $x \to y$ in M is a *points-to edge* indicating that $x \in \mathsf{P}$ contains the address of $y \in \mathsf{L}$.

Example 7. In Figure 1.5, memory as a graph has only nodes (variables) and no edges in the absence of pointers as shown in the first column (M). In the presence of pointers, the pointer variables are connected to each other as shown by the edges as shown in the second and the third column (M_1 and M_2).

Unless noted explicitly, all subsequent references to memory locations and transformers refer to the abstract view.

1.3.1.2 Strong and Weak Updates

A memory update changes the contents of a memory location. In concrete memory, every assignment overwrites the contents of the memory location corresponding to the LHS of the assignment. However, in abstract memory, we may be uncertain as to which of several locations a variable (say p) points to. Hence an indirect assignment such as $*p = \&x$ does not overwrite any of its pointees, but merely *adds* x to the possible pointees. This is a *weak update*. Sometimes however, there is only one possible abstract location described by the LHS of an assignment, and in this case we may, in general, *replace* the contents of this location. This is a *strong update*. There is just one subtlety which we return to later: prior to the above assignment we may only have one assignment to p (say $p = \&a$). If this latter assignment dominates the former, then a strong update is appropriate. But if the latter assignment only appears on some control flow paths to the former, then we say that the read of p in $*p = \&x$ is *upwards exposed* (live on entry to the current procedure) and therefore may have additional pointees unknown to the current procedure. Thus, the criterion for a strong update in an assignment is that its LHS references a single location *and* the location referenced is not upwards exposed (for more details, see Section 4.5). An important special case is that a direct assignment to a variable (e.g. $p = \&x$) is always a strong update.

When a value is stored in a location, we say that the location is *defined* without specifying whether the update is strong or weak. We make this distinction only where it is required.

```
int a, b, c, d;

p()
{
    c = a*b;
    q(); /* call 1 */
    a = c*d;
    q(); /* call 2 */
}

q()
{
    a = b*c;
}
```

Let f_q denote the summary for procedure q for available expressions analysis

(I) f_q as a context-independent and context-sensitive summary

$$f_q(X) = X \cdot 011 + 010$$

Here '·' and '+' are bit-vector 'and' and 'or'.

(II) f_q as a context-dependent and context-sensitive summary

$$f_q = \{100 \mapsto 010,\ 011 \mapsto 011\}$$

(III) f_q as a context-insensitive summary information

$$f_q = 010$$

Figure 1.6: Illustrating different kinds of procedure summaries for available expressions analysis. The set of expressions $\{a{*}b,\ b{*}c,\ c{*}d\}$ is represented by the bit vector 111.

1.3.1.3 Memory Transformer

A procedure summary for points-to analysis should represent memory updates in terms of copying locations, loading from locations, or storing to locations. It is called a *memory transformer* (a collection of memory updates) because it updates the memory before a call to the procedure to compute the memory after the call. Given a memory M and a memory transformer $\varDelta$, the updated memory M' is computed by $M' = \varDelta(M)$ as illustrated in the below example.

Example 8. In Figure 1.5, a memory transformer $\varDelta$ contains a single update in the form of store $({*x} = \&a)$. The thick edge in the figure is the points-to edge generated by $\varDelta$ and the thin edge represents the carried forward input memory. When the input to memory transformer $\varDelta$ is the memory M_1, we get the output memory M_2, i.e.,

$M_2 = \mathit{\Delta}(M_1)$. In this example, the pointee of x (which is y) was earlier pointing to z, now starts pointing to a as shown in M_2. The contents of the memory location y updated from $\&z$ to $\&a$.

1.3.2 Procedure Summaries

We distinguish between three kinds of summaries of a procedure that can be constructed for minimizing the number of times a procedure is re-analyzed:

(a) A bottom-up parameterized procedure summary (e.g. summary (I) in Figure 1.6) which is context independent. The context is supplied by the values of the parameters of the summary.

(b) A top-down enumeration of procedure summary (e.g. summary (II) in Figure 1.6) in the form of input-output pairs for the input values reaching a procedure.

(c) A bottom-up parameterless (and hence context-insensitive) summary information (e.g. summary (III) in Figure 1.6).

Context independence (in (a) above) achieves context sensitivity through parameterization and should not be confused with context insensitivity (in (c) above).

Example 9. Figure 1.6 illustrates the three different kinds of procedure summaries for available expressions analysis. Procedure q kills the availability of expression $a{*}b$, generates the availability of $b{*}c$, and is transparent to the availability of $c{*}d$.

- In Figure 1.6, summary (I) is a parameterized flow function, summary (II) is an enumerated flow function, whereas summary (III) is a data flow value (i.e. it is a summary information as against a summary) representing the effect of all calls of procedure q.

- Procedure summaries (I) and (II) are context-sensitive (because they compute distinct values for different calling contexts of q) whereas summary (III) is context-insensitive (because it represents the same value regardless of the calling context).

- Summaries (I) and (III) are context-independent (because they can be constructed without requiring any information from the calling contexts of q) whereas summary (II) is context-dependent (because it requires information from the calling contexts - input values reaching a procedure).

Our work focuses on the procedure summary of type (I) of Figure 1.6.

1.3.3 Challenges in Constructing Procedure Summaries for Points-to Analysis

Construction of procedure summaries for other analyses in the absence of pointers is trivial. In such a situation, the data dependence between memory updates within a procedure can be inferred by using variable names without requiring any information from the callers. Procedure summaries for some analyses, including various bit-vector data flow analyses (such as live variables analysis), can be precisely represented by constant *gen* and *kill* sets [51] or graph paths discovered using reachability [82]. In the presence of pointers, these (bit-vector) summaries can be constructed using externally supplied points-to information.

Procedure summaries for points-to analysis, however, cannot be represented in terms of constant *gen* and *kill* sets because the association between pointer variables and their pointee locations could change in the procedure and may depend on the aliases between pointer variables established in the callers of the procedure.

The main challenge in constructing procedure summaries for points-to analysis is:

> *The memory transformers need to handle indirectly accessed unknown pointees and preserve data dependence between them without any imprecision.*

Since a memory transformer's goal is to compute points-to information, its construction cannot assume availability of points-to information as an input unlike other procedure summaries which can use externally supplied points-to information.

Often, and particularly for points-to analysis, we have a situation where a procedure summary must either lose information or retain internal details which can only be resolved when its caller is known.

Example 10. Consider procedure f on the right. For many calls, $f()$ simply returns $\&a$ but until we are certain that $*p$ does not alias with x, we cannot perform this constant-propagation optimization. We say that the assignment 04 *blocks* this optimization. (We also use the word '*barrier*' for a blocking assignment.) There are four possibilities:

```
01  int a, b, *x, **p;
02  int * f() {
03      x = &a;
04      *p = &b;
05      return x;
06  }
```

(i) If it is known that $*p$ and x *always* alias then we can optimize f to return $\&b$.

(ii) If it is known that $*p$ and x alias on some control flow paths containing a call to f but not on all, then the procedure returns $\&a$ in some cases and $\&b$ in other cases. While procedure f cannot be optimized to do this, a static analysis can compute such a summary.

(iii) If it is known that they *never* alias we can optimize this code to return $\&a$.

(iv) If nothing is known about the alias information, then to preserve precision, we must retain this blocking assignment in the procedure summary for f.

The key idea is that information from the calling context(s) can determine whether a potentially blocking assignment really blocks an optimization or not. As such we say that we *postpone* optimizations that we would like to do until it is safe to do them.

The above example illustrates the following challenges in constructing flow-sensitive memory transformers: (a) representing indirectly accessed unknown pointees, (b) identifying blocking assignments and postponing some optimizations, and (c) recording control flow between memory updates so that potential data dependence between them is neither violated nor over-approximated.

Thus, the main problem in constructing flow-sensitive memory transformers for points-to analysis is to find a representation that is compact and yet captures memory updates and the minimal control flow between them succinctly.

Pointer Statement		Flow Function $f \in \mathbf{F} = \{\mathit{ad}, \mathit{cp}, \mathit{st}, \mathit{ld}\}$, $f : \mathsf{PTG} \mapsto \mathsf{PTG}$	Placeholders in X
Address	$x = \&y$	$\mathit{ad}_{xy}(X) = \; X - \{(x, l_1) \mid l_1 \in L\} \cup$ $\{(x, y)\}$	$\emptyset$
Copy	$x = y$	$\mathit{cp}_{xy}(X) = \; X - \{(x, l_1) \mid l_1 \in L\} \cup$ $\{(x, \phi_1) \mid (y, \phi_1) \in X\}$	ϕ_1
Store	$*x = y$	$\mathit{st}_{xy}(X) = \; X - \{(\phi_1, l_1) \mid (x, \phi_1) \in X, l_1 \in L\} \cup$ $\{(\phi_1, \phi_2) \mid \{(x, \phi_1), (y, \phi_2)\} \subseteq X\}$	ϕ_1, ϕ_2
Load	$x = *y$	$\mathit{ld}_{xy}(X) = \; X - \{(x, l_1) \mid (x, l_1) \in L\} \cup$ $\{(x, \phi_2) \mid \{(y, \phi_1), (\phi_1, \phi_2)\} \subseteq X\}$	ϕ_1, ϕ_2

Figure 1.7: Points-to analysis flow functions for basic pointer assignments.

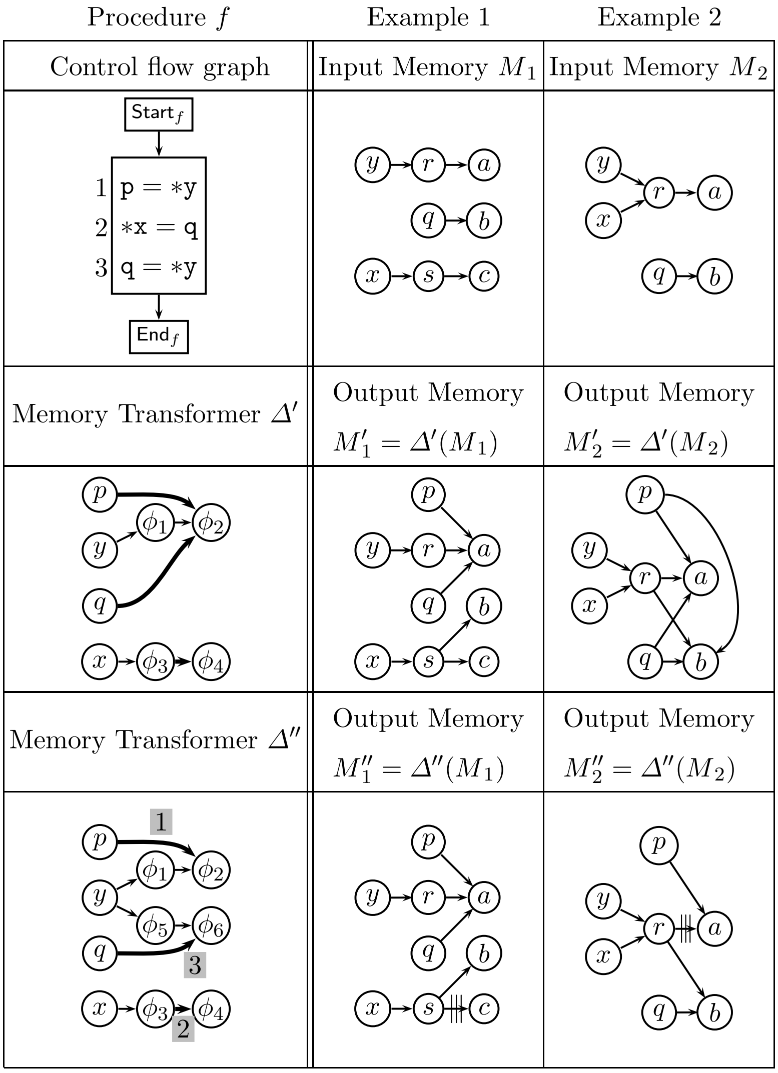
1.3.4 Limitations of Existing Procedure Summaries for Points-to Analysis

A common solution for modelling indirect accesses of unknown pointees in a memory transformer is to use placeholders[1] (illustrated in Figure 1.7) which are pattern-matched against the input memory to compute the output memory.

The use of placeholders may lead to a representation of procedure summaries that is not closed under composition. Let L and $P \subseteq L$ denote the sets of locations and pointers in a program. Then, the points-to information is a member of $\mathsf{PTG} = 2^{P \times L}$, the set of all relations (or equivalently graphs) associating pointers to locations. For a given statement, a flow function for points-to analysis computes points-to information after the statement by incorporating its effect on the points-to information that holds before the statement. It has the form $f : \mathsf{PTG} \to \mathsf{PTG}$. Figure 1.7 enumerates the space of flow functions for basic pointer assignments.[2] These basic flow functions are named in terms of the variables appearing in the assignment statement and are parameterized on the input

[1] Placeholders have also been known as external variables [65, 103, 109] and extended parameters [110]. They are parameters of the procedure summary and not necessarily of the procedure for which the summary is constructed.

[2] Other pointer assignments involving structures and heap are handled as described in Chapter 8.



Procedure f
Example 1
Example 2
Control flow graph
Input Memory M_1
Input Memory M_2
Start$_f$
1 p = *y
2 *x = q
3 q = *y
End$_f$
y r a q b x s c
Memory Transformer Δ'
Output Memory $M'_1 = \Delta'(M_1)$
Output Memory $M'_2 = \Delta'(M_2)$
p y q x ϕ_1 ϕ_2 ϕ_3 ϕ_4
Memory Transformer Δ''
Output Memory $M''_1 = \Delta''(M_1)$
Output Memory $M''_2 = \Delta''(M_2)$
1
ϕ_5 ϕ_6
3
2

The memory transformer Δ' is compact but imprecise because it uses the same placeholder for every access of a pointee. Thus it over-approximates the memory.

The memory transformer Δ'' shows that precision can be improved by using a separate placeholder for every access of a pointee. However, the size of the memory transformer increases.

Figure 1.8: An STF-style memory transformer Δ' and its associated transformations. Δ'' is its flow-sensitive version. Unknown pointees are denoted by placeholders ϕ_i. Thick edges in a memory transformer represent the points-to edges *generated* by it, other edges are carried forward from the input memory. Labels of the points-to edges in Δ'' correspond to the statements indicating the sequencing of edges. Edges that are *killed* in the memory are struck off.

points-to information X which may depend on the calling context. The information from the calling context is described in terms of placeholders in X denoted by ϕ_1 and ϕ_2. It is easy to see that the function space $\mathsf{F} = \{\mathit{ad}, \mathit{cp}, \mathit{st}, \mathit{ld}\}$ is not closed under composition as shown by the example below.

Example 11. Let f represent the composition of flow functions for the statement sequence $x = *y$; $z = *x$. Then

$$\begin{aligned} f(X) = \mathsf{Id}_{zx}(\mathsf{Id}_{xy}(X)) = \; & \big(X - (\{(x, l_1) \mid (x, l_1) \in \mathsf{L}\} \; \cup \{(z, l_1) \mid (z, l_1) \in \mathsf{L}\} \;)\big) \\ & \cup \{(x, \phi_2) \mid \{(y, \phi_1), (\phi_1, \phi_2)\} \subseteq X\} \\ & \cup \{(z, \phi_3) \mid \{(y, \phi_1), (\phi_1, \phi_2), (\phi_2, \phi_3)\} \subseteq X\} \end{aligned}$$

The flow function f has three placeholders and cannot be reduced to any of the four primitive flow functions in the set.

The use of placeholders explicates the unknowns pointees and hence the function space is not closed under composition. Thus, in the case of accessing recursive data structures[3], the size of procedure summaries would grow unboundedly and an explicit summarization technique is required to bound it.

We describe two broad approaches that use placeholders. The first approach, which we call a *multiple transfer functions* (MTF) approach, proposed a precise representation of a procedure summary for points-to analysis as a collection of *partial transfer functions* (PTFs) [9, 47, 110, 116].[4] Each PTF corresponds to a combination of aliases that might occur in the callers of a procedure.

Our work is inspired by the second approach, which we call a *single transfer function* (STF) approach [65, 103, 109]. This approach does not customize procedure summaries for combinations of aliases.

However, the existing STF approach fails to be precise. We illustrate this approach and its limitations to motivate our key ideas using Figure 1.8. It shows a procedure and two memory transformers (Δ' and Δ'') for it and the associated input and output memories. The effect of Δ' is explained in Example 12 and that of Δ'' in Example 13.

Example 12. Transformer Δ' is constructed by the STF approach [65, 103, 109]. It can be viewed as an abstract points-to graph containing placeholders ϕ_i for modelling unknown pointees of the pointers accessed in procedure f. For example, ϕ_1 represents

[3] Pointers to scalars cannot be used for creating recursive data structures, but pointers to structures and heap could be a part of recursive data structures that are unbounded.

[4] In level-by-level analysis [116], multiple PTFs are combined into a single function with a series of condition checks for different points-to information occurring in the calling contexts.

the pointees of y and ϕ_2 represents the pointees of pointees of y, both of which are not known in the procedure. The placeholders are pattern matched against the input memory (e.g. M_1 or M_2) to compute the corresponding output memory (M'_1 and M'_2 respectively). A crucial difference between a memory and a memory transformer is: a memory is a snapshot of points-to edges whereas a memory transformer needs to distinguish the points-to edges that are generated by it (shown by thick edges) from those that are carried forward from the input memory (shown by thin edges). However, in the memory, there is no such distinction.

The two accesses of y in statements 1 and 3 may or may not refer to the same location because of a possible side-effect of the intervening assignment in statement 2. If x and y are aliased in the input memory (e.g. in M_2), statement 2 redefines the pointee of y and hence p and q will not be aliased in the output memory. However, Δ' uses the same placeholder for all accesses of a pointee. Further, Δ' also suppresses strong updates because the control flow ordering between memory updates is not recorded. Hence, points-to edge $s \rightarrow c$ in M'_1 is not deleted. Similarly, points-to edge $r \rightarrow a$ in M'_2 is not deleted and q spuriously points to a. Additionally, p spuriously points-to b. Hence, p and q appear to be aliased in the output memory M'_2.

The use of control flow ordering between the points-to edges that are *generated* by a memory transformer can improve its precision as shown by the following example.

Example 13. In Figure 1.8, the memory transformer Δ'' differs from Δ' in two ways. Firstly it uses a separate placeholder for every access of a pointee to avoid an over-approximation of memory (e.g. placeholders ϕ_1 and ϕ_2 to represent $*y$ in statement 1, and ϕ_5 and ϕ_6 to represent $*y$ in statement 3). This, along with control flow, allows strong updates thereby killing the points-to edge $r \rightarrow a$ and hence q does not point to a (as shown in M''_2). Secondly, the points-to edges generated by the memory transformer are ordered based on the control flow of a procedure, thereby adding some form of flow-sensitivity which Δ' lacks. To see the role of control flow, observe that if the points-to edge corresponding to statement 2 is considered first, then p and q will always be aliased because the possible side-effect of statement 2 will be ignored.

The output memories M_1'' and M_2'' computed using Δ'' are more precise than the corresponding output memories M_1' and M_2' computed using Δ'.

Observe that, although Δ'' is more precise than Δ', it uses a larger number of placeholders and also requires control flow information. The large size of Δ'' affects the scalability of points-to analysis.

A fundamental problem with placeholders is that they use a low-level representation of memory expressed in terms of classical points-to edges. Hence a placeholder-based approach is forced to explicate unknown pointees by naming them, resulting in either a large number of placeholders (in the STF approach) or multiple PTFs (in the MTF approach). The need of control flow ordering further increases the number of placeholders in the former approach. The latter approach obviates the need of ordering because the PTFs are customized for combinations of aliases.

1.4 Our Key Ideas and Contributions

At a practical level, our main contribution is a method of flow-sensitive, field-sensitive, and context-sensitive exhaustive points-to analysis of C programs that scales to large real-life programs.

The core ideas of GPGs have been presented in [25] and the complete analysis is given in [26]. We describe our formulations for a C-like language.

1.4.1 Key Ideas

We propose a *generalized points-to graph* (GPG) as a representation for a memory transformer of a procedure; special cases of GPGs also represent memory as a points-to relation.[5] GPGs summarizes the effect of a procedure and contain GPUs (generalized points-to updates) representing individual memory updates along with the control flow between them. A GPG is characterized by the following key ideas that overcome the two limitations described in Section 1.3.4.

- A GPG leaves the unknown pointees (pointees that are indirectly accessed and

[5] This is analogous to a matrix which can be seen both as a transformer (for a linear translation in space) and also as an absolute value.

defined in the callers) implicit by using the counts of indirection levels. This obviates the need for placeholders. Simple arithmetic on the counts of indirection levels allows us to combine the effects of multiple memory update.

- A GPG uses a flow relation to order memory updates. An interesting property of the flow relation is that it can be compressed dramatically without losing precision and can be transformed into a compact acyclic flow relation in most cases, even if the procedure it represents has loops or recursive calls.

1.4.2 Highlights of GPG-based Points-to Analysis

GPGs are compact—their compactness is achieved by a careful choice of a suitable representation and a series of optimizations as described below.

1. Our representation of memory updates, called the *generalized points-to update* (GPU) leaves accesses of unknown pointees implicit without losing precision.

2. GPGs undergo aggressive optimizations that are applied repeatedly to improve the compactness of GPGs incrementally. These optimizations are similar to the optimizations performed by compilers and are governed by the data dependence between two memory updates. They are as follows:

    - Strength reduction optimization.
    - Redundancy elimination optimizations.
    - Call inlining optimization.
    - Type-based non-aliasing.

3. Interleaving call inlining and strength reduction of GPGs facilitates a novel optimization that computes flow- and context-sensitive points-to information in the first phase of a bottom-up approach. This obviates the need for the usual second phase where procedure summaries are used to compute points-to information.

In order to perform these optimizations:

- We define operations of *GPU composition* (to create new GPUs by eliminating data dependence between two GPUs), and *GPU reduction* (to eliminate the data dependence of a GPU with the GPUs in a given set).

Original	Transformed	Original	Transformed	Original	Transformed
1 $x = \&a;$ 2 $y = x;$	$x = \&a;$ $y = \&a;$	1 $x = \&a;$ 2 $y = \&a;$ 3 $x = \&b;$	$y = \&a;$ $x = \&b;$	1 $y = \&a;$ 2 $*x = \&b;$	1 $y = \&a;$ 2 $*x = \&b;$
(a) Cases A1 and B		(b) Cases A3 and B		(c) Case C	

Figure 1.9: Data dependence between memory updates. The data dependence is eliminated wherever possible and redundant control flow is eliminated in the absence of data dependence. Case A2 is explained in Example 19 in Section 3.1.

- We propose novel data flow analyses such as two variants of *reaching GPUs analysis* (to identify the effects of memory updates reaching a given statement) and *coalescing analysis* (to eliminate the redundant control flow in the GPG).
- We handle recursive calls through a fixed-point computation. These calls are eliminated by a bounded inlining of callee GPGs without over-approximation. Calls through function pointers are proposed to be handled through delayed inlining.

1.4.3 The Role of Data Dependence in GPG Optimizations

GPG optimizations are governed by the following possibilities of data dependence between two memory updates (a detailed illustration is given in Example 10 in Section 1.3.3) that are described below:

- **Case A.** The memory updates have a data dependence between them (Cases (i) and (ii) in Example 10 in Section 1.3.3). The data dependence could be:
  - **Case 1.** a read-after-write (RaW) dependence,
  - **Case 2.** a write-after-read (WaR) dependence, or
  - **Case 3.** a write-after-write (WaW) dependence.

  A read-after-read (RaR) dependence is irrelevant.
- **Case B.** The memory updates do not have a data dependence between them (Case (iii) in Example 10 in Section 1.3.3).

- **Case C.** More information is needed to find out whether the memory updates have a data dependence between them (Case (iv) in Example 10 in Section 1.3.3).

These cases are exploited by the optimizations described below (Figure 1.9 gives examples for the cases described below):

1. *Strength reduction* optimization exploits case A1. It simplifies memory updates by using the information from other memory updates to eliminate data dependence between them.

   Example 14. In Figure 1.9(a), the data dependence between memory updates 1 and 2 can be eliminated as shown in the transformed code. After eliminating the data dependence, the control flow becomes redundant and can be eliminated, indicated by no ordering in the transformed code.

2. *Redundancy elimination* optimizations handle cases A2, A3, and B. They remove redundant memory updates (case A3) and minimize control flow (case B). The latter is based on exploiting (lack of) data dependence between memory updates. These opportunities are enhanced by strength reduction optimization.

   Example 15. In Figure 1.9(b), there is a WaW data dependence between memory updates 1 and 3 making the memory update 1 redundant. Hence, it can be eliminated. Notice that there is no data dependence between the memory updates 2 and 3 and hence the control flow ordering between them can be eliminated as shown in the transformed code.

   Case A2 is an anti-dependence (WaR) and is modelled by eliminating control flow and ensuring that it is not viewed as a RaW dependence. The control flow is redundant in this case because the memory updates with WaR data dependence are modelled by us as parallel assignments (Example 19 in Section 3.1).

3. *Call inlining* optimization handles case C by progressively providing more information. It inlines the summaries of the callees of a procedure at the call sites in the summary of the procedure. This enhances the opportunities of strength reduction and redundancy elimination and enables context-sensitive analysis.

Example 16. In Figure 1.9(c), the data dependence between memory updates 1 and 2 is unknown and depends on the alias information between x and y in the calling context. Thus, control flow between the two memory updates is important for a flow-sensitive analysis and hence should be retained as shown in the transformed code. Also notice that, pointee of x is accessed in the original code, however x is defined in the callers and hence we need a good representation for indirectly accessed unknown pointees.

More details are discussed in Section 1.3 and Chapter 3.

Our measurements suggest that

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 computed data may be subjected to in search of precision.

Our optimizations are effective because they eliminate data dependence wherever possible and discard irrelevant control flow without losing precision. The approaches that use flow and context insensitivity, discard control flow but over-approximate data dependence causing imprecision.

1.4.4 Additional Techniques for Achieving Efficiency

Additionally, we employ the following techniques to achieve scalability:

- Type-based non-aliasing: We use the types specified in the program to rule out data dependence between memory updates in order to eliminate redundant control flow and resolve some additional instances of case C into case B (for more details, see Section 5.4).

- We reduce the size of the control flow graphs (hence, the size of GPGs) by eliding the statements that do not access or generate points-to information (hence, irrelevant to points-to analysis). The control flow between the other basic blocks is maintained. This is similar to the sparse evaluation graph (SEG) [12, 18, 80].

- For optimizing the reaching GPUs analysis and the coalescing analysis on GPGs, the nodes in the worklist awaiting processing are prioritized by the reverse postorder traversal of the GPG such that the nodes with smaller order are processed before the nodes with a higher order. This optimization reduces the number of times a node is reprocessed.

- For optimizing the process of construction of GPGs, we traverse the def-use chains of SSA for efficiently simplifying memory updates involving SSA variables (top-level pointers that are not referenced indirectly via other pointers). Thus, our analysis employs a partial SSA to perform sparse analysis on top-level variables.

- We extend the concept of bypassing [73, 74] to pointers thereby filtering out the points-to information that is not accessed in a procedure. This enhances the efficiency of the second phase of bottom-up approach (the phase in which points-to information is computed using procedure summaries).

1.5 The Organization of the Thesis

Chapter 2 provides a survey on the literature of points-to analysis. Chapter 3 introduces the concept of generalized points-to updates (GPUs) that form the basis of GPGs and provides a brief overview of GPG construction through a motivating example. Chapter 4 describes the strength-reduction optimization performed on GPGs by formalizing the operations such as GPU composition and GPU reduction and defining data flow equations for reaching GPUs analyses. Chapter 5 describes redundancy elimination optimizations performed on GPGs. Chapter 6 explains the interprocedural use of GPGs by defining call inlining and shows how recursion and calls through function pointers are handled. Chapter 7 shows how GPGs are used for performing points-to analysis. Chapter 8 describes the handling of structures, unions and the heap. Chapter 9 presents empirical evaluation on SPEC benchmarks and Chapter 10 concludes the thesis with future work.

Chapter 2

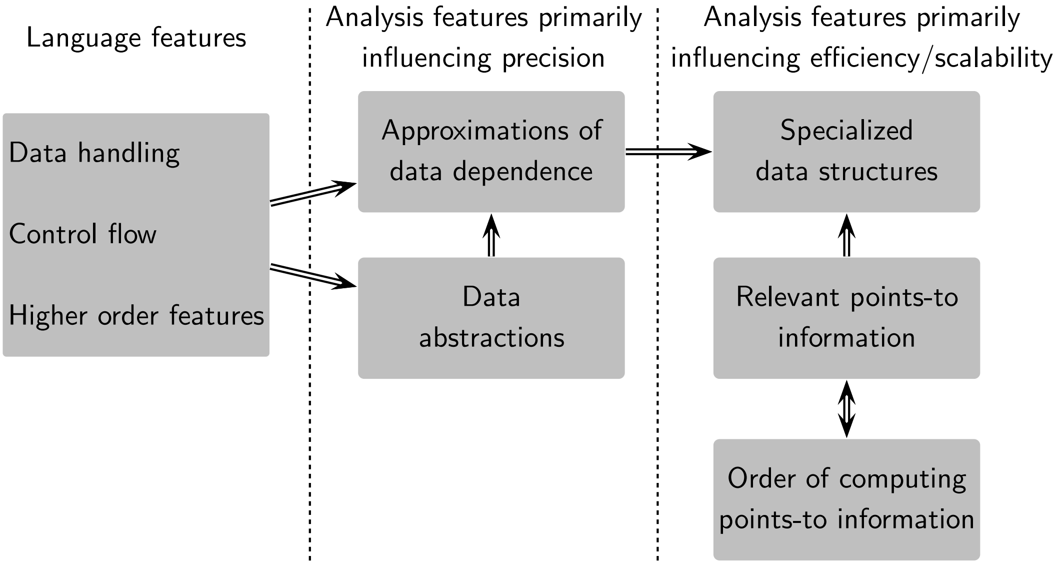
Literature Survey

Many investigations reported in the literature have described the popular points-to analysis methods and have presented a comparative study of the methods with respect to scalability and precision [42, 43, 48, 60, 95, 99, 101]. Instead of describing the methods, we devise a metric of features that influence the precision and efficiency/scalability of points-to analysis. This metric can be used for identifying important characteristic of any points-to analysis at an abstract level.

Section 2.1 describes the big picture view of our metric and also lists the popular approaches based on the characteristics of the analysis. Section 2.2 describes the interaction between the characteristic features. Section 2.3 contextualizes our work in the big picture. Section 2.4 describes the state-of-the-art approaches for bottom-up interprocedural points-to analysis.

2.1 Factors Influencing the Precision, Efficiency, and Scalability of Points-to Analysis

Figure 2.1 presents our metric. At the top level, we have language features and analysis features (the top-row of the picture). The analysis features have been divided further based on whether their primary influence is on the precision or efficiency/scalability of points-to analysis (the three-columns in the picture). The categorization of language features is obvious. Here we describe our categorization of analysis features.


Language features
Analysis features primarily influencing precision
Analysis features primarily influencing efficiency/scalability
Data handling
Control flow
Higher order features
Approximations of data dependence
Data abstractions
Specialized data structures
Relevant points-to information
Order of computing points-to information

Feature		Examples
Language	Data handling	Addressof (&) operator, type casts, unions, dynamic memory allocation, pointer arithmetic, container objects
	Control flow	Function pointers, receiver objects of calls, virtual calls, concurrency
	Higher order features	Reflection, *eval* in Javascript
Analysis	Approximations of data dependence	Path-sensitivity, flow-sensitivity, context-sensitivity, SSA form
	Data abstractions	Field-sensitivity, object-sensitivity, allocation-site-based or type-based abstraction of heap, heap cloning, summarized access paths, summarization of aggregates
	Relevant points-to information	All pointers (exhaustive analysis), relevant pointers in incremental, demand-driven, staged, level-by-level, or liveness-based analyses
	Order of computing points-to information	Governed by relevance of pointers, or by algorithmic features (e.g. top-down, bottom-up, parallel, or randomized algorithms)
	Specialized data structures	BDDs, bloom filters, disjoint sets (for union-find), points-to graphs with placeholders, GPGs

Figure 2.1: Language and analysis features affecting the precision, efficiency, and scalability of points-to analyses. An arrow from feature A to feature B indicates that feature A influences feature B. The features influencing precision, influence efficiency and scalability indirectly.

2.1.1 Features Influencing Precision

Two important sources of imprecision in an analysis are approximation of data dependence and abstraction of data (the middle column of the picture in Figure 2.1).

2.1.1.1 Approximations of Data Dependence

Observe that control flow in imperative languages is a proxy for implicit data dependence created by a temporal ordering between the definitions and uses of variables. The flow- or context-insensitive approaches over-approximate the data dependence because they over-approximate the control flow. In other words, control flow over-approximation may introduce spurious data dependences that may have not existed if the analysis respected the control flow. This causes imprecision. In this section, we describe various ways in which data dependences have been approximated for points-to analysis.

Flow Sensitivity versus Flow Insensitivity

Flow-insensitivity effectively creates a complete graph out of a control flow graph causing over-approximation in the control flow and hence in the data dependence. The classical inclusion-based [2] and equality-based [102] points-to analyses are the most popular flow-insensitive approaches. The time complexity of equality-based approach is almost linear, however it is very imprecise because it does not distinguish between the LHS and the RHS of an assignment. The inclusion-based approach makes this distinction and hence is more precise compared to the equality-based approach but has cubic complexity. Shapiro and Horowitz [93] propose an algorithm which is tuned such that its precision ranges from equality-based to inclusion-based points-to analysis. Many algorithms were proposed to accelerate the inclusion-based approach [21, 31, 39, 69]. The inclusion-based points-to analysis is also extended to object-oriented languages [83]. Spark [59] provides a flexible framework for experimenting with points-to analyses for Java. Spark supports equality- and subset-based analyses, variations in field sensitivity, and several solving algorithms. Spark is composed of building blocks on which new analyses can be based. Rountev et al. [83] propose points-to analysis for Java using annotated constraints.

Some approaches propose partial flow sensitivity; they treat some variables flow sensitively and the rest flow insensitively. Whaley and Lam [107] have analyzed Java programs flow sensitively only for local variables. Lhoták et al. [58] combined flow-insensitivity with strong updates. They claim that this algorithm is efficient like flow-insensitive analysis, with the same worst-case bounds, yet its precision benefits from strong updates like flow-sensitive analysis. The work on program decomposition [118] helps an analysis choose different parts of a program to be analyzed with varying levels of precision. It enables exploration of a trade-off between algorithm efficiency and precision by allowing flow-insensitive and flow-sensitive alias analyses to be used on independent parts of the program. Region-based selective flow-sensitive pointer analysis [113] operates on the regions partitioned from a program. Flow-sensitivity is maintained between the regions but not within them, making traditional flow-insensitive and flow-sensitive as well as recent sparse flow-sensitive analyses all special instances of the framework.

Several flow-sensitive algorithms have also been proposed in the literature [11, 53, 62, 64, 88, 107]. Hind et al. [41] propose an approximation algorithm for interprocedural alias analysis. This work presents a flow-sensitive and a flow-insensitive interprocedural pointer alias analysis algorithm. It also presents a flow-insensitive interprocedural pointer alias analysis algorithm that incorporates kill information to improve precision. It also provides empirical measurements of the efficiency and precision of the three interprocedural alias analysis algorithms. This work claims that a flow-insensitive analysis with kill does not improve precision over a flow-insensitive analysis without kill.

Hind and Pioli performed several studies on precision/scalability trade-off of flow sensitivity [40, 42, 43] indicating that the precision gain is not worth the price one has to pay for flow sensitivity. However, the work by Hardekopf and Lin [32, 33] found very good results for flow-sensitive points-to analysis. The importance of flow sensitivity has been further strengthened by Stefan Staiger-Stöhr [101] who criticizes the claim of flow sensitivity being not worth the price with counter explanations.

SSA-Based Flow Sensitivity

Static single assignment (SSA) form also discards control flow but avoids over-approximation in data dependences by creating def-use chains in the form of SSA edges. This is possible because of referential transparency of renamed variables in SSA form. Thus, an analysis over SSA version of a program becomes a sparse analysis because it can ignore the statements appearing between definitions and uses of data. Besides, for the use and definition statements, it computes the information only for the variables being defined and used.

Hasti and Horwitz [34] combined a flow-insensitive pointer analysis with SSA form [14]. They use an iterative process to obtain progressively better results. The algorithm can be 'tuned' to provide a range of results that fall between the results of flow-insensitive and flow-sensitive analysis.

One of the earliest sparse flow-sensitive points-to analysis which incrementally constructs SSA was described by Chase et al. [8]. It is similar to Tok's work [105, 106]. Hardekopf and Lin [32] proposed a semi-sparse flow-sensitive analysis for efficient handling of strong updates. They convert non-address-taken or top-level local variables to SSA form to improve analysis efficiency. Another work of Hardekopf and Lin [33] is based on a sparse representation of program code created by a staged, flow-insensitive pointer analysis. This approach scales flow-sensitive points-to analysis to programs with millions of lines of code.

The classical SSA form for arrays cannot provide the element-level data flow information required for such analyses. An Array SSA form [54] captures precise element-level data flow information for array variables in all cases. It is general and simple, and coincides with standard SSA form when applied to scalar variables. It can also be used for structures and other variable types that can be modeled as arrays.

Context Sensitivity versus Context Insensitivity

Context insensitivity treats calls and returns as goto statements as far as the control transfer between procedures is concerned. This over-approximates the interprocedural control flow by propagating data flow information along the interprocedurally invalid paths and hence over-approximates the data dependence between assignments across procedures.

Several approaches [2, 21, 31, 32, 33, 34, 39, 69, 102] are context-insensitive. Fahndrich et al.[22] proposed a context-sensitive flow-analysis using instantiation constraints. They show that flow information can be computed efficiently while considering only the paths with well-defined call-return sequences, even for higher-order programs. Context-sensitivity using value flow graphs (VFGs) [62] is achieved by simultaneously applying function cloning and computing context-free language reachability (CFL-reachability) by restricting the flow of values to interprocedural paths in which procedure calls and returns are matched.

Milanova et al. [68] proposed object sensitivity, a new form of context sensitivity for flow-insensitive points-to analysis for Java. The results show that object sensitivity significantly improves the precision of side-effect analysis and call graph construction, compared to (a) context-insensitive analysis, and (b) context-sensitive points-to analysis that models context using the invoking call site. These experiments demonstrate that object-sensitive analyses can achieve substantial precision improvement, while at the same time remaining efficient and practical. Smaragdakis et al. [96] defined a full-object-sensitive analysis that results in significantly higher precision and often performance. They also introduced type-sensitivity as an explicit approximation of object-sensitivity that preserves high context quality at substantially reduced cost.

Lhoták et al. [61] proposed PADDLE framework that supports several variations of context-sensitive analyses, including call site strings and object sensitivity, and context-sensitively specializes both pointer variables and the heap abstraction. They claim that object-sensitive analyses are more precise than comparable variations of the other approaches, and that specializing the heap abstraction improves precision more than extending the length of context strings. Hybrid context-sensitivity [50] shows that a selective combination of call-site- and object-sensitivity for Java points-to analysis is highly profitable. A selective combination of both kinds of contexts not only vastly outperforms non-selective combinations but is also faster than a mere object-sensitive analysis.

Lattner et al. [57] proposed a heap-cloning based context-sensitive points-to analysis. For achieving a scalable implementation, several algorithmic and engineering design choices were made in this approach. Some of these choices are: a flow-insensitive and unification-based analysis, and sacrificing context-sensitivity across recursive procedures.

Cheng and Mei [10] proposed a modular interprocedural pointer analysis based on access-paths for C programs. They illustrate that access-paths can reduce the overhead of representing context-sensitive transfer functions. Lian et al. [63] presents a flow-insensitive, context-sensitive points-to analysis algorithm that computes alias information that is almost as precise as that computed by Andersen's algorithm and almost as efficient as Steensgaard's algorithm.

Whaley et al. [108] achieves context sensitivity by creating a clone of a method for every context of interest, and run a context-insensitive algorithm over the expanded call graph to get context-sensitive results. For precision, a clone for every acyclic path through a program's call graph is created, treating methods in a strongly connected component (i.e., methods in a cycle of recursion) as a single node.

2.1.1.2 Data Abstractions

An abstract location usually represents a set of concrete locations. An over-approximation of this set of locations leads to spurious data dependences because a large number of locations are treated alike. This causes imprecision in points-to analysis. In this section, we describe common techniques that use different kinds of data abstractions in points-to analysis.

Field Sensitivity versus Field Insensitivity

A field-sensitive analysis treats each field of a structure variable as a separate variable. A field-insensitive analysis over-approximates the fields of a structure variable by a common single variable thereby over-approximating the concrete locations and introducing imprecision.

Yong et al. [114] proposed a points-to analysis to handle structures and type-casting. They observe that supporting field-sensitivity can significantly improve the analysis precision. Pearce et al. [76, 77] proposes a field-sensitive points-to analysis for modeling aggregates and function pointers. They found that a field-sensitive analysis is more expensive to compute, but yields significantly better precision over a field-insensitive analysis.

Heap Abstraction

Heap data is potentially unbounded and seemingly arbitrary. Hence, unlike stack and static data, heap data cannot be abstracted in terms of a fixed set of program variables. Specialization of heap objects is critical for points-to analysis to effectively analyze complex memory activity. The most common heap abstraction uses an allocation site to treat all locations allocated by a given allocation statement as same. An alternative less precise abstraction partitions the heap locations based on types. Kanvar et al. [48] provide a big-picture view of heap abstractions.

Nystrom st al. [72] discusses heap specialization with respect to call chains. Due to the sheer number of distinct call chains, exhaustive specialization can be cumbersome. On the other hand, insufficient specialization can miss valuable opportunities to prevent spurious data flow, which results in not only reduced accuracy but also increased overhead. Lattner et al. [57] propose a heap-cloning based context-sensitive points-to analysis. Boomerang [97] abstracts heap using access graphs which is a storeless abstraction for handling heap precisely.

2.1.2 Features Influencing Efficiency and Scalability

Different methods use different techniques to achieve scalability. We characterize them based on the following three criteria.

2.1.2.1 Specialized Data Structures

A method may use specialized data structures for encoding information efficiently (e.g. BDDs or GPUs and GPGs) or may use them for modelling relevant points-to information (e.g. use of placeholders to model accesses of unknown pointees in a bottom-up method). Several innovative data structures for representing the points-to (or alias) information have been proposed.

Earlier approaches stored alias pairs explicitly [55]. However, this representation is storage-intensive and hence a compact representation is proposed which stores only a few basic alias pairs explicitly and new alias pairs are derived based on transitivity and symmetry [11]. Later, a more compact representation in the form of points-to pairs was introduced [20] which significantly reduced the storage requirement because points-to pairs store only the edges in a memory graph unlike aliases that store pairs of paths incident on the same node in a memory graph.

Heintze and Tardieu [39] proposed the use of sparse bitmaps for storing points-to information. However, bitmaps cannot take advantage of the commonality across various points-to sets. Therefore, for a context-sensitive analysis, the use of bitmaps requires a large amount of memory.

Zhu [122] observed that a vast amount of points-to information can be encoded in a space-efficient manner using binary decision diagrams (BDD) [6]. Until then, BDDs were used in symbolic model checking [7] and to represent large sets and maps [66]. Due to the storage efficiency, BDDs were quickly adapted for solving points-to analysis algorithms. Berndl et al. [4], Hardekopf and Lin [31, 32, 33], Lhoták et al. [59], Whaley and Lam [108], and Zhu and Calman [123] proposed variants of points-to analysis algorithms using BDDs.

Nasre et al. [70] used a specially designed multi-dimensional bloom filter for storing points-to information for a flow-insensitive and context-sensitive analysis. This probabilistic data structure allows trading precision for memory usage of the analysis.

Lattner et al. [57] proposed a data structure graph (DS graph) for each function in a program, summarizing the memory objects accessible within the function along with their connectivity patterns. Value flow graph (VFG) [62] and pointer assignment graph (PAG) [92] represent flow of values along edges between memory objects and pointer variables (represented as nodes). Additionally, they contain edges to represent interprocedural control flow. Spark [59] uses a pointer assignment graph as its internal representation of the program being analyzed. Several approaches [12, 18, 80] use a sparse evaluation graph (SEG) which is computed by eliding the statements that do not access or generate points-to information (hence, irrelevant to points-to analysis). The analysis becomes more efficient because it needs to process fewer basic blocks and control flow edges. Hardekopf and Lin [32] uses a dataflow graph (DFG) as a data structure which is a combination of a sparse evaluation graph (SEG) and def-use chains.

There are many data structures in the literature that represent procedure summaries (e.g. VFG, PAG, DS graph). However, they have not been explicitly projected as procedure summaries.

2.1.2.2 Relevant Points-to Information

Many methods choose to prioritize computing a specific kind of points-to information which is then used to compute further points-to information. For example, staged points-to analyses [33] begin with conservative points-to information which is then made more precise. Similarly, some methods [47, 116] begin by computing points-to information for top-level pointers whose indirections are then eliminated. This uncovers a different set of pointers as top-level pointers whose points-to information is then computed.

Demand-driven algorithms [28, 29, 38, 93, 97, 98, 100, 121] compute points-to information that is relevant to the demand raised by the client analysis. Boomerang [97] is based on IFDS which is a top-down tabulation-based approach. A distinctive feature of Boomerang is that it extends the IFDS framework to support non-distributive analyses such as alias and points-to analysis.

An incremental analysis handles dynamic addition and/or deletion of a set of statements to the already analyzed program and processes the statements without having to analyze the complete program (original program plus new statements) from scratch. Thus, for an incremental points-to analysis, the relevant points-to information is associated with the changes in the program. Saha and Ramakrishnan [90] describe a framework based on logic programming for implementing various incremental and demand-driven program analyses formulated using deductive rules. Yur et al. [117] propose an incremental flow-sensitive and context-sensitive points-to analysis algorithm to handle addition and deletion of single statements in a C program.

Liveness-based points-to analysis [53] computes points-to information only for live variables. Thus, in this case, the relevant points-to information is governed by liveness.

2.1.2.3 Order of Computing Points-to Information

Most methods order computations based on relevant points-to information which may also be defined in terms of a chosen order of traversal over the call graph (eg. top-down or bottom-up).

The order of computation of points-to information varies in a points-to analysis of multi-threaded programs and parallel points-to analysis. Due to numerous thread-interleavings possible in a multi-threaded program, the analysis of such programs poses significant challenges from precision and scalability perspectives. Salcianu and Rinard [91] proposed a combined pointer and escape analysis for multi-threaded programs. There has been some work on parallelizing the pointer analysis algorithm. Some of the approaches [47, 84, 118] simply mention that their algorithms could be parallelized, a parallel pointer analysis is proposed by Mendez-Lojo et al. [67]. Edvinsson et al. [19] proposed parallel points-to analysis for object oriented programs. Rugina and Rinard [85, 86] proposed an interprocedural, context-sensitive and flow-sensitive pointer analysis for multi-threaded programs. Putta and Nasre [78] proposed a parallel version of context-sensitive inclusion-based points-to analysis for C programs. They make use of replication of points-to sets to improve parallelism.

Zhao et al. [120] proposed a parallel sparse flow-sensitive points-to analysis. It uses Array SSA form [54] (for capturing precise element-level data flow information for arrays) and Heap SSA form (for modelling each field as a distinct logical “heap array”).

2.2 Interaction between Analysis Features

In this section we explain the interaction between the features indicated by the arrows shown in Figure 2.1.

- *Data abstraction influences approximations of data dependence.* An abstract location may be over-approximated to represent a larger set of concrete locations in many situations such as in field-insensitivity, type-based abstraction, allocation site-based abstraction, etc. This over-approximation creates spurious data dependence between the concrete locations represented by an abstract location.

- *Approximation of data dependence influences the choice of efficient data structures.* Some flow-insensitive methods use disjoint sets for efficient union-find algorithms. Several methods use BDDs for scaling context-sensitive analyses.

- *Relevant points-to information affects the choice of data structures.* Points-to information is stored in the form of graphs, points-to pairs, or BDDs for top-down approaches. For bottom-up approaches, points-to information is computed using procedure summaries that use placeholders or GPUs.

- *Relevant points-to information and order of computing influence each other mutually.* In level-by-level analysis [116], points-to information is computed one level at a time. The relevant information to be computed at a given level requires points-to information computed by the higher levels. Thus, in this case the relevance of points-to information influences the order of computation. In liveness-based points-to analysis (LFCPA) [53] only the live pointers are relevant. Thus, points-to information is computed only when the liveness of pointers is generated. Thus, the order of computing dictated by the generation of liveness information influences the relevant points-to information to be computed.

2.3 Positioning Our Work in the Feature Metric

GPG-based points-to analysis preserves data dependence by being flow- and context-sensitive. It is path-insensitive and uses SSA form for top-level local variables. Unlike the approaches that over-approximate control flow indiscriminately, we discard control flow as much as possible but only when there is a guarantee that it does not over-approximate data dependence. Our analysis is field-sensitive. It over-approximates arrays by treating all its elements alike. We use allocation-site-based abstraction for representing heap locations and use k-limiting for summarizing the unbounded accesses of heap where allocation sites are not known. Like every bottom-up approach, points-to information is computed when all the information is available in the context. Our analysis is exhaustive and computes points-to information for all pointers.

GPGs are different from other data structures as follows:

- Unlike DFGs [32], GPGs do not have data dependence edges (def-use chains). Before GPG optimizations, data dependences are implicitly preserved by control flow edges and GPG optimizations eliminate data dependences thereby minimizing the control flow edges.
- Unlike PAGs [92] and VFGs [62], GPGs maintain a clean separation between the paths in a memory graph (represented by GPUs in a GPB) and the paths in a control flow graph (represented by control flow edges across GPBs).

2.4 Approaches of Constructing Procedure Summaries

This section describes the investigations that use a bottom-up approach for points-to analysis because our work falls in this category. This category can be further subdivided into the multiple transfer functions (MTF) and the single transfer function (STF) approach.

2.4.1 The Multiple Transfer Functions (MTF) Approach

In this approach control flow is not required to be recorded between memory updates [47, 110, 116, 119]. The data dependence between memory updates (even those that access unknown pointers) is known because either the alias information or the points-to information from the calling context is used. Figure 2.2 gives an example where multiple procedure summaries are created for a code snippet given in the figure based on the alias information in the calling context. These approaches construct symbolic procedure summaries. This involves computing preconditions and corresponding postconditions (in

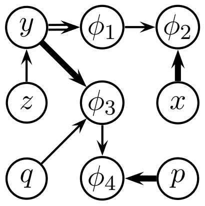
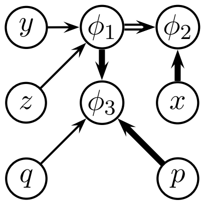
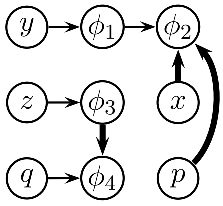
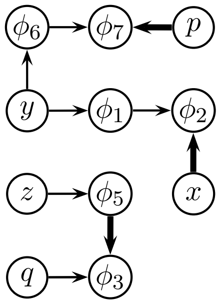
	MTF Approach			STF Approach
1. $x = *y$; 2. $*z = q$; 3. $p = *y$;				 ϕ_6 ϕ_7 p y ϕ_1 ϕ_2 z ϕ_5 x q ϕ_3
Example	$*z$ and y are aliases	z and y are aliases	z and y are not related	no assumption about aliases

Figure 2.2: An example for demonstrating MTF and STF approach. Two dereferences of y are separated by a possibly side-effect causing statement through z. Edges with double lines represent the killed information. Thick edges represent the generated information. Black edges represent carried forward input information. MTF approach creates multiple summaries for a procedure for each combination of aliases. Only relevant aliases are considered. STF approach makes no assumption about aliases.

terms of aliases or points-to information). A calling context is matched against a precondition and the corresponding postcondition gives the result.

Level-by-level analysis [116] constructs a procedure summary with multiple interprocedural conditions. It matches the calling contexts with these conditions and chooses the appropriate summary for a given context. This method partitions the pointer variables in a program into different levels based on the Steensgaard's points-to graph for the program. It constructs a procedure summary for each level (starting with the highest level) and uses the points-to information from the previous level. This method constructs interprocedural def-use chains by using extended SSA form. When used in conjunction with conditions based on points-to information from calling contexts, the chains become context sensitive.

The scalability of these approaches depends on the number of aliases/points-to pairs in the calling contexts, which could be large. Further, this approach may not be useful for constructing summaries for library functions which have to be analyzed without the benefit of calling contexts. Saturn [30] creates sound summaries but they may not be precise across applications because of their dependence on context information.

Relevant context inference [9] constructs a procedure summary by inferring the relevant potential aliasing between unknown pointees that are accessed in the procedure.

Although, it does not use the information from the context, it has multiple versions of the summary depending on the alias and the type context. This analysis could be inefficient if the inferred possibilities of aliases and types do not actually occur in the program. It also over-approximates the alias and the type context as an optimization thereby being only partially context-sensitive.

2.4.2 The Single Transfer (STF) Approach

This approach does not make any assumptions about the calling contexts [62, 65, 92, 103, 109] but constructs large procedure summaries causing inefficiency in fixed-point computation at the intraprocedural level. It introduces separate placeholders for every distinct access of a pointee (Section 1.3.4). Figure 2.2 gives an example for STF approach with a large procedure summary when no information from the calling context is available.

In this approach, the data dependence is not known in the case of indirect accesses of unknown pointees and hence control flow is required for constructing the summary for a flow-sensitive points-to analysis. However, these methods do not record control flow between memory updates in the summaries so constructed. Hence, for soundness, the use of a procedure summary at a call site does not kill any information. The assumption of any ordering between memory updates and absence of kill introduces imprecision. However, it may not have much adverse impact on programs written in Java because all local variables in Java have SSA versions, thanks to the absence of indirect assignments to variables (there is no addressof operator). Besides, there are few static variables in Java programs and absence of kill for them may not matter much; the points-to relations of heap locations are not killed in any case.

Note that the MTF approach is precise even though no control flow in the procedure summaries is recorded because the information from calling context obviates the need for control flow.

2.4.3 The Hybrid Approach

Hybrid approaches use customized summaries and combine the top-down and bottom-up analyses to construct summaries [119]. This choice is controlled by the number of times a procedure is called. If this number exceeds a fixed threshold, a summary is constructed using the information of the calling contexts that have been recorded for that procedure. A new calling context may lead to generating a new precondition and hence a new summary. If the threshold is set to zero, then a summary is constructed for every procedure and hence we have a pure bottom-up approach. If the threshold is set to a very large number, then we have a pure top-down approach and no procedure summary is constructed.

Additionally, we can set a threshold on the size of procedure summary or the percentage of context-dependent information in the summary or a combination of these choices. In our implementation, we have used the percentage of context-dependent information as a threshold—when a procedure has a significant amount of context-dependent information, it is better to introduce a small touch of top-down analysis (Section 9.4.2). If this threshold is set to 0%, our method becomes purely bottom-up approach; if it is set to 100%, our method becomes a top-down approach.

2.5 Chapter Summary

In this chapter we presented a survey of various points-to analysis algorithms. We classified the analyses based on the metric of features that influence the precision and efficiency/scalability of points-to analysis.

Chapter 3

The Generalized Points-to Graphs

In this chapter, we define a *generalized points-to graph* (GPG) which serves as our memory transformer. It is a graph with *generalized points-to blocks* (GPBs) as nodes which contain a set of *generalized points-to updates* (GPUs). The ideas and algorithms for defining and computing these three representations of memory transformers can be seen as a collection of abstractions, operations, data flow analyses, and optimizations. Their relationships are shown in Figure 3.1. A choice of key abstractions enables us to define GPU operations which are used for performing three data flow analyses. The information computed by these analyses enables optimizations over GPGs.

Section 3.1 defines GPGs and Section 3.2 provides an overview of GPG operations. Section 3.3 presents an overview of our approach in a limited setting of our motivating example of Figure 3.2. Towards the end of this chapter, Figure 3.7 fleshes out Figure 3.1 to list specific abstractions, operations, analyses, and optimizations.

3.1 Defining a Generalized Points-to Graph (GPG)

We model the effect of a pointer assignment on an abstract memory by defining the concept of *generalized points-to update* (GPU) in Definition 1. We use the statement label s to capture weak versus strong updates and for computing points-to information.[1] Definition 1 gives the abstract semantics of a GPU. The concrete semantics of a GPU $x \xrightarrow[s]{i|j} y$

[1] We omit the statement labels in GPUs at some places when they are not required.

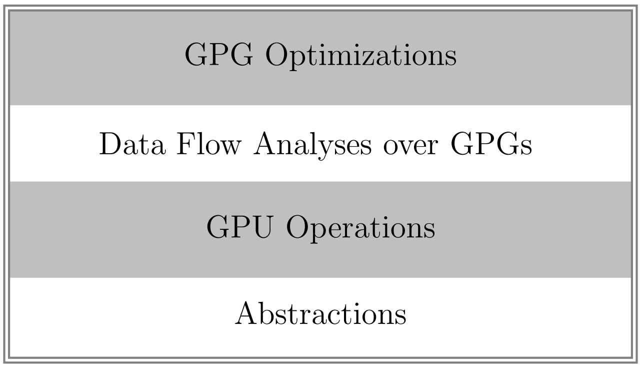


Figure 3.1: Inter-relationships between ideas and algorithms for defining and computing GPUs, GPBs, and GPGs. Each layer is defined in terms of the layers below it. Figure 3.7 fleshes out this picture by listing specific abstractions, GPU operations, data flow analyses, and optimizations.

Given variables x and y and $i > 0$, $j \geq 0$, a *generalized points-to update* (GPU) $x \xrightarrow[s]{i|j} y$ represents a memory transformer in which all locations reached by $i - 1$ indirections from x in the abstract memory are defined by the pointer assignment labelled s, to hold the address of all locations reached by j indirections from y. The pair $i|j$ represents indirection levels and is called the *indlev* of the GPU (i is the *indlev* of x, and j is the *indlev* of y). The letter γ is used to denote a GPU unless named otherwise.

Definition 1: *Generalized Points-to Update (GPU).*

can be viewed as the following C-style pointer assignment with $i - 1$ dereferences of x[2] and j dereferences of $\&y$:

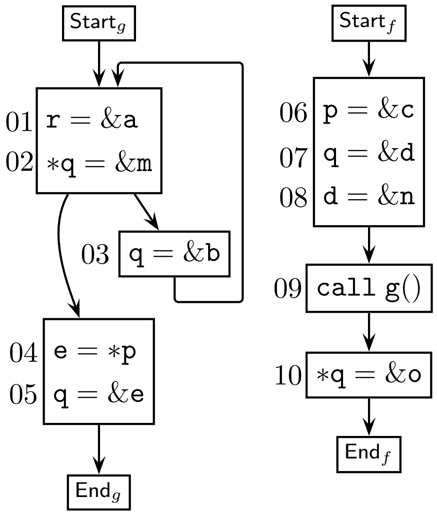
$$\underbrace{*\,*\ldots*}_{(i-1)}\mathtt{x} = \underbrace{*\,*\ldots*}_{j}\&\mathtt{y}$$

A GPU $\gamma : x \xrightarrow[s]{i|j} y$ generalizes a points-to edge[3] with the following properties:

- The direction indicates that the source x with *indlev* i identifies the locations being defined by the assignment s and the target y with *indlev* j identifies the locations

[2] Alternatively, i dereferences of $\&x$. We choose $i - 1$ dereference from x because the left-hand side cannot be $\&x$.

[3] Although a GPU can be drawn as an arrow just like a points-to edge, we avoid the term 'edge' for a GPU because of the risk of confusion with a 'control flow edge' in a GPG.



Variables	Types
`m, n, o`	`int`
`a`	`float*`
`b, c, d, e`	`int*`
`p, q`	`int**`
`r`	`float**`

Figure 3.2: A motivating example. Procedures are represented by their control flow graphs (CFGs). All variables are global.

whose addresses are read.

- The GPU γ abstracts away $i - 1 + j$ placeholders.
- The GPU γ represents *may* information because different locations may be reached from x and y along different control flow paths reaching statement s in the procedure.

We refer to a GPU with $i = 1$ and $j = 0$ as a *classical points-to edge* as it encodes the same information as edges in classical points-to graphs.

Example 17. The pointer assignment in statement 01 in Figure 3.2 is represented by a GPU $r \xrightarrow[01]{1|0} a$ where the indirection levels "1|0" appear above the arrow and the statement number "01" appears below the arrow. The indirection level 1 in "1|0" indicates that r is defined by the assignment and the indirection level 0 in "1|0" indicates that the address of a is read. Similarly, statement 02 is represented by a GPU $q \xrightarrow[02]{2|0} m$. The indirection level 2 for q indicates that some pointee of q is being defined and the indirection level 0 indicates that the address of m is read.

Figure 3.3 presents the GPUs for basic pointer assignments in C. (To deal with C structs and unions, GPUs are augmented to encode lists of field names—for details see Figure 8.2 in Chapter 8).

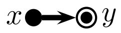
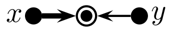
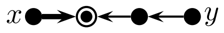
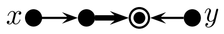
Pointer assignment	GPU	Relevant memory graph after the assignment
$s\colon \texttt{x} = \&\texttt{y}$	$x \xrightarrow[s]{1\|0} y$	
$s\colon \texttt{x} = \texttt{y}$	$x \xrightarrow[s]{1\|1} y$	
$s\colon \texttt{x} = *\texttt{y}$	$x \xrightarrow[s]{1\|2} y$	
$s\colon *\texttt{x} = \texttt{y}$	$x \xrightarrow[s]{2\|1}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

Figure 3.3: GPUs for basic pointer assignments in C. In the memory graphs, a double circle indicates the location whose address is being assigned, a thick arrow shows the generated edges. Unnamed nodes may represent multiple pointees (implicitly representing placeholders).

A *generalized points-to block* (GPB), denoted δ, is a set of GPUs abstracting memory updates. A *generalized points-to graph* (GPG) of a procedure, denoted $\varDelta$, is a graph (N, E) whose nodes in N are labelled with GPBs and edges in E abstract the control flow of the procedure. By common abuse of notation, we often conflate nodes and their GPB labellings.

Definition 2: *Generalized Points-to Blocks (GPBs) and Generalized Points-to Graphs (GPGs).*

GPUs are useful rubrics of our abstractions because they can be composed to construct new GPUs with smaller indirection levels whenever possible thereby converting them progressively to classical points-to edges. The composition between GPUs eliminates the data dependence between them and thereby, the need for control flow ordering between them. Section 3.2 briefly describes the operations of *GPU composition* and *GPU reduction* which are used for the purpose; they are defined formally in Chapter 4.

A GPU can be seen as an atomic transformer which is used as a building block for the *generalized points-to graph* (GPG) as a memory transformer for a procedure (Definition 2). The GPG for a procedure differs from its control flow graph (CFG) in the following way:

- The CFG of a procedure could have procedure calls whereas its GPG does not.[4] Besides, a GPG is acyclic in almost all cases, even if the procedure it represents

[4] In the presence of recursion and calls through function pointers (Sections 6.3 and 6.4), we need an intermediate form of GPG called an *incomplete* GPG containing unresolved calls that are resolved when more information becomes available.

has loops or recursive calls. Our empirical measurements (Table 9.2 in Chapter 9) show that very few procedures (the number is in single digits) out of hundreds of procedures have back edges in their optimized GPGs.

- The GPBs which form the nodes in a GPG are analogous to the basic blocks of a CFG except that the basic blocks are sequences of statements but GPBs are (unordered) sets of GPUs representing parallel assignments.

A concrete semantic reading of a GPB δ is defined in terms of the semantics of executing a GPU (Definition 1). Execution of δ implies that the GPUs in δ are executed non-deterministically in any order. Effectively all GPUs in δ represent parallel assignments in which all right hand sides are evaluated before any left hand side is written. This gives a correct abstract reading of a GPB as a *may* property. But a stronger concrete semantic reading also holds as a *must* property: Let δ contain GPUs corresponding to some statement s. Define $X_s \subseteq \delta$ by $X_s = \{x \xrightarrow[s]{i|j} y \in \delta\}$, $X_s \neq \emptyset$. Then, whenever statement s is reached in any execution, at least one GPU in X_s *must* be executed. This semantics corresponds to that of the points-to information generated for a statement in the classical points-to analysis. This gives GPBs their expressive power—multiple GPUs arising from a single statement, produced by GPU-reduction (see later), represent *may*-alternative updates, but one of these *must* be executed.[5]

Example 18. Consider a GPB $\{\gamma_1 : x \xrightarrow[11]{1|0} a, \gamma_2 : x \xrightarrow[11]{1|0} b, \gamma_3 : y \xrightarrow[12]{1|0} c, \gamma_4 : z \xrightarrow[13]{1|0} d, \gamma_5 : t \xrightarrow[13]{1|0} d\}$. After executing this GPB we know that the points-to sets of x is overwritten to become $\{a, b\}$ (i.e. x definitely points to one of a and b) because GPUs γ_1 and γ_2 both represent statement 11 and define a single location x. Similarly, the points-to set of y is overwritten to become $\{c\}$ because γ_3 defines a single location c in statement 12. However, this GPB causes the points-to sets of z and t to *include* $\{d\}$ (without removing the existing pointees) because γ_4 and γ_5 both represent statement 13 but

[5] A subtlety is that a GPB δ may contain a spurious GPU that can never be executed because the flow functions of points-to analysis are non-distributive [51]. This is a consequence of introducing over-approximation to compute a decidable version of an undecidable analysis (we perform flow-sensitive analysis as opposed to path-sensitive analysis).

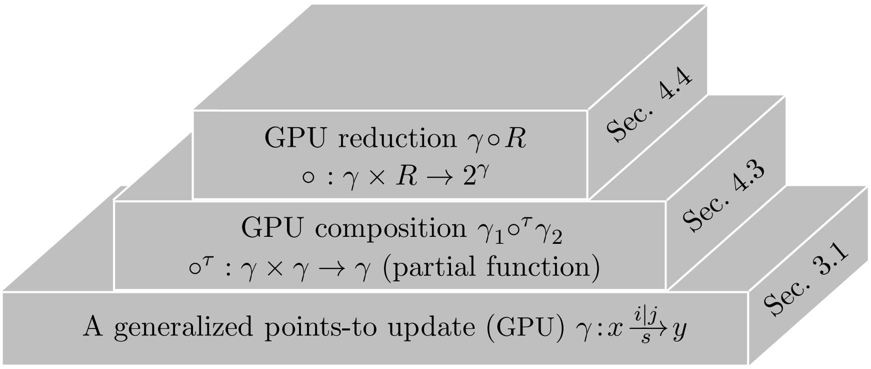

GPU reduction $\gamma \circ R$
$\circ : \gamma \times R \rightarrow 2^{\gamma}$
Sec. 4.4
GPU composition $\gamma_1 \circ^{\tau} \gamma_2$
$\circ^{\tau} : \gamma \times \gamma \rightarrow \gamma$ (partial function)
Sec. 4.3
A generalized points-to update (GPU) $\gamma : x \xrightarrow[s]{i|j} y$
Sec. 3.1

Figure 3.4: A hierarchy of core operations involving GPUs. The set of GPUs reaching a GPU γ (computed using the reaching GPUs analyses of Sections 4.6 and 4.7) is denoted by R. By abuse of notation, we use γ, δ, and R also as types to indicate the signatures of the operations. The operator "$\circ$" can be disambiguated using the types of the operands.

define separate locations. Thus, x and y are strongly updated (their previous pointees are removed) but z and t are weakly updated (their previous pointees are augmented).

The above example also illustrates how GPU statement labels capture the distinction between strong and weak updates.

The *may* property of the absence of control flow between the GPUs in a GPB (i.e., the effect of parallel assignments) allows us to model a WaR dependence as illustrated in the following example:

Example 19. Consider the code snippet on the right. There is a WaR data dependence between statements 01 and 02. If the control flow is not maintained, the statements could be executed in the reverse order and y could erroneously point to a.

```
01   y = x;
02   x = &a;
```

We construct a GPB $\{y \xrightarrow[01]{1|1} x, x \xrightarrow[02]{1|0} a\}$ for the code snippet. The *may* property of this GPB ensures that there is no data dependence between these GPUs. The execution of this GPB (containing a set of GPUs representing parallel assignments) in the context of the memory represented by the GPU $x \xrightarrow[12]{1|0} b$, computes the points-to information $\{y \rightarrow b, x \rightarrow a\}$. It does not compute the erroneous points-to information $y \rightarrow a$ thereby preserving the WaR dependence. Thus, WaR dependence can be handled without maintaining control flow.

3.2 An Overview of GPG Operations

Figure 3.4 lists the GPG operations based on the concept of generalized points-to updates (GPUs). Each layer is defined in terms of the layers below it. For each operation, Figure 3.4 describes the types of its operands and result, and lists the section in which the operation is defined.

3.2.1 GPU Composition

In a compiler, the sequence $p = \&a; *p = x$ is usually simplified to $p = \&a; a = x$ to facilitate further optimizations. Similarly, the sequence $p = \&a; q = p$ is usually simplified to $p = \&a; q = \&a$. While both simplifications are forms of constant propagation, they play rather different roles, and in the GPG framework, are instances of (respectively) *SS* and *TS* variants of *GPU composition* (Section 4.3).

Suppose a GPU γ_1 precedes γ_2 on some control flow path and γ_2 reads a pointer defined by γ_1 (i.e., there is a RaW dependence between γ_1 and γ_2). Then, GPU composition $\gamma_2 \circ^\tau \gamma_1$ eliminates the data dependence and computes a new GPU where τ is type of composition. The resulting GPU γ_3 is a simplified version of the *consumer* GPU γ_2 obtained by using the points-to information in the *producer* GPU γ_1 such that:

- The *indlev* of γ_3 (say $i'|j'$) does not *exceed* that of γ_2 (say $i|j$), i.e. $i' \leq i$ and $j' \leq j$. The two GPUs γ_2 and γ_3 are equivalent in the context of GPU γ_1 (i.e., γ_3 is a simplified form of γ_2).

- The type of GPU composition (denoted τ) is governed by the role of the common node (later called the 'pivot') between γ_1 and γ_2. The forms of GPU composition important here are *TS* and *SS* compositions (Section 4.3). In *TS* composition, the pivot is the target of consumer GPU γ_2 and the source of producer GPU γ_1, whereas in *SS* composition, the pivot is the source of both γ_1 and γ_2.

Both forms of GPU composition are partial functions—either succeeding with a simplified GPU or signalling failure. A comparison of *indlev*s allow us to determine whether a GPU composition is possible; if so, simple arithmetic on *indlev*s allows us to compute the *indlev* of the resulting GPU.

Example 20. For statement sequence $p = \&a; *p = x$, the consumer GPU $\gamma_2 : p \xrightarrow[2]{2|1} x$ (statement 2) is simplified to $\gamma_3 : a \xrightarrow[2]{1|1} x$ by replacing the source p of γ_2 using the producer GPU $\gamma_1 : p \xrightarrow[1]{1|0} a$ (statement 1). GPU γ_3 can be further simplified to one or more points-to edges (i.e. GPUs with *indlev* $1|0$) when GPUs representing the pointees of x (the target of γ_3) become available.

The above example illustrates the following:

- Multiple GPU compositions may be required to reduce the *indlev* of a GPU to convert it to an equivalent GPU with *indlev* $1|0$ (a classical points-to edge).
- *SS* and *TS* variants of GPU composition respectively allow a source or target of a consumer GPU to be resolved into a simpler form.

3.2.2 GPU Reduction

We generalize the operation of GPU composition as follows. If we have a set RGIn_s of GPUs (representing generalized-points-to knowledge from previous statements and obtained from the *reaching GPUs analyses* of Sections 4.6 and 4.7) and a single GPU $\gamma_s \in \delta_s$, representing a GPU for statement s, then *GPU reduction* $\gamma_s \circ \mathsf{RGIn}_s$ constructs a set of one or more GPUs, all of which correspond to statement s. This is considered as the information generated for statement s and is denoted by RGGen_s. It is a union of all such sets created for every GPU $\gamma_s \in \delta_s$ and is semantically equivalent to δ_s in the context of RGIn_s and may beneficially replace δ_s.

GPU reduction plays a vital role in constructing GPGs in two ways. First, inlining the GPG of a callee procedure and performing GPU reduction eliminates procedure calls. Further, GPU reduction helps in removing redundant control flow wherever possible and resolving recursive calls. In particular, a GPU reduction $\gamma_s \circ \mathsf{RGIn}_s$ eliminates the RaW data dependence of γ_s on RGIn_s thereby eliminating the need for a control flow between γ_s and the GPUs in RGIn_s.

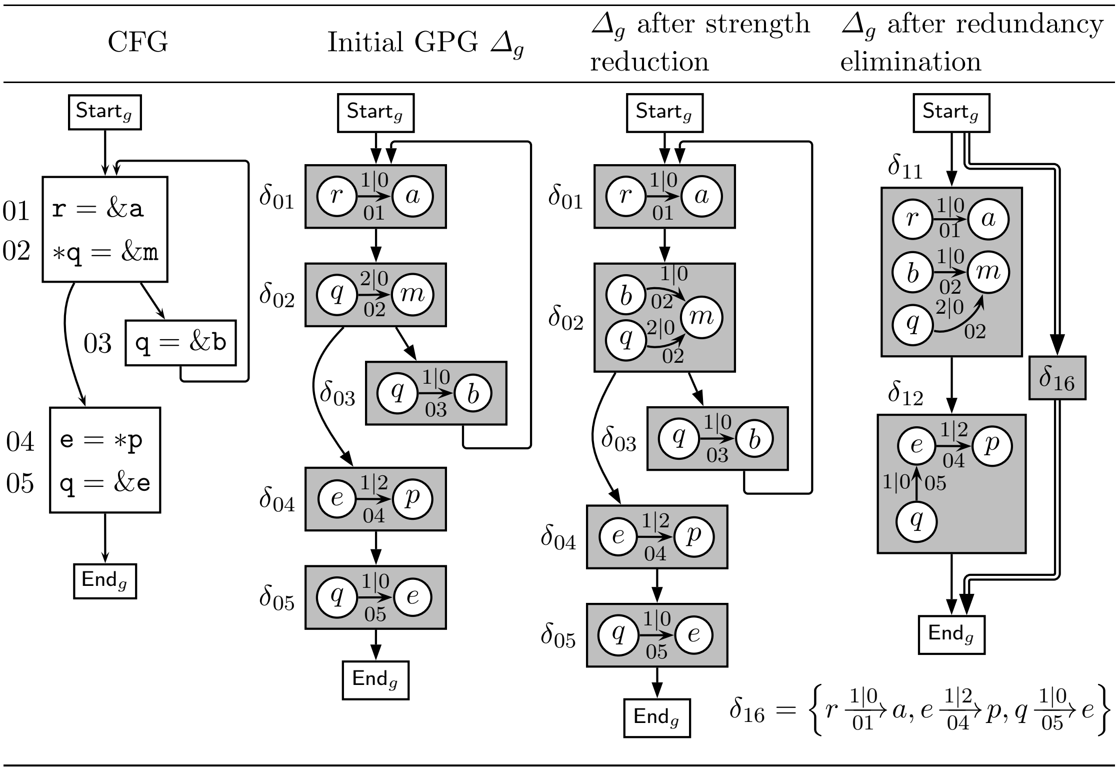

CFG
Initial GPG Δ_g
Δ_g after strength reduction
Δ_g after redundancy elimination
Start$_g$
End$_g$
01 r = &a
02 *q = &m
03 q = &b
04 e = *p
05 q = &e
δ_{01}
δ_{02}
δ_{03}
δ_{04}
δ_{05}
δ_{11}
δ_{12}
δ_{16}
$r \xrightarrow[01]{1|0} a$
$q \xrightarrow[02]{2|0} m$
$b \xrightarrow[02]{1|0} m$
$q \xrightarrow[03]{1|0} b$
$e \xrightarrow[04]{1|2} p$
$q \xrightarrow[05]{1|0} e$
$\delta_{16} = \left\{ r \xrightarrow[01]{1|0} a, e \xrightarrow[04]{1|2} p, q \xrightarrow[05]{1|0} e \right\}$

Figure 3.5: Constructing the GPG for procedure g (see Figure 3.2). The edges with double lines in the last column are not different from control flow edges but have been shown separately because they are not present in the CFG. They represent definition-free paths for the sources of all GPUs that do not appear in GPB δ_{16}. Thus, it is a definition-free path for the sources $(b, 1)$ and $(q, 2)$ of GPUs $b \xrightarrow[02]{1|0} m$ and $q \xrightarrow[02]{2|0} m$.

3.3 An Overview of GPG Construction

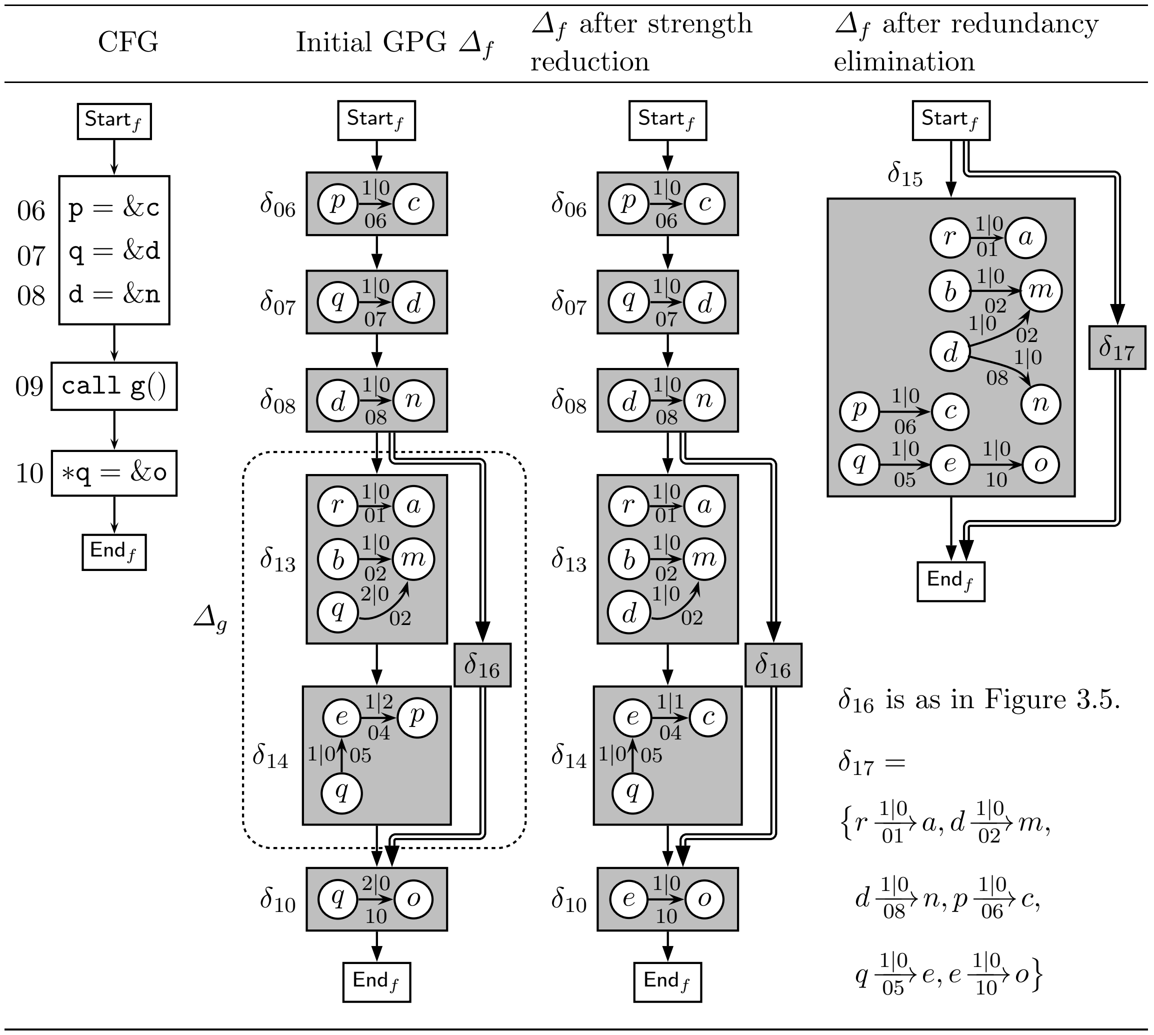
Recall that a GPG of procedure f (denoted Δ_f) is a graph whose nodes are GPBs (denoted δ) abstracting sets of memory updates in terms of GPUs. The edges between GPBs are induced by the control flow of the procedure. Δ_f is constructed using the following steps:

1. *creation* of the initial GPG, and *inlining* optimized GPGs of called procedures[6] within Δ_f,

2. *strength reduction* optimization to simplify the GPUs in Δ_f by performing *reaching GPUs analyses* and transforming GPBs using *GPU reduction*,

3. *redundancy elimination* optimizations to improve the compactness of Δ_f.

[6] This requires a bottom-up traversal of a spanning tree of the call graph starting with its leaf nodes.



δ_{16} is as in Figure 3.5.

$\delta_{17} =$

$\{r \xrightarrow[01]{1|0} a, d \xrightarrow[02]{1|0} m,$

$d \xrightarrow[08]{1|0} n, p \xrightarrow[06]{1|0} c,$

$q \xrightarrow[05]{1|0} e, e \xrightarrow[10]{1|0} o\}$

Figure 3.6: Constructing the GPG for procedure f (see Figures 3.2 and 3.5). GPBs δ_{13}, δ_{14}, and δ_{16} in the GPG are the (renumbered) GPBs representing the inlined optimized GPG of procedure g. The statement labels in the GPUs of these GPBs remain unchanged. Redundancy elimination of Δ_f coalesces all of its GPBs creating a new GPB δ_{15}. GPB δ_{17} is required for modelling definition-free paths. The edges with double lines are control flow edges shown separately because they are introduced to represent definition-free paths.

Steps (2) and (3) are required to construct a compact GPG for efficient analysis. This section illustrates GPG construction intuitively using the motivating example in Figure 3.2. The formal details of these steps are provided in later chapters.

3.3.1 Creating a GPG and Call Inlining

In order to construct a GPG from a CFG, we first map the CFG naively into a GPG by the following transformations:

- Non-pointer assignments and condition tests are removed (treating the latter as non-deterministic control flow). GPG flow edges are induced from those of the control flow graph CFG. This is similar to the construction of sparse evaluation graph (SEG).
- Since local variables are not in the scope of the callers and a GPG represents a summary to be used in the callers, a GPG of a procedure does not retain GPUs containing local variables of the procedure. We use def-use chains of the SSA form to eliminate all local variables from the GPG of a procedure. This is similar to semi-sparse analysis [32]. Later in Section 5.4, we explain that our analysis is actually better than semi-sparse analysis because we minimize the control flow.
- Each pointer assignment labelled s is transliterated to its GPU (denoted γ_s). Figure 3.3 presented the GPUs for basic pointer assignments in C.
- A singleton GPB is created for every pointer assignment in the CFG.
- The procedure calls are replaced by the optimized GPGs of the callees. The resulting GPG may still contain unresolved calls in the case of recursion and function pointers (Sections 6.3 and 6.4).

Example 21. The initial GPG for procedure g of Figure 3.2 is given in Figure 3.5. Each assignment is replaced by its corresponding GPU. The initial GPG for procedure f is shown in Figure 3.6 with the call to procedure g on line 09 replaced by its optimized GPG.

Examples 22 to 24 in the rest of this section explain the analyses and optimizations over $\varDelta_f$ and $\varDelta_g$ at an intuitive level.

3.3.2 Strength Reduction Optimization

This step simplifies all GPUs in GPB δ_s by

- performing reaching GPUs analysis; this performs GPU reduction $\gamma \circ \mathsf{RGIn}_s$ for each $\gamma \in \delta_s$ which computes a set of GPUs that are equivalent to δ_s, and

- replacing δ_s by the resulting GPUs.

In some cases, the reaching GPUs analysis needs to *block* certain GPUs from participating in GPU reduction (as in Example 10 in Section 1.3.3) to ensure the soundness of strength reduction. When this happens, redundancy elimination optimizations need to know if the blocked GPUs in a GPG are useful for potential composition after the GPG is inlined in the callers. These two conflicting requirements (of ignoring some GPUs for strength reduction but remembering them for redundancy elimination) are met by performing two variants of reaching GPUs analysis: first with blocking, and then without blocking. Our motivating example (Figure 3.2) does not have any instance of blocking, hence we provide an overview only of reaching GPUs analysis without blocking.

Strength reduction simplifies each GPB as much as possible given the absence of knowledge of aliasing in the caller (Example 10 in Section 1.3.3). In the process, data dependences are eliminated to the extent possible thereby paving way for redundancy elimination (Section 3.3.3).

In order to reduce the *indlev*s of the GPUs within a GPB, we need to know the GPUs reaching the GPB along all control flow paths from the Start GPB of the procedure. We compute such GPUs through a data flow analysis in the spirit of the classical reaching definitions analysis except that it is not a bit-vector framework because it computes sets of GPUs by processing pointer assignments. This analysis annotates nodes δ_s of the GPG with the sets $\mathsf{RGIn}_s, \mathsf{RGOut}_s, \mathsf{RGGen}_s$ and RGKill_s. It computes RGIn_s as a union of RGOut of the predecessors of s. Then it computes RGGen_s by performing GPU reduction $\gamma \circ \mathsf{RGIn}_s$ for each GPU $\gamma \in \delta_s$. By construction, all resulting GPUs are equivalent to γ and have indirection levels that do not exceed that of γ. Because of the presence of $\gamma \in \delta_s$, some GPUs in RGIn_s are killed and are not included in RGOut_s.

This process may require a fixed-point computation in the presence of loops. Since this step follows inlining of GPGs of callee procedures, procedure calls have already been eliminated and hence this analysis is effectively intraprocedural.

There is one last bit of detail which we allude to here and explain in Section 4.5 where the analysis is presented formally: For the start GPB of the GPG, RGIn is initialized to *boundary definitions*[7] that help track definition-free paths to identify variables that are upwards exposed (i.e. live on entry to the procedure and therefore may have additional

[7] The boundary definitions represent *boundary conditions* [1].

pointees unknown to the current procedure). This is required for making a distinction between strong and weak updates (Sections 1.3.1.2 and 4.5). For the purpose of this overview, we do not show boundary definitions in our example below. They are explained in Example 30 in Section 4.5.

Example 22. We intuitively explain the reaching GPUs analysis for procedure g over its initial GPG (Figure 3.5). The final result is shown later in Figure 4.6. Since we ignore boundary definitions for now, the analysis begins with $\mathsf{RGIn}_{01} = \emptyset$. Further, since we compute the least fixed point, RGOut values are initialized to $\emptyset$ for all statements. The GPU corresponding to the assignment in statement 01 $\gamma_1 : r \xrightarrow[01]{1|0} a$, forms RGOut_{01} and RGIn_{02}. For statement 02, $\mathsf{RGIn}_{02} = \{r \xrightarrow[01]{1|0} a\}$ and $\mathsf{RGGen}_{02} = \{q \xrightarrow[02]{2|0} m\}$. $\mathsf{RGKill}_{02} = \emptyset$ and RGOut_{02} is computed using RGIn_{02} which also forms RGIn_{03} which is $\{r \xrightarrow[01]{1|0} a, q \xrightarrow[02]{2|0} m\}$. For statement 03, $\gamma_3 : q \xrightarrow[03]{1|0} b$ forms RGGen_{03}. In the second iteration of the analysis over the loop, we have $\mathsf{RGIn}_{01} = \mathsf{RGOut}_{03} = \{r \xrightarrow[01]{1|0} a, q \xrightarrow[02]{2|0} m, q \xrightarrow[03]{1|0} b\}$. RGIn_{02} is also the same set. Composing $\gamma_2 : q \xrightarrow[02]{2|0} m$ with $q \xrightarrow[03]{1|0} b$ in RGIn_{02} results in the GPU $b \xrightarrow[02]{1|0} m$. Also, the pointee information of q is available only along one path (identified with the help of boundary definitions that are not shown here) and hence the assignment causes a weak update and the GPU $q \xrightarrow[02]{2|0} m$ is also retained. Thus, RGGen_{02} is now updated and now contains two GPUs: $b \xrightarrow[02]{1|0} m$ and $q \xrightarrow[02]{2|0} m$. This process continues until the least fixed point is reached.

The strength reduction optimization after reaching GPUs analysis gives the GPG shown in the third column of Figure 3.5 (the fourth column represents the GPG after redundancy elimination optimizations and is explained in Section 3.3.3).

3.3.3 Redundancy Elimination Optimizations

This step performs the following optimizations across GPBs to improve the compactness of a GPG.

First, we perform dead GPU elimination to remove *redundant* GPUs in δ_s, i.e. those that are killed along every control flow path from s to the End GPB of the procedure. If a GPU $\gamma \notin \mathsf{RGOut}_{\mathsf{End}}$, then γ is removed from all GPBs. In the process, if a GPB becomes empty, it is eliminated by connecting its predecessors to its successors.

Example 23. In procedure g of Figure 3.5, pointer q is defined in statement 03 but is redefined in statement 05 and hence the GPU $q \xrightarrow[03]{1|0} b$ is eliminated. Hence GPB δ_{03} becomes empty and is removed from the GPG of procedure g ($\varDelta_g$). Note that GPU $q \xrightarrow[02]{2|0} m$ does not define q but its pointee and hence is not killed by statement 05. Thus it is not eliminated from $\varDelta_g$.

For procedure f in Figure 3.6, the GPU $q \xrightarrow[07]{1|0} d$ in δ_{07} is killed by the GPU $q \xrightarrow[05]{1|0} e$ in δ_{14}. Hence the GPU $q \xrightarrow[07]{1|0} d$ is eliminated from the GPB δ_{07} which then becomes empty and is removed from the optimized GPG. Similarly, the GPU $e \xrightarrow[04]{1|1} c$ in GPB δ_{14} is removed because e is redefined by the GPU $e \xrightarrow[10]{1|0} o$ in the GPB δ_{10} (after strength reduction in $\varDelta_f$). However, GPU $d \xrightarrow[08]{1|0} n$ in GPB δ_{08} is not removed even though δ_{13} contains a definition of d expressed by GPU $d \xrightarrow[02]{1|0} m$. This is because δ_{13} also contains GPU $b \xrightarrow[02]{1|0} m$ which defines b, indicating that statement 02 defines two pointers b and d. Hence, d is not defined along all paths and the previous definition of d cannot be killed—giving a weak update.

Finally, we eliminate the redundant control flow in a GPG by performing coalescing analysis (Section 5.4). It partitions the GPBs of a GPG (into *parts*) such that all GPBs in a part are coalesced (i.e., a new GPB is formed by taking a union of the GPUs of all GPBs in the part) and control flow is retained only across the new GPBs representing the parts. Given a GPB δ_s in part π_i, we can add its adjacent GPB δ_t to π_i provided the ***may*** property (Section 3.1) of π_i is preserved. This is possible if the GPUs in π_i and δ_t do not have a data dependence between them.

The data dependences that can be identified using the information available within a procedure (or its callees) are eliminated by strength reduction. However, when a GPU involves an unresolved dereference which requires information from calling contexts, its data dependences with other GPUs is unknown. Coalescing decisions involving such unknown data dependences are resolved using types. The control flow is retained only when type matching indicates the possibility of RaW or WaW data dependence. In other cases the two GPBs are considered for coalescing.

The new GPB after coalescing is numbered with a new label because GPBs are distinguished using labels for maintaining control flow within a GPG. A callee GPG may be inlined at multiple call sites within a procedure. Hence, we renumber the GPB labels after call inlining and coalescing. Note that strength reduction does not create new GPBs; it only creates new (equivalent) GPUs within the same GPB.

Coalescing two GPBs that do not have control flow between them may eliminate a definition-free path for the GPUs in it (see the Example 24 below). We handle this situation as follows: We create an artificial GPB by collecting all GPUs that do not have a definition-free path in the GPG. We add a path from start to end via this GPB. This introduces a definition-free path for all GPUs that do not appear in this GPB.

Note that the GPBs are renumbered after coalescing and during call inlining, however, the statement labels associated with the GPUs in the GPBs are not renumbered. This is because, we need to maintain the association between the GPUs and the corresponding statements in the program (for computing points-to information, see Chapter 7).

Example 24. For procedure g in Figure 3.5, the GPBs δ_{01} and δ_{02} can be coalesced: there is no data dependence between their GPUs because GPU $r \xrightarrow[01]{1|0} a$ in δ_{01} defines r whose type is `float` $**$ whereas the GPUs in δ_{02} read the address of m, pointer b, and pointee of q. The type of latter two is `int` $*$. Since types do not match, there is no data dependence.

The GPUs in δ_{02} and δ_{04} contain a dereference whose data dependence is unknown. We therefore use the type information. Since both q and p have the same types, there is a possibility of RaW data dependence between the GPUs $q \xrightarrow[02]{2|0} m$ and $e \xrightarrow[04]{1|2} p$ (p and q could be aliased in the caller). Thus, we do not coalesce the GPBs δ_{02} and δ_{04}. Also, there is no RaW dependence between the GPUs in the GPBs δ_{04} and δ_{05} and we coalesce them; recall that potential WaR dependence does not matter because of the *may*-property of GPBs (see Example 19 in Section 3.1).

The GPB resulting from coalescing GPBs δ_{01} and δ_{02} is labelled δ_{11}. Similarly, the GPB resulting from coalescing GPBs δ_{04} and δ_{05} is labelled δ_{12}. The loop formed by the back edge $\delta_{02} \rightarrow \delta_{01}$ in the GPG after dead GPU and empty GPB elimination and before coalescing now reduces to a self loop over δ_{11}. Since the GPUs in a GPB do not have a dependence between them, the self loop $\delta_{11} \rightarrow \delta_{11}$ is redundant and is removed.

For procedure f in Figure 3.6, after performing dead GPU elimination, the remaining GPBs in the GPG of procedure f are all coalesced into a single GPB δ_{15}

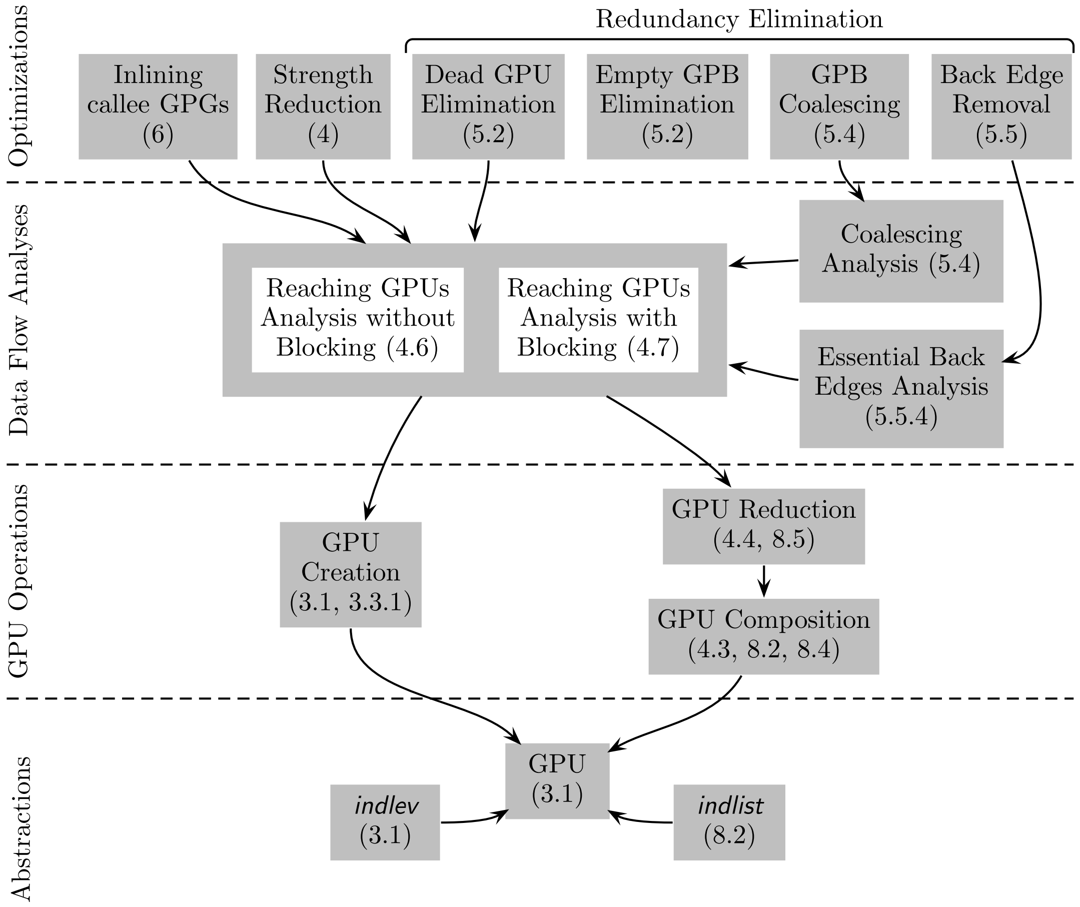

Optimizations
Data Flow Analyses
GPU Operations
Abstractions
Redundancy Elimination
Inlining callee GPGs (6)
Strength Reduction (4)
Dead GPU Elimination (5.2)
Empty GPB Elimination (5.2)
GPB Coalescing (5.4)
Back Edge Removal (5.5)
Coalescing Analysis (5.4)
Reaching GPUs Analysis without Blocking (4.6)
Reaching GPUs Analysis with Blocking (4.7)
Essential Back Edges Analysis (5.5.4)
GPU Creation (3.1, 3.3.1)
GPU Reduction (4.4, 8.5)
GPU Composition (4.3, 8.2, 8.4)
GPU (3.1)
indlev (3.1)
indlist (8.2)

Figure 3.7: The big picture of GPG construction as a fleshed out version of Figure 3.1. The arrows show the dependence between specific instances of optimizations, analyses, operations, and abstractions. The results of the two variants of reaching GPUs analysis are required together. The optimization of empty GPB removal does not depend on any data flow analysis. The labels in parentheses refer to relevant sections.

because there is no data dependence within the GPUs of its GPBs.

As exemplified in Example 23, the sources of the GPUs $b \xrightarrow[02]{1|0} m$ and $q \xrightarrow[02]{2|0} m$ in procedure g are not defined along all paths from Start_g to End_g leading to a weak update. This is modelled by introducing a definition-free path (shown by edges with double lines in the fourth column of Figure 3.5). Thus for procedure g, we have GPB δ_{16} that contains all GPUs of Δ_g that are defined along all paths to create a definition-free path for those that are not. Similarly, for procedure f, we have a definition-free path for the source of GPU $b \xrightarrow[02]{1|0} m$ (as shown in the fourth column of Figure 3.6). The

GPB δ_{17} contains all GPUs of Δ_f except $b \xrightarrow[02]{1|0} m$. GPU $q \xrightarrow[02]{2|0} m$ which has a definition-free path in Δ_g, reduces to $d \xrightarrow[02]{1|0} m$ in Δ_f. Since d is also defined in δ_{08}, it does not have a definition-free path in Δ_f.

3.4 The Big Picture

In this section, we have defined the concepts of GPUs, GPBs, and GPGs as memory transformers and described their semantics. We have also provided an overview of GPG construction in the context of our motivating example.

Figure 3.7 is a fleshed out version of Figure 3.1. It provides the big picture of GPG construction by listing specific abstractions, operations, data flow analyses, and optimizations and shows dependences between them. The optimizations use the results of data flow analyses. The two variants of reaching GPUs analysis are the key analyses; they have been clubbed together because their results are required together. They use the GPU operations which are defined in terms of key abstractions. Empty GPB removal does not require a data flow analysis.

The rest of the thesis defines these abstractions, operations, analyses, and optimizations formally.

Chapter 4

Strength Reduction Optimization

In this chapter, we formalize the basic operations that compute the information required for performing strength reduction optimization of GPBs in a GPG.

4.1 Chapter Overview

Figure 4.1 gives an overview of all the optimizations and also gives a hierarchical relationship between the analyses, GPG operations, and optimizations. For soundness, strength reduction optimization requires some GPU compositions to be postponed by blocking them (Section 4.7.1). However, these GPUs may be useful when the GPGs are inlined in the callers and hence should not be subjected to redundancy elimination optimizations. These conflicting requirements are addressed by performing two variants of reaching GPUs analysis whose dependency is shown in Figure 4.1. The caller dependent accesses of pointers (pointers which are defined in the callers but accessed in callees) are represented by upwards-exposed versions of variables (Section 4.5). The relationship between variables and their corresponding upwards exposed versions is established through GPUs representing boundary definitions. These GPUs are used to model caller dependent GPUs and also enable strong updates. Boundary definitions are used by both the variants of reaching GPUs analysis as shown in Figure 4.1.

Section 4.2 gives an overview of strength reduction optimization. Section 4.3 defines GPU composition as a family of partial operations. Section 4.4 defines GPU reduction. Section 4.5 describes how strong/weak updates are handled in the presence of definition-free paths by modelling caller-defined pointer variables. Section 4.6 provides data flow

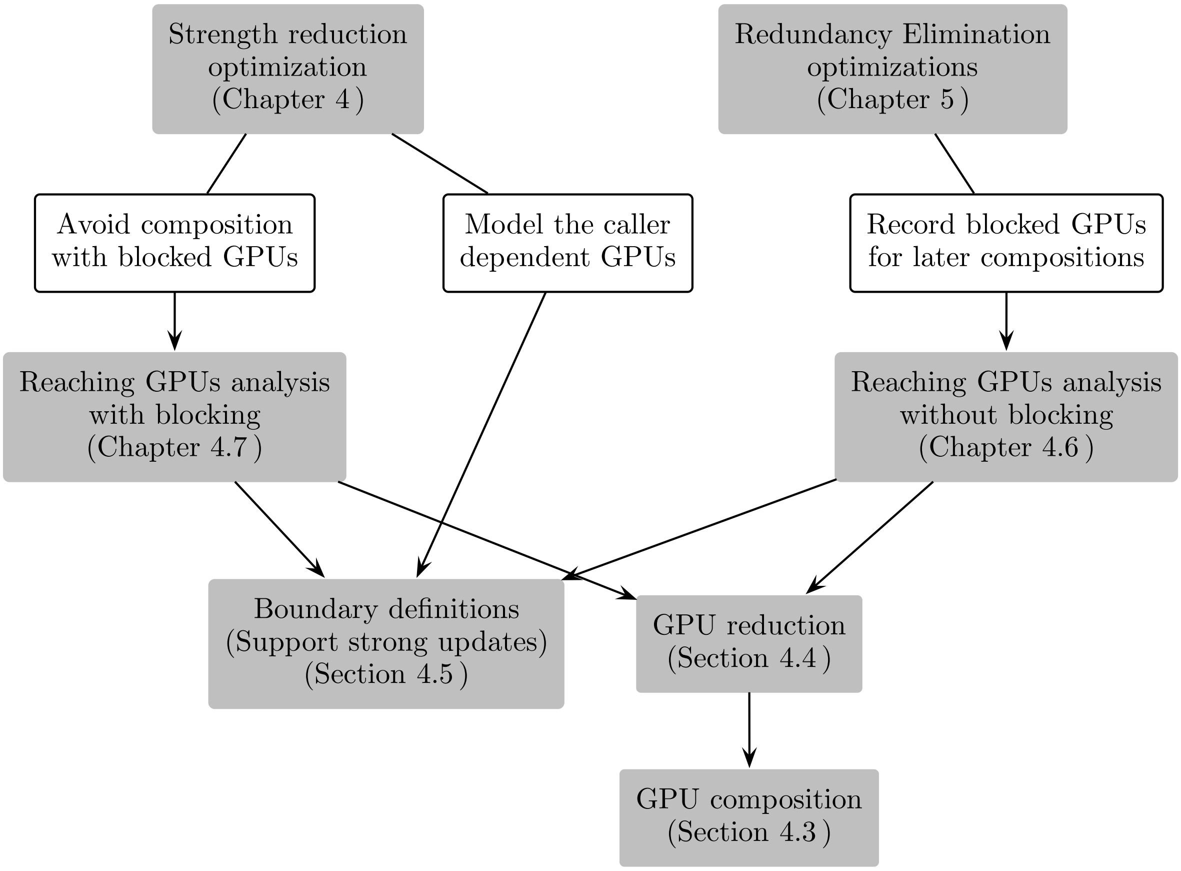

Strength reduction optimization (Chapter 4)
Redundancy Elimination optimizations (Chapter 5)
Avoid composition with blocked GPUs
Model the caller dependent GPUs
Record blocked GPUs for later compositions
Reaching GPUs analysis with blocking (Chapter 4.7)
Reaching GPUs analysis without blocking (Chapter 4.6)
Boundary definitions (Support strong updates) (Section 4.5)
GPU reduction (Section 4.4)
GPU composition (Section 4.3)

Figure 4.1: An overview of dependencies between optimizations, variants of reaching GPUs analysis, and GPU operations. An undirected edge indicates the requirements of optimizations. Strength reduction optimization needs to identify barrier GPUs to avoid composition with some GPUs for soundness, and handle caller dependent GPUs. The blocked GPUs may not be redundant and could be used for later compositions in the callers. Thus, they should not be subjected to redundancy elimination optimizations and should be recorded. A directed edge $X \rightarrow Y$ indicates that X requires Y.

equations for reaching GPUs analysis without blocking while Section 4.7 provides data flow equations for reaching GPUs analysis with blocking. Section 4.8 proves the termination of reaching GPUs analyses formally.

4.2 An Overview of Strength Reduction

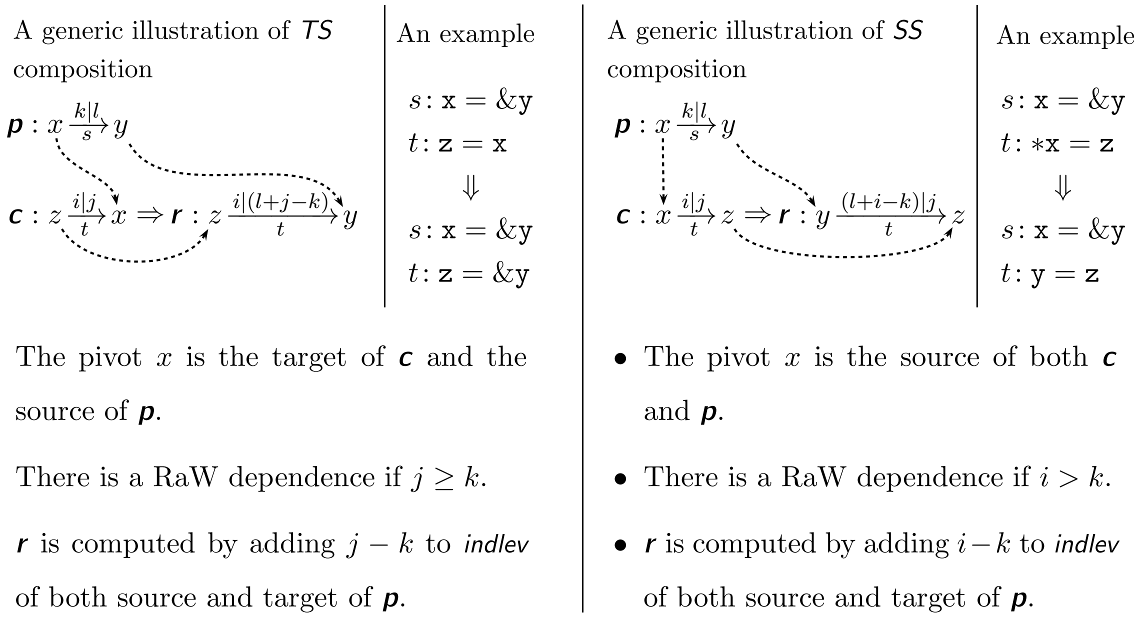
Recall that the construction of a GPG of a procedure begins by transliterating each pointer assignment labelled s in the CFG of the procedure into a GPB δ_s containing the singleton[1] GPU corresponding to the assignment. Then the GPUs are simplified by composing them with other GPUs. This simplification progressively converts a GPU to a classical points-to edge as noted in Sections 1.3.3 and 3.3.2. Some simplifications can be done immediately while others are blocked awaiting knowledge of aliasing in the callers and so are postponed. They are reconsidered in the calling context after the GPG is inlined as a procedure summary in its callers. The strength reduction optimization then replaces every GPU $\gamma \in \delta_s$ with its simplified version.

Based on the knowledge of a (*producer*) GPU $\boldsymbol{p}$, a *consumer* GPU $\boldsymbol{c}$ is simplified through an operation called *GPU composition* denoted $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ (where τ is SS or TS). A consumer GPU may require multiple GPU compositions to reduce it to an equivalent GPU with *indlev* $1|0$ (a classical points-to edge). This is achieved by *GPU reduction* $\boldsymbol{c} \circ R$ which involves a series of GPU compositions with appropriate producer GPUs in R in order to simplify the consumer GPU $\boldsymbol{c}$ maximally. The set R of GPUs used for simplification provides a context for $\boldsymbol{c}$ and represents generalized-points-to knowledge from previous statements. It is obtained by performing a data flow analysis called the *reaching GPUs analysis* which computes the sets RGIn_s, RGOut_s, RGGen_s, and RGKill_s. The set RGGen_s is semantically equivalent to δ_s in the context of RGIn_s and may beneficially replace δ_s. We have two variants of reaching GPUs analysis for the reasons described below.

In some cases, the location read by $\boldsymbol{c}$ could be different from the location defined by $\boldsymbol{p}$ due to the presence of a GPU $\boldsymbol{b}$ (called a *barrier*) corresponding to an intervening assignment. The GPU $\boldsymbol{p}$ may be updated by the GPU $\boldsymbol{b}$ depending on the aliases in the calling context (Section 1.3.3). This may alter the data dependence between $\boldsymbol{c}$ and $\boldsymbol{p}$. It could happen because the *indlev* of the source of $\boldsymbol{p}$ or $\boldsymbol{b}$ is greater than 1 indicating that the pointer being defined by this GPU is still not known. In such a situation (characterized formally in Section 4.7.1), replacing δ_s by RGGen_s during strength reduction may be unsound. To ensure soundness, we need to *postpone* the composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ explicitly by eliminating those GPUs from R that are blocked by a barrier.[2] We do this by performing a variant of reaching GPUs analysis called the *reaching GPUs analysis with blocking* that removes GPUs blocked by a barrier to compute reaching GPUs that are not blocked (Sec-

[1] GPU reduction and GPB coalescing may include multiple GPUs in a GPB. Hence for generality, we treat a GPB as a set of possibly multiple GPUs.

[2] Formally the term 'barrier' applies to a GPU, but we abuse it and refer to its associated statement as a barrier too.


A generic illustration of *TS* composition
$\boldsymbol{p} : x \xrightarrow[s]{k|l} y$
$\boldsymbol{c} : z \xrightarrow[t]{i|j} x \Rightarrow \boldsymbol{r} : z \xrightarrow[t]{i|(l+j-k)} y$
An example
s: x = &y
t: z = x
⇓
s: x = &y
t: z = &y
A generic illustration of *SS* composition
$\boldsymbol{p} : x \xrightarrow[s]{k|l} y$
$\boldsymbol{c} : x \xrightarrow[t]{i|j} z \Rightarrow \boldsymbol{r} : y \xrightarrow[t]{(l+i-k)|j} z$
An example
s: x = &y
t: ∗x = z
⇓
s: x = &y
t: y = z

- The pivot x is the target of ***c*** and the source of ***p***.
- There is a RaW dependence if $j \geq k$.
- ***r*** is computed by adding $j - k$ to *indlev* of both source and target of ***p***.

- The pivot x is the source of both ***c*** and ***p***.
- There is a RaW dependence if $i > k$.
- ***r*** is computed by adding $i - k$ to *indlev* of both source and target of ***p***.

Figure 4.2: Composing a consumer GPU ***c*** with a producer GPU ***p*** to compute a new GPU ***r*** which is equivalent to ***c*** in the context of ***p***. Both *SS* and *TS* compositions exploit a RaW dependence of statement at t on the statement at s because the pointer defined in ***p*** is used to simplify a pointer used in ***c***. The other two possible compositions *TT* and *ST* are less useful.

tion 4.7). We distinguish the two variants by using the phrase *reaching GPUs analysis without blocking* for the earlier reaching GPUs analysis. For strength reduction, it is sufficient to perform reaching GPUs analysis with blocking. However, redundancy elimination optimizations need to know whether the blocked GPUs in a GPG are useful for potential composition after the GPG is inlined in the callers. These two conflicting requirements force us to perform both the variants of reaching GPUs analysis: with blocking, and without blocking.

4.3 GPU Composition

We define GPU composition as a family of partial operations. These operations simplify a consumer GPU ***c*** using a producer GPU ***p*** and compute a semantically equivalent GPU.

4.3.1 The Intuition Behind GPU Composition

The composition of a consumer GPU $\boldsymbol{c}$ and a producer GPU $\boldsymbol{p}$, denoted $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$, computes a resulting GPU $\boldsymbol{r}$ by simplifying $\boldsymbol{c}$ using $\boldsymbol{p}$. This is possible when $\boldsymbol{c}$ has a RaW dependence on $\boldsymbol{p}$ through a common variable called the *pivot* of composition. This requires the pivot to be the source of $\boldsymbol{p}$ but it could be the source or the target of $\boldsymbol{c}$.

We name the compositions as *TS* or *SS* where the first letter indicates the role of the pivot in $\boldsymbol{c}$ and second letter indicates its role in $\boldsymbol{p}$. If the pivot is the target of $\boldsymbol{c}$ and the source of $\boldsymbol{p}$, the composition is called a *TS* composition. If the pivot is the source of both $\boldsymbol{c}$ and $\boldsymbol{p}$, the composition is called an *SS* composition. We remark for completeness that there are two further GPU-composition operations which can be applied when the pivot is the target of $\boldsymbol{p}$. These are called *ST* and *TT* compositions which correspond to anti- and output dependence. They are optional and we do not use them here. This is because *TS* and *SS* compositions are sufficient to convert a GPU to a classical points-to edge (i.e., a GPU with *indlev* “1|0”).

Figure 4.2 illustrates *TS* and *SS* compositions. For *TS* composition, consider GPUs $\boldsymbol{c} : z \xrightarrow[t]{i|j} x$ and $\boldsymbol{p} : x \xrightarrow[s]{k|l} y$ with a pivot x which is the target of $\boldsymbol{c}$ and the source of $\boldsymbol{p}$. The goal of GPU composition is to join the source z of $\boldsymbol{c}$ and the target y of $\boldsymbol{p}$ by using the pivot x as a bridge. This requires the *indlev*s of x to be made the same in the two GPUs. For example, if $j \geq k$ (other cases are explained later in the section), this can be achieved by adding $j - k$ to the *indlev*s of the source and target of $\boldsymbol{p}$ to view the base GPU $\boldsymbol{p}$ in its derived form as $x \xrightarrow{j|(l+j-k)} y$. This balances the *indlev*s of x in the two GPUs allowing us to create a simplified GPU $\boldsymbol{r} : z \xrightarrow{i|(l+j-k)} y$. (Given a GPU $x \xrightarrow[s]{i|j} y$, we can create a GPU $x \xrightarrow[s]{(i+1)|(j+1)} y$ based on the type restrictions on the *indlev*s of x and y.)

Although this computes the transitive effect of GPUs, in general, it cannot be modelled using multiplication of matrices representing graphs as explained in Section 4.3.3.

4.3.2 Defining GPU Composition

Before we define the GPU composition formally, we need to establish the properties of *validity* and *desirability* that allow us to characterize meaningful GPU compositions.

(a) A composition $\boldsymbol{r} = \boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ is *valid* only if $\boldsymbol{c}$ reads a location defined by $\boldsymbol{p}$ and this read/write happens through the pivot of the composition.

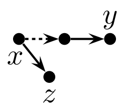
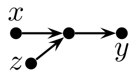
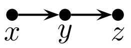
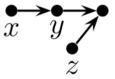
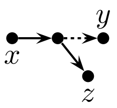
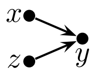
Possible *SS* Compositions			Possible *TS* Compositions		
Statement sequence	Memory graph after the stmt. sequence	GPUs	Statement sequence	Memory graph after the stmt. sequence	GPUs
$i < k$			$j < k$		
$\boxed{\text{Ex. } ss1}$ ∗x = &y x = &z		***p***: $x \xrightarrow{2\|0} y$ ***c***: $x \xrightarrow{1\|0} z$ (*invalid*)	$\boxed{\text{Ex. } ts1}$ ∗x = &y z = x		***p***: $x \xrightarrow{2\|0} y$ ***c***: $z \xrightarrow{1\|1} x$ (*invalid*)
$i > k$			$j > k$		
$\boxed{\text{Ex. } ss2}$ x = &y ∗x = &z		***p***: $x \xrightarrow{1\|0} y$ ***c***: $x \xrightarrow{2\|0} z$ ***r***: $y \xrightarrow{1\|0} z$	$\boxed{\text{Ex. } ts2}$ x = &y z = ∗x		***p***: $x \xrightarrow{1\|0} y$ ***c***: $z \xrightarrow{1\|2} x$ ***r***: $z \xrightarrow{1\|1} y$
$i = k$			$j = k$		
$\boxed{\text{Ex. } ss3}$ ∗x = &y ∗x = &z		***p***: $x \xrightarrow{2\|0} y$ ***c***: $x \xrightarrow{2\|0} z$ (*invalid*)	$\boxed{\text{Ex. } ts3}$ x = &y z = x		***p***: $x \xrightarrow{1\|0} y$ ***c***: $z \xrightarrow{1\|1} x$ ***r***: $z \xrightarrow{1\|0} y$

Figure 4.3: Illustrating the *validity* of *SS* and *TS* compositions based on the *indlev*s of pivot (x in these examples) in the consumer GPU ***c*** and producer GPU ***p***.

(b) A composition $\boldsymbol{r} = \boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ is *desirable* only if the *indlev* of ***r*** does not exceed the *indlev* of ***c*** (i.e., ***r*** is closer to classical points-to edge in terms of *indlev*s than ***c***).

We say that a GPU composition is *admissible* if and only if it is *valid* and *desirable*.

Validity requires the *indlev* of the pivot in ***c*** to be greater than the *indlev* of pivot in ***p***. For the generic *indlev*s used in Figure 4.2, this requirement for *validity* translates to the following constraints:

$$j \geq k \qquad (\textit{TS} \text{ composition}) \tag{4.1}$$

$$i > k \qquad (\textit{SS} \text{ composition}) \tag{4.2}$$

Observe that *SS* composition condition (4.2) prohibits equality unlike the condition for *TS* composition (4.1). This is because of the fact that *SS* composition involves the source nodes of both the GPUs and when $i = k$, ***c*** overwrites the location written by ***p***; for a location written by ***p*** to be read by ***c*** in its source, i must be strictly greater than k.

$$\left(z \xrightarrow[t]{i|j} x\right) \circ^{\text{ts}} \left(v \xrightarrow[s]{k|l} y\right) := \begin{cases} z \xrightarrow[t]{i|(l+j-k)} y & (v = x) \wedge (l \leq k \leq j) \\ \text{fail} & \text{otherwise} \end{cases}$$

$$\left(x \xrightarrow[t]{i|j} z\right) \circ^{\text{ss}} \left(v \xrightarrow[s]{k|l} y\right) := \begin{cases} y \xrightarrow[t]{(l+i-k)|j} z & (v = x) \wedge (l \leq k < i) \\ \text{fail} & \text{otherwise} \end{cases}$$

Definition 3: *GPU Composition* $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$

Example 25. The following (attempted) compositions in Figure 4.3 are *invalid* because ***c*** does not read a location defined by ***p***.

- In example *ss1* (*SS* composition), $k = 2$ and $i = 1$ violating Constraint (4.2). GPU ***c*** redefines x instead of reading a location defined by ***p***.
- In example *ss3* (*SS* composition), $k = i = 2$ violating Constraint (4.2). GPU ***c*** redefines pointee of x (i.e., $*x$) instead of reading a location defined by ***p***.
- In example *ts1* (*TS* composition), $k = 2$ and $j = 1$ violating Constraint (4.1). GPU ***c*** reads x instead of reading pointee of x (i.e, $*x$) defined by ***p***. In other words, there is no data dependence between ***c*** and ***p*** which is evident from the fact that the order of the statements can be changed and yet the meaning of the program remains same.

Following compositions in Figure 4.3 are *valid* because ***c*** reads a location defined by ***p***.

- In example *ss2* (*SS* composition), $k = 1$ and $i = 2$ satisfies Constraint (4.2).
- In example *ts2* (*TS* composition), $k = 1$ and $j = 2$ satisfies Constraint (4.1).
- In example *ts3* (*TS* composition), $k = 1$ and $j = 1$ satisfies Constraint (4.1).

The *desirability* of GPU composition characterizes progress in conversion of GPUs into classical points-to edges by ensuring that the *indlev* of the new source and the new target in ***r*** does not exceed the corresponding *indlev* in the consumer GPU ***c***. This requires the *indlev* in the simplified GPU ***r*** and the consumer GPU ***c*** to satisfy the following constraints. In each constraint, the first term in the conjunct compares the *indlev*s of the sources of $\boldsymbol{c}$ and $\boldsymbol{r}$ while the second term compares those of the targets (see Figure 4.2):

$$(i \le i) \wedge (l + j - k \le j) \quad \text{or equivalently} \quad l \le k \qquad (\mathit{TS}\ \text{composition}) \tag{4.3}$$

$$(l + i - k \le i) \wedge (j \le j) \quad \text{or equivalently} \quad l \le k \qquad (\mathit{SS}\ \text{composition}) \tag{4.4}$$

Example 26. Consider the statement sequence $x = *y; z = x$. A *TS* composition of the corresponding GPUs $\boldsymbol{p} : x \xrightarrow{1|2} y$ and $\boldsymbol{c} : z \xrightarrow{1|1} x$ is *valid* because $j = k = 1$ satisfying Constraint 4.1. However, if we perform this composition, we get $\boldsymbol{r} : z \xrightarrow{1|2} y$. Intuitively, this GPU is not useful for computing a points-to edge because the *indlev* of $\boldsymbol{r}$ is "1|2" which is greater than the *indlev* of $\boldsymbol{c}$ which is "1|1". Formally, this composition is flagged *undesirable* because $l = 2$ which is greater than $k = 1$ violating Constraint 4.3.

We take a conjunction of the constraints of *validity* (4.1 and 4.2) and *desirability* (4.3 and 4.4) to characterize *admissible* GPU compositions.

$$l \le k \le j \qquad (\mathit{TS}\ \text{composition}) \tag{4.5}$$

$$l \le k < i \qquad (\mathit{SS}\ \text{composition}) \tag{4.6}$$

Note that an *undesirable* GPU composition is *valid* but *inadmissible*. It will eventually become *desirable* after the producer GPU is simplified further through strength reduction optimization after the GPG is inlined in a caller's GPG.

Definition 3 defines GPU composition formally. It computes a simplified GPU $\boldsymbol{r} = \boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ by balancing the *indlev* of the pivot in both the GPUs provided the composition (*TS* or *SS*) is *admissible*. Otherwise it fails—being a partial operation. Note that *TS* and *SS* compositions are mutually exclusive for a given pair of $\boldsymbol{c}$ and $\boldsymbol{p}$ because a variable cannot occur both in the RHS and the LHS of a pointer assignment in the case of pointers to scalars. Since our language is modelled on C, GPUs for statements such as $*x = x$ or $x = *x$ are prohibited by typing rules; GPUs for statements such as $*x = *x$ are ignored as inconsequential. Further, we assume as allowed by C-standard *undefined behavior* that the programmer has not abused type-casting to simulate such prohibited statements. Chapter 8 considers a richer situation with structs and unions where we can

```
Input: $\boldsymbol{c}$              // The consumer GPU to be simplified.
       $R$              // The context (set of GPUs) in which $\boldsymbol{c}$
                        //  is to be simplified.
Output: Red             // The set of simplified GPUs equivalent to $\boldsymbol{c}$.
GPU_reduction ($\boldsymbol{c}$, $R$)
{  if ($R = \emptyset \wedge \boldsymbol{c} = \cdot \xrightarrow[.]{1|0} \cdot$)  // $\boldsymbol{c}$ is a points-to edge
   {  Red = $\{\boldsymbol{c}\}$
      $W = \emptyset$
   }
   else
   {  Red = $\emptyset$
      $W = \{\boldsymbol{c}\}$
   }
   while ($W \neq \emptyset$)
   {  extract $w$ from $W$
      $composed$ = false
      for each $\gamma \in R$
      {  if ($r = w \circ^{\text{ts}} \gamma$) succeeds
         {  $W = W \cup \{r\}$
            $composed$ = true
         }
         else if ($r = w \circ^{\text{ss}} \gamma$) succeeds
         {  $W = W \cup \{r\}$
            $composed$ = true
         }
      }
      if ($\neg$ $composed$ )
         Red = Red $\cup \{w\}$
   }
   return Red
}
```

Definition 4: *GPU Reduction* $\boldsymbol{c} \circ R$

have an assignment $x \to n = x$ which might have both *TS* and *SS* compositions with a GPU $\boldsymbol{p}$ that defines x and $\boldsymbol{c}$ reads and defines some pointee of x simultaneously.

4.3.3 Modelling GPU Composition as Matrix Multiplication?

GPU composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ computes transitive effects of edges $\boldsymbol{c}$ and $\boldsymbol{p}$. This is somewhat similar to the reachability computed in a graph: If there are edges $x \rightarrow y$ and $y \rightarrow z$ representing the facts that y is reachable from x and z is reachable from y, then it follows that z is reachable from x and an edge $x \rightarrow z$ can be created. If the graph is represented by an adjacency matrix A in which the element (x, y) represents reachability of y from x, matrix multiplication $A \times A$ can be used to compute the transitive effect.

It is difficult to model GPU composition as matrix multiplication because of the following reasons:

- GPUs have labels as pairs of numbers representing indirection levels and also statement labels. Hence we will need to device an appropriate operator and the usual multiplication would not work.

- GPU composition has some additional constraints over reachability because of *validity* and *desirability*; *invalid* and *undesirable* compositions are not performed. These restrictions are difficult to model in matrix multiplication.

- Transitive reachability considers only the edges of the kind $x \rightarrow y$ and $y \rightarrow z$; i.e. the pivot should be the target of the first edge and the source of the second edge. GPU composition considers pivot as both source as well as target in $\boldsymbol{c}$ and source in $\boldsymbol{p}$ and hence considers two compositions (*TS* and *SS*). For example, we compose $\boldsymbol{p} : x \xrightarrow[s]{1|0} z$ and $\boldsymbol{c} : x \xrightarrow[t]{2|0} y$ in an *SS* composition to create a new GPU $\boldsymbol{r} : z \xrightarrow[t]{1|0} y$. Transitive reachability computed using matrix multiplication considers only *TS* composition.

Thus, matrix multiplication does not model GPU composition naturally.

4.4 GPU Reduction

GPU reduction $\boldsymbol{c} \circ R$ uses the GPUs in R to compute a set of GPUs Red whose *indlev*s do not exceed that of $\boldsymbol{c}$. The result of GPU reduction $\boldsymbol{c} \circ R$ must ensure the semantic equivalence of Red with $\boldsymbol{c}$ in the context of R. The set R is computed using reaching GPUs analysis without blocking (Section 4.6). In some cases, we need to restrict R using the reaching GPUs analysis with blocking (Section 4.7) to ensure this semantic equivalence.

For $\boldsymbol{c} \circ R$, the *indlev* of $\boldsymbol{c}$ is reduced progressively using the GPUs from R through a series of *admissible* GPU compositions. For example, a GPU $x \xrightarrow{1|2} y$ requires two *TS* compositions to transform it into a classical points-to edge: first one for identifying the pointees of y and second one for identifying the pointees of pointees of y. Similarly, for a GPU $x \xrightarrow{2|1} y$, an *SS* composition is required to identify the pointees of x which are being defined and a *TS* composition is required to identify the pointees of y whose addresses are being assigned. Thus, the result of GPU reduction $\boldsymbol{c} \circ R$ is a fixed-point of cascaded GPU compositions in the context of R.

4.4.1 Defining GPU Reduction $\boldsymbol{c} \circ R$

Definition 4 gives the algorithm for GPU reduction. The worklist W is initialized to $\{\boldsymbol{c}\}$. A reduced GPU is added to W for further GPU compositions. When a GPU w cannot be reduced any further, the flag *composed* remains **false** and w is added to Red (lines 23 and 24 of Definition 4). When the input to GPU reduction R is $\emptyset$ and the GPU $\boldsymbol{c}$ is already in the reduced form (GPU with *indlev* "1|0"), $\boldsymbol{c}$ is added to Red (lines 2, 3, and 4 of Definition 4) which then forms the output of GPU reduction. This algorithm assumes that the graph induced by the GPUs in R is acyclic. This holds for pointers to scalars. However, in the presence of structures, the graph induced by GPUs in R may contain cycles via fields of structures; Section 8.5 extends the algorithm to handle cycles.

Example 27. Consider the statements in this example. For $\boldsymbol{c} : x \xrightarrow[23]{1|2} y$, the set of GPUs reaching statement 23 is $R = \{y \xrightarrow[21]{1|0} a, a \xrightarrow[22]{1|0} b\}$. The reduction $\boldsymbol{c} \circ R$ involves two consecutive *TS* compositions. The first composition involves $y \xrightarrow[21]{1|0} a$ as $\boldsymbol{p}$, resulting in $\boldsymbol{r} = x \xrightarrow[23]{1|1} a$ which is added to the worklist. In the second iteration of the **while** loop on line 10 of Definition 4, the reduced GPU $x \xrightarrow[23]{1|1} a$ in the previous iteration now becomes the consumer GPU. It is composed with $a \xrightarrow[22]{1|0} b$ which results in a reduced GPU $x \xrightarrow[23]{1|0} b$. This GPU is added to the worklist. However, since it cannot be reduced further as it is already in the classical points-to form, the loop terminates. The flag *composed* remains **false** for the final GPU $x \xrightarrow[23]{1|0} b$ because no further composition is possible and Red $= \{x \xrightarrow[23]{1|0} b\}$.

```
21   y = &a;
22   a = &b;
23   x = *y;
```

The termination of GPU reduction is guaranteed by the following reasons:

- A GPU w extracted from the worklist will never be added to it again. If there is no reduction, then w is added to Red directly. This is ensured by setting the flag *composed* appropriately.

- Reduction of *indlev* of source and target of a GPU w is performed independently, hence there is no oscillation across iterations of fixed-point computation.

- The process terminates only when the GPUs in Red are either in their simplified form or no more GPUs are available in R for further compositions.

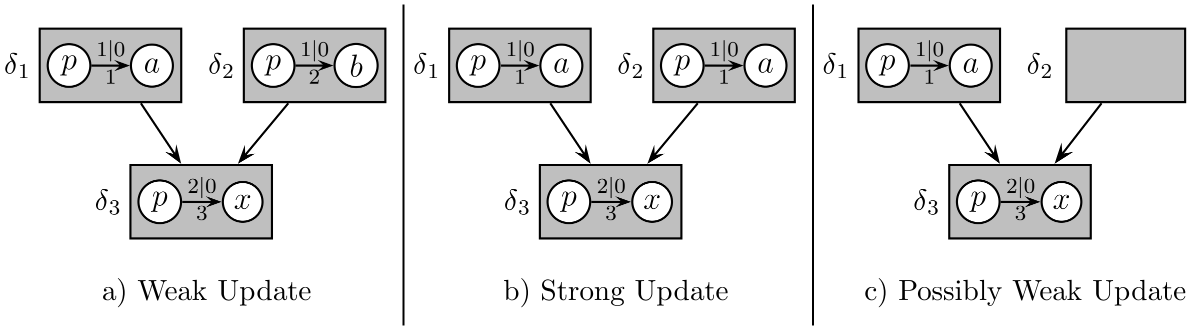
- The order in which a GPU γ is selected from R for composition with w does not matter because of the following properties of R that are established by the reaching GPUs analysis with and without blocking (Sections 4.6 and 4.7).

  Consider two GPUs γ_1 and γ_2 in R. Then γ_1 and γ_2 cannot compose with each other: If the composition $\gamma_2 \circ \gamma_1$ were possible, it would have been performed during the reaching GPUs analysis (Section 4.6) and γ_2 would not exist in R because it would be replaced by the result of the composition. Similarly if the composition $\gamma_1 \circ \gamma_2$ were possible, γ_1 would not exist in R. Hence we examine the possible reasons of existence of both γ_1 and γ_2 in R and explain why the order of performing the compositions $w \circ \gamma_1$ and $w \circ \gamma_2$ does not matter.

  (a) There is no data dependence between γ_1 and γ_2 because there is no pivot between them or one does not follow the other on any control flow path. Hence a composition between them is ruled out. In this case, the order between $w \circ \gamma_1$ and $w \circ \gamma_2$ is irrelevant.

  (b) There is data dependence between γ_1 and γ_2 potentially enabling a composition. Without any loss of generality, consider the composition $\gamma_2 \circ \gamma_1$. Then there are two possibilities that may have prohibited the composition:

    (i) $\gamma_2 \circ \gamma_1$ is *inadmissible* because it is *undesirable*. Then, $w \circ \gamma_1$ also is *undesirable* because the *desirability* constraint is based solely on the *indlev* of γ_1 (Constraints 4.3 and 4.4). Thus w may compose only with γ_2 and the issue of an order between $w \circ \gamma_1$ and $w \circ \gamma_2$ does not arise.



a) Weak Update

b) Strong Update

c) Possibly Weak Update

Figure 4.4: An example demonstrating the need for identifying a definition-free path to make a distinction between strong and weak update.

(ii) $\gamma_2 \circ \gamma_1$ is *admissible* but has been postponed because of a barrier (introduced in Section 1.3.3 and explained later in Section 4.7) between γ_2 and γ_1. In this case, the barrier also prohibits a composition of w with γ_1 and it can compose only with γ_2. Thus the issue of an order between $w \circ \gamma_1$ and $w \circ \gamma_2$ does not arise.

4.4.2 A Comparison with Dynamic Transitive Closure

It is tempting to compare GPU reduction $\boldsymbol{c} \circ R$ with dynamic transitive closure [15, 16]: a series of GPU compositions are performed until the GPU $\boldsymbol{c}$ cannot be simplified any further. However, the analogy stops at this abstract level. Apart from the reasons mentioned in Section 4.3.3, the following differences make it difficult to model GPU reduction in terms of dynamic transitive closure:

- GPU reduction does not compute unrestricted transitive effects. Dynamic transitive closure computes unrestricted transitive effects.

- We do not compute closure. Strength reduction optimization replaces the result of GPU reduction with the GPUs in the GPB. Dynamic transitive closure implies retaining all GPUs including the GPUs computed in the intermediate steps.

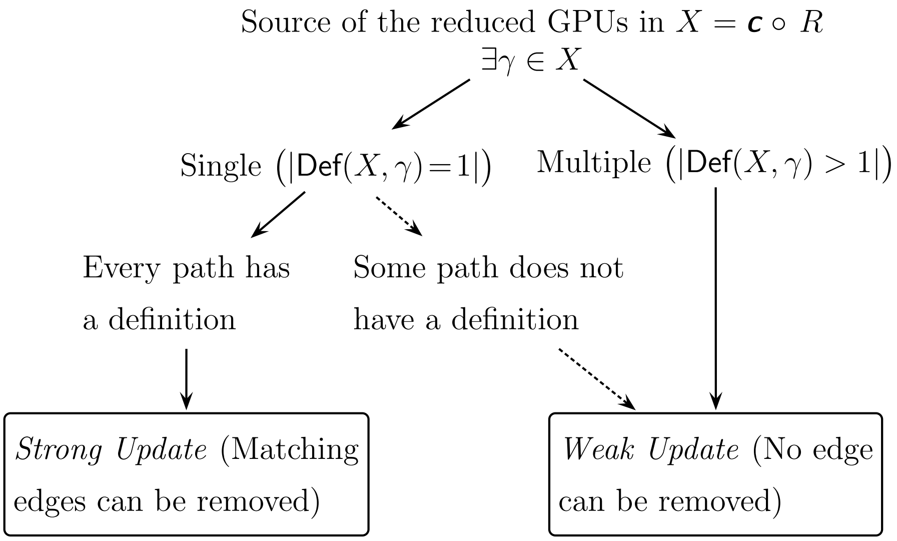

Source of the reduced GPUs in $X = \boldsymbol{c} \circ R$
$\exists \gamma \in X$
Single $\big(|\mathsf{Def}(X,\gamma)=1|\big)$
Multiple $\big(|\mathsf{Def}(X,\gamma) > 1|\big)$
Every path has a definition
Some path does not have a definition
Strong Update (Matching edges can be removed)
Weak Update (No edge can be removed)

Figure 4.5: Criteria for strong and weak updates in $\varDelta$. Our formulations eliminate the dashed edge simplifying strong updates. $\mathsf{Def}(X,\gamma)$ is defined in Definition 4.

4.5 Modelling Caller-Defined Pointer Variables

This section describes the need for identifying definition-free paths for strong updates. This is required to handle statements such as $*x = p$ where one of the definition of x reaching the statement appears in a caller. This section describes how the definitions in the callers (which are not available during GPG construction) are abstractly captured and strong updates are performed precisely.

Recall from Section 1.3.1.2 that, in abstract memory, we may be uncertain as to which of several locations a variable points to.

Example 28. In Figure 4.4(a), multiple pointees of p reach GPB δ_3. GPU reduction for the GPU in δ_3 returns a set of GPUs $\{a \xrightarrow[3]{1|0} x, b \xrightarrow[3]{1|0} x\}$ which define multiple pointers leading to a weak update. In this case, we do not overwrite the pointees of a and b, but merely add $\&x$ to the possible values they can contain.

In Figure 4.4(b), p has a single pointee reaching GPB δ_3. The result of GPU reduction is $\{a \xrightarrow[3]{1|0} x\}$ indicating that there is only one possible abstract location defined by GPU $p \xrightarrow[3]{2|0} x$. In this case we may, in general, *replace* the contents of location a.

This is a strong update. However, this is necessary but not sufficient for a strong update because the pointer may not be defined along all paths—there may be a path along which the pointer may not be defined within the procedure but may be defined in a caller (Figure 4.4(c)). In the presence of such a definition-free path in a procedure, even if we find a single pointee of p in the procedure, we cannot guarantee that a single abstract location is being defined. This makes it difficult to distinguish between strong and weak updates.

The role of definition-free path in making a distinction between strong and weak update is summarized in Figure 4.5. Also, the effect of definition-free paths has to be taken into account during strength reduction optimization: if γ_1 is simplified to γ_2, γ_2 can replace γ_1 provided there is no definition-free path reaching γ_1; otherwise γ_1 should also be included with γ_2 to allow the composition of γ_1 with the producer GPUs in a caller.

Example 29. Figure 3.5 shows the set of GPUs corresponding to statement 02 (δ_{02} in the GPG after strength reduction) of procedure g of Figures 3.2 and 3.5. There is a definition-free path for q meaning that δ_{11} in the optimized $\varDelta_g$ must include GPU $q \xrightarrow[02]{2|0} m$ along with its reduced GPU $b \xrightarrow[02]{1|0} m$.

We identify definition-free paths by introducing *boundary definitions* (explained below) which help us to preserve definition-free paths that may be eliminated by coalescing.

The boundary definitions are introduced for global variables and formal parameters because they could be read in a procedure before being defined. They are symbolic in that they are not introduced in the GPG of a procedure but are included in RGIn of the Start GPB during reaching GPUs analysis. They are of the form $x \xrightarrow[00]{\ell|\ell} x'$ where x' is a symbolic representation of the initial value of x at the start of the procedure and ℓ ranges from 1 to the maximum depth of the indirection level which depends on the type of x, and 00 is the label of the Start GPB. For type (int $**$), ℓ ranges from 1 to 2. Variable version x' is called the *upwards exposed* [51] version of x. This is similar to Hoare-logic style specifications in which postconditions use (immutable) *auxiliary variables* x' to be able to talk about the original value of variable x (which may have since changed). Our upwards-exposed versions serve a similar purpose, so that logically on entry to each procedure the statement $x = x'$ provides a definition of x.

A reduced GPU $x \xrightarrow[s]{i|j} y$ along any path kills the boundary definition $x \xrightarrow[00]{i|i} x'$ on that path indicating that $(i-1)^{th}$ pointees of x are redefined. Including boundary definitions

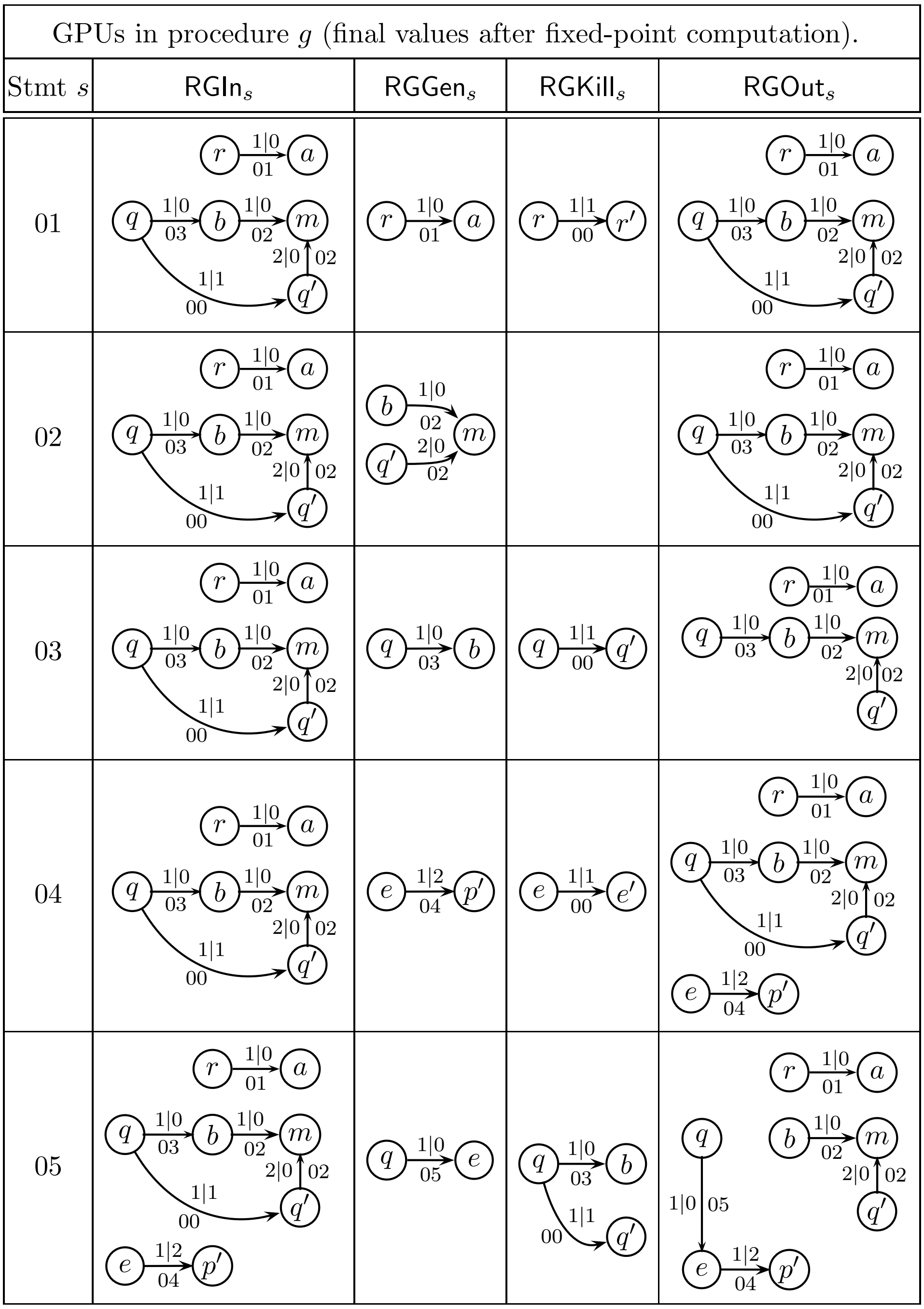


Figure 4.6: The data flow information computed by reaching GPUs analysis for procedure g of the motivating example given in Figure 3.2. In RGIn and RGOut, we show only one boundary definition $q \xrightarrow[00]{1|1} q'$ because other boundary definitions do not participate in GPU reduction for this example. However, the boundary definitions that are removed are shown in RGKill.

at the start ensures that if a boundary definition $x \xrightarrow[00]{i|i} x'$ reaches a program point s, there is a definition-free path from Start to s; its absence at s guarantees that the source of

$x \xrightarrow[00]{i|i} x'$ has been defined along all paths reaching s. This leads to a simple, necessary and sufficient condition for strong updates: all GPUs corresponding to a statement s must define the same location thereby eliminating the possibility represented by the dashed path in Figure 4.5.

The boundary definitions also participate in GPU compositions thereby modelling the semantics of definition-free paths. They enable strong updates thereby improving the precision of analysis.

Example 30. Consider reaching GPUs analysis for the GPB corresponding to statement 02 in the initial GPG of procedure g (δ_{02} in Figure 3.5). We include the boundary definitions for each global variable and the parameters of a procedure as RGIn of the Start GPB of the GPG of procedure g. Although Figure 3.5 does not show boundary definitions for simplicity, they are shown in Figure 4.6 for variable q (boundary definitions of other variables are not required for strong updates in this example). These boundary definitions capture the effect of definition-free paths to distinguish between weak and strong updates.

The GPU $\gamma_2 : q \xrightarrow[02]{2|0} m$ is composed with GPUs from RGIn_{02} which contains a GPU $q \xrightarrow[03]{1|0} b$ indicating that pointer b is being defined by statement 02. However, this is not the case of strong update as b is not the only pointer that is being defined by the assignment. There is a definition-free path along which pointee of q is not available indicating that q may have a definition in the callers of procedure g which is also required in statement 02 of g but is currently unavailable. The presence of boundary definition $q \xrightarrow[00]{1|1} q'$ in RGIn_{02} indicates the presence of a definition-free path and the composition of this GPU results in a reduced GPU $q' \xrightarrow[02]{2|0} m$ which is also a part of δ_{02}. The GPU $q' \xrightarrow[02]{2|0} m$ has been represented by the GPU $q \xrightarrow[02]{2|0} m$ in Figure 3.5 because it ignores boundary definitions.

At the call site in procedure f, after the composition of GPU $q \xrightarrow[07]{1|0} d$ and $q \xrightarrow[02]{2|0} m$ (the upwards-exposed version q' is replaced by q during call inlining; for more details see Chapter 6), the set of reduced GPUs corresponding to statement 02 in procedure f (GPB δ_{13}) contains two GPUs $b \xrightarrow[02]{1|0} m$ and $d \xrightarrow[02]{1|0} m$ (Figure 3.6). Since, statement 02 defines two pointers d and b, no GPU is removed and hence the GPU $d \xrightarrow[08]{1|0} n$ in GPB δ_{08} is retained owing to a weak update.

An important property of boundary definitions is that they appear only in RGIn and RGOut of the reaching-GPUs analysis—they never appear in the GPBs or in RGGen, although the upwards-exposed versions of variables could be involved in the GPUs in RGGen. Also, the algorithm for GPU reduction does not change with the introduction of boundary definitions because a GPU can be composed with boundary definitions just like with any other GPUs.

4.6 Reaching GPUs Analysis without Blocking

In this section, we ignore the effect of barriers and present the data flow equations for computing RGIn and RGOut for every GPB δ in the GPG of a procedure. Section 4.7 incorporates the effects of barriers and performs reaching GPUs analysis with blocking to compute $\overline{\mathsf{RGIn}}$ and $\overline{\mathsf{RGOut}}$ for every GPB δ. The data flow information $\overline{\mathsf{RGGen}}$ computed by this analysis is then used to perform strength reduction optimization of a GPG.

The reaching GPUs analysis is an intraprocedural forward data flow analysis in the spirit of the classical reaching definitions analysis. It computes the set RGIn_s of GPUs reaching a given GPB δ_s by processing the GPBs that precede δ_s on control flow paths reaching δ_s. Then it incorporates the effect of δ_s on the GPUs in RGIn_s through GPU reduction to compute a set of GPUs after s (RGOut_s). The result of GPU reduction, denoted RGGen_s, is semantically equivalent to that of δ_s. The GPUs in RGGen_s have *indlev*s that do not exceed the *indlev*s of the corresponding GPUs in δ_s. Thus, δ_s can be replaced by RGGen_s as a part of strength reduction optimization after the analysis reaches its fixed point.

RGOut_s is computed using RGGen_s and RGKill_s. RGGen_s contains all GPUs computed by GPU reduction $\gamma \circ \mathsf{RGIn}_s$ (for all $\gamma \in \delta_s$). RGKill_s contains the GPUs to be removed. They are under-approximated when a strong update cannot be performed. When a strong update is performed, we kill those GPUs of RGIn_s whose source and *indlev* match that of the shared source of the reduced GPUs (identified by $\mathsf{Match}(\gamma, \mathsf{RGIn}_s)$). For a weak update, $\mathsf{Kill}(\mathsf{RGGen}_s, \mathsf{RGIn}_s) = \emptyset$.

GPU reduction allows us to model Kill (i.e., GPU removal from RGIn) in the case of strong update as follows: The reduced GPUs should define the same abstract location along every control flow path reaching the statement represented by γ. This is captured

$$
\begin{aligned}
\mathsf{RGIn}_s &:= \begin{cases} \left\{x \xrightarrow[s]{\ell|\ell} x' \mid x \in \mathsf{P}, 0 < \ell \leq \kappa\right\} & s = \mathsf{Start}, \kappa \text{ is the largest } \textit{indlev} \\ \displaystyle\bigcup_{p\, \in\, \mathit{pred}(s)} \mathsf{RGOut}_p & \text{otherwise} \end{cases} \\
\mathsf{RGOut}_s &:= (\mathsf{RGIn}_s - \mathsf{RGKill}_s) \cup \mathsf{RGGen}_s \\
\mathsf{RGGen}_s &:= \mathsf{Gen}\,(\delta_s,\ \mathsf{RGIn}_s) \\
\mathsf{RGKill}_s &:= \mathsf{Kill}\,(\mathsf{RGGen}_s,\ \mathsf{RGIn}_s) \\
\mathsf{Gen}(X, R) &:= \bigcup_{\gamma\, \in X} \gamma \circ R \\
\mathsf{Kill}(X, R) &:= \left\{\gamma_1 \mid \exists\, \gamma \in X \text{ such that } |\mathsf{Def}(X, \gamma)| = 1 \wedge \gamma_1 \in \mathsf{Match}(\gamma, R)\right\} \\
\mathsf{Match}(x \xrightarrow[s]{i|j} y, R) &:= \left\{\gamma \in R \mid \gamma = u \xrightarrow[t]{k|l} v,\ x = u,\ i = k\right\} \\
\mathsf{Def}\big(X, w \xrightarrow[s]{k|l} z\big) &:= \left\{(x, i) \mid x \xrightarrow[s]{i|j} y \in X\right\}
\end{aligned}
$$

Definition 5: *Data flow equations for Reaching GPUs Analysis without Blocking*

by the requirement $|\mathsf{Def}(X, \gamma)| = 1$ in the definition of $\mathsf{Kill}(X, R)$ in Definition 5 where $\mathsf{Def}(X, \gamma)$ extracts the source nodes and their indirection levels of the GPUs (i.e. pair (x, i) for GPU $x \xrightarrow[s]{i|j} y$) in X that are constructed for the same statement s. The GPUs that are killed are determined by the GPUs in RGGen_s and not those in δ_s. For defining Kill, we need to identify the pointers defined by a statement. Since a GPB may have GPUs corresponding to multiple pointer assignment statements on account of structural optimization of coalescing of GPBs, we partition the set of GPUs in a GPB according to the assignments they correspond to.

Example 31. Figure 4.6 gives the final result of reaching GPUs analysis for procedure g of our motivating example. We have shown the boundary GPU $q \xrightarrow[00]{1|1} q'$ for q. Other boundary GPUs are not required for strong updates in this example and have been omitted. This result has been used to construct GPG Δ_g shown in Figure 3.5. For procedure f, we do not show the complete result of the analysis but make some observations. The GPU $q \xrightarrow[10]{2|0} o$ is composed with the GPU $q \xrightarrow[05]{1|0} e$ to create a reduced GPU $e \xrightarrow[10]{1|0} o$. Since, only a single pointer (in this case e) is being defined by the assignment, this is a case of strong update and hence kills $e \xrightarrow[04]{1|1} c$. The GPU to be killed is identified by $\mathsf{Match}(e \xrightarrow[10]{1|0} o, \mathsf{RGIn}_{10})$ which matches the source and the *indlev* of the

```
int a, b, *p, *q, **x;
void h()
{  p = &a;      /* GPU p */
   *x = &b;     /* GPU b */
   q = p;       /* GPU c */
}
```

If x points-to p then q points-to b else q points-to a.

(a) Composition across an indirect GPU ***b***

```
int a, b, *p, *q, **x;
void h()
{  *x = &a;     /* GPU p */
   p = &b;      /* GPU b */
   q = *x;      /* GPU c */
}
```

If x points-to p then q points-to b else q points-to a.

(b) Composition with an indirect GPU across the GPU ***b***

Figure 4.7: Risk of unsoundness in GPU reduction caused by a barrier GPU.

GPU to be killed to that of the reduced GPU. Thus, kill is determined by the reduced GPU (in this case $e \xrightarrow[10]{1|0} o$) and not the consumer GPU (in this case $q \xrightarrow[10]{2|0} o$).

4.7 Reaching GPUs Analysis with Blocking

In a GPU reduction, it is possible that ***c*** has an *admissible* composition with some producer GPU ***p***, but the location read by ***c*** could be different from the location defined by ***p*** due to the presence of a barrier GPU ***b*** (Sections 1.3.3 and 4.2).[3]

To ensure soundness, we perform a variant of reaching GPUs analysis that identifies barriers and excludes blocked GPUs from the set of reaching GPUs. The unblocked GPUs are contained in the sets $\overline{\mathsf{RGIn}}_s$ and $\overline{\mathsf{RGOut}}_s$ computed through a data flow analysis. The data flow information $\overline{\mathsf{RGGen}}_s$ computed by this analysis is then used to replace δ_s thereby ensuring the soundness of strength reduction optimization.

4.7.1 The Need of Blocking

The combined effect of an intervening assignment between a producer GPU ***p*** and a consumer GPU ***c***, and the GPUs in a calling context may change the pointer chain

[3] In the MTF approach, all the data dependences are known because of the alias information, hence there is no blocking. In the STF approach, since the data dependence is unknown in the case of dereferences, blocking is required. Such situations are handled by introducing different placeholders for different accesses of the same variable. These accesses may be separated by a barrier in a procedure.

established by ***p***. This alters the data dependence between ***p*** and ***c***. We call such an assignment (or the corresponding GPU), a *barrier*.

In this case, ***c*** should not be composed with ***p*** and should be left unsimplified. If $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ is performed, then RGGen_s will not contain ***c***. Hence, when strength reduction optimization replaces δ_s by RGGen_s, ***c*** will be replaced by the result of composition, possibly leading to unsoundness.

We characterize these situations by building on Section 1.3.3 and defining the notion of a *barrier GPU*. A barrier GPU ***b*** blocks the GPU ***p*** and prevents it from reaching ***c*** effectively *postponing* the composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$. The composition is postponed to avoid the possibility of unsound strength reduction optimization. After inlining the GPG in a caller, more information may become available. Thus, it may resolve any uncertain data dependence between ***c*** and ***p***—so a composition which was earlier postponed may now safely be performed. This is explained in the rest of the section.

We define a barrier as follows. Let an *indirect* GPU refer to a GPU whose *indlev* of the source is greater than 1 (i.e., the pointer being defined by the GPU is not known), for example, $x \xrightarrow[s]{i|j} y$ $i > 1$. Then, a GPU ***b*** corresponding to an assignment between ***c*** and ***p*** on some control flow path is a barrier if:

- ***b*** is an indirect GPU. This is a composition across an indirect GPU ***b*** (Figure 4.7(a)).

- ***p*** is an indirect GPU (***b*** need not be an indirect GPU). This is a composition with an indirect GPU across the GPU ***b*** (Figure 4.7(b)).

We illustrate these situations in the following example.

Example 32. Consider the procedure in Figure 4.7(a). The composition between the GPUs for statements 02 and 04 is *admissible*. However, statement 03 may cause a side-effect by indirectly defining p (if x points to p in the calling context). Thus, q in statement 04 would point to b if x points to p; otherwise it would point to a. If we replace the GPU $q \xrightarrow[04]{1|1} p$ by $q \xrightarrow[04]{1|0} a$ (which is the result of composing $q \xrightarrow[04]{1|1} p$ with $p \xrightarrow[02]{1|0} a$), then we would miss the GPU $q \xrightarrow[04]{1|0} b$ if x points to p in the calling context—leading to unsoundness. Since the calling context is not available during GPG construction, we postpone this composition to eliminate the possibility of unsoundness. This is done by blocking the GPU $p \xrightarrow[02]{1|0} a$ by an indirect GPU $x \xrightarrow[03]{2|0} b$ which acts as a barrier. This corresponds to the first case described above.

For the second case, consider statement 02 of the procedure in Figure 4.7(b) which may indirectly define p (if x points to p). Statement 03 directly defines p. Thus, q in statement 04 would point to b if x points to p; otherwise it would point to a. We postpone the composition $\boldsymbol{c}\!:\!q \xrightarrow[04]{1|2} x$ with $\boldsymbol{p}\!:\!x \xrightarrow[02]{2|0} a$ by blocking the GPU $\boldsymbol{p}$ where the GPU $p \xrightarrow[03]{1|0} b$ acts as a barrier.

A barrier GPU is likely to have a WaW or WaR dependence with some preceding GPUs which cannot be ascertained without the alias information in the calling context.[4] In the absence of alias information from the calling context, we use the type information to identify some such GPUs as non-blocking. The barrier blocks such GPUs, so that the compositions of $\boldsymbol{c}$ with them are postponed (Section 1.3.3). Consider a GPU $\boldsymbol{p}$ originally blocked by a barrier $\boldsymbol{b}$ where $\boldsymbol{p}$ or $\boldsymbol{b}$ could be an indirect GPU. After inlining the GPG in its callers and performing strength reduction optimization in the calling contexts, the following situations could arise:

1. The *indlev* of the source of the indirect GPU ($\boldsymbol{p}$ or $\boldsymbol{b}$) is reduced to 1 thereby identifying the pointer being defined by the GPU. In this case, $\boldsymbol{b}$ ceases being a barrier and so no longer blocks $\boldsymbol{p}$ leading to the following two situations:

   (a) $\boldsymbol{b}$ redefines some location in the pointer chain established by $\boldsymbol{p}$ thereby obviating the composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$. Blocking in this case is essential for soundness.

   (b) $\boldsymbol{b}$ does not redefine a location in the pointer chain established by $\boldsymbol{p}$ thereby allowing the composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$. Blocking in this case is redundant.

2. The *indlev* of the source of the indirect GPU ($\boldsymbol{p}$ or $\boldsymbol{b}$) remains greater than 1. In this case, $\boldsymbol{b}$ continues to block $\boldsymbol{p}$ awaiting further inlining.

In case 1(a), an eager reduction of $\boldsymbol{c}$ without blocking $\boldsymbol{p}$ would cause $\boldsymbol{c}$ to be replaced by the result of composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$, thereby causing unsoundness. Reaching GPUs analysis with blocking helps to postpone the composition until all information becomes available. Our measurements (Chapter 9) show that situation 1(a) rarely arises in practice because

[4] Since we are interested in the writes of a barrier, a RaW data dependence is immaterial.

it amounts to defining the same pointer multiple times through different aliases in the same context. The below example explains all the above cases with an example given in Figure 4.7(a).

Example 33. Case 1(a) above could arise if x points to p in the calling context of the procedure in Figure 4.7(a). As a result, GPU $p \xrightarrow[02]{1|0} a$ is killed by the barrier GPU $p \xrightarrow[03]{1|0} b$ (which is the simplified version of the barrier GPU $x \xrightarrow[03]{2|0} b$) and hence the composition is prohibited and q points to b for statement 04. Case 1(b) could arise if x points to any location other than p in the calling context. In this case, the composition between $q \xrightarrow[04]{1|1} p$ and $p \xrightarrow[02]{1|0} a$ is sound and q points to a for statement 04. Case 2 could arise if pointee of x is not available even in the calling context. In this case, the barrier GPU $x \xrightarrow[03]{2|0} b$ continues to block $p \xrightarrow[02]{1|0} a$.

Example 34 describes blocking to ensure soundness of strength reduction.

Example 34. To see how reaching GPUs analysis with blocking helps, consider the example in Figure 4.7(b). The set of GPUs reaching the statement 04 is $\mathsf{RGIn}_{04} = \{x \xrightarrow[02]{2|0} a, p \xrightarrow[03]{1|0} b\}$. The GPU $x \xrightarrow[02]{2|0} a$ is blocked by the barrier GPU $p \xrightarrow[03]{1|0} b$ and hence $\overline{\mathsf{RGIn}}_{04} = \{p \xrightarrow[03]{1|0} b\}$. Thus, GPU reduction for $w\!:\!q \xrightarrow[04]{1|2} x$ (in the context of $\overline{\mathsf{RGIn}}_{04}$) computes Red as $\{w\}$ with the flag *composed* set to **false** because w cannot be reduced further within the GPG of the procedure. However, w is still not a points-to edge and can be simplified further after the GPG is inlined in its callers. Hence we postpone the composition of w with $\boldsymbol{p}\!:\!x \xrightarrow[02]{2|0} a$ until $\boldsymbol{p}$ is simplified.

4.7.2 Data Flow Equations for Reaching GPUs Analysis With Blocking

A barrier may not necessarily block all preceding GPUs. We use the type information to identify absence of data dependence between a barrier and the GPUs reaching it. This allows us to minimize blocking by identifying GPUs that need not be blocked. A barrier $\boldsymbol{b} \in \overline{\mathsf{RGGen}}_s$ may block a producer GPU $\boldsymbol{p} \in \overline{\mathsf{RGIn}}_s$ if it writes into a location read by or written by $\boldsymbol{p}$. Thus, they could share a WaW or a WaR data dependence. Recall

$$\overline{\mathsf{RGIn}}_s := \begin{cases} \left\{x \xrightarrow[s]{\ell|\ell} x' \mid x \in \mathsf{P}, 0 < \ell \leq \kappa\right\} & s = \mathsf{Start}, \kappa \text{ is the largest } \textit{indlev} \\ \displaystyle\bigcup_{p \,\in\, \mathit{pred}(s)} \overline{\mathsf{RGOut}}_p & \text{otherwise} \end{cases}$$

$$\overline{\mathsf{RGOut}}_s := \left(\overline{\mathsf{RGIn}}_s - \left(\overline{\mathsf{RGKill}}_s \cup \mathsf{Blocked}\left(\overline{\mathsf{RGIn}}_s, \overline{\mathsf{RGGen}}_s\right)\right)\right) \cup \overline{\mathsf{RGGen}}_s$$

$$\mathsf{Blocked}\,(I, G) := \begin{cases} \emptyset & G = \emptyset \\ \{\gamma \mid \gamma \in I, \overline{\mathsf{DDep}}(\mathsf{IndGPUs}(G), \{\gamma\})\} & |\mathsf{IndGPUs}(G)| > 1 \\ \{\gamma \mid \gamma \in \mathsf{IndGPUs}(I), \overline{\mathsf{DDep}}(G, \{\gamma\})\} & \text{otherwise} \end{cases}$$

$$\mathsf{IndGPUs}\,(X) := \{x \xrightarrow[s]{i|j} y \mid x \xrightarrow[s]{i|j} y \in X, i > 1\}$$

$$\overline{\mathsf{RGGen}}_s := \mathsf{Gen}\,\left(\delta_s,\ \overline{\mathsf{RGIn}}_s\right)$$

$$\overline{\mathsf{RGKill}}_s := \mathsf{Kill}\,\left(\overline{\mathsf{RGGen}}_s,\ \overline{\mathsf{RGIn}}_s\right)$$

$$\overline{\mathsf{DDep}}(B, I) \Leftrightarrow \mathsf{TDef}(B) \cap (\mathsf{TDef}(I) \cup \mathsf{TRef}(I)) \neq \emptyset$$

$$\mathsf{TDef}(X) := \left\{\mathsf{typeof}(x, i) \mid x \xrightarrow[s]{i|j} y \ \in \ X\right\}$$

$$\begin{aligned} \mathsf{TRef}(X) := {} & \left\{\mathsf{typeof}(x, k) \mid 1 \leq k < i, x \xrightarrow[s]{i|j} y \in X\right\} \cup \\ & \left\{\mathsf{typeof}(y, k) \mid 1 \leq k < j, x \xrightarrow[s]{i|j} y \in X\right\} \end{aligned}$$

Note: The definitions of Gen and Kill are same as in Definition 5

Definition 6: *Data flow equations for Reaching GPUs Analysis with Blocking.*

that a barrier GPU ***b*** is either an indirect GPU or a GPU that follows an indirect GPU (Section 4.7.1). Thus the following GPUs should be blocked:

- If $\overline{\mathsf{RGGen}}_s$ contains an indirect GPU ***b***, then all GPUs reaching δ_s that share a data dependence with ***b*** should be blocked regardless of the nature of other GPUs (if any) in $\overline{\mathsf{RGGen}}_s$.

- If $\overline{\mathsf{RGGen}}_s$ does not contain an indirect GPU and is not $\emptyset$, then all indirect GPUs reaching δ_s that share a data dependence with a GPU in $\overline{\mathsf{RGGen}}_s$ should be blocked.

We define a predicate $\overline{\mathsf{DDep}}(B, I)$ to check the presence of data dependence between the set of GPUs B and I (Definition 6). When the types of $\boldsymbol{b} \in B$ and $\boldsymbol{p} \in I$ match[5], we

[5] Although C11 standard allows type casting for pointers, there is no guarantee of the expected behavior if there is alignment mismatch. For example, the runtime behavior of assigning '`int *`' to '`float *`'

assume the possibility of data dependence and ***b*** blocks ***p***. $\mathsf{TDef}(B)$ is the set of types of locations being written by a barrier whereas $(\mathsf{TDef}(I) \cup \mathsf{TRef}(I))$ represents the set of types of locations defined or read by the GPUs in I thereby checking a WaW and WaR dependence. The type of the i^{th} pointee of x is given by $\mathsf{typeof}(x, i)$ illustrated as below.

Example 35. If the declaration of a pointer x is '`int` $**$`x`', then $\mathsf{typeof}(x, 1)$ is '`int` $**$' and $\mathsf{typeof}(x, 2)$ is '`int`$*$'. Note that $\mathsf{typeof}(x, 0)$ is not a pointer and $\mathsf{typeof}(x, 3)$ is undefined because x cannot be dereferenced thrice.

The data flow equations in Definition 6 identify the GPUs in $\overline{\mathsf{RGGen}}_s$ that can act as a barrier. The main difference between $\overline{\mathsf{RGOut}}_s$ (Definition 6) and RGOut_s (Definition 5) is that the former uses function $\mathsf{Blocked}$ which computes blocked GPUs as follows:

- Case 1 in $\mathsf{Blocked}$ equation corresponds to not blocking any GPU because $\overline{\mathsf{RGGen}}_s$ is empty.
- Case 2 in $\mathsf{Blocked}$ equation corresponds to blocking appropriate GPUs reaching s (i.e. $\overline{\mathsf{RGIn}}_s$) because $\overline{\mathsf{RGGen}}_s$ contains an indirect GPU.
- Case 3 in $\mathsf{Blocked}$ equation corresponds to blocking appropriate indirect GPUs reaching s because $\overline{\mathsf{RGGen}}_s$ does not contains an indirect GPU and is not $\emptyset$.

Example 36. For the procedure in Figure 4.7(b), $\overline{\mathsf{RGIn}}_{02} = \emptyset$ and $\overline{\mathsf{RGGen}}_{02}$ is $\{x \xrightarrow[02]{2|0} a\}$. Although $\overline{\mathsf{RGGen}}_{02}$ contains an indirect GPU, since no GPUs reach 02 (because it is the first statement), $\overline{\mathsf{RGOut}}_{02}$ is $\{x \xrightarrow[02]{2|0} a\}$ indicating that no GPUs are blocked.

For statement 03, $\overline{\mathsf{RGIn}}_{03} = \{x \xrightarrow[02]{2|0} a\}$ and $\overline{\mathsf{RGGen}}_{03} = \{p \xrightarrow[03]{1|0} b\}$. $\overline{\mathsf{RGGen}}_{03}$ is non-empty and does not contain an indirect GPU and thus $\overline{\mathsf{RGOut}}_{03} = \{p \xrightarrow[03]{1|0} b\}$ according to the third case in the $\mathsf{Blocked}$ equation in Definition 6 indicating that the GPU $x \xrightarrow[02]{2|0} a$ is blocked and should not be used for composition by the later GPUs. The indirect GPU in $\overline{\mathsf{RGIn}}_{03}$ is excluded from $\overline{\mathsf{RGOut}}_{03}$. Note that the indirect GPU $x \xrightarrow[02]{2|0} a$ is blocked by the GPU $p \xrightarrow[03]{1|0} b$ because $\mathsf{typeof}(x, 2)$ matches with $\mathsf{typeof}(p, 1)$ indicating

depends on the compiler and the architecture. However, assigning '`void`$*$' to '`int`$*$' does not result in misalignment. In our implementation, we trust the types recorded in the GIMPLE IR used by gcc and assume that there is no undefined behavior of the program.

a possibility of WaW dependence.

For statement 04, $\overline{\mathsf{RGIn}}_{04} = \{p \xrightarrow[03]{1|0} b\}$ and $\overline{\mathsf{RGGen}}_{04}$ is $\{q \xrightarrow[04]{1|2} x\}$. For this statement, the composition $(q \xrightarrow[04]{1|2} x \circ^{\text{ts}} x \xrightarrow[02]{2|0} a)$ is postponed because the GPU $x \xrightarrow[02]{2|0} a$ is blocked. In this case, $\overline{\mathsf{RGGen}}_{04}$ does not contain an indirect GPU and $\overline{\mathsf{RGOut}}_{04} = \{p \xrightarrow[03]{1|0} b, q \xrightarrow[04]{1|2} x\}$.

Similarly in Figure 4.7(a), the GPU $p \xrightarrow[02]{1|0} a$ is blocked by the barrier GPU $x \xrightarrow[03]{2|0} b$ because $\mathsf{typeof}(p, 1)$ matches with $\mathsf{typeof}(x, 2)$. Hence, the composition $(q \xrightarrow[04]{1|1} p \circ^{\text{ts}} p \xrightarrow[02]{1|0} a)$ is postponed.

In the GPG of procedure g of our motivating example in Figure 3.5, the GPUs $r \xrightarrow[01]{1|0} a$ and $q \xrightarrow[03]{1|0} b$ are not blocked by the GPU $q \xrightarrow[02]{2|0} m$ because they have different types. However, the GPU $e \xrightarrow[04]{1|2} p$ blocks the indirect GPU $q \xrightarrow[02]{2|0} m$ because there is a possible WaW data dependence (e and q could be aliased in the callers of g).

4.8 Convergence of Reaching GPUs Analyses

In this section, we show the convergence of reaching GPUs analyses by defining a lattice and proving that the flow functions of the analyses are monotonic.

We use the usual guarantee of the convergence of a data flow analysis on the maximum fixed point solution if the following conditions are satisfied [51]:

- The lattice L of data flow values is a complete lattice.[6]
- Flow functions $f : L \rightarrow L$ are monotonic.
- All strictly descending chains in L are finite.

4.8.1 Lattice for Reaching GPUs Analyses

Let Γ be a set of all possible GPUs representing every pointer pointing to every location based on the type constraints. It is easy to see that Γ is finite: For two variables x and y, the number of GPUs $x \xrightarrow[s]{i|j} y$ depends on the number of possible *indlev*s $(i|j)$ and the number of statements. Since the number of statements is finite, we need to examine the number of *indlev*s. For pointers to scalars, the number of *indlev*s between any two

[6] Actually, we only need the lattice to be a meet-semilattice but our lattices are complete and every complete lattice is a meet-semilattice.

variables is bounded because of type restrictions. For pointers to structures (Chapter 8), *indlev*s are replaced by indirection lists (*indlist*s). Sections 8.3 and 8.4 summarize *indlist*s restricting them to a finite number between any two pointer variables.

In order to define a partial order between the sets of GPUs computed by reaching GPUs analyses, we define two useful properties of the sets. Recall that the presence of boundary definitions eliminate definition-free paths reaching a GPB δ (Section 4.5). In other words, every pointer is defined along every path reaching δ. Thus, for every pointer variable x, sets RGIn_s and RGOut_s (equivalently, $\overline{\mathsf{RGIn}_s}$ and $\overline{\mathsf{RGOut}_s}$) contain at least one GPU with source x and *indlev* ℓ where ℓ ranges from 1 to the maximum indirection level. This has the following consequences:

- Let $\mathsf{S}(X)$ denote the set of sources (with *indlev*s) of all GPUs in X.

  $$\mathsf{S}(X) = \{(x, i) \mid \gamma \in X, \gamma = x \xrightarrow[s]{i|j} y\}$$

  For non-empty sets R_1 and R_2 such that $R_1 \subseteq R_2$, the set of sources is identical, i.e., $\mathsf{S}(R_1) = \mathsf{S}(R_2)$.

- Let $\mathsf{T}(X)$ denote the set of targets (with *indlev*s) of all GPUs in X.

  $$\mathsf{T}(X) = \{(y, j) \mid \gamma \in X, \gamma = x \xrightarrow[s]{i|j} y\}$$

  Given R_1 and R_2 such that $R_1 \subseteq R_2$, since $\mathsf{S}(R_1) = \mathsf{S}(R_2)$, the likely existence of additional GPUs in R_2 is possible only when $\mathsf{T}(R_1) \subseteq \mathsf{T}(R_2)$.

We use the above properties to define a partial order between the sets of GPUs computed by reaching GPUs analyses as follows:

$$\forall R_1, R_2 \subseteq \mathit{\Gamma} \quad R_1 \sqsubseteq R_2 \iff (R_1 \supseteq R_2) \;\wedge\; \big((R_2 = \emptyset) \;\vee (\mathsf{S}(R_1) = \mathsf{S}(R_2))\big)$$

Observe that $R_1 \supseteq R_2$ and $\mathsf{S}(R_1) = \mathsf{S}(R_2)$ ensure that $\mathsf{T}(R_1) \supseteq \mathsf{T}(R_2)$ and hence the definition of partial order does not require a relationship between the targets of GPUs to be included explicitly.

Example 37. Consider two sets of GPUs R_1 and R_2 such that $R_1 = \{a \xrightarrow[1]{2|1} b\}$ and $R_2 = \{a \xrightarrow[1]{2|1} b, c \xrightarrow[2]{1|0} d\}$. Though $R_1 \subseteq R_2$, $R_2 \not\sqsubseteq R_1$ because pointer c is not defined in R_1. However, if $R_1 = \{a \xrightarrow[1]{2|1} b, c \xrightarrow[0]{1|1} c'\}$ and $R_2 = \{a \xrightarrow[1]{2|1} b, c \xrightarrow[0]{1|1} c', c \xrightarrow[2]{1|0} d\}$, then $R_2 \sqsubseteq R_1$ because $R_2 \supseteq R_1$ and $\mathsf{S}(R_1) = \mathsf{S}(R_2) = \{(a, 2), (c, 1)\}$.

The lattice $\left(2^{\Gamma}, \sqsubseteq\right)$ is a complete lattice (because Γ is finite). The top element of this lattice is $\emptyset$ and the bottom element is Γ.

4.8.2 Monotonicity of GPBs

We prove the monotonicity of GPBs for reaching GPUs analysis with blocking and argue later why the result also holds for reaching GPUs analysis without blocking. For readability, we use notations I_s and O_s for $\overline{\mathsf{RGIn}}_s$ and $\overline{\mathsf{RGOut}}_s$, G_s and K_s for $\overline{\mathsf{RGGen}}_s$ and $\overline{\mathsf{RGKill}}_s$, and $\overline{\mathsf{B}}_s$ for blocked GPUs.

Lemma 4.1. *A GPB δ_s is monotonic if* $\mathsf{I}'_s \sqsubseteq \mathsf{I}_s \Rightarrow \mathsf{O}'_s \sqsubseteq \mathsf{O}_s$

Proof. From Definition 6, we know that,

$$\mathsf{O}_s = (\,\mathsf{I}_s - (\,\mathsf{K}_s \cup \mathsf{B}_s)) \cup \mathsf{G}_s$$

From the definition of our partial order, we know that,

$$\begin{aligned}
\mathsf{I}'_s \sqsubseteq \mathsf{I}_s &\Rightarrow \mathsf{I}'_s \supseteq \mathsf{I}_s \\
&\Rightarrow \bigcup_{\gamma\,\in\,\delta_s} \gamma \circ \mathsf{I}'_s \supseteq \bigcup_{\gamma\,\in\,\delta_s} \gamma \circ \mathsf{I}_s \\
&\Rightarrow \mathsf{G}'_s \supseteq \mathsf{G}_s \qquad\qquad \text{(GPU reduction, Definition 4)}
\end{aligned}$$

K_s is computed by matching the source and *indlev* of GPUs in Γ with the source and *indlev* of the GPUs in G_s ($\mathsf{K}_s = \mathsf{Kill}\ (\mathsf{G}_s)$).[7]

$$\mathsf{Kill}(X) = \left\{\gamma_1 \mid \exists\, \gamma \in X \text{ such that } |\mathsf{Def}(X, \Gamma)| = 1 \wedge \gamma_1 \in \mathsf{Match}(\gamma, \Gamma)\right\}$$

where $\mathsf{Def}(X, \Gamma)$ computes a set of pairs (source and its *indlev*) for the GPUs in X with the same statement label. Kill is performed only when a single pointer is being defined by a statement resulting in a strong update.

[7] The earlier definition of Kill (Definition 5) restricted it to the GPUs in the argument RGIn_s. The definition used here omits the second argument of Kill by extending it to the entire Γ for convenience. This does not affect $(\mathsf{RGIn}_s - \mathsf{Kill}(\ldots))$.

For $\mathsf{I}'_s \sqsubseteq \mathsf{I}_s$, we know that $\mathsf{S}(\mathsf{I}'_s) = \mathsf{S}(\mathsf{I}_s)$ and $\mathsf{T}(\mathsf{I}'_s) \supseteq \mathsf{T}(\mathsf{I}_s)$. Thus, a strong update with I'_s as input implies a strong update with I_s as input. However a strong update in I_s as input does not imply a strong update in I'_s. Hence, $\mathsf{I}'_s \sqsubseteq \mathsf{I}_s \Rightarrow \mathsf{K}_s \supseteq \mathsf{K}'_s$. Similarly, we can reason about the blocked GPUs and get, $\mathsf{I}'_s \sqsubseteq \mathsf{I}_s \Rightarrow \mathsf{B}_s \supseteq \mathsf{B}'_s$. Thus,

$$\begin{aligned}\mathsf{I}'_s \sqsubseteq \mathsf{I}_s &\Rightarrow \mathsf{I}'_s \supseteq \mathsf{I}_s\\ &\Rightarrow \big(\big(\mathsf{I}'_s - \big(\mathsf{K}'_s \cup \mathsf{B}'_s\big)\big) \cup \mathsf{G}'_s\big) \supseteq \big(\big(\mathsf{I}_s - (\mathsf{K}_s \cup \mathsf{B}_s)\big) \cup \mathsf{G}_s\big)\\ &\Rightarrow \mathsf{O}'_s \supseteq \mathsf{O}_s\end{aligned}$$

In order to prove $\mathsf{O}'_s \sqsubseteq \mathsf{O}_s$, we show below that $\mathsf{S}(\mathsf{O}'_s) = \mathsf{S}(\mathsf{O}_s)$. Since, Kill is determined by matching the sources of the GPUs in G_s, the set of sources in I_s and O_s (equivalently, those in I'_s and O'_s) are identical. In other words, for every GPU $x \xrightarrow{i|j} y$ that is killed, some GPU $x \xrightarrow{i|k} z$ is generated. Hence, $(\mathsf{S}(\mathsf{I}'_s) = \mathsf{S}(\mathsf{I}_s)) \Rightarrow (\mathsf{S}(\mathsf{O}'_s) = \mathsf{S}(\mathsf{O}_s))$.

Thus, $\mathsf{I}'_s \sqsubseteq \mathsf{I}_s \Rightarrow \mathsf{O}'_s \sqsubseteq \mathsf{O}_s$ and hence a GPB is monotonic. □

The set of blocked GPUs B, is $\emptyset$ for reaching GPUs analysis without blocking and hence the monotonicity of GPBs for reaching GPUs analysis without blocking follows as a corollary to the above lemma.

4.8.3 Convergence on the Maximum Fixed Point

The data flow equations for reaching GPUs analyses are modelled as follows. Let a GPG Δ contain n GPBs numbered 1 through n. The initialization for the analysis is as follows: We initialize I_1 to the set D of boundary definitions. All other I_i and O_i for GPB δ_i are initialized to $\top$ (i.e, $\emptyset$). A fixed point for the Equation (4.7) is computed by repetitive application of flow functions.

$$\begin{aligned}\langle \mathsf{I}_1, \mathsf{O}_1, \mathsf{I}_2, \mathsf{O}_2, \ldots, \mathsf{I}_n, \mathsf{O}_n\rangle = \langle\ \ &\mathsf{I}_1,\\ &f_{\mathsf{O}_1}(\mathsf{I}_1, \mathsf{O}_1, \mathsf{I}_2, \mathsf{O}_2, \ldots, \mathsf{I}_n, \mathsf{O}_n),\\ &f_{\mathsf{I}_2}(\mathsf{I}_1, \mathsf{O}_1, \mathsf{I}_2, \mathsf{O}_2, \ldots, \mathsf{I}_n, \mathsf{O}_n),\\ &\qquad\qquad \ldots\\ &f_{\mathsf{I}_n}(\mathsf{I}_1, \mathsf{O}_1, \mathsf{I}_2, \mathsf{O}_2, \ldots, \mathsf{I}_n, \mathsf{O}_n),\\ &f_{\mathsf{O}_n}(\mathsf{I}_1, \mathsf{O}_1, \mathsf{I}_2, \mathsf{O}_2, \ldots, \mathsf{I}_n, \mathsf{O}_n)\rangle\end{aligned} \tag{4.7}$$

where each f_{I_i} is a meet operation and f_{O_i} is application of GPB δ. The flow functions f_{I_i} and f_{O_i} compute the values of I_i and O_i respectively.

Theorem 4.2. *Reaching GPUs analysis with blocking converges on the maximum fixed point.*

Proof. Convergence on a fixed-point follows from the following two facts:

- All strictly descending chains in the lattice $(2^{\Gamma}, \sqsubseteq)$ are finite because the lattice is finite.
- The meet operation (i.e., a set union) is monotonic and is defined for all subsets of 2^{Γ} because $\left(2^{\Gamma}, \sqsubseteq\right)$ is a complete lattice. Hence the flow functions f_{I_i} in Equation (4.7) are monotonic. Further, the flow functions f_{O_i} are also monotonic because GPBs are monotonic.

Since we use the initialization $\top$ (i.e., $\emptyset$) and the lattice is complete, the analysis converges on the maximum fixed point. □

4.9 Chapter Summary

In this chapter, we defined GPU operations (such as GPU composition and GPU reduction) and data flow analyses (such as reaching GPUs analysis with and without blocking) to perform strength reduction optimization. This optimization is local to a GPB in the GPG and is one of the most important optimizations to compute points-to information using GPGs.

Our measurements (Chapter 9) show that the indirect effects that cause unsoundness without blocking are extremely rare in practical programs, thereby indicating that reaching GPUs analysis with blocking may not be required.

Chapter 5

Redundancy Elimination Optimizations

In this chapter, we formalize the various redundancy elimination optimizations.

Section 5.1 gives an overview of redundancy elimination optimizations. Section 5.2 formalizes dead GPUs elimination and describes empty GPB elimination. Section 5.4 provides data flow equations for coalescing of GPBs. Section 5.5.4 provides data flow equations for back edge removal in a GPG.

5.1 An Overview of Redundancy Elimination Optimizations

Strength reduction simplifies GPUs and eliminates data dependences between them. This paves way for redundancy elimination optimizations which remove redundant GPUs and minimize control flow. As a consequence, they improve the compactness of a GPG and reduce the repeated re-analysis of GPBs caused by inlining at call sites. We propose the following three categories of redundancy elimination optimizations:

- Dead GPU and empty GPB elimination.
- GPB Coalescing.
- Back edge removal.

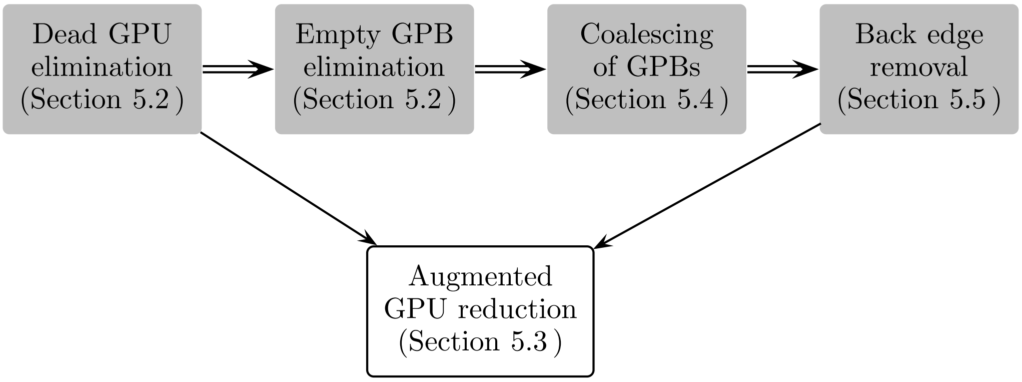

Dead GPU elimination (Section 5.2)
Empty GPB elimination (Section 5.2)
Coalescing of GPBs (Section 5.4)
Back edge removal (Section 5.5)
Augmented GPU reduction (Section 5.3)

Figure 5.1: The order of redundancy elimination optimizations. The edges with double lines indicate a sequence. An edge $X \Rightarrow Y$ indicates that X precedes Y in the sequence. Other edges indicate dependencies. An edge $X \rightarrow Y$ indicate that X depends on Y.

Figure 5.1 depicts the order of redundancy elimination optimizations as a sequence. Dead GPU elimination and back edge removal optimizations require the set Queued which is computed by an augmented version of GPU reduction (Definition 7 in Section 5.3).

Recall that the strength reduction optimization may postpone the reduction of certain GPUs. This requires us to postpone optimizations such as dead GPU elimination, coalescing, and back edge removal in order to ensure soundness. In this chapter, we describe each of the optimizations in detail and characterize when to postpone them.

5.2 Dead GPU and Empty GPB Elimination

We perform dead GPU elimination to remove a redundant GPU $\gamma \in \delta_s$ that is killed along every control flow path from s to the End GPB of the procedure. However, the following two kinds of GPUs should not be removed even if they are killed in reaching GPUs analyses: (a) GPUs that are blocked, or (b) GPUs that are producer GPUs for compositions that have been postponed (Sections 4.3.2 and 4.7). For the former, we check that a GPU considered for dead GPU elimination does not belong to $\mathsf{RGOut}_{\mathsf{End}}$ (the result of reaching GPUs analysis without blocking) and $\overline{\mathsf{RGOut}}_{\mathsf{End}}$ (the result of reaching GPUs analysis with blocking); for the latter we check that the GPU is not a producer GPU for a postponed composition. We record such GPUs in the set Queued computed for every GPG. It is computed during GPU reduction. The revised definition of GPU reduction that computes the Queued set is provided in Definition 7 (Section 5.3). Thus, we perform dead

GPU elimination and remove a GPU $\gamma \in \delta_s$ only if $\gamma \notin (\mathsf{RGOut}_{\mathsf{End}} \cup \overline{\mathsf{RGOut}}_{\mathsf{End}} \cup \mathsf{Queued})$.

Example 38 provides an instance of dead GPU elimination.

Example 38. In procedure g of Figure 3.5, pointer q is defined in statement 03 but is redefined in statement 05 and hence the GPU $q \xrightarrow[03]{1|0} b$ is killed and does not reach the End GPB. Since no composition with the GPU $q \xrightarrow[03]{1|0} b$ is postponed, it does not belong to set Queued either. Hence the GPU $q \xrightarrow[03]{1|0} b$ is eliminated from the GPB δ_{03} as an instance of dead GPU elimination.

Similarly, the GPUs $q \xrightarrow[07]{1|0} d$ (in δ_{07}) and $e \xrightarrow[04]{1|1} c$ (in δ_{14}) in the GPG of procedure f (Figure 3.6) are eliminated from their corresponding GPBs. Observe that the GPU $d \xrightarrow[08]{1|0} n$ in GPB δ_{08} is not removed even though δ_{13} contains a definition of d expressed by the GPU $d \xrightarrow[02]{1|0} m$. This is because δ_{13} also contains GPU $b \xrightarrow[02]{1|0} m$ which defines b, indicating that d is not defined along all paths. Hence the previous definition of d cannot be killed resulting in a weak update.

Example 39 provides an instance where the optimization of dead GPU elimination has to be suppressed because of blocking.

Example 39. For the procedure in Figure 4.7(a), the GPU $p \xrightarrow[02]{1|0} a$ is not killed but is blocked by the barrier $x \xrightarrow[03]{2|0} b$; hence it is present in RGOut_{05} but not in $\overline{\mathsf{RGOut}}_{05}$ (05 is the End GPB). This GPU may be required when the barrier $x \xrightarrow[03]{2|0} b$ is reduced after call inlining (and ceases to block $p \xrightarrow[02]{1|0} a$). Thus, it is not removed by dead GPU elimination.

In the process of dead GPU elimination, if a GPB becomes empty, it is eliminated by connecting its predecessors to its successors as illustrated by the following example.

Example 40. In the GPG of procedure g of Figure 3.5, the GPB δ_{03} becomes empty after dead GPU elimination. Hence, δ_{03} can be removed by connecting its predecessors to successors. This transforms the back edge $\delta_{03} \rightarrow \delta_{01}$ to $\delta_{02} \rightarrow \delta_{01}$. Similarly, the GPB δ_{07} is deleted from the GPG of procedure f in Figure 3.6.

```
Input:  $\boldsymbol{c}$        // The consumer GPU to be simplified.
        $R$        // The set of GPUs using which $\boldsymbol{c}$ is to be simplified.
        $\overline{R}$        // The set of GPUs that have been blocked by a barrier.
Output: Red      // The set of simplified GPUs equivalent to $\boldsymbol{c}$.
        Queued   // The set of GPUs which may be used later.
Augmented_GPU_reduction ($\boldsymbol{c}$, $R$, $\overline{R}$)
{ if ($R = \emptyset \land \boldsymbol{c} = \cdot \xrightarrow[\cdot]{1|0} \cdot$)  // $\boldsymbol{c}$ is a points-to edge
    { Red = $\{\boldsymbol{c}\}$
        $W$ = Queued = $\emptyset$
    }
    else
    { Red = Queued = $\emptyset$
        $W = \{\boldsymbol{c}\}$
    }
    while ($W \neq \emptyset$)
    { extract $w$ from $W$
        for each $\gamma \in R$
        { $\langle W, tscomp, tspost \rangle$ = Compose_GPUs($ts, w, \gamma, W, \overline{R}$)
            $\langle W, sscomp, sspost \rangle$ = Compose_GPUs($ss, w, \gamma, W, \overline{R}$)
            if ($tspost$ or $sspost$)
                Queued = Queued $\cup \{\gamma\}$
        }
        if ($\neg$ ($tscomp$ or $sscomp$))
            Red = Red $\cup \{w\}$
    }
    return (Red , Queued)
}
Compose_GPUs($\tau, w, \gamma, W, \overline{R}$)
{ $composed$ = $postpone$ = false
    if ($r = w \circ^{\tau} \gamma$) succeeds
    { if ($\gamma \in \overline{R}$)
        { $W = W \cup \{r\}$
            $composed$ = true
        }
        else $postpone$ = true
    }
    else if (Undes_comp($\tau, w, \gamma$))
        $postpone$ = true
    return $\langle W, composed, postpone \rangle$
}
```

Definition 7: *Augmented edge reduction algorithm for computing* Queued *GPUs*

5.3 Augmented GPU Reduction Algorithm for Computing Queued GPUs

Calculating the set of GPUs Queued for dead GPU elimination can be performed parallely with GPU reduction. We thus define a new algorithm given in Definition 7 for GPU reduction which augments the method of computing set Queued. The set Queued is also required for back-edge removal as explained in Section 5.5. The new algorithm needs two arguments (unblocked GPUs $\overline{\mathsf{RGOut}}$ as well as blocked+unblocked GPUs RGOut) unlike the earlier definition of GPU reduction.

A GPU $\boldsymbol{p} \in \mathsf{Queued}$ could belong to any of the following categories:

- Composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ may be postponed because $\boldsymbol{p}$ may be a GPU blocked by the presence of a barrier. It is possible that the barrier which may be simplified after Δ is inlined in a caller and may not block $\boldsymbol{p}$ any more enabling its composition with the consumer GPU $\boldsymbol{c}$.

- Composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ may be *undesirable*. It is possible that $\boldsymbol{p}$ may be simplified after Δ is inlined in a caller making the composition *desirable*.

In the first case, a GPU composition is *admissible* when RGIn is used for GPU reduction but with the GPU $\boldsymbol{p}$ being blocked ($\boldsymbol{p} \notin \overline{\mathsf{RGIn}}$), the composition is postponed. These conditions are checked at line numbers 25 and 26 in Definition 7 and accordingly the flag *postpone* is set.

In the second case, we identify a *valid* but *undesirable* GPU composition using the predicate Undes_comp (Equation 5.1) which checks that a pivot exists ($v = x$ for *TS* and $u = x$ for *SS*) and the composition is *undesirable* ($l > k$). This check is performed at line number 32 in Definition 7 and accordingly the flag *postpone* is set.

$$\mathsf{Undes\_comp}\big(\tau, u \xrightarrow[t]{i|j} v, x \xrightarrow[s]{k|l} y\big) = \begin{cases} \textbf{true} & (\tau = \text{ts}) \wedge (v = x) \wedge (l > k) \\ \textbf{true} & (\tau = \text{ss}) \wedge (u = x) \wedge (l > k) \\ \textbf{false} & \text{otherwise} \end{cases} \tag{5.1}$$

Example 41 illustrates the computation of set Queued in the case of blocking.

Example 41. The set of GPUs reaching statement 04 is $\{p \xrightarrow[02]{1|0} a, x \xrightarrow[03]{2|0} b\}$. The barrier GPU (in this case $x \xrightarrow[03]{2|0} b$) blocks the GPU $p \xrightarrow[02]{1|0} a$ and hence $\overline{\mathsf{RGIn}}_{04} = \{x \xrightarrow[03]{2|0} b\}$. Thus, the composition between GPUs ***c*** : $q \xrightarrow[04]{1|1} p$ and ***p*** : $p \xrightarrow[02]{1|0} a$ is *admissible* if RGIn_{04} is used for GPU reduction and the condition on line 25 of Definition 7 is **true**. However, since ***p*** $\notin \overline{\mathsf{RGIn}}_{04}$, the condition on line 26 (Definition 7) is **false** and the flag *postpone* is set to **true** indicating that ***p*** is blocked and a composition is postponed and ***p*** should be added to Queued as seen on line numbers 15 and 16 of the algorithm.

```
01  void h()
02  {  p = &a;
03     *x = &b;
04     q = p;
05  }
```

Example 42 illustrates the computation of set Queued for *undesirable* compositions.

Example 42. The composition between GPUs ***c*** : $p \xrightarrow[3]{1|2} y$ and ***p*** : $y \xrightarrow[4]{1|2} x$ is *undesirable* because the result of composition is a GPU $p \xrightarrow[3]{1|3} x$ whose *indlev* exceeds that of ***c***. This composition will be performed once the ***p*** is simplified. The predicate Undes_comp returns **true** because $(l > k)$ (in this case $l = 2$ and $k = 1$) indicating that the composition is *undesirable* and adds ***p*** to the set Queued.

5.4 Minimizing the Control Flow by Coalescing GPBs

Strength reduction eliminates data dependence between GPUs rendering the control flow redundant. Eliminating redundant control flow is important to make a GPG as compact as possible—in the absence of control flow minimization, the size of the GPG of a procedure tends to increase exponentially because of transitive inlinings of calls in the procedure. This effect is aggravated by the fact that many procedures are called multiple times in the same procedure as shown by our empirical measurements. Besides, recursion causes multiple inlinings of the GPGs of procedures in the cycle of recursion in order to compute a fixed point (Section 6.3).

In this section, we first present our first attempt for coalescing adjacent GPBs in a GPG. This approach did not give us the desired efficiency benefit, hence we devised the second approach for coalescing. The first approach imposed stronger conditions for coalescing than the second approach. It is important to note that there is no loss of precision in either of them.

5.4.1 Our First Approach: Coalescing of Adjacent GPBs

This approach coalesces two GPBs δ_s and δ_t in a GPG where $\delta_t \in succ(\delta_s)$ into a single GPB under the following conditions:

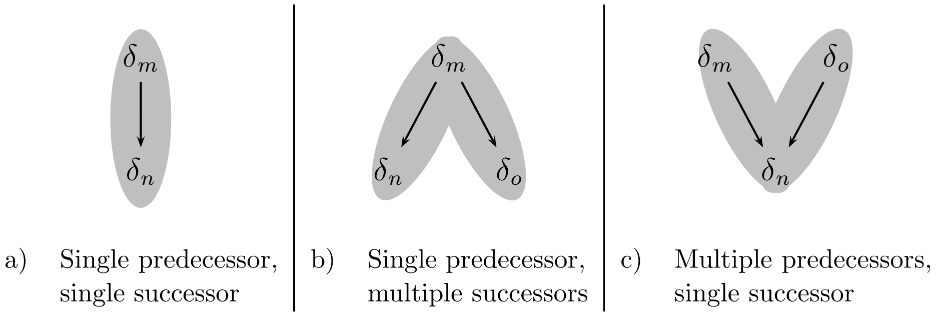
(i) No GPU in δ_t is data-dependent on a GPU in δ_s, i.e., $\mathsf{DDep}(\delta_s, \delta_t)$ returns false (DDep is defined in Definition 8). For RaW and WaW data dependence, we maintain the control flow.

(ii) GPBs δ_s and δ_t are adjacent in the control flow, δ_s dominates δ_t, and δ_t post-dominates δ_s. This ensures that the control flow relation with other GPBs is not over-approximated.

When we coalesce two GPBs, the resulting GPB is numbered with a new label. This does not change the statement number of the GPUs in the GPB. Observe that initially, each GPB contains a single GPU and the label of the GPB corresponds to the statement number of the GPU in the GPB. After GPU reduction of the lone GPU in a GPB, the GPB may contain multiple GPUs but all of them would have the same statement number and the label of the GPB continues to correspond to the statement number of its GPUs. However, after coalescing a GPB typically contains multiple GPUs whose statement numbers differ from each other.

Example 43. In the GPG $\varDelta_g$ in Figure 3.5, GPB δ_{01} dominates GPB δ_{02} and δ_{02} post-dominates δ_{01}. However, since δ_{02} contains a since the pointee of q is not known, the data dependence between the GPU $q \xrightarrow[02]{2|0} m$ in GPB δ_{02} and the GPUs in other δ_{01} cannot be determined. Hence, the GPB δ_{02} cannot be coalesced with δ_{01} in the GPG of procedure g. Similarly, the GPB δ_{04} cannot be coalesced with the GPB δ_{05} because the pointee of p is not known in the GPU $e \xrightarrow[04]{1|2} p$ of GPB δ_{04} indicating the possibility of a data dependence between the two GPBs.

The elimination of control flow because of coalescing is rare in this approach because of the strong condition of dominance and post-dominance between the adjacent GPBs. Also, a dereference in the GPUs restricts the coalescing further because data dependence of a dereference is unknown.



a) Single predecessor, single successor

b) Single predecessor, multiple successors

c) Multiple predecessors, single successor

Figure 5.2: Our strategy for partitioning the GPG for coalescing of GPBs. The case of multiple predecessors and multiple successors can be viewed as a combination of cases (b) and (c).

5.4.2 Our Second Approach: Coalescing GPBs by Partitioning a GPG

In this approach, we relax the requirement of dominance and post-dominance and eliminate redundant control flow by coalescing adjacent GPBs. This amounts to partitioning the set of GPBs in a GPG such that each part contains the GPBs whose GPUs do not have a data dependence between them and hence can be seen as parallel assignments in accordance with abstract semantics of a GPB (Section 3.1).

5.4.2.1 The Intuition behind Coalescing

Since partitioning is driven by preserving and exploiting the absence of data dependence, it is characterized by the following properties:

- A GPG can be partitioned in multiple ways to minimize the control flow. The absence of data dependence is not a transitive relation: Consider GPBs δ_l, δ_m, and δ_n such that $m \in \mathit{succ}(l)$ and $n \in \mathit{succ}(m)$. Assume that $\gamma_m \in \delta_m$ does not have a data dependence with $\gamma_l \in \delta_l$ and $\gamma_n \in \delta_n$ does not have a data dependence with $\gamma_m \in \delta_m$. However, there may be a data dependence between $\gamma_l \in \delta_l$ and $\gamma_n \in \delta_n$. If the data dependence exists, then the following two partitions have minimal control flow: $\Pi_1 = \{\{\delta_l, \delta_m\}, \{\delta_n\}\}$ and $\Pi_2 = \{\{\delta_l\}, \{\delta_m, \delta_n\}\}$. Our heuristics (described below) construct partition Π_1.
- The possibility of data dependence between GPBs δ_m and δ_n matters only if there is control flow between them. Otherwise, they are executed in different execution

instances of the program and there is no data dependence between them even if the variables or abstract locations accessed by them are same. Hence the successors of a GPB can be coalesced with each other in the same part provided there is no control flow between them.

- As a design choice, a successor (predecessor) of a GPB is included in the part containing the GPB *iff all* successors (predecessors) of the GPB are included in the part (Figure 5.2): Consider GPBs δ_l, δ_m and δ_n such that $\mathit{succ}(l) = \{m, n\}$ and neither m is a successor of n nor vice-versa. Let $\delta_l \in \pi_i$. Since there is no control flow between δ_m and δ_n, including only one of them in π_i will create a spurious control flow between them. This ordering could introduce a spurious data dependence between their GPUs. A spurious RaW dependence may create spurious GPUs thereby causing imprecision, or a spurious WaR or WaW dependence may kill some GPUs leading to unsoundness.

- Coalescing may eliminate a definition-free path for the source of a GPU. This may convert the GPU from *may*-def (i.e., source is defined along some path) to ***must***-def (i.e., source is defined along all paths) in the GPG. Consider GPBs δ_l, δ_m, δ_n, and δ_o such that $\mathit{succ}(l) = \{m, n\}$ and $\mathit{pred}(o) = \{m, n\}$. Let $\pi_i = \{\delta_l, \delta_m, \delta_n\}$ and $\pi_j = \{\delta_o\}$. The source of some GPU $\gamma_m \in \delta_m$ may have a definition-free path $\delta_l \to \delta_n \to \delta_o$. After coalescing, this definition-free path ceases to exist because of the control flow edge $\pi_i \to \pi_j$. This may lead to strong updates instead of weak updates thereby leading to unsoundness. Hence, we add a separate definition-free path for such GPUs.

Due to the possibility of multiple partitions satisfying the above criteria, identifying the "best" partition would require defining a cost model. Instead, we compute a unique partition by imposing additional restrictions described below. Our empirical measurements show significant compression by our heuristic partitioning below and any attempt of finding the best partitioning may provide only marginal overall benefits because the process would become inefficient. Hence we use the following greedy heuristics:

- Start GPB and End GPB form singleton parts and no other GPB is included in these parts. This is required for modelling definition-free paths from Start to End to distinguish between strong and weak updates by a callee GPG in a caller GPG.

- The process of identifying the partition begins with Start GPB. Thus Start forms $\pi_1 \in \mathit{\Pi}$. As a consequence, a part $\pi_i \in \mathit{\Pi}$ grows only in the "forward" direction including only successor GPBs. It never grows in the "backward" direction by considering predecessors.

- Consider δ_n and δ_s, $s \in \mathit{succ}(n)$ such that $\delta_n \rightarrow \delta_s$ is a back edge. Then δ_n and δ_s belong to the same partition π_i *iff* all GPBs in the loop formed by the back edge (i.e. all GPBs that appear on all paths from δ_s to δ_n) belong to π_i.

In principle, partitioning could be performed using a greedy process interleaved with coalescing such that each part grows incrementally. However, this incremental expansion cannot be done by coalescing one successor at a time because all successors and all predecessors of all these successors must be included in the same partition, and this property needs to be applied transitively. Hence, we use the usual dichotomy of analysis and transformations and separate the process of discovering the partition (analysis) from the process of coalescing (transformation). We perform a data flow analysis that computes the partitions by examining the data dependencies and enforcing all requirements of partitioning. Actual coalescing is performed only after all partitions have been identified. Hence, we define a data flow analysis that constructs a part π_i inductively by considering the possibility of including the successors of the GPBs that are already in π_i.

5.4.2.2 The Role of Data Dependence in Blocking and Coalescing

The main differences between the use of data dependence for blocking (Definition 6 in Section 4.7) and for coalescing are:

- *The motivation behind using data dependence.* When analyzing for blocking, we identify the possibility of a barrier updating a location accessed by a previous GPU. In coalescing we wish to establish that no control flow needs to be maintained between two GPUs.

- *The way data dependence is used.* For blocking, we use the *possible presence* of data dependence between a barrier and reaching GPUs to block some of the reaching GPUs. For coalescing, we use the *guaranteed absence* of data dependence between the GPUs of a GPB and those reaching it from within a part to coalesce the GPB with the part.

$$\mathsf{CIn}_n := \begin{cases} \mathsf{\textit{false}} & n \text{ is } \mathsf{Start} \\ \bigwedge\limits_{p \in pred(n)} \mathsf{coalesce}(p, n) & \text{otherwise} \end{cases}$$

$$\mathsf{COut}_n := \begin{cases} \mathsf{\textit{false}} & n \text{ is } \mathsf{End} \\ \bigwedge\limits_{s \in succ(n)} \mathsf{CIn}_s & \text{otherwise} \end{cases}$$

$$\mathsf{coalesce}(p, n) \Leftrightarrow \mathsf{COut}_p \wedge \big(\mathsf{GOut}_p = \emptyset \vee \mathsf{gpuFlow}(p, n) \neq \emptyset\big)$$

$$\mathsf{GIn}_n := \begin{cases} \emptyset & n \text{ is } \mathsf{Start} \\ \bigcup\limits_{p \in pred(n)} \mathsf{gpuFlow}(p, n) & \text{otherwise} \end{cases}$$

$$\mathsf{GOut}_n := \begin{cases} \mathsf{GIn}_n \cup \delta_n & \mathsf{CIn}_n = \mathsf{\textit{true}} \\ \delta_n & \text{otherwise} \end{cases}$$

$$\mathsf{gpuFlow}(p, n) := \begin{cases} \emptyset & \neg\mathsf{CIn}_n \wedge \mathsf{DDep}(\mathsf{GOut}_p, \delta_n) \\ \mathsf{GOut}_p & \text{otherwise} \end{cases}$$

$$\begin{aligned} \mathsf{DDep}(X, Y) \Leftrightarrow {} & \big(\mathsf{deref}(X) \vee \mathsf{deref}(Y)\big) \wedge \\ & \big(\mathsf{TDef}(Y) \cup \mathsf{TRef}(Y)\big) \cap \mathsf{TDef}(X - Y) \neq \emptyset \end{aligned}$$

$$\mathsf{deref}(X) \Leftrightarrow \exists\, x \xrightarrow[s]{i|j} y \in X \;\text{ s.t. }\; (i > 1) \vee (j > 1)$$

Definition 8: *Data flow equations for Coalescing Analysis.*

- *Relevant data dependences.* Coalescing removes control flow between two GPUs enabling their non-deterministic execution with respect to each other which is oblivious to any data dependence between the GPUs. Hence, a RaW and WaW dependences need to be preserved by prohibiting coalescing. However, a WaR dependence is not affected by coalescing (see Example 19). On the other hand, blocking by a barrier does not involve RaW dependence because only writes by a barrier are of interest and hence blocking needs to handle only WaW and WaR dependences.

- *The role of dereference in data dependence.* As noted above, for blocking, only the write by a barrier is important and not a read. Hence, we check for a dereference only in the source of a barrier GPU. For coalescing analysis, we need to consider dereferences both in the source and the target.

These differences change the modelling of data dependence for coalescing in the following ways:

- The check for a dereference is now included within the predicate for data dependence.
- Consider a GPB δ_n for coalescing in a part π_i. We now check for both reads and writes in the GPUs of δ_n and only writes in the GPUs of π_i.

Compare the predicates $\overline{\mathsf{DDep}}$ (Definition 6) for blocking and DDep (Definition 8) for coalescing to see the above differences. For establishing the absence of dependence, we match the types of $\gamma_1 \in X$ with the types of $\gamma_2 \in Y$. This is meaningful only when $\gamma_1 \neq \gamma_2$. The term $X - Y$ in the definition of predicate DDep ensures this.

5.4.2.3 Partitioning Analysis

We define two interdependent data flow analyses that inductively

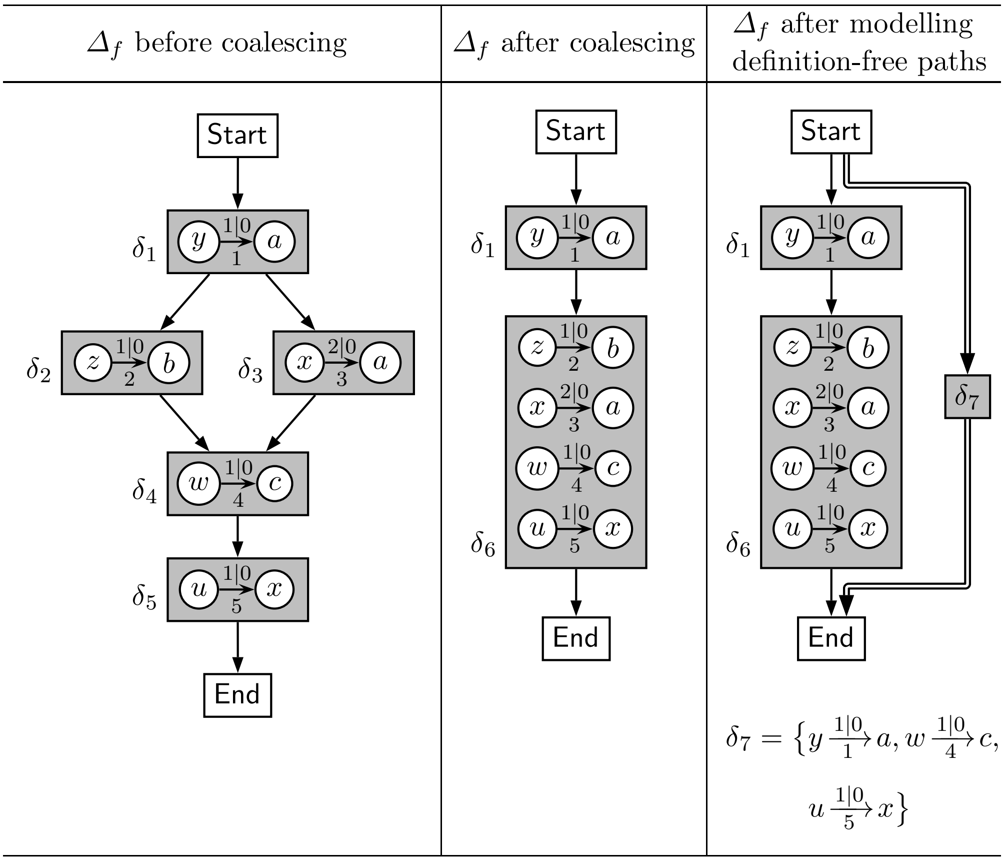
- construct part π_i using data flow variables $\mathsf{CIn}_n/\mathsf{COut}_n$, and
- compute the GPUs accumulated in the part reaching the GPB δ_n in data flow variables $\mathsf{GIn}_n/\mathsf{GOut}_n$.

The latter is required to identify the RaW or WaW data dependence between the GPUs in part π_i.

Unlike the usual data flow variables that typically compute a set of facts, $\mathsf{CIn}_n/\mathsf{COut}_n$ are predicates. If CIn_n is *true*, it indicates that δ_n belongs to the same part as that of *all* of its predecessors. If COut_n is *true*, it indicates that δ_n belongs to the same part as that of *all* of its successors. Thus our analysis does not enumerate the parts as sets of GPBs explicitly; instead, parts are computed implicitly by setting predicates $\mathsf{CIn}/\mathsf{COut}$ of adjacent GPBs.

A GPB δ_n belongs to the same part as that of its predecessor δ_p only if COut_p and CIn_n are set to *true*. However, the values of COut_p and CIn_n are computed by using the set of GPUs GOut_p (which contains all the GPUs that reach δ_p from all the GPBs that are in the part same as δ_p) and δ_n. The function $\mathsf{gpuFlow}$ (Definition 8) checks for the presence of data dependence between the GPUs in GOut_p and δ_n and accordingly sets the values of COut_p and CIn_n.

```
int *** u;    float * z, *w;
int ** x;     int a, b, c;
int * y;
```


Δ_f before coalescing
Δ_f after coalescing
Δ_f after modelling definition-free paths
Start
δ_1
$y \xrightarrow[1]{1|0} a$
δ_2
$z \xrightarrow[2]{1|0} b$
δ_3
$x \xrightarrow[3]{2|0} a$
δ_4
$w \xrightarrow[4]{1|0} c$
δ_5
$u \xrightarrow[5]{1|0} x$
δ_6
δ_7
End

$$\delta_7 = \left\{ y \xrightarrow[1]{1|0} a, w \xrightarrow[4]{1|0} c, u \xrightarrow[5]{1|0} x \right\}$$

Figure 5.3: An example demonstrating the effect of coalescing. Control flow edges with double lines represent a definition-free path for z and pointee of x.

The data flow equations to compute $\mathsf{CIn}_n/\mathsf{COut}_n$ are given in Definition 8. The initialization is *true* for all GPBs. Predicate $\mathsf{coalesce}(p, n)$ uses $\mathsf{gpuFlow}(p, n)$ to check if GPUs in GOut_p are allowed to flow from p to n—if yes, then p and n belong to the same part. If GOut_p is $\emptyset$ (when δ_p is $\emptyset$), they belong to the same part regardless of $\mathsf{gpuFlow}(p, n)$. The presence of COut_p in the equation of $\mathsf{coalesce}$ (Definition 8) ensures that GPB δ_p is considered for coalescing with δ_n only if δ_p has not been found to be a "boundary" in coalescing because it cannot coalesce with some successor.

Name for GPUs. Statement ids do not matter									
γ_1	$y \xrightarrow[1]{1\vert 0} a$	γ_2	$z \xrightarrow[2]{1\vert 0} b$	γ_3	$x \xrightarrow[3]{2\vert 0} a$	γ_4	$w \xrightarrow[4]{1\vert 0} c$	γ_5	$u \xrightarrow[5]{1\vert 0} x$

GPB n	TDef (n)	TRef (n)	GIn_n	GOut_n	CIn_n	COut_n
δ_1	$\{$int $*\}$	$\emptyset$	$\emptyset$	$\{\gamma_1\}$	F	F
δ_2	$\{$float $*\}$	$\emptyset$	$\{\gamma_1\}$	$\{\gamma_2\}$	F	T
δ_3	$\{$int $*,$ int $**\}$	$\emptyset$	$\{\gamma_1\}$	$\{\gamma_3\}$	F	T
δ_4	$\{$float $*\}$	$\emptyset$	$\{\gamma_2, \gamma_3\}$	$\{\gamma_2, \gamma_3, \gamma_4\}$	T	T
δ_5	$\{$int $***\}$	$\emptyset$	$\{\gamma_2, \gamma_3, \gamma_4\}$	$\{\gamma_2, \gamma_3, \gamma_4, \gamma_5\}$	T	F

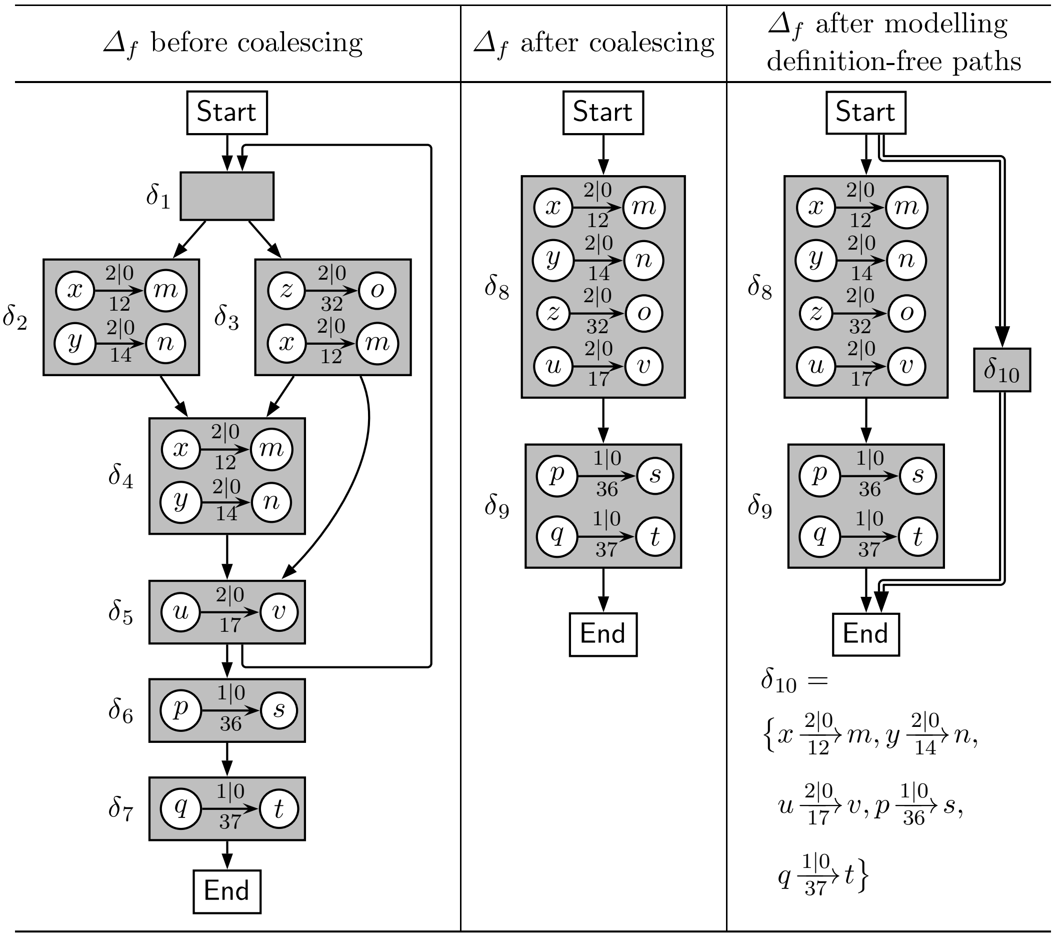
Figure 5.4: The data flow information computed by coalescing analysis for example in Figure 5.3. The CIn and COut values indicate that GPBs δ_2, δ_3, δ_4, δ_5 can be coalesced.

Another striking difference between the equations for CIn/COut in Definition 8 and the usual data flow equations is that the data flow variables CIn_n and COut_n for GPB n are independent of each other—CIn_n depends only on the COut of its predecessors and COut_n depends only on the CIn of its successors. Intuitively, this form of data flow equations attempts to *melt* the boundaries of GPB n to explore fusing it with its successors and predecessors as follows:

- When CIn_n is true, it melts the boundary at the top of the GPB and glues it with all its predecessors that are already in the part. This deletes the in-edges of n and the part grows in a forward direction.
- When COut_n is true, it melts the boundary at the bottom of the GPB and includes all its successors in the part. This deletes the out-edges of n and the part grows in the forward direction.

The incremental expansion of a part in a forward direction influences the flow of GPUs accumulated in a part leading to a forward data flow analysis using data flow variables $\mathsf{GIn}_n/\mathsf{GOut}_n$. The data flow equations to compute them are given in Definition 8. Function $\mathsf{gpuFlow}(p, n)$ in the equation for GIn computes the set of GPUs that flow from p to n. It establishes the absence of data dependences using predicate DDep defined

```
int **x;       int ***u;
float **y;     int *p, *q, *v;
short **z;     int m, n, o, s, t;
```



$\varDelta_f$ before coalescing
$\varDelta_f$ after coalescing
$\varDelta_f$ after modelling definition-free paths
Start
δ_1
δ_2
$x \xrightarrow[12]{2|0} m$
$y \xrightarrow[14]{2|0} n$
δ_3
$z \xrightarrow[32]{2|0} o$
$x \xrightarrow[12]{2|0} m$
δ_4
$x \xrightarrow[12]{2|0} m$
$y \xrightarrow[14]{2|0} n$
δ_5
$u \xrightarrow[17]{2|0} v$
δ_6
$p \xrightarrow[36]{1|0} s$
δ_7
$q \xrightarrow[37]{1|0} t$
End
Start
δ_8
$x \xrightarrow[12]{2|0} m$
$y \xrightarrow[14]{2|0} n$
$z \xrightarrow[32]{2|0} o$
$u \xrightarrow[17]{2|0} v$
δ_9
$p \xrightarrow[36]{1|0} s$
$q \xrightarrow[37]{1|0} t$
End
Start
δ_8
$x \xrightarrow[12]{2|0} m$
$y \xrightarrow[14]{2|0} n$
$z \xrightarrow[32]{2|0} o$
$u \xrightarrow[17]{2|0} v$
δ_{10}
δ_9
$p \xrightarrow[36]{1|0} s$
$q \xrightarrow[37]{1|0} t$
End
$\delta_{10} = \{x \xrightarrow[12]{2|0} m, y \xrightarrow[14]{2|0} n, u \xrightarrow[17]{2|0} v, p \xrightarrow[36]{1|0} s, q \xrightarrow[37]{1|0} t\}$

Figure 5.5: An example demonstrating the effect of coalescing. The loop formed by the back edge $\delta_5 \rightarrow \delta_1$ reduces to a self loop over GPB δ_8 after coalescing. Since self loops are redundant, they are eliminated. Double lines represent definition-free paths.

in Section (5.4.2.2). If no data dependence exists, the GPUs accumulated in GOut_p are propagated to n (i.e., GIn_n). The presence of $\neg\mathsf{CIn}_p$ in equation for $\mathsf{gpuFlow}$ ensures that GPUs in GOut_p are propagated to δ_n only if δ_n has not been found to be a "boundary" in coalescing because it cannot coalesce with some predecessor.

Example 44. Figure 5.4 gives the data flow information for the example of Figure 5.3. Even though there is no data dependence between the GPUs in GPBs δ_1 and δ_2 (indicated by $\mathsf{DDep}(\mathsf{GOut}_1, \delta_2)$ returning *false*), they cannot be coalesced. This is

Name for GPUs. Statement ids do not matter					
γ_1 $x \xrightarrow[12]{2\|0} m$	γ_2 $y \xrightarrow[14]{2\|0} n$	γ_3 $z \xrightarrow[32]{2\|0} o$	γ_4 $u \xrightarrow[17]{2\|0} v$	γ_5 $p \xrightarrow[36]{1\|0} s$	γ_6 $q \xrightarrow[37]{1\|0} t$

GPB n	TDef (n)	TRef (n)	GIn$_n$	GOut$_n$	CIn$_n$	COut$_n$
δ_1	$\emptyset$	$\emptyset$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	F	T
δ_2	{int∗, float∗}	$\emptyset$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	T	T
δ_3	{short∗, int∗}	$\emptyset$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	T	T
δ_4	{int∗, float∗}	$\emptyset$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	T	T
δ_5	{int∗∗}	$\emptyset$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	T	F
δ_6	{int∗}	$\emptyset$	$\emptyset$	$\{\gamma_5\}$	F	T
δ_7	{int∗}	$\emptyset$	$\{\gamma_5\}$	$\{\gamma_5, \gamma_6\}$	T	F

Figure 5.6: The data flow information computed by coalescing analysis for example in Figure 5.5. The CIn and COut values indicate that GPBs δ_1, δ_2, δ_3, δ_4, δ_5 can be coalesced. Similarly, GPBs δ_6 and δ_7 can be coalesced. GPBs δ_5 and δ_6 must remain in different coalesced groups.

because there is data dependence between the GPUs in the GPBs δ_1 and its other successor δ_3 (indicated by DDep(GOut$_1, \delta_3$) returning *true* because typeof $(x, 2)$ matches with typeof$(y, 1)$). Since, a successor of a GPB cannot be coalesced with the GPB unless all the successors of the GPB can be coalesced with it, δ_2 cannot be coalesced with δ_1. Thus, GOut$_2 = \{z \xrightarrow[2]{1|0} b\}$ and GOut$_3 = \{x \xrightarrow[3]{2|0} a\}$ and the GPU $y \xrightarrow[1]{1|0} a$ is not included in the sets.

Since the GPUs in GOut$_2$ and δ_4 does not contain a dereference, it is easy to identify that there is no data dependence between the GPUs. On the other hand, the GPUs in GOut$_3$ contain a dereference and we check the data dependence between the GPUs in GOut$_3$ and δ_4 using the type information. The check DDep(GOut$_3, \delta_4$) returns *false* indicating that there is no type matching and hence no possibility of data dependence, thereby allowing the coalescing of the two GPBs. Since there is no data dependence between the GPUs in the GPB δ_4 and both its predecessors δ_2 and δ_3, the GPBs can be coalesced. Similarly, GPBs δ_4 and δ_5 can be coalesced. Thus COut$_2$, COut$_3$, CIn$_4$, COut$_4$, and CIn$_5$ are *true* and GPBs δ_2, δ_3, δ_4, and δ_5 belong to the same part and are coalesced into a new GPB δ_6.

Examples 45 and 46 illustrate coalescing of GPBs in a loop.

Example 45. Figure 5.6 gives the data flow information for the example of Figure 5.5. GPBs δ_1 and δ_2 can be coalesced because COut_1 is *true* and GOut_1 is $\emptyset$. Thus, $\mathsf{DDep}(\mathsf{GOut}_1, \delta_2)$ returns *false* indicating that types do not match and hence there is no possibility of a data dependence between the GPUs of δ_1 and δ_2. Similarly, GPBs δ_1 and δ_3 can be coalesced. Thus COut_1, CIn_2, and CIn_3 are *true*. We check the data dependence between the GPUs of GPBs δ_2 and δ_4 using the type information. However, $\mathsf{DDep}(\mathsf{GOut}_2, \delta_4)$ returns *false* because the term $(\mathsf{GOut}_2 - \delta_4)$ is $\emptyset$. Thus, GPBs δ_2 and δ_4 belong to the same part and can be coalesced. For GPBs δ_3 and δ_4, the possibility of data dependence is resolved based on the type information. The term $(\mathsf{GOut}_3 - \delta_4)$ returns $z \xrightarrow[32]{2|0} o$ whose $\mathsf{typeof}(z, 1)$ does not match that of the pointers being read in the GPUs in δ_4. Thus, GPBs δ_3 and δ_4 can be coalesced. GPBs δ_4 and δ_5 both contain a GPU with a dereference, however $\mathsf{DDep}(\mathsf{GOut}_4, \delta_5)$ returns *false* indicating that there is no type matching and hence no possibility of data dependence, thereby allowing the coalescing of the two GPBs. The $\mathsf{DDep}(\mathsf{GOut}_5, \delta_6)$ returns *true* (type of source of the GPU $x \xrightarrow[12]{2|0} m \in \mathsf{GOut}_5$ matches the source of the GPU $p \xrightarrow[36]{1|0} s \in \delta_6$) indicating a possibility of data dependence in the caller through aliasing and hence the two GPBs cannot be coalesced. Thus, the first part in the partition contains only GPBs δ_1, δ_2, δ_3, δ_4, and δ_5. GPB δ_6 now marks the first GPB of the new part. GPBs δ_6 and δ_7 can be coalesced as there is no data dependence between their GPUs. The loop $\delta_5 \rightarrow \delta_1$ before coalescing now reduces to self loop over GPB δ_8 after coalescing. The self loop is redundant and hence eliminated. GPBs δ_5 and δ_1 can be coalesced because all the GPBs of the loop belong to the same part.

Observe that some GPUs appear in multiple GPBs of a GPG (before coalescing). This is because we could have multiple calls to the same procedure. Thus, even though the GPBs are renumbered, the statement labels in the GPUs remain unchanged resulting in repetitive occurrence of a GPU. This is a design choice because it helps us to accumulate the points-to information of a particular statement in all contexts.

Example 46. In the example of Figure 3.5, GPBs δ_1 and δ_2 can be coalesced because $\mathsf{DDep}(\mathsf{GOut}_1, \delta_2)$ returns *false* indicating that there is no type matching and hence no possible data dependence between their GPUs. Thus, COut_1 and CIn_2 are set to *true*. The loop formed by the back edge $\delta_2 \rightarrow \delta_1$ reduces to a self loop over GPB δ_{11} after coalescing. The self loop is redundant and hence it is eliminated. For GPBs δ_2 and δ_4, $\mathsf{DDep}(\mathsf{GOut}_2, \delta_4)$ returns *true* because $\mathsf{typeof}(q, 2)$ (for the GPU $q \xrightarrow[02]{2|0} m$ in δ_{02}) matches $\mathsf{typeof}(p, 2)$ (for the GPU $e \xrightarrow[04]{1|2} p$ in δ_{04}) which is `int *`. This indicates the possibility of a data dependence between the GPUs of GPBs δ_2 and δ_4 (q and p could be aliased in the caller) and hence these GPBs cannot be coalesced. Thus, COut_2 and CIn_4 are set to *false*. For GPBs δ_4 and δ_5, $\mathsf{DDep}(\mathsf{GOut}_4, \delta_5)$ returns *false* because there is no possible data dependence. Hence COut_4 and CIn_5 are set to *true* and the two GPBs can be coalesced.

Recall that our coalescing heuristics requires us to prohibit

- coalescing with Start and End GPBs so that the definition-free paths that were subsumed because of coalescing other GPBs can be modelled, and
- coalescing of the source and target GPBs of a back edge unless all GPBs in the loop formed by the back edge are included in the same part.

The data flow equations for Coalescing (CIn/COut in Definition 8) do not have any provision of these requirements; they are enforced separately during the actual transformation.

5.4.2.4 Preserving Definition-Free Paths

Coalescing can have the side effect of eliminating definition-free paths. Consider a GPU γ that reaches the exit of a GPG along some path but not all. It means that there is some path in the GPG along which the source of γ is not defined (i.e., the source of γ is *may*-defined in the GPG). According to our heuristics of coalescing, a GPB is coalesced either with all its successors or with none. Hence, after coalescing with all successors, a definition-free path may get subsumed and γ may reach the exit of a GPG along all paths indicating that the source of γ is now *must*-defined. This would lead to a strong update instead of a weak update thereby introducing unsoundness. Hence, we need to add an explicit definition-free path for such GPUs. The GPUs with definition-free paths are identified by the corresponding boundary definitions. A definition-free path for the source of GPU : $x \xrightarrow[s]{i|j} y$ exists in a GPG only if the boundary definition $x \xrightarrow[00]{i|i} x'$ reaches the exit of the GPG.

Example 47 illustrates the modelling of definition-free path after coalescing for the example in Figure 5.3.

Example 47. In the example of Figure 5.3, the definition-free path is shown by edges with double lines in the GPG obtained after coalescing. The GPU $z \xrightarrow[2]{1|0} b$ does not reach the exit along the path $\delta_1 \rightarrow \delta_3 \rightarrow \delta_4 \rightarrow \delta_5$ which forms the definition-free path. Similarly, the GPU $x \xrightarrow[3]{2|0} a$ does not reach the exit along the path $\delta_1 \rightarrow \delta_2 \rightarrow \delta_4 \rightarrow \delta_5$ indicating a definition-free path. We add a definition-free path between Start and End GPBs of a GPG with a GPB that contains all GPUs that do not have any definition-free path. Thus, we have a GPB δ_7 which contains all GPUs except $z \xrightarrow[2]{1|0} b$ and $x \xrightarrow[3]{2|0} a$.

Example 48 illustrates the modelling of definition-free path after coalescing for the example in Figure 5.5.

Example 48. In the example of Figure 5.5, the definition-free path is shown by edges with double lines in the GPG obtained after coalescing. The GPU $z \xrightarrow[32]{2|0} o$ does not reach the exit along the path $\delta_1 \rightarrow \delta_2 \rightarrow \delta_4 \rightarrow \delta_5 \rightarrow \delta_6 \rightarrow \delta_7$ which forms the definition-free path. We add a definition-free path between Start and End GPBs of a GPG with a GPB that contains all GPUs that do not have any definition-free path. Thus, we have a GPB δ_{10} which contains all GPUs except $z \xrightarrow[32]{2|0} o$.

The example below illustrates the modelling of definition-free path after coalescing for the the motivating example in Figure 3.2.

Example 49. In Figures 3.5 and 3.6, definition-free paths are shown by edges with double lines in the GPGs of procedures f and g obtained after coalescing. For procedure g, the GPUs $b \xrightarrow[02]{1|0} m$ and $q \xrightarrow[02]{2|0} m$ undergo a weak update and hence do not kill their corresponding boundary definitions. This indicates that the source of these GPUs are *may*-defined and hence a definition-free path is required for these GPUs.

Thus, we add a definition-free path between Start and End GPBs of Δ_g with GPB δ_{16} which contains the set of GPUs $\{r \xrightarrow[01]{1|0} a, e \xrightarrow[04]{1|2} p, q \xrightarrow[05]{1|0} e\}$.

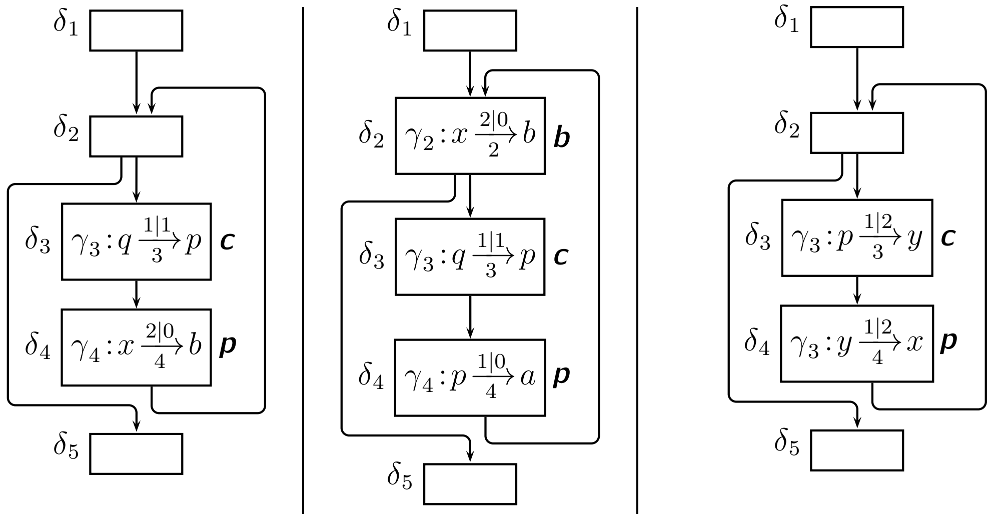
For procedure f, the boundary definition $b \xrightarrow[00]{1|1} b'$ reaches the exit of Δ_f indicating that b is *may*-defined. Hence a definition-free path is added with GPB δ_{17} containing all GPUs of Δ_f except $b \xrightarrow[02]{1|0} m$. GPU $q \xrightarrow[02]{2|0} m$, which has a definition-free path in Δ_g, reduces to $d \xrightarrow[02]{1|0} m$ in Δ_f. However, d is defined in δ_{08} also, hence it does not have a definition-free path in Δ_f.

Coalescing is most effective for recursive procedures. This is because the size of the GPG of a procedure tends to increase exponentially because of multiple inlinings of recursive calls until a fixed point is achieved. In such a case, the number of unique GPUs in a GPG is small but the number of GPBs is large. Coalescing helps to construct a compact GPG for such procedures.

Note that we do not need to preserve definition-free paths in the first approach because the dominance and post-dominance relation between adjacent GPBs ensures that no existing definition-free path is ever subsumed.

5.4.3 Coalescing Facilitates Sparse Analysis

Recall that we use the partial SSA created by GCC to construct the initial GPG. The assignments defining SSA variables are ignored and the assignments involving the use/read of SSA variables are handled using use-def chains of SSA. Thus, a GPG of a procedure does not retain GPUs containing local variables (because local variables are not in the scope of the callers). We traverse the SSA chains transitively until we reach a statement whose right hand side has an address-taken variable, a global variable, or a formal parameter. SSA resolution is a very efficient implementation of GPU composition in the setting of SSA variables; we just need to follow the SSA edges to eliminate the pivot. This is similar to semi-sparse analysis [32] where the information is propagated along def-use chains for top-level pointer variables and along the control flow edges for the rest of the pointer variables. Semi-sparse analysis does not eliminate control flow between assignments that use non-SSA variables, which we do. Hence, our method is much efficient than the semi-sparse method. Our approach connects all definitions of (local) pointer variables to their respective uses similar to a sparse analysis [33].


δ_1
δ_2
δ_3 $\gamma_3 : q \xrightarrow[3]{1|1} p$ c
δ_4 $\gamma_4 : x \xrightarrow[4]{2|0} b$ p
δ_5
δ_1
δ_2 $\gamma_2 : x \xrightarrow[2]{2|0} b$ b
δ_3 $\gamma_3 : q \xrightarrow[3]{1|1} p$ c
δ_4 $\gamma_4 : p \xrightarrow[4]{1|0} a$ p
δ_5
δ_1
δ_2
δ_3 $\gamma_3 : p \xrightarrow[3]{1|2} y$ c
δ_4 $\gamma_3 : y \xrightarrow[4]{1|2} x$ p
δ_5

(a) Indirect GPU

(b) Blocked by barrier

(c) *Undesirable* GPU composition

Figure 5.7: Examples characterizing the producer GPUs that suppress dead GPU elimination and back-edge removal optimizations. The consumer, producer, and barrier GPUs are denoted by ***c***, ***p***, and ***b***, respectively.

5.5 Back-Edge Removal

A GPG Δ is constructed by incorporating the effect of loops in the GPG through a fixed-point computation during the reaching GPUs analyses. When Δ is inlined in its callers, a GPB δ contained in a loop in Δ would be considered repeatedly for constructing the GPGs of its callers because the loop would also be included in the GPG of the callers. This repeated fixed-point processing is redundant in many cases and can be beneficially avoided by removing back edges from Δ before it is inlined in its callers. This section characterizes the situations when it is possible to do so.

We begin by examining the situations in which a producer GPU ***p*** satisfies all of the following conditions: (a) ***p*** is contained in loop in Δ, (b) ***p*** is likely to be required to simplify a consumer GPU ***c***, (c) ***c*** also belongs to Δ, (d) ***c*** $\circ^{\tau}$ ***p*** is performed after Δ is inlined in a caller. We then identify the situations when ***p*** requires the presence of back edges to reach ***c***. When no such consumer-producer pair of GPUs in Δ requires a particular back edge, the back edge is redundant and can be removed.

5.5.1 Characterizing Prospective Producer GPUs

In the context of back-edge removal, the following two cases may arise when a consumer GPU $\boldsymbol{c}$ in a GPG Δ is blocked from being composed with a producer GPU $\boldsymbol{p}$ in Δ, but may compose after Δ is inlined in the body of the caller:

(a) GPU $\boldsymbol{p}$ may be a producer GPU whose composition has been postponed either because it is blocked or the composition is *undesirable*. These GPUs are recorded in set Queued for every GPG. It is computed during GPU reduction (Definition 7 in section 5.3).

(b) Composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ is not possible because there is no pivot of composition; in this case, if $\boldsymbol{p}$ is an indirect GPU, it is possible that it may be simplified after Δ is inlined in a caller enabling the composition of the resulting GPU with $\boldsymbol{c}$ in the caller's body.

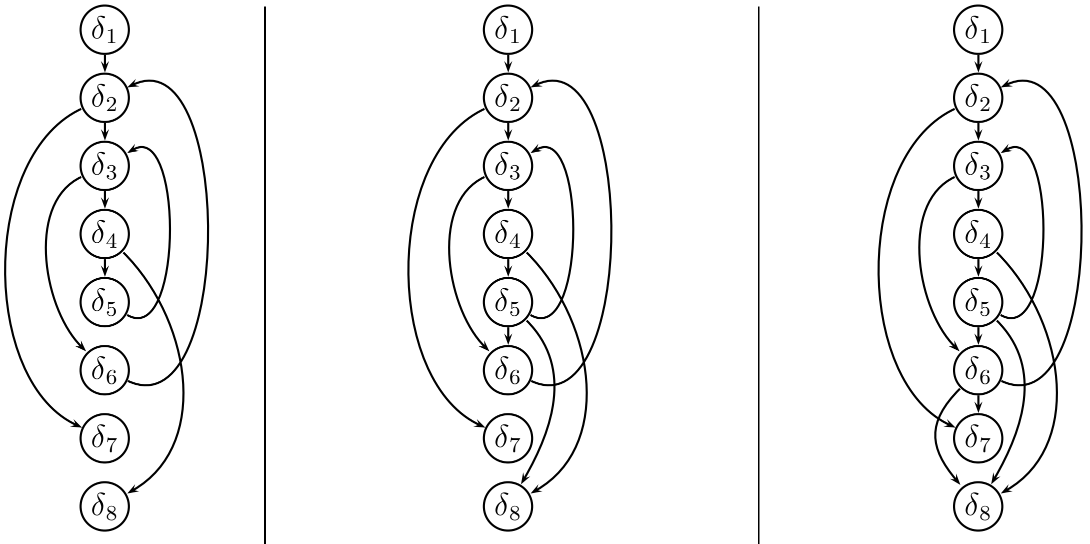
In each of these cases, if $\boldsymbol{p}$ can reach $\boldsymbol{c}$ only when a back edge is traversed, the back edge cannot be removed. In such a case, we say that the back edge is *essential.*

Example 50 illustrates how to identify essential back edges.

Example 50. Consider the example in Figure 5.7(a) where an indirect GPU $x \xrightarrow[4]{2|0} b$ reaches the GPB δ_3 along the back edge $\delta_4 \rightarrow \delta_2$. The GPU $q \xrightarrow[3]{1|1} p$ cannot be composed with $x \xrightarrow[4]{2|0} b$ because there is no pivot between them. However, if x points to p in the caller then the GPU $x \xrightarrow[4]{2|0} b$ is simplified to $p \xrightarrow[4]{1|0} b$ after inlining at the call site. The simplified GPU can then compose with $q \xrightarrow[3]{1|1} p$ which is simplified to give $q \xrightarrow[3]{1|0} b$. Thus, GPU in δ_4 may be required at δ_3 and hence the back edge $\delta_4 \rightarrow \delta_2$ cannot be eliminated.

In GPB δ_3 in Figure 5.7(b), the composition between $\boldsymbol{c} : q \xrightarrow[3]{1|1} p$ and $\boldsymbol{p} : p \xrightarrow[4]{1|0} a$ is postponed because the barrier GPU $x \xrightarrow[2]{2|0} b$ blocks $\boldsymbol{p}$ which reaches $\boldsymbol{c}$ along the back edge $\delta_4 \rightarrow \delta_2$. However, the blocked GPU may be required for composition once the barrier GPU is simplified. Hence the back edge $\delta_4 \rightarrow \delta_2$ is essential for the GPU to reach a consumer for a possible composition in the callers.

In GPB δ_3 Figure 5.7(c), the composition between GPUs $\boldsymbol{c} : p \xrightarrow[3]{1|2} y$ and $\boldsymbol{p} : y \xrightarrow[4]{1|2} x$ is *undesirable* because the result of composition is a GPU $p \xrightarrow[3]{1|3} x$ whose *indlev* exceeds that of $\boldsymbol{c}$. This composition may be performed if the GPU $\boldsymbol{p}$ is simplified. Hence the back edge $\delta_4 \rightarrow \delta_2$ is essential because the simplified form of GPU $y \xrightarrow[4]{1|2} x$ reaching along the back edge is required for composition later.



(a) Original GPG

(b) After making the inner loop regular

(c) After making outer loop regular

Figure 5.8: Converting single-entry irregular loops into regular loops. GPBs δ_4 and δ_5 get included in the outer loop after the inner loop is made regular.

We define a set of *prospective producer* GPUs for which essential-back-edges analysis is performed. This set is computed for every GPG by taking the union of Queued (case (a)) and all indirect GPUs generated across all GPBs δ_s (case (b)):

$$\mathsf{Prospective\_Producer\_GPUs} = \mathsf{Queued} \cup \{\gamma \mid \gamma \in \mathsf{RGGen}_s, \gamma \text{ is an indirect GPU}\} \quad (5.2)$$

5.5.2 Different Possibilities for Handling Postponed GPU Compositions

A GPU composition is postponed because it is *undesirable* or the producer GPU is blocked by a barrier. In either case, we may need to retain back edges in a GPG requiring a fixed-point computation when the GPG is inlined in a caller. A composition postponed by a barrier may lead to unsoundness but an *undesirable* composition does not. For the latter, we hence have two possibilities. Either,

- we perform *undesirable* GPU compositions and eliminate the back edges, or
- we do not perform *undesirable* GPU compositions and retain the back edges.

Since pointers to scalars cannot be used for creating recursive data structures, the fixed-point computation for them is not expensive. Hence, we choose the latter option for them. However, pointers to structures and heap could be a part of recursive data structures that are unbounded. Hence, fixed-point computation may be expensive and we choose the former option (see Section 8.4).

For GPU compositions that are postponed because of the presence of a barrier, we must retain the back edges until the barrier GPUs are sufficiently simplified.

5.5.3 Loop and Loop Characteristics

We define the *loop* of a back edge $\delta_s \rightarrow \delta_t$ in a GPG as a strongly connected component (SCC) consisting of the set of GPBs appearing on all control flow paths starting at δ_t (the *first* GPB of the loop) and ending on δ_s (the *last* GPB of the loop) such that no control flow path involves a back edge. The *exit* of a loop is a GPB in the loop with a successor that is not in the loop. The *successors* of a loop are the successor GPBs of its *exit*s. A loop is *single-entry* if its *first* GPB dominates its *last* GPB. A loop is *regular* if it is single-entry and its *last* GPB is also an *exit* of the loop. A regular loop is *strongly regular* if its *last* GPB is the only *exit* of the loop. A regular loop could have a premature *exit* but a strongly regular loop cannot.

Example 51. A C-style **while** loop is not regular because its *last* GPB (i.e. the source of the back edge) is not an *exit* of the loop. A **do-while** loop is regular; it is strongly regular if it does not contain a **break** statement or a **goto** statement to a label outside the loop. For the GPG in Figure 5.8(a), the loops of back edges $\delta_5 \rightarrow \delta_3$ and $\delta_6 \rightarrow \delta_2$ are not regular because their *last* GPBs (δ_5 and δ_6) are not their *exit*s. Both the loops have a single entry. Further, in the case of nested **while** loops, an outer **while** loop may not contain some GPBs of the inner **while** loop. In our example, if the inner loop is not made regular, δ_5 does not appear on any control flow path from δ_2 to δ_6 that does not pass through any back edge.

We are interested in regular loops because, unlike an irregular loop, the GPUs in a regular loop may reach its successor GPBs without traversing the back edge corresponding to the loop. Hence, a regular loop may not require its back edge for propagating its GPUs to the loop successors. This leads to the following two design choices:

- We restrict ourselves to single-entry SCCs for the purpose of back-edge removal. A multi-entry SCC is *irreducible* [35, 36, 37]. Since such an SCC does not have a unique *first* node, back edges cannot be identified uniquely and a cycle starting at some *first* node in the SCC could be completed by adding an edge identified as a forward edge rather than an edge marked as a back edge. Thus, the removal of such a back edge could under-approximate the acyclic paths from *first* nodes to *last* nodes leading to unsoundness. Hence, we choose not to consider a back edge for removal if it is contained in an irreducible SCC.

- We convert a single-entry irregular loop into a regular loop by adding edges from the *last* GPB of the loop to its successors. These edges are necessarily forward edges.

Note that, by default, the GPBs of an inner **while** loop are not part of the outer **while** loop. They get included in the outer **while** loop after the inner **while** loop is converted to a regular loop. Since this conversion updates outer loops by including more GPBs, correctness requires the conversion to be done innermost-first to ensure that all GPBs in an inner loop get included in the outer loop. Since we restrict back-edge-removal optimizations to reducible SCCs, the process of conversion begins with a loop whose *first* GPB has the greatest post-order number—it is guaranteed to be dominated by the *first* GPBs of all enclosing loops. This ensures that inner loops are processed for conversion before outer loops.

Example 52. For the GPG in Figure 5.8(a), the inner loop with back edge $\delta_5 \rightarrow \delta_3$ is made regular by adding edges $\delta_5 \rightarrow \delta_6$ and $\delta_5 \rightarrow \delta_8$ (Figure 5.8(b)) so that the *last* GPB has successors δ_6 and δ_8 which are not in the loop and hence the GPB δ_5 now becomes the *exit* of the inner loop.

Observe that the set of GPBs contained in the loop represented by the back edge $\delta_6 \rightarrow \delta_2$ is $\{\delta_2, \delta_3, \delta_6\}$ before the inner loop is made regular (Figure 5.8(a)). It does not contain the GPBs δ_4 and δ_5 as there is no path which starts at δ_2 and ends on δ_6 passing through δ_4 and δ_5 without traversing a back edge. With the addition of edge $\delta_5 \rightarrow \delta_6$, the set of GPBs in the loop of $\delta_6 \rightarrow \delta_2$ becomes $\{\delta_2, \delta_3, \delta_4, \delta_5, \delta_6\}$ (Figure 5.8(b)).

$$\begin{aligned}
\mathsf{EBIn}_s &:= \mathsf{Compress}\left(\bigcup_{t \in \mathit{pred}(s)} \mathsf{Edgeflow}\,(t \rightarrow s, \mathsf{EBOut}_t)\right) \\
\mathsf{EBOut}_s &:= \mathsf{Gen}_s \;\cup\; \mathsf{Filter}\,(s,\; \mathsf{EBIn}_s - \mathsf{Kill}_s) \\
\mathsf{Gen}_s &:= \{\langle\gamma, \emptyset\rangle \mid \gamma \in \mathsf{RGGen}_s \cap \mathsf{Prospective\_Producer\_GPUs}\} \\
\mathsf{Kill}_s &:= \{\langle\gamma, \beta\rangle \mid \gamma \in \mathsf{RGKill}_s\}
\end{aligned}$$

Definition 9: *Useful back edges analysis for computing* EBOut_s

We make the outer loop (i.e. the loop of the back edge $\delta_6 \rightarrow \delta_2$) regular by adding edges $\delta_6 \rightarrow \delta_7$ and $\delta_6 \rightarrow \delta_8$ so that the *last* GPB δ_6 also becomes the *exit* of the loop as shown in Figure 5.8(c). Note that we could add the edge $\delta_6 \rightarrow \delta_8$ because the GPB δ_4 has been made a part of the loop by making the inner loop regular. Further, its successor δ_8 which is not a part of the loop is the successor of the outer loop.

5.5.4 Essential Back Edges Analysis

A GPU $\gamma \in \mathsf{Prospective\_Producer\_GPUs}$ (Equation 5.2) may be required for a later composition. Hence, if γ is contained in a loop, the corresponding back edge may have to be retained. This situation arises if the back edge is required for γ to reach its consumer GPU. We call such a back edge an essential back edge. In order to identify essential back edges, we perform a data flow analysis that computes a set EBIn_s for each statement s. EBIn_s is a set of pairs $\langle\gamma, \beta\rangle$ such that γ is a prospective producer GPU and β is a set of back edges such that γ reaches s only after traversing some back edge in β. In other words, if γ could reach the program point along a back-edge-free control flow path, then β is $\emptyset$. We use this information in the following manner: if $\langle\gamma, \beta\rangle$ reaches a consumer GPU γ, then the edges in β are essential and should not be removed from the GPG.

The data flow equations for computing EBIn_i and EBOut_i are given in Definition 9. Observe that this analysis does not associate a set of back edges with all GPUs. It needs to be done only for the prospective producer GPUs characterized in Section 5.5.1. We use the following auxiliary functions:

- Function $\mathsf{Edgeflow}\,(m \rightarrow n, X)$ adds edge $m \rightarrow n$ to the set of back edges of the

GPUs in X if the edge $m \rightarrow n$ is a back edge.

$$\mathsf{Edgeflow}\,(m \rightarrow n, X) = \begin{cases} X & m \rightarrow n \text{ is a forward edge} \\ \{\langle \gamma, \beta \cup \{m \rightarrow n\}\rangle \mid \langle \gamma, \beta\rangle \in X\} & \text{otherwise} \end{cases}$$

- Function $\mathsf{Filter}(m, X)$ removes a back edge $m \rightarrow n$ from β when $\langle \gamma, \beta\rangle$ reaches node m because it has already gone over $m \rightarrow n$ once. In other words, $\langle \gamma, \beta\rangle$ has completed a cycle and has reached the end of the loop. If it goes over a forward edge coming out of m, it can go out of m without traversing the back edge $m \rightarrow n$.

$$\mathsf{Filter}\,(m, X) = \{\langle \gamma, \beta - \{m \rightarrow n\}\rangle \mid \langle \gamma, \beta\rangle \in X, m \rightarrow n \text{ is a back edge}\}$$

- Function $\mathsf{Compress}(X)$ compresses set X of pairs $\langle \gamma, \beta\rangle$ by partitioning it on the basis of GPUs using function $\mathsf{Select}\,(\gamma, X)$ and then merging each equivalence class into a single pair using function $\mathsf{Merge}\,(Y)$.

$$\mathsf{Compress}\,(X) = \big\{\mathsf{Merge}\,(\mathsf{Select}\,(\gamma, X)) \mid \langle \gamma, \beta\rangle \in X\big\}$$
$$\mathsf{Select}\,(\gamma, X) = \{\langle \gamma, \beta\rangle \mid \langle \gamma, \beta\rangle \in X\}$$

- Function $\mathsf{Merge}\,(Y)$ takes as an argument a set $Y \subseteq X$ of all pairs involving GPU γ and computes a single pair for γ defined as follows:
  - If a pair $\langle \gamma, \emptyset\rangle$ exists in Y, there is a back-edge-free path along which γ can reach the program point. In such a case, Merge discards all back edges associated with γ and returns a single pair $\langle \gamma, \emptyset\rangle$.
  - If β is not empty in any pair in Y, it means that every path along which γ has reached the program point contains a back edge. In such a situation, Merge takes a union of all back edge sets and returns a single pair $\langle \gamma, \beta'\rangle$ such that β' is non-empty.

$$\mathsf{Merge}\,(Y) = \begin{cases} \langle \gamma, \emptyset\rangle & \exists\ \langle \gamma, \emptyset\rangle \in Y \\ \langle \gamma, \{m \rightarrow n \mid \langle \gamma, \beta\rangle \in Y, m \rightarrow n \in \beta\}\rangle & \text{otherwise} \end{cases}$$

The following two examples illustrate essential back edges analysis using the GPGs in Figure 5.8. Although, the picture does not show any GPU, we assume the presence of suitable GPUs for the purpose of illustration.

Example 53. Consider the case where GPB $\delta_5 = \{\gamma_5 : p \xrightarrow[5]{1|0} a\}$ and GPB $\delta_6 = \{\gamma_6 : p \xrightarrow[6]{1|0} b\}$ in all the three GPGs. For simplicity, assume that all other GPBs can be ignored. In the original GPG given in Figure 5.8(a) where the loops are not made regular, the pairs $\langle \gamma_5, \{\delta_5 \rightarrow \delta_3, \delta_6 \rightarrow \delta_2\}\rangle$ and $\langle \gamma_6, \{\delta_6 \rightarrow \delta_2\}\rangle$ reach the GPB δ_8. When the inner loop is made regular by adding the edges $\delta_5 \rightarrow \delta_6$ and $\delta_5 \rightarrow \delta_8$ (Figure 5.8(b)), there exists a path ($\delta_5 \rightarrow \delta_8$) along which the GPU γ_5 reaches δ_8 without traversing any back edges and hence $\langle \gamma_5, \emptyset\rangle$ reaches δ_8. However, the GPU γ_6 still needs a back edge to reach δ_8 and hence the pair $\langle \gamma_6, \{\delta_6 \rightarrow \delta_2\}\rangle$ continues to reach δ_8. The set β associated with γ_6 becomes $\emptyset$ after the outer loop is made regular with the addition of edge $\delta_6 \rightarrow \delta_8$.

After computing the minimum fixed point of EBIn_s, we define the set of essential back edges in Δ using the result of essential back edges analysis as follows:

$$\mathsf{EssentialBackEdges} = \{b \mid \langle \gamma, \beta \rangle \in \mathsf{EBIn}_i, b \in \beta, \gamma \in \mathsf{Prospective\_Producer\_GPUs}\}$$

A back edge that is not essential can be deleted from Δ.

Example 54. Continuing with Figure 5.8(c), consider the case where GPB $\delta_2 = \{\gamma_2 : x \xrightarrow[2]{2|0} b\}$, GPB $\delta_4 = \{\gamma_4 : q \xrightarrow[4]{1|1} p\}$, GPB $\delta_5 = \{\gamma_5 : p \xrightarrow[5]{1|0} a\}$, and GPB $\delta_6 = \{\gamma_6 : p \xrightarrow[6]{1|0} b\}$. For simplicity, assume that all other GPBs can be ignored. Then, the set of prospective GPUs contain γ_2 and γ_6 because GPU γ_2 is an indirect GPU and GPU γ_6 is in Queued. The composition $\gamma_4 \circ^{\text{ts}} \gamma_6$ is postponed because γ_6 is blocked by a barrier GPU γ_2; there is no path from δ_6 to δ_4 that does not go through δ_2 hence $\gamma_6 \notin \overline{\mathsf{RGIn}}_4$ because $\gamma_6 \notin \overline{\mathsf{RGOut}}_2$.

The other two GPUs (γ_4 and γ_5) are not in $\mathsf{Prospective\_Producer\_GPUs}$ because the GPU γ_4 is neither an indirect GPU nor a producer GPU for any composition (variable q does not appear in any other GPU). On the other hand, the GPU γ_5 is a producer GPU for the composition $\gamma_4 \circ^{\text{ts}} \gamma_5$ which is *admissible*. It is not blocked by the barrier γ_2 because it can reach γ_4 without going through δ_2 (i.e. $\gamma_5 \in \overline{\mathsf{RGIn}}_4$ although $\gamma_5 \notin \overline{\mathsf{RGOut}}_2$).

Since the GPU γ_6 may compose with γ_4 once the barrier GPUs is simplified and the pair $\langle \gamma_6, \{\delta_6 \rightarrow \delta_2\}\rangle$ reaches δ_4, our analysis indicates that the back edge $\delta_6 \rightarrow \delta_2$

is essential and so cannot be removed. Further, there is no GPU for which the back edge $\delta_5 \rightarrow \delta_3$ is required. Hence it is inessential and so can be deleted from $\varDelta$.

Coalescing eliminates almost all back edges in the GPGs with very little work left for back-edge removal. Moreover, essential back edges analysis is very expensive and hence we have not implemented this optimization.

5.6 Chapter Summary

Dead GPU elimination identifies the GPUs in a GPG that do not impact the GPGs of the callers and eliminates them thereby paving way for empty GPB elimination optimization. Coalescing identifies the redundant control flow between the GPBs and eliminates it thereby making the GPGs more compact. Back-edge removal identifies the inessential back edges in the GPGs and eliminates them thereby reducing the number of iterations in a fixed-point computation when the GPGs are inlined at the call sites. Practically, coalescing eliminates most of the back edges. Also, essential back edges analysis is expensive and hence our implementation does not perform it.

Chapter 6

Call Inlining

In order to construct the GPG of a procedure, the optimized GPGs of its callees are inlined at the call sites and the resulting GPG of the procedure is then optimized. After a GPG is inlined at a call site, its GPBs undergo another round of optimization in the calling context. This repeated optimization in the context of each transitive caller of a GPG enhances the compactness of the GPGs significantly.

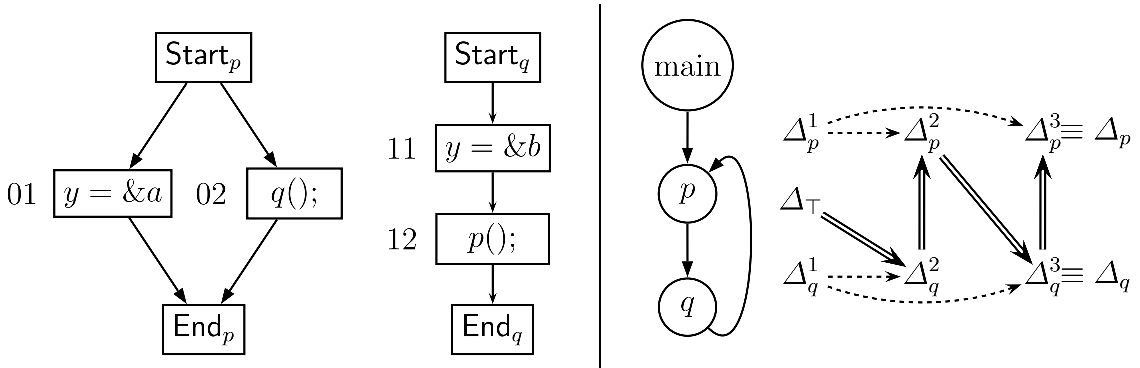
The GPG of a procedure can be constructed completely only when (a) all callees are known, and (b) their GPGs have been constructed completely. The first condition is violated by a call through function pointer and the second condition is violated by a recursive call. We classify procedure calls into the following three categories and explain their handling in this chapter.

- Callee is known and the call is non-recursive.
- Callee is known and the call is recursive.
- Callee is not known.

The third category requires the concept of modelling a use statement to represent the use of a pointer to represent a call through function pointer. This modelling of a use statement is introduced in Chapter 7.

6.1 Callee is Known and the Call is Non-Recursive

In this case, the GPG of the callee can be constructed completely before the GPG of its callers if we traverse the call graph bottom up.



(a) Mutually recursive procedures

(b) Call graph and the order of constructing GPGs for second approach

Figure 6.1: An example demonstrating the construction of GPGs for recursive procedures through a fixed-point computation.

We inline the optimized GPGs of the callees at the call sites in the caller procedures. GPB labels are used for maintaining control flow within a GPG. Hence, we renumber the GPB labels after call inlining and coalescing. Note that if a GPG is inlined multiple times then each inlining uses a fresh numbering for the GPBs that get inlined. Since the statement labels are unique across procedures, their occurrences in GPUs do not change by inlining even if a GPG is inlined at two different call sites within the same procedure. As noted earlier, this is a design choice because it helps us to accumulate the points-to information of a particular statement in all contexts.

When inlining a callee's (optimized) GPG, we add two new GPBs, a predecessor to its Start GPB and a successor to its End GPB. These new GPBs contain respectively:

- GPUs that correspond to the actual-to-formal-parameter mapping (or zero GPUs for a function with no parameters).
- A GPU that maps the return variable of the callee to the receiver variable of the call in the caller (or zero GPUs for a `void` function).

Some GPUs in the GPG of the callee may have upwards-exposed versions of variables (see Section 4.5). These are the variables whose values may be defined in a caller and are read by the callee. For example, global variables or formal parameters (which are defined through actual parameters). Hence when a GPG is inlined in a caller procedure,

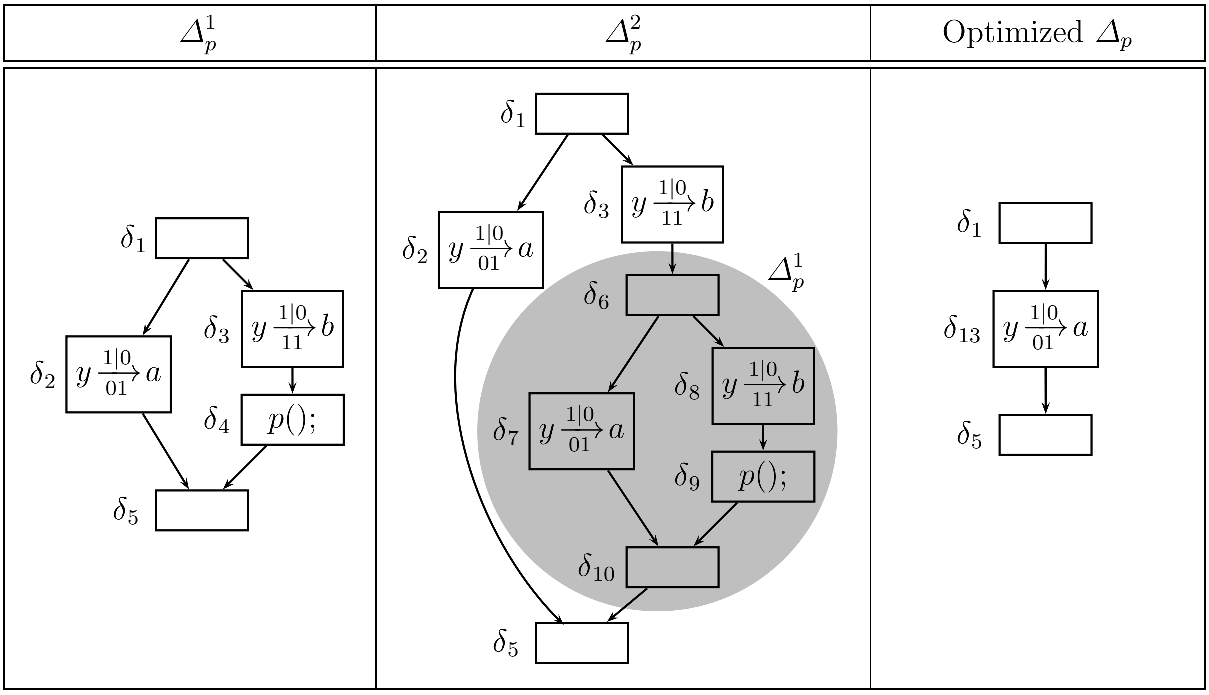

Δ_p^1
Δ_p^2
Optimized Δ_p
δ_1
δ_2 $y \xrightarrow[01]{1|0} a$
δ_3 $y \xrightarrow[11]{1|0} b$
δ_4 $p();$
δ_5
δ_1
δ_2 $y \xrightarrow[01]{1|0} a$
δ_3 $y \xrightarrow[11]{1|0} b$
Δ_p^1
δ_6
δ_7 $y \xrightarrow[01]{1|0} a$
δ_8 $y \xrightarrow[11]{1|0} b$
δ_9 $p();$
δ_{10}
δ_5
δ_1
δ_{13} $y \xrightarrow[01]{1|0} a$
δ_5

Figure 6.2: Series of GPGs of procedure p of Figure 6.1. Δ_p^1 is the initial GPG of the self recursive version of procedure p which is then used to compute Δ_p^2 (highlighted in the second column above). The optimized Δ_p is then used for computing Δ_q^2.

we substitute the callee's upwards-exposed variable x' occurring in a callee's GPU by the original variable x when the GPU is included in the caller's GPG.

Inlining of procedure calls with the callee's optimized GPG allows reaching GPUs analyses to remain intraprocedural analyses. However, recursive and indirect calls need to be handled specially. These cases are discussed in Sections 6.3 and 6.4.

6.2 Callee is Known and the Call is Recursive: Our First Approach

This section describes our first approach for handling recursive calls which failed to scale to large programs because it required creating large GPGs in intermediate stages. Section 6.3 describes an alternative approach which scales by keeping GPGs small in each step and at the same time preserves precision.

Consider procedure p which calls procedure q and q calls p (Figure 6.1). The GPG

```
**Input**:  $p, \Delta_p^1$   // A self-recursive procedure and its first incomplete
                 // GPG containing only self-recursive calls.
**Output**: $\Delta_p$        // Optimized complete GPG for procedure $p$.
Refine_GPG $(p, \Delta_p^1)$
{  $i = 1$
    Perform both variants of reaching GPUs analysis over $\Delta_p^i$
    $Rcurr = \mathsf{RGOut}_{\mathsf{End}}(\Delta_p^i)$
    $\overline{R}curr = \overline{\mathsf{RGOut}}_{\mathsf{End}}(\Delta_p^i)$
    **repeat**
    {  $Rprev = Rcurr$
       $\overline{R}prev = \overline{R}curr$
       $i = i + 1$
       Compute $\Delta_p^i$ by inlining calls of $p$ in $\Delta_p^{i-1}$ with $\Delta_p^1$
       Perform both variants of reaching GPUs analysis over $\Delta_p^i$
       $Rcurr = \mathsf{RGOut}_{\mathsf{End}}(\Delta_p^i)$
       $\overline{R}curr = \overline{\mathsf{RGOut}}_{\mathsf{End}}(\Delta_p^i)$
    }  **until**  $\big((Rcurr \neq Rprev) \vee (\overline{R}curr \neq \overline{R}prev)\big)$
    Delete all call GPBs from $\Delta_p^i$ without connecting their
    predecessors to their successors. Remove the GPBs that do not
    appear on any path from the Start GPB of $\Delta_p^i$ to its End GPB.
    Perform strength reduction and redundancy elimination
    optimizations over $\Delta_p^i$
    **return** $\Delta_p^i$
}
```

Definition 10: *Computing GPGs for self-recursive procedures through fixed-point computation.*

of q depends on that of p and vice-versa. Thus, we have *incomplete* GPGs in which some calls are not inlined because the GPG of the callees are incomplete. We handle this mutual dependency of incomplete GPGs of p and q with the following two steps.

- *Eliminating indirect recursion.* We use a known algorithm [49] to convert indirect recursion into self recursion. The resulting self-recursive version of procedure p is shown in the second column in Figure 6.2.

- *Repeated inlining of self-recursive calls.*

We explain the second step using $\Delta_\top$ which is the $\top$ element of the lattice of all pos-

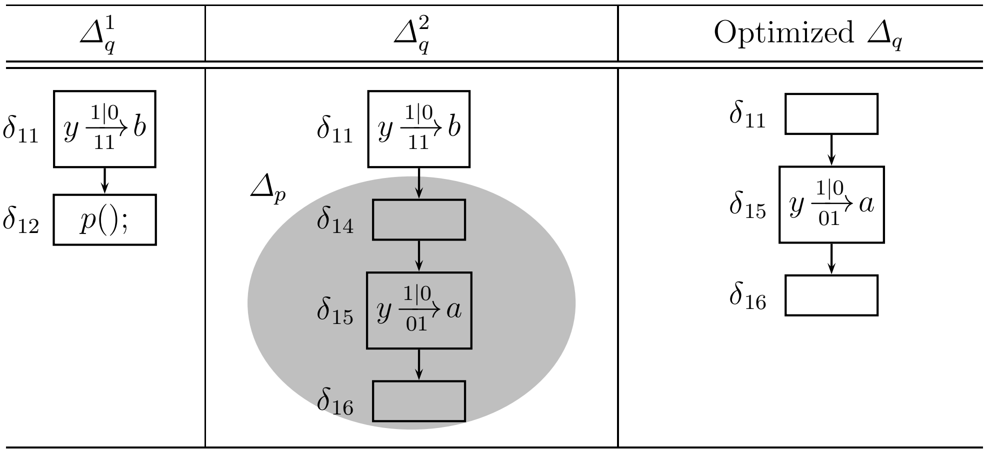


Figure 6.3: Series of GPGs of procedure q of Figure 6.1. Δ_q^1 is the initial GPG of procedure q. The optimized Δ_p (highlighted in the second column above) computed after reaching the fixed point, is used for constructing Δ_q^2. The GPBs are numbered from 11 because the numbers upto 10 are used to represent the GPBs of procedure p.

sible procedure summaries. $\Delta_\top$ is used to represent the effect of a call when the callee's complete GPG is not available. It kills all GPUs and generates none (thereby, when applied, computes the $\top$ value— $\emptyset$—of the lattice for *may* points-to analysis) [51]. Semantically, $\Delta_\top$ corresponds to a procedure call which never returns (e.g. loops forever). It consists of a special GPB called the *call* GPB whose flow functions are constant functions computing the empty set of GPUs for both variants of reaching GPUs analysis. Note that $\Delta_\top$ differs from the empty GPG which is an identity function as a memory transformer. A GPG representing identity flow function does not generate or kill any GPUs.

We perform the reaching GPUs analyses over incomplete GPGs containing self-recursive calls by repeated inlining until no further inlining is required.

Since data flow analysis over incomplete GPGs under-approximates the effect of some calls through $\Delta_\top$, the data flow values need to be recomputed. This is achieved by inlining the calls by including incomplete GPGs of the callees to compute a new GPG over which the data flow analysis is repeated. Let Δ_p^1 denote the GPG of procedure p in which indirect recursion has been converted to self-recursion. Then, we get a series of GPGs whose termination is defined as follows: We compute Δ_p^i, $i > 1$ by inlining Δ_p^1 in each iteration of fixed-point computation. The overall effect of a procedure is reflected by the values reaching its End block. Hence at each stage, we compare the data flow values of the reaching GPUs analyses for the End GPB of a GPG. If these values for a GPG are same as those in the previous GPG in the series, the process stops.

The convergence of above fixed-point computation differs subtly from the usual fixed-point computation in the following manner: in each step of computation, the GPGs continue to change. We stop the fixed-point computation when the *data flow* values converge, not when the resultant GPGs converge (i.e., the GPGs in the last two iterations of fixed-point computation have the same effect when they are applied at the call site in the caller). This requires us to consider the unblocked GPUs ($\overline{\textsf{RGOut}}$) as well as the blocked+non-blocked GPUs (RGOut). Thus, the two GPGs across the iterations are considered to be identical only when they have identical $\overline{\textsf{RGOut}}_{\textsf{End}}$ and $\textsf{RGOut}_{\textsf{End}}$. Section 6.5 formally proves the convergence of GPG construction.

Definition 10 provides an algorithm for computing the fixed point by repeated inlining of self-recursive calls. Once the fixed point is achieved, the remaining call blocks are redundant. Since they represent $\Delta_\top$, i.e., procedure summary representing a call that never returns, they are eliminated without connecting their predecessors to their successors. All GPBs which no longer appear on a control flow path from the Start GPB to the End GPB, are also removed from the GPG, thereby garbage-collecting unreachable GPBs.

Example 55. Consider procedure p in Figure 6.1. Reaching GPUs analyses with and without blocking are performed on the self-recursive GPG of procedure p (Δ_p^1 in the first column of Figure 6.2) and the values are stored in *Rprev* and $\overline{R}\mathit{prev}$. In this example, $\mathit{Rprev} = \{y \xrightarrow[01]{1|0} a\}$ and $\overline{R}\mathit{prev} = \{y \xrightarrow[01]{1|0} a\}$. The call to p in Δ_p^1 is inlined by Δ_p^1 to compute Δ_p^2 (second column of Figure 6.2). Reaching GPUs analyses with and without blocking are performed on Δ_p^2 and the values are stored in *Rcur* and $\overline{R}\mathit{currr}$. In this example, $\mathit{Rcur} = \{y \xrightarrow[01]{1|0} a\}$ and $\overline{R}\mathit{curr} = \{y \xrightarrow[01]{1|0} a\}$. These values are same as those of *Rprev* and $\overline{R}\mathit{prev}$ indicating that a fixed point is reached and no further inlining is required.

The GPG so constructed (Δ_p^2) still contains a call to p (in δ_9). It is now deleted because inlining this call and computing a new GPG will have the same effect as the previous one. This deletion makes δ_5 (the End GPB) unreachable from δ_8. Hence δ_8 is deleted. Subsequent redundancy elimination optimizations give the final GPG Δ_p

as shown in the third column of Figure 6.2.

The **repeat -until** loop in Definition 10 is guaranteed to terminate because of the finiteness of the set of GPUs *Rprev*, $\overline{R}$*prev*, *Rcurr*, $\overline{R}$*curr*. This is explained as follows. For two variables x and y, the number of GPUs $x \xrightarrow[s]{i|j} y$ depends on the number of possible *indlev*s $(i|j)$ and the number of statements. Since the number of statements is finite, we need to examine the number of *indlev*s. For pointers to scalars, the number of *indlev*s between any two variables is bounded because of type restrictions. For pointers to structures (Chapter 8), *indlev*s are replaced by indirection lists (*indlist*s). Sections 8.3 and 8.4 summarize *indlist*s restricting them to a finite number between any two pointer variables. Hence the number of GPUs is also finite.

An important point is that it suffices to eliminate indirect recursion from one procedure in every strongly connected component (SCC) in a call graph; the choice of this procedure is immaterial. After performing fixed-point computation over the selected procedure, its optimized GPG can be inlined in other procedures.

Example 56. For the example in Figure 6.1, we create a self-recursive GPG Δ_p^1 for procedure p and compute its fixed point using repeated inlining. The optimized GPG Δ_p is then inlined in the initial GPG (Δ_q^1 shown in the first column of Figure 6.3) of recursive procedure q to compute Δ_q^2 (second column of Figure 6.3) which needs no fixed point computation as its effect is already incorporated in Δ_p. It is further optimized to compute Δ_q. Note that we have eliminated indirect recursion from p; choosing q for the purpose (or eliminating indirect recursion from both) would give the same GPGs for both p and q.

This approach of converting indirect recursion to self recursion, and repeatedly inlining of the recursive calls failed because it required inlining of unoptimized GPGs. Thus in many cases, the size of GPG became too big and our analyses and optimizations did not scale. Hence, instead of first creating a naively large GPG and then optimizing it to bring down the size, we decided to keep the GPGs small at every stage by inlining only optimized GPGs. We describe this approach in the next section.

6.3 Callee is Known and the Call is Recursive: Our Second Approach

The mutual dependency between the GPGs of procedures p and q (Figure 6.1) is handled by successive construction of incomplete GPGs of p and q through fixed-point computation without converting indirect recursion to self recursion. This avoids the eager inlining of unoptimized GPGs required by the previous approach (Section 6.2).

A set of recursive procedures is represented by a strongly connected component in a call graph which is formed by a collection of back edges that represent recursive calls. Since we traverse a call graph bottom up, the construction of GPGs for a set of recursive procedures begins with the procedures that are the sources of back edges. The GPGs of some callees of these procedures (i.e. the callees that are targets of back edges in the call graph) have not been constructed yet. We handle such situations by using a special GPG $\Delta_\top$ that represents the effect of a call when the callee's GPG is not available. ($\Delta_\top$ is the $\top$ element of the lattice of all possible procedure summaries). Recall that $\Delta_\top$ corresponds to the call to a procedure that never returns (e.g. loops forever). It consists of a special GPB called the *call* GPB whose flow functions are constant functions computing the empty set of GPUs for both variants of reaching GPUs analysis.

We perform the reaching GPUs analyses over incomplete GPGs containing recursive calls by repeated inlining of callees starting with $\Delta_\top$ as their initial GPGs, until no further inlining is required. This is achieved as follows: Since data flow analysis over incomplete GPGs under-approximates the effect of some calls through $\Delta_\top$, the data flow values need to be recomputed. This is achieved by inlining the calls by including incomplete GPGs of the callees to compute a new GPG over which the data flow analysis is repeated. Let Δ_p^1 denote the GPG of procedure p in which all the calls to the procedures that are not part of the strongly connected component are inlined by their respective optimized GPGs. Note that the GPGs of these procedures have already been constructed because of the bottom up traversal over the call graph. The calls to procedures that are part of the strongly connected component are retained in Δ_p^1. In each step, the recursive calls in Δ_p^1 are inlined either

- by $\Delta_\top$ when no GPG of the callee has been constructed, or

```
Input: $p, \Delta_p^1, \Delta_p^i$   // A recursive procedure, its first incomplete GPG
                                     // containing only recursive calls, and its $i^{th}$ GPG
                                     // in the fixed-point computation.
Output: $\Delta_p^{i+1}$             // Optimized $(i+1)^{th}$ GPG for procedure $p$.
Refine_GPG $(p, \Delta_p^1, \Delta_p^i)$
{
    $Rprev = \mathsf{RGOut}_{\mathsf{End}}(\Delta_p^i)$
    $\overline{R}prev = \overline{\mathsf{RGOut}}_{\mathsf{End}}(\Delta_p^i)$
    Compute $\Delta_p^{i+1}$ by inlining recursive calls in $\Delta_p^1$ with their latest GPGs
    Perform both variants of reaching GPUs analysis over $\Delta_p^{i+1}$
    $Rcurr = \mathsf{RGOut}_{\mathsf{End}}(\Delta_p^{i+1})$
    $\overline{R}curr = \overline{\mathsf{RGOut}}_{\mathsf{End}}(\Delta_p^{i+1})$
    if $\big((Rcurr \neq Rprev) \vee (\overline{R}curr \neq \overline{R}prev)\big)$
        Push callers of $p$ on the worklist
    Perform strength reduction and redundancy elimination optimizations
    over $\Delta_p^{i+1}$
    return $\Delta_p^{i+1}$
}
```

Definition 11: *Computing GPGs for recursive procedures through fixed-point computation.*

- by an incomplete GPG of a callee in which some calls are under-approximated using the initial value $\Delta_\top$.

Thus we compute a series of GPGs Δ_p^i, $i > 1$ for every procedure p in a strongly connected component until the termination of fixed-point computation. For this purpose, we initialize a worklist with all procedures in a strongly connected component. This worklist is ordered by the postorder relation between the procedures in the call graph. A procedure is added to the worklist based on the following criterion; the process terminates when the worklist becomes empty. Once Δ_p^i is constructed, we decide to construct Δ_q^j for a caller q of p if the data flow values of the End GPB of Δ_p^i differ from those of the End GPB of Δ_p^{i-1}. This is because the overall effect of a procedure on its callers is reflected by the values reaching its End GPB (because of forward flow of information in points-to analysis). If the data values of the End GPBs of Δ_p^{i-1} and Δ_p^i are same, then they would have identical effect on their callers. Thus, the GPGs are semantically identical as procedure summaries even if they differ structurally. This step is described in Definition 11.

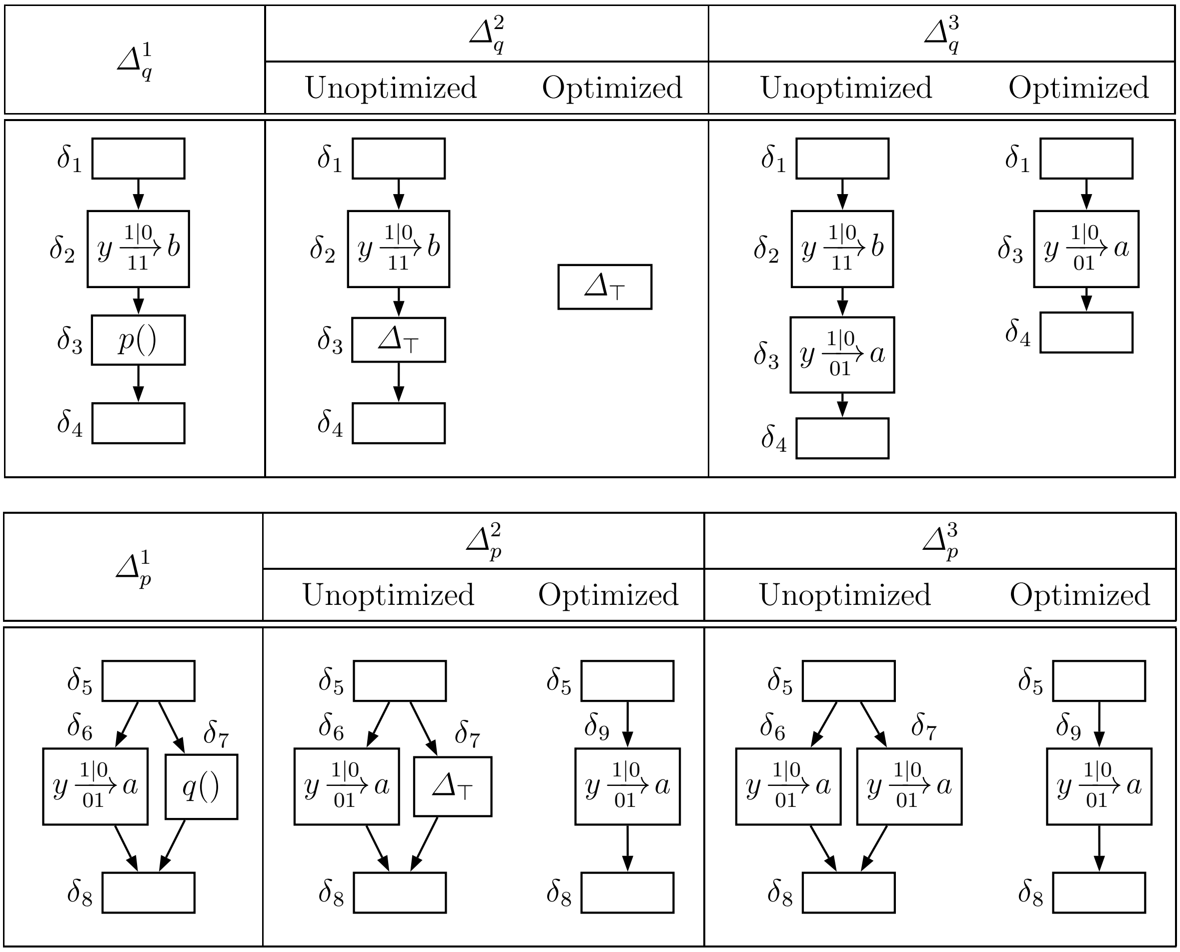


Figure 6.4: Series of GPGs of procedures p and q of Figure 6.1. They are computed in the order shown in Figure 6.1(b). See Example 57 for explanation. The data flow values reaching End_p are identical for Δ_p^2 and Δ_p^3 hence the optimized version of Δ_p^3 is considered the final Δ_p. Although the data flow values reaching End GPB of Δ_q^3 are different from those in Δ_q^2, Δ_q^3 uses Δ_p^2 whose effect is same as that of Δ_p^3. Hence, Δ_q^4 will have the same effect as Δ_q^3 and hence is not constructed.

The convergence of this fixed-point computation is similar to that in the first approach: in each step of computation, the GPGs continue to change. And yet, we stop the fixed-point computation when the *data flow* values of the End GPB converge across the changing GPGs, not when the resultant GPGs converge. Section 6.5 formally proves the convergence of GPG construction.

Note that in this approach, there are no call GPBs in a GPG after the fixed-point computation which need to be eliminated explicitly as in the case of first approach. In the first approach, the call GPBs are present in the GPG even after fixed-point computation because indirect recursion is converted to self-recursion.

Example 57. In the example of Figure 6.1, the sole strongly connected component contains procedures p and q. Since procedure q is the source of the back edge in the call graph, the GPG of procedure q is constructed first. There are no calls in procedure q to procedures outside the strongly connected component. Thus, Δ_q^1 contains a single call to procedure p whose GPG is not constructed yet and hence the construction of Δ_q^2 requires inlining of $\Delta_\top$. Since $\Delta_\top$ represents a procedure call which never returns, the GPB End_q becomes unreachable from the rest of the GPBs in Δ_q^2. The optimized Δ_q^2 is $\Delta_\top$ because all GPBs that no longer appear on a control flow path from the Start GPB to the End GPB are removed from the GPG, thereby garbage-collecting unreachable GPBs. Δ_p^1 contains a single call to procedure q whose incomplete GPG Δ_q^2, which is $\Delta_\top$, is inlined during construction of Δ_p^2. The optimized version of Δ_p^2 is shown in Figure 6.4. Then, Δ_p^2 is used to construct Δ_q^3. Reaching GPUs analyses with and without blocking are performed on Δ_q^2 and Δ_q^3. The data flow values for Δ_q^2 are $Rprev = \overline{R}prev = \emptyset$ whereas the data flow values for Δ_q^3 are $Rcurr = \overline{R}curr = \{y \xrightarrow[01]{1|0} a\}$. Since the data flow values have changed, caller of q i.e., p is pushed on the worklist and Δ_p^3 is constructed by inlining Δ_q^3. The data flow values computed for Δ_p^2 and Δ_p^3 are identical $Rprev = \overline{R}prev = Rcurr = \overline{R}curr = \{y \xrightarrow[01]{1|0} a\}$ and hence caller of p i.e., procedure q is not added to the worklist. The worklist becomes empty and hence the process terminates. Note that the data flow values of Δ_q^2 and Δ_q^3 differ and yet we do not construct the GPG Δ_q^4. This is because Δ_q^4 constructed by inlining Δ_p^3 will have the same effect as that of Δ_q^3 constructed by inlining Δ_p^2 since the impact of Δ_p^2 and Δ_p^3 is identical. Thus, the GPUs reaching the End_q in Δ_q^4 are same as the GPUs reaching the End_q in Δ_q^3.

The process of fixed-point computation is guaranteed to terminate because of the finiteness of the set of GPUs $Rprev$, $\overline{R}prev$, $Rcurr$, $\overline{R}curr$ as described in Section 6.2.

Observe that for the example in Figure 6.1, the GPGs of procedures p and q in both the approaches are identical. This may not always be true. However, the set of GPUs reaching the End GPBs of the GPGs constructed in both the approaches for a procedure will always be identical.

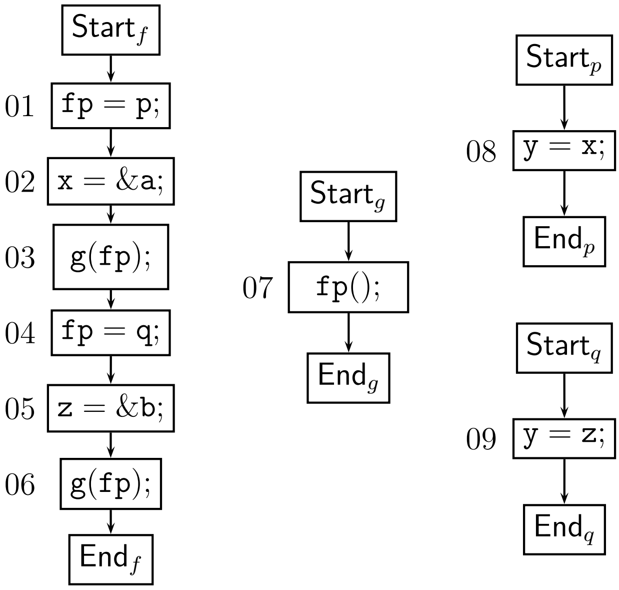


Figure 6.5: An example demonstrating the handling of function pointers.

6.4 Callee is Not Known

Recall that in the case of recursion, we may have incomplete GPGs because the GPGs of the callees are incomplete. Similarly, in the presence of a call through a function pointer, we have incomplete GPGs for a different reason—the callee procedure of such a call is not known. We model a call through function pointer (say *fp*) at call site s as a use statement with a GPU $\boldsymbol{u} \xrightarrow[s]{1|1} \mathit{fp}$ (Chapter 7).

Our goal is to convert a call through a function pointer into a direct call for every pointee of the function pointer. Interleaving of strength reduction and call inlining reduces the GPU $\boldsymbol{u} \xrightarrow[s]{1|1} \mathit{fp}$ and provides the pointees of *fp*. This is identical to computing points-to information (Chapter 7). Until the pointees become available, the GPU $\boldsymbol{u} \xrightarrow[s]{1|1} \mathit{fp}$ acts as a barrier. Once the pointees become available, the indirect call converts to a set of direct calls and are handled as explained in Sections 6.1 and 6.3.

Example 58. Figure 6.5 provides an example of procedures containing calls through function pointers. Figure 6.6 provides the GPGs of the procedures before and after resolving all calls through function pointers. Procedure g has an indirect call through function pointer *fp* in statement 07 and is modelled by a GPB containing a single

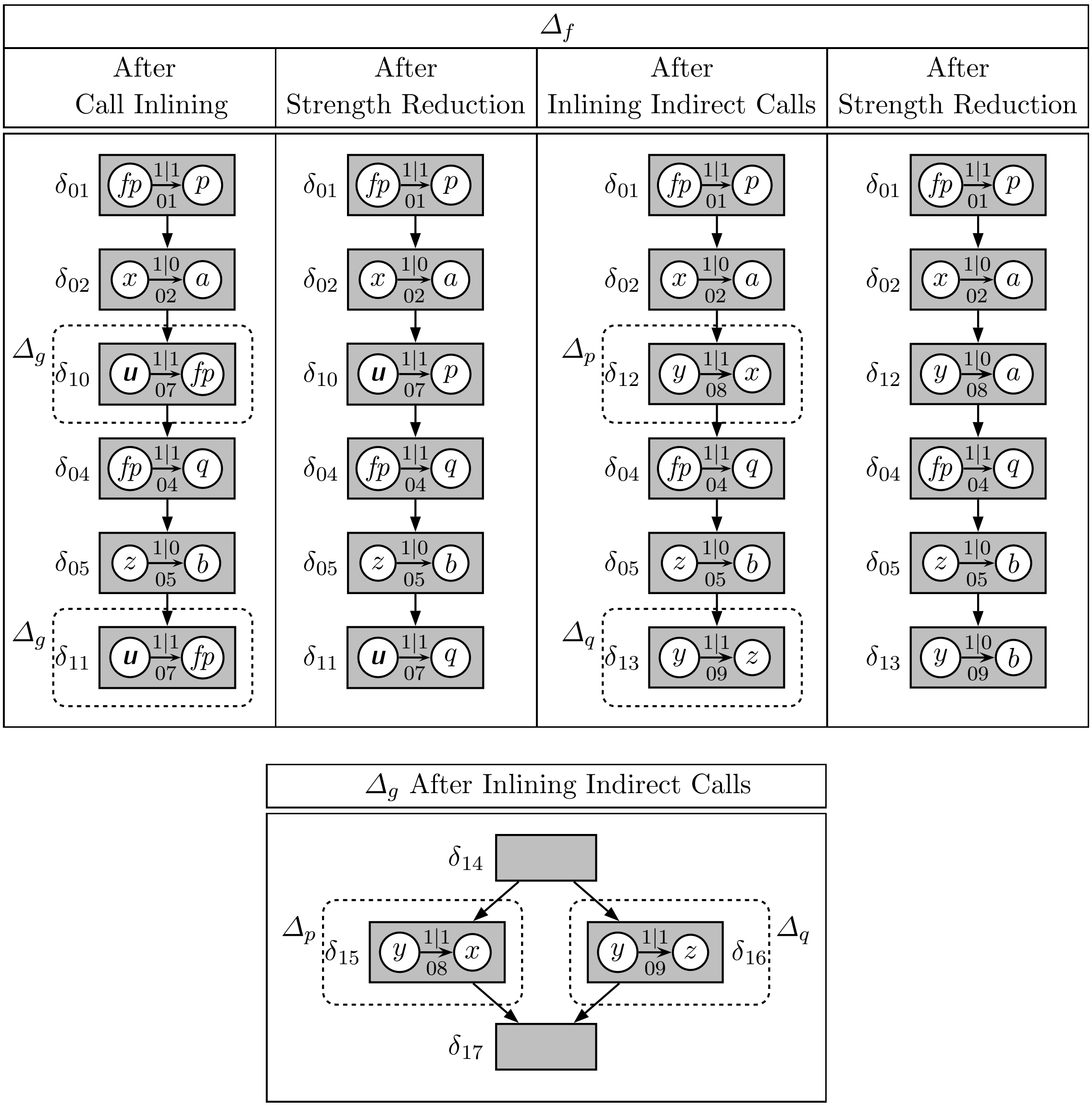


Figure 6.6: Handling function pointers for the example in Figure 6.5. First, the direct calls are inlined leading to the discovery of pointees of the function pointer *fp* causing further inlining and strength reduction. See Example 58 for explanation.

GPU $\boldsymbol{u} \xrightarrow[07]{1|1} fp$ where $\boldsymbol{u}$ models a use (Chapter 7). This GPG is inlined in procedure f in statement 03 as δ_{10} and in statement 06 as δ_{11}.

Since we have $fp \xrightarrow[01]{1|1} p \in \overline{\mathsf{RGIn}}_{10}$, the GPU in δ_{10} reduces to $\boldsymbol{u} \xrightarrow[07]{1|1} p$ indicating that the callee of this indirect call is p. Similarly, the callee for the indirect call in δ_{11} is q. Hence we inline Δ_p in δ_{10} which then becomes δ_{12}. Similarly, Δ_q is inlined in δ_{11} which then becomes δ_{13}. This information is reflected in g by recording p and q as the pointees of *fp* in statement 07. The indirect call in g is converted to two direct calls leading to the inlining of $\varDelta_p$ and $\varDelta_q$ in $\varDelta_g$.

In δ_{03} in procedure f, only procedure p is called because *fp* points to p in statement 03 whereas in δ_{06}, only q is called because *fp* points to q in statement 06. However, in procedure g, either p is called in the context of call at 03 (represented by the GPB δ_{15} in the final GPG) or q is called in the context of call at 06 (represented by the GPB δ_{16} in the final GPG).

6.5 Convergence of GPG Construction in the Presence of Recursion

Similar to Section 4.8, we use the usual guarantee of the convergence of a data flow analysis on the maximum fixed point solution if the following conditions are satisfied [51]:

- The lattice L of data flow values is a complete lattice.
- Flow functions $f : L \rightarrow L$ are monotonic.
- All strictly descending chains in L are finite.

6.5.1 Modelling GPG Construction as a Data Flow Analysis

We model the construction of GPGs as a data flow analysis with the following data flow equations. Let a program contain m procedures numbered 1 through m with their GPGs numbered $\varDelta_1$ through $\varDelta_m$. Let F_i denote the function that computes the GPG $\varDelta_i$ for procedure numbered i. Then,

$$\begin{aligned}\langle \varDelta_1, \varDelta_2, \ldots, \varDelta_m \rangle = \langle\ & F_1(\varDelta_1, \varDelta_2, \ldots, \varDelta_m),\\ & F_2(\varDelta_1, \varDelta_2, \ldots, \varDelta_m),\\ & \qquad\qquad \ldots\\ & F_m(\varDelta_1, \varDelta_2, \ldots, \varDelta_m)\end{aligned} \tag{6.1}$$

The arguments to the function F_i contain all GPGs but only the callee GPGs are inlined, the rest of the GPGs can be ignored. The flow function F_i is a vector of pair of flow functions for reaching GPUs analyses (see Equation 4.7 in Section 4.8).

$$F_i = \langle (f_{\mathsf{I}_1}, f_{\overline{\mathsf{I}}_1}), (f_{\mathsf{O}_1}, f_{\overline{\mathsf{O}}_1}), \ldots, (f_{\mathsf{I}_n}, f_{\overline{\mathsf{I}}_n}), (f_{\mathsf{O}_n}, f_{\overline{\mathsf{O}}_n}) \rangle \tag{6.2}$$

where $f_{\overline{\mathsf{I}}_j}$ and f_{I_j} are the flow functions representing the meet operator for reaching GPUs analysis with and without blocking. Similarly, $f_{\overline{\mathsf{O}}_j}$ and f_{O_j} are the flow functions representing the application of a GPB (in case of a non-call block) or the application of a callee's GPG (in case of a call block) for reaching GPUs analysis with and without blocking.

A fixed point for the Equation (6.1) is computed by repetitive application of flow functions with initialization $\varDelta_\top$ for each $\varDelta_i$.

6.5.2 Lattice of GPGs

The overall effect of a GPG of a procedure in its callers is represented by the GPUs reaching its End GPB. The two components that capture the effect of a GPG are:

- A set of all GPUs reaching the End GPB ($\mathsf{RGOut}_{\mathsf{End}}$).
- A set of all unblocked GPUs reaching the End GPB ($\overline{\mathsf{RGOut}_{\mathsf{End}}}$).

Let $\varGamma$ denote the set of all GPUs (Section 4.8.1). Let $\varPi$ be a set of pairs defined as follows:

$$\varPi = \left\{ \langle R, \overline{R} \rangle \mid R \subseteq \varGamma, \overline{R} \subseteq \varGamma \right\}$$

where R is the set of all (blocked+unblocked) GPUs reaching δ_s (or is RGIn_s) and $\overline{R}$ is the set of unblocked GPUs reaching δ_s (or is $\overline{\mathsf{RGIn}_s}$).

We define a partial order over $\varPi$ as follows. Let $\pi_1 \in \varPi$ be $\langle R_1, \overline{R_1} \rangle$ and $\pi_2 \in \varPi$ be $\langle R_2, \overline{R_2} \rangle$. Then,

$$\pi_1 \sqsubseteq \pi_2 \iff (R_1 \sqsubseteq R_2) \land (\overline{R_1} \sqsubseteq \overline{R_2})$$

where the partial order $R_1 \sqsubseteq R_2$ is as defined in Section 4.8.1. A pair π_1 is weaker than π_2 if it represents a larger set of GPUs that reaches a GPB (both RGOut and $\overline{\mathsf{RGOut}}$). Replacing π_2 with π_1 is a sound approximation. The lattice $(\varPi, \sqsubseteq)$ is a complete lattice (because $\varGamma$ is finite). The top element of this lattice is $\langle \emptyset, \emptyset \rangle$ and the bottom element is $\langle \varGamma, \varGamma \rangle$.

A GPG $\mathit{\Delta}$ of a procedure is a map $\mathit{\Delta} : \mathit{\Pi} \rightarrow \mathit{\Pi}$. The input to a GPG $\mathit{\Delta}$ is a pair $\pi \in \mathit{\Pi}$ where π could represent:

- Boundary definitions $\langle D, D\rangle$ where D is a set of boundary definitions.
- Memory $\langle M, M\rangle$ for computing points-to information within the procedure where M is the memory represented by a set of points-to edges (GPUs with *indlev* "1|0").
- GPUs $\langle \mathsf{RGIn}_c, \overline{\mathsf{RGIn}_c}\rangle$ reaching a call to the procedure at call site c.

We denote the output of a GPG $\mathit{\Delta}$ for an input π by $\mathit{\Delta}(\pi)$. This represents the GPUs reaching the End GPB of $\mathit{\Delta}$.

We define a partial order over the set of all GPGs as follows:

$$\mathit{\Delta}_1 \sqsubseteq \mathit{\Delta}_2 \iff \forall\, \pi \in \mathit{\Pi} \quad \mathit{\Delta}_1(\pi) \sqsubseteq \mathit{\Delta}_2(\pi)$$

This partial order allows us to treat all structurally different GPGs that represent the same mapping as identical.

The top element $\mathit{\Delta}_\top$ of the lattice is a constant function that computes $\langle\emptyset, \emptyset\rangle$ (i.e., $\forall\, \pi \in \mathit{\Pi}\;\; \mathit{\Delta}_\top(\pi) = \langle\emptyset, \emptyset\rangle$). Irrespective of the input to $\mathit{\Delta}_\top$, it does not generate any GPUs and kills all the GPUs reaching its Start GPB. Similarly, the bottom element of the lattice, denoted $\mathit{\Delta}_\bot$ is a constant function defined as $\forall\, \pi \in \mathit{\Pi}\;\; \mathit{\Delta}_\bot(\pi) = \langle \mathit{\Gamma}, \mathit{\Gamma}\rangle$.

The finiteness of the lattice of GPGs is guaranteed by the finiteness of $\mathit{\Gamma}$ and hence the lattice is complete.

6.5.3 Convergence on the Maximum Fixed Point

In this section, we show that GPG construction computes a unique GPG for every procedure even in presence of recursion.

Theorem 6.1. *GPGs computed by Equation (6.1) converge on the maximum fixed point.*

Proof. Let $\mathit{\Delta}_i^k$ denote the GPG of procedure numbered i in k^{th} step of fixed-point computation of Equation (6.1). Then, Equation (6.1) computes the following sequence of vectors of GPGs of all procedures.

$$\langle \mathit{\Delta}_1^0, \mathit{\Delta}_2^0, \ldots, \mathit{\Delta}_m^0\rangle, \langle \mathit{\Delta}_1^1, \mathit{\Delta}_2^1, \ldots, \mathit{\Delta}_m^1\rangle, \ldots, \langle \mathit{\Delta}_1^i, \mathit{\Delta}_2^i, \ldots, \mathit{\Delta}_m^i\rangle, \ldots$$

Lemma 6.2 shows that this sequence follows a descending chain.

$$\langle \varDelta_1^0, \varDelta_2^0, \ldots, \varDelta_m^0 \rangle \sqsubseteq \langle \varDelta_1^1, \varDelta_2^1, \ldots, \varDelta_m^1 \rangle \sqsubseteq \ldots \sqsubseteq \langle \varDelta_1^i, \varDelta_2^i, \ldots, \varDelta_m^i \rangle \sqsubseteq \ldots$$

Since the lattice of $\varDelta_i$'s is finite, so is the lattice of m length vectors of $\varDelta_i$. Hence, every strictly descending chain is finite. Thus, there exists a k such that,

$$\langle \varDelta_1^{k+1}, \varDelta_2^{k+1}, \ldots, \varDelta_m^{k+1} \rangle = \langle \varDelta_1^k, \varDelta_2^k, \ldots, \varDelta_m^k \rangle$$

We begin the fixed-point computation with the initial value:

$$\langle \varDelta_\top, \varDelta_\top, \ldots, \varDelta_\top \rangle$$

Since the lattice if finite, it is also complete. Hence, the sequence converges on the maximum fixed point. □

Lemma 6.2. *The sequence of vectors of GPGs computed by Equation (6.1) follows a descending chain.*

Proof. We prove the lemma by using induction on the number of steps for computing GPGs using Equation (6.1).

Basis: Since the initialization for all GPGs is $\varDelta_\top$ ($\top$ value),

$$\begin{aligned} \langle \varDelta_1^1, \varDelta_2^1, \ldots, \varDelta_m^1 \rangle \ &\sqsubseteq \langle \varDelta_1^0, \varDelta_2^0, \ldots, \varDelta_m^0 \rangle \\ &\sqsubseteq \langle \varDelta_\top, \varDelta_\top, \ldots, \varDelta_\top \rangle \end{aligned}$$

Inductive hypothesis: Assume that after k steps, the following relation holds

$$\langle \varDelta_1^k, \varDelta_2^k, \ldots, \varDelta_m^k \rangle \ \sqsubseteq \langle \varDelta_1^{k-1}, \varDelta_2^{k-1}, \ldots, \varDelta_m^{k-1} \rangle$$

Thus, for GPG $\varDelta_i$ in k and $(k-1)$ steps, we have,

$$\forall\, \delta_j\colon\ (\mathsf{I}_j^k \sqsubseteq \mathsf{I}_j^{k-1}) \ \wedge\ (\bar{\mathsf{I}}_j^k \sqsubseteq \bar{\mathsf{I}}_j^{k-1}) \ \wedge\ (\mathsf{O}_j^k \sqsubseteq \mathsf{O}_j^{k-1}) \ \wedge\ (\overline{\mathsf{O}}_j^k \sqsubseteq \overline{\mathsf{O}}_j^{k-1})$$

where I_j^k and O_j^k are the sets of GPUs reaching the entry and exit of δ_j respectively in GPG $\varDelta_i^k$ after fixed point computation of reaching GPUs analysis without blocking. Similarly, $\bar{\mathsf{I}}_j^k$ and $\overline{\mathsf{O}}_j^k$ are the sets of GPUs reaching the entry and exit of δ_j respectively in GPG $\varDelta_i^k$ after fixed point computation of reaching GPUs analysis with blocking.

Inductive step: In $(k+1)^{th}$ step, we compute Δ_i^{k+1} by inlining callee GPGs computed in k^{th} step. We need to prove that,

$$\langle \Delta_1^{k+1}, \Delta_2^{k+1}, \ldots, \Delta_m^{k+1} \rangle \sqsubseteq \langle \Delta_1^{k}, \Delta_2^{k}, \ldots, \Delta_m^{k} \rangle$$

This can be proved by showing that,

$$\forall\, \delta_j:\ \ (\mathsf{I}_j^{k+1} \sqsubseteq \mathsf{I}_j^{k}) \ \wedge\ (\bar{\mathsf{I}}_j^{k+1} \sqsubseteq \bar{\mathsf{I}}_j^{k}) \ \wedge\ (\mathsf{O}_j^{k+1} \sqsubseteq \mathsf{O}_j^{k}) \ \wedge\ (\overline{\mathsf{O}}_j^{k+1} \sqsubseteq \overline{\mathsf{O}}_j^{k})$$

Here for brevity, we consider only the sets of GPUs computed by reaching GPUs analysis without blocking (I_j and O_j). However, all arguments hold for the sets of GPUs computed by reaching GPUs analysis with blocking ($\bar{\mathsf{I}}_j$ and $\overline{\mathsf{O}}_j$) also.

By inductive hypothesis, for all GPBs δ_j in Δ_i, we know that,

$$\mathsf{O}_j^{k} \sqsubseteq \mathsf{O}_j^{k-1} \Rightarrow \prod_{p \in pred(j)} \mathsf{O}_p^{k} \sqsubseteq \prod_{p \in pred(j)} \mathsf{O}_p^{k-1} \qquad \text{(meet is monotonic)}$$

$$\Rightarrow\ \mathsf{I}_j^{k+1} \sqsubseteq \mathsf{I}_j^{k} \qquad \text{(From Definition 5)}$$

For the value O_j^{k+1}, we need to consider two cases. The flow functions f_{O_j} in Equation (6.2) for computing the sets O_j represent

- GPB application for non-call blocks and
- GPG application for call blocks.

For the first case, we have already proved that GPBs are monotonic (Section 4.8.2). Here we prove the second case.

Let GPB δ_j contain a call to procedure q. Then, O_j^{k} is computed by inlining the GPG Δ_q^{k-1} computed in the $(k-1)^{th}$ step. Similarly for $(k+1)^{th}$ step, O_j^{k+1} is computed by inlining the GPG Δ_q^{k} computed in the k^{th} step. By inductive hypothesis,

$$\Delta_q^{k} \sqsubseteq \Delta_q^{k-1} \Rightarrow\ \mathsf{O}_j^{k+1} \sqsubseteq \mathsf{O}_j^{k}$$

Since the relation $\mathsf{O}_j^{k+1} \sqsubseteq \mathsf{O}_j^{k}$ holds for every GPB δ_j, it also holds for the End GPB. Similarly, $\overline{\mathsf{O}}_j^{k+1} \sqsubseteq \overline{\mathsf{O}}_j^{k}$ also holds for the End GPB. Hence, $\Delta_i^{k+1} \sqsubseteq \Delta_i^{k}$. This holds for every Δ_i proving the inductive step. Hence the lemma. □

6.6 Chapter Summary

Call inlining incorporates the effect of callee by inlining the optimized GPGs of the callees at its call site. It also maps the arguments in the caller to their corresponding parameters in the callee as well as the return variable of the callee to the receiver variable of the call in the caller.

Recursive calls are handled through a fixed-point computation. These calls are eliminated by a bounded inlining of callee GPGs without over-approximation.

A call through function pointer can be converted to multiple direct calls based on the number of pointees of the function pointer. Calls through function pointers are handled through delayed inlining when the pointees of the function pointer are not locally available. Such calls are modelled by use statements and are resolved when the pointees of function pointers become available.

Chapter 7

Computing Points-to Information using GPGs

Recall that a flow- and context-sensitive interprocedural analysis using procedure summaries is performed in two phases: the first phase constructs the procedure summaries and the second phase uses the procedure summaries to compute the desired information, which, in our case, is the classical points-to information. In this chapter, we discuss the second phase of computing points-to information using GPGs. Section 7.1 models a use statement corresponding to a non-pointer assignment or an expression accessing pointer variables. Section 7.2 describes the second phase that computes points-to information at every program point using GPGs.

Although it is good to know the points-to information at every program point, it is sufficient to know only the pointees of the pointers occurring in assignments and other statements. Section 7.3 describes how the desired points-to information for every assignment can be computed as a side-effect of the first phase thereby, making the second phase redundant.

7.1 Modelling a Use Statement

The use of pointers in assignments is modelled by our definition of GPUs. We model the use of pointers in other statements in the form of a *use* statement. This is required because even though a pointer is not being defined in such statements, there is a use of pointer variables and hence one needs to know the pointees of these pointer variables.

Consider a use of a pointer variable in a non-pointer assignment or an expression. We represent such a use with a GPU whose source is a fictitious node $\boldsymbol{u}$ with *indlev* 1 and the target is the pointee that is being read. Thus a condition '`if (x == *y)`' where both x and y are pointers, is modelled as a GPB $\left\{\boldsymbol{u} \xrightarrow[s]{1|1} x, \boldsymbol{u} \xrightarrow[s]{1|2} y\right\}$ whereas an integer assignment '`*x = 5;`' is modelled as a GPB $\left\{\boldsymbol{u} \xrightarrow[s]{1|2} x\right\}$.

Example 59. Consider the code snippet on the right. There is a non-pointer assignment statement 02 in which the pointee of x (which is the location a) is being defined. A client analysis would like to know the pointees of x for statement 02. We model this use of pointee of x as a GPU $\boldsymbol{u} \xrightarrow[02]{1|2} x$. This GPU can be composed with $x \xrightarrow[01]{1|0} a$ to get a reduced GPU $\boldsymbol{u} \xrightarrow[02]{1|1} a$ indicating that pointee of x in statement 2 is a.

```
01   x = &a;
02   *x = 5;
```

When a use involves multiple pointers such as '`if (x == *y)`', the corresponding GPB contains multiple GPUs. After their reduction, pointers x and y are eliminated and we may get a GPB containing GPUs $\boldsymbol{u} \xrightarrow[s]{1|1} a$ and $\boldsymbol{u} \xrightarrow[s]{1|1} b$ and we may not know which of 'a' and 'b' is a pointee of x or y. If the exact pointer-pointee relationship is required, rather than just the reduced form of the use (devoid of pointers), we need additional minor bookkeeping to record GPUs and the corresponding pointers.

7.2 Computing Points-to Information At Every Program Point

In this section, we describe how to compute points-to information for each statement within a procedure. This requires computing the points-to information reaching the call points of the procedure in its callers. This boundary information is denoted by *BI*.

7.2.1 Operations for Computing Points-to Information

Recall that the points-to information in abstract memory is seen as a relation (Section 1.3.1). We define the operations of *memory application* and *memory composition* for computing points-to information PIn and POut for every pointer assignment within a

$$
\begin{aligned}
\mathsf{PIn}_s &:= \begin{cases} \mathit{BI} & s = \mathsf{Start} \\ \bigcup\limits_{p \,\in\, \mathit{pred}(s)} \mathsf{POut}_p & \text{otherwise} \end{cases} \\
\mathsf{POut}_s &:= (\mathsf{PIn}_s - \mathsf{PKill}_s) \cup [\![\delta_s]\!]\mathsf{PIn}_s \\
\mathsf{PKill}_s &:= \mathsf{Memkill}\left([\![\delta_s]\!]\mathsf{PIn}_s,\ \mathsf{PIn}_s\right) \\
\mathsf{Memkill}(M, P) &:= \left\{\gamma_1 \mid \gamma_1 \in \mathsf{Match}(\gamma, P), \gamma \in M, \mathsf{Singledef}\,(M,\ \gamma)\right\} \\
\mathsf{Singledef}\,(M,\ x \xrightarrow[s]{i|j} y) &:= |M^{\,i-1}\{x\}| = 1
\end{aligned}
$$

Definition 12: *Computing Points-to information using GPG, $\Delta\,(M)$*

procedure. The pointees of a set of pointers $X \subseteq \mathsf{P}$ in M are computed by the memory application $M\ X = \{y \mid (x, y) \in M, x \in X\}$. Let M^i denote a relation composition of degree i, i.e. $M^i = M \circ M^{i-1}$ where M^0 is an identity relation. Then, $M^i\{x\}$ discovers the i^{th} pointees of x which involves i transitive reads from x: first $i-1$ addresses are read followed by the content of the last address. For composability of M, we extend its domain to L by the inclusion map. Since M^0 is an identity relation, $M^0\{x\} = \{x\}$.

Computing points-to information within a procedure requires the following two operations which compute a new memory M' from a given memory M.

- *GPU evaluation* $[\![\gamma]\!]M$ computes the set of points-to edges that would be created by executing the GPU $\gamma : x \xrightarrow{i|j} y$ in memory M. It is given by:

  $$[\![x \xrightarrow{i|j} y]\!]M = \left\{w \xrightarrow{1|0} z \mid w \in M^{i-1}\{x\},\ z \in M^j\{y\}\right\}.$$

  Recall that we identify classical points-to edges as GPUs with *indlev* $1|0$.

- *GPB evaluation* $[\![\delta]\!]M$ simply iterates GPU evaluation for the GPUs in GPB δ.

  $$[\![\delta]\!]M = \bigcup_{\gamma \,\in\, \delta} [\![\gamma]\!]M$$

Definition 12 gives the data flow equations for computing points-to information for individual statements within a procedure. It is achieved by traversing the GPG of the procedure and using the GPBs corresponding to the statements.

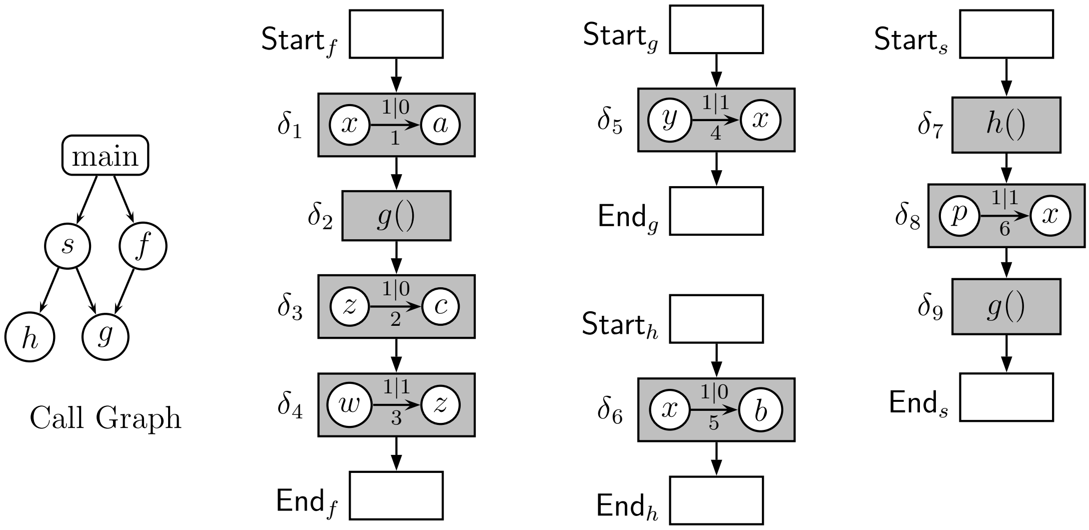


Figure 7.1: Interaction between GPGs for computing points-to information. The GPG of procedure *main* has been omitted.

7.2.2 Computation of Boundary Information *BI*

The boundary information *BI* for a procedure is computed as the union of the points-to information reaching the procedure from all of its call points in all its callers.[1] For the main function, *BI* is computed from static initializations. In the presence of recursion, a fixed-point computation is required to compute *BI*.

The predicate Singledef $(M,\ x\xrightarrow[t]{i|j}y)$ in Definition 12 asserts that a GPU $x\xrightarrow[t]{i|j}y$ in δ_s defines a single pointer. Observe that Singledef $(M,\ x\xrightarrow[s]{i|j}y)$ trivially holds for $i = 1$ (i.e. for direct assignments). The GPUs to be removed (Memkill) are characterized much along the lines of Kill (Definition 5 in Section 4.6). Besides, function Match is same as in Definition 5 in Section 4.6.

Example 60. For the program in Figure 7.1, the *BI* of procedure g (denoted BI_g) is the points-to information reaching g from its callers f and s and hence a union of POut_{δ_1} and POut_{δ_8} where $\mathsf{POut}_{\delta_1} = \{x\xrightarrow[1]{1|0}a\}$ and $\mathsf{POut}_{\delta_8} = \{x\xrightarrow[5]{1|0}b, p\xrightarrow[6]{1|0}b\}$. Thus, $\mathsf{PIn}_{\mathsf{Start}_g} = BI_g = \mathsf{POut}_{\delta_1} \cup \mathsf{POut}_{\delta_8}$. The points-to information generated for statement 4 is $[\![\delta_5]\!]\mathsf{PIn}_{\mathsf{Start}_g} = \{y\xrightarrow[4]{1|0}a, y\xrightarrow[4]{1|0}b\}$.

[1] *BI* as defined above is the boundary information for computing points-to information. The boundary information for reaching GPUs analyses consists of boundary definitions (Section 4.5).

7.2.3 Bypassing of *BI*

The *BI* of a procedure computed from all call sites of the procedure contains many points-to pairs that are not accessed by the procedure. This causes inefficiency in computing PIn and POut (Definition 12) because many irrelevant points-to pairs are processed for each statement. Thus the efficiency of computing points-to information can be enhanced significantly by filtering out the points-to information that is irrelevant to a procedure. Our earlier implementation [25] shows efficiency gain that was achieved by bypassing. This concept of *bypassing* has been successfully used for data flow values of scalars [73, 74] but not for pointers.

GPGs support bypassing naturally for pointers with the help of upwards-exposed versions of variables. The occurrence of an upwards-exposed version of a variable in a GPU indicates that there is a use of the variable in the GPU requiring pointee information from the callers. In other words, the variable is live on entry to the procedure. Thus, the points-to information of such a variable is relevant and should be a part of *BI*. For variables that do not have their corresponding upwards-exposed versions occurring in a GPU, their points-to information is irrelevant and can be discarded from the *BI* of the procedure, effectively bypassing the calls to the procedure.

Example 61. In our example of Figure 7.1, the set of GPUs containing upwards exposed variables in procedure g after call inlining and strength reduction optimization is $\{y \xrightarrow[4]{1|1} x'\}$. This implies that some pointees of x from the calling context are accessed in the procedure g. Hence BI_g should contain the GPUs involving x only. The points-to information of p is not required in procedure g and hence can be bypassed. Thus, BI_g is no longer the union POut_{δ_1} and POut_{δ_8} as it excludes GPU $p \xrightarrow[6]{1|0} b$ as it is irrelevant to procedure g and hence is bypassed.

The computation of boundary information (*BI*) is expensive. As a result, in our earlier implementation [25], the second phase took more time than the first phase of the analysis. The optimization of bypassing enhanced the efficiency of the second phase. We further optimize the computation of points-to information by rendering the second phase redundant with the help of statement labels that are part of the GPU abstraction.

7.3 Obviating the Second Phase of Bottom-Up Approach

Recall that the statement labels used in GPUs were used to distinguish between strong and weak updates (Section 4.5). However, the statement labels are also useful to compute points-to information by making the second phase of a bottom-up approach which uses procedure summaries (created in the first phase) redundant. This is because our first phase computes the points-to information as a side-effect of the construction of GPGs.

The process of computing points-to information can be seen as a two step process:

- creating def-use or use-def chains for pointers to view producer GPUs as definitions of pointers and consumer GPUs as uses of pointers, and
- performing strength reduction of the consumer GPUs using the information from the producer GPUs to reduce the *indlev*s of the consumer GPUs.

Since our first phase does this for constructing procedure summaries, it is sufficient to compute points-to information. The statement labels help to uniquely identify the definition of pointer variables and their corresponding uses.

This process is easy to visualize if the definitions and uses are in the same procedure. Consider a producer GPU ***p*** and a consumer GPU ***c*** that are not in the same procedure. We can facilitate strength reduction involving them by propagating either

(a) ***p*** to the caller containing ***c***,

(b) ***c*** to the caller containing ***p***,

(c) both ***p*** and ***c*** to a common procedure, or

(d) neither (if they are same in the procedure).

The propagation of information in cases (a) and (b) is similar to that in a top-down analysis; case (a) corresponds to a forward analysis and case (b) corresponds to a backward analysis. However, case (c) is only possible in bottom-up analysis.

A typical second phase of a bottom-up approach described in Section 7.2 involves propagation of information similar to cases (a) and (b). This is illustrated in Example 62. We use propagation similar to case (c) which is subsumed in the first phase of a bottom-up approach rendering the second phase redundant. It is illustrated in Example 63.

Example 62. Consider procedures f, g, h and s defined in Figure 7.1. We can facilitate strength reduction in the following ways for cases (a) and (b):

- *Propagating* $\boldsymbol{\mathsf{p}}$ *to the procedure containing* $\boldsymbol{\mathsf{c}}$. A top-down forward analysis would propagate the GPU $x \xrightarrow[1]{1|0} a$ from procedure f to procedure g.
- *Propagating* $\boldsymbol{\mathsf{c}}$ *to the procedure containing* $\boldsymbol{\mathsf{p}}$. A top-down backward analysis in the spirit of liveness could propagate the GPU $y \xrightarrow[4]{1|1} x$ from procedure g to procedure f.

We handle case (c) by interleaved call inlining and strength reduction. Call inlining enhances the opportunities for strength reduction by providing more information from the callers. The interleaving of strength reduction and call inlining gradually converts a GPU $x \xrightarrow[s]{i|j} y$ to a set of points-to edges $\{a \xrightarrow[s]{1|0} b \mid a \text{ is } i^{th} \text{ pointee of } x,\ b \text{ is } j^{th} \text{ pointee of } y\}$.

Since statement numbers are unique across all procedures and are not renamed on inlining, the points-to edges computed across different contexts for a given statement represent the flow- and context-sensitive points-to information for the statement.

Example 63. The four variants of hoisting $\boldsymbol{\mathsf{p}}$ and $\boldsymbol{\mathsf{c}}$ to a common procedure in the first phase of a bottom-up method are illustrated below. Effectively, they make the second phase redundant.

(c.1) When Δ_g is inlined in f, $\boldsymbol{\mathsf{c}}\!: y \xrightarrow[4]{1|1} x$ from procedure g is hoisted to procedure f that contains GPU $\boldsymbol{\mathsf{p}}\!: x \xrightarrow[1]{1|0} a$ thereby propagating the use of pointer x in procedure g to caller f. Strength reduction reduces $\boldsymbol{\mathsf{c}}$ to $y \xrightarrow[4]{1|0} a$.

(c.2) When Δ_h is inlined in s, $\boldsymbol{\mathsf{p}}\!: x \xrightarrow[5]{1|0} b$ from procedure h is hoisted to procedure s that contains $\boldsymbol{\mathsf{c}}\!: p \xrightarrow[6]{1|1} x$ thereby propagating the definition of x in procedure h to the caller s. Strength reduction reduces $\boldsymbol{\mathsf{c}}$ to $p \xrightarrow[6]{1|0} b$.

(c.3) When Δ_g and Δ_h are inlined in s, $\boldsymbol{\mathsf{c}}\!: y \xrightarrow[4]{1|1} x$ in procedure g and $\boldsymbol{\mathsf{p}}\!: x \xrightarrow[5]{1|0} b$ in procedure h are both hoisted to procedure s thereby propagating both the use and definition of x in procedure s. Strength reduction reduces $\boldsymbol{\mathsf{c}}$ to $y \xrightarrow[4]{1|0} b$.

(c.4) Both the definition and use of pointer z are available in procedure f with $\boldsymbol{\mathsf{c}}$

$: w \xrightarrow[3]{1|1} z$ and $\boldsymbol{p} : z \xrightarrow[2]{1|0} c$. Strength reduction reduces $\boldsymbol{c}$ to $w \xrightarrow[3]{1|0} c$.

Thus, y points-to a along the call from procedure f and it points-to b along the call from procedure s. Thus, the points-to information $\{y \xrightarrow{1|0} a, y \xrightarrow{1|0} b\}$ represents flow- and context-sensitive information for statement 4.

7.4 Chapter Summary

In a traditional bottom-up approach, a top-down traversal over the call graph for computing points-to information follows a bottom-up traversal over the call graph for constructing procedure summaries. We have defined two methods for computing points-to information using procedure summaries. We implemented the first approach for computing points-to information [25] and discovered that the second phase took more time than the first phase of the analysis because computation of *BI* is expensive. The optimization of bypassing enhanced the efficiency of the second phase and yet the second phase needed more time than that of the first phase. Hence in our recent implementation, we further optimized the computation of points-to information by rendering the second phase redundant with the help of statement labels that are part of the GPU abstraction.

Chapter 8

Handling Heap for Points-to Analysis using GPGs

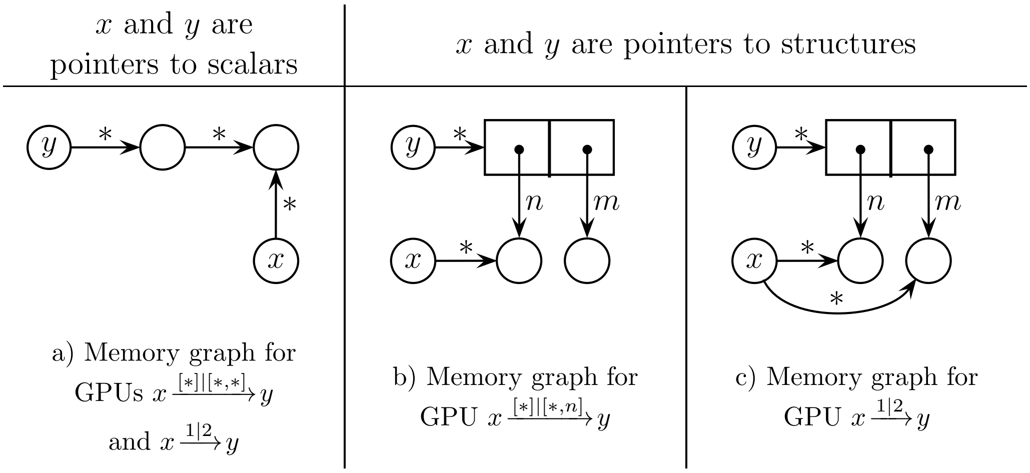
So far we have created the concept of GPGs for pointers to scalars allocated on the stack or in the static area. This chapter extends the concepts to data structures containing named fields created using C style **struct** or **union** and possibly allocated on heap (as well as on the stack or in static memory). For simplicity, we show only the set of GPUs reaching a given statement and do not show the complete GPG of a procedure.

8.1 Extending GPUs to Handle Structures and Heap

We extend GPGs to handle structures as follows:

- We generalize the concept of *indlev*s to indirection lists (*indlist*s) to handle structures and heap accesses field sensitively (Section 8.2).
- We abstract heap locations using allocation sites (Section 8.3). In this abstraction, all locations allocated at a particular allocation site are treated alike. This approximation allows us to handle the unbounded nature of heap as if it were bounded [48]. Hence only weak updates can be performed on heap locations.[1]

[1] We also perform weak updates for address-escaped variables (Section 9.1) because they share many similarities with heap locations. Like heap locations, address-escaped variables could outlive the lifetime of the procedures that create them. They potentially represent multiple concrete locations because of multiple calls to procedures. Further, this number could be unbounded in the case of recursive calls.


x and y are pointers to scalars
x and y are pointers to structures
y
*
*
*
x
y
*
n
m
x
*
y
*
n
m
x
*
*

a) Memory graph for GPUs $x \xrightarrow{[*]|[*,*]} y$ and $x \xrightarrow{1|2} y$

b) Memory graph for GPU $x \xrightarrow{[*]|[*,n]} y$

c) Memory graph for GPU $x \xrightarrow{1|2} y$

Figure 8.1: An example demonstrating the need for *indlist*s in the case of structures. No distinction between dereferences is required for pointers to scalars (case a). Distinction between dereferences is required for pointers to structures for field sensitivity (case b). When no distinction between dereferences for pointers to structures, imprecision is introduced owing to field insensitivity (case c).

- We use an additional summarization based on k-limiting to bound the accesses in a loop (Section 8.4). This is because the locations being accessed in a loop may be allocated in a caller and hence their allocation sites may not be available when the GPG of the procedure is being constructed.
- We introduce *indlist*s and k-limiting summarization to extend the concept of GPU composition to handle them (Section 8.4.2).
- We extend GPU reduction to handle cycles that may be created by allocation-site-based abstraction and k-limiting summarization (Section 8.5).

The allocation-site-based-abstraction and k-limiting summarization techniques are required to create a decidable version of our method of constructing procedure summaries in the form of GPGs. The resulting points-to analysis is a precise flow-sensitive, field-sensitive, and context-sensitive analysis (relative to these two summarization techniques).[2]

[2] In a top-down analysis, k-limiting is not required because allocation sites are propagated from callers to callees. While the use of k-limiting in a bottom-up approach seems like an additional restriction, unless the locations involved in a pointer chain are allocated by $m > k$ distinct allocation sites, there is no loss of precision compared to a top-down approach.

Pointer assignment	GPU	Remark
$x = malloc(\ldots)$	$x \xrightarrow{[*]\|[\,]} h_i$	The allocation site name is i
$x =$ NULL	$x \xrightarrow{[*]\|[\,]}$ NULL	NULL is a distinguished location
$x = y.n$	$x \xrightarrow{[*]\|[n]} y$	
$x.n = y$	$x \xrightarrow{[n]\|[*]} y$	
$x = y \rightarrow n$	$x \xrightarrow{[*]\|[*,n]} y$	
$x \rightarrow n = y$	$x \xrightarrow{[*,n]\|[*]} y$	

Figure 8.2: GPUs with indirection lists (*indlist*s) for basic pointer assignments in C for structures and dynamically allocated heap locations.

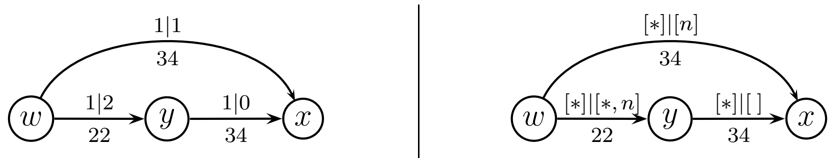
The optimizations performed on GPGs and the required data flow analyses (reaching GPUs analyses and coalescing analysis) remain the same. Hence, the discussion in these sections is driven mainly by examples that illustrate how the theory developed earlier is adapted to handle structures (typically, but not necessarily, heap-allocated).

8.2 Extending GPU Composition to Indirection Lists

The *indlev* "$i|j$" of a GPU $x \xrightarrow[s]{i|j} y$ represents i dereferences of x and j dereferences of y using the dereference operator $*$. We can also view the *indlev* "$i|j$" as lists (also referred to as indirection list or *indlist*) containing i and j occurrences of $*$. This representation naturally allows field-sensitive handling of structures by using indirection lists containing field dereferences. Consider the statements $x = *y$ and $x = y \rightarrow n$ involving pointer dereferences. Since $x = y \rightarrow n$ is equivalent to $x = (*y).n$, we can represent the two statements by GPUs as shown below:

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

We achieve field sensitivity by enumerating field names. For statement $x = *y$, having a field-insensitive representation which does not distinguish between different fields, makes



• *Difference* of *indlev* of pivot y $(2-1)$ is computed. • Difference $(2-1)$ is *positive.* • *Add* the difference to *indlev* of x.	• *Remainder* of *indlist* of pivot y (Rem $([*],[*,n])$) is computed. • $[*]$ is *prefix* of $[*,n]$. • *Append* the remainder to *indlist* of x.

Figure 8.3: GPU composition using the abstraction of *indlev*s and *indlist*s.

no difference, but the GPU for statement $x = y \to n$ loses precision (Figure 8.1). Figure 8.2 illustrates the GPUs corresponding to the basic pointer assignments involving structures.

The dereference in the pointer expression $y \to n$ is represented by an *indlist* written as $[*, n]$ associated with pointer variable y. It means that, first the address in y is read and then the address in field n is read. On the other hand, the access $y.n$ as shown in the third row of Figure 8.2 can be mapped to location by adding the offset of field n to the virtual address of y at compile time. Hence, it can be treated as a separate variable which is represented by a node $y.n$ with an *indlist* $[*]$. We can also represent $y.n$ with a node y and an *indlist* $[n]$. For our implementation, we chose the former representation. However, the latter representation is more convenient for explaining the GPU compositions and hence we use it in the rest of this chapter. For structures, we ensure field sensitivity by maintaining *indlist* in terms of field names. We choose to handle unions field-insensitively to capture aliasing between its fields.

Recall that a GPU composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ involves balancing the *indlev* of the pivot in $\boldsymbol{c}$ and $\boldsymbol{p}$ (Section 4.3). With *indlist* replacing *indlev*, the operations remain similar in spirit, although now they become operations on lists rather than operations on numbers. To motivate the operations on *indlist*s, let us recall the operations on *indlev*s: GPU composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ requires balancing *indlev*s of the pivot which involves computing the difference between the *indlev* of the pivot in $\boldsymbol{c}$ and $\boldsymbol{p}$. This difference is then added to the *indlev*

$$\left(z \xrightarrow[t]{il_1|il_2} x\right) \circ^{\text{ts}} \left(v \xrightarrow[u]{il_3|il_4} y\right) := \begin{cases} z \xrightarrow[t]{il_1|il_5} y & (v = x) \wedge (il_2 = il_3@il_6) \wedge (il_5 = il_4@il_6) \\ \text{fail} & \text{otherwise} \end{cases}$$

$$\left(x \xrightarrow[t]{il_1|il_2} z\right) \circ^{\text{ss}} \left(v \xrightarrow[u]{il_3|il_4} y\right) := \begin{cases} y \xrightarrow[t]{il_5|il_2} z & (v = x) \wedge (il_1 = il_3@il_6) \wedge (il_5 = il_4@il_6) \\ & \wedge\, il_6 \neq [\,] \\ \text{fail} & \text{otherwise} \end{cases}$$

Definition 13: *GPU Composition* $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ *using indlists.*

of the non-pivot node in $\boldsymbol{p}$. Recall that a GPU composition is *valid* (Section 4.3.2) only when the *indlev* of the pivot in $\boldsymbol{c}$ is greater than or equal to the *indlev* of the pivot in $\boldsymbol{p}$. For convenience, we illustrate it again in the following example.

Example 64. Consider $\boldsymbol{p}: y \xrightarrow{1|0} x$ and $\boldsymbol{c}: w \xrightarrow{1|2} y$ where y is the pivot (Figure 8.3). Then a *TS* composition $\boldsymbol{c} \circ^{\text{ts}} \boldsymbol{p}$ is *valid* because *indlev* of y in $\boldsymbol{c}$ (which is 2) is greater than *indlev* of y in $\boldsymbol{p}$ (which is 1). The difference $(2 - 1)$ is added to the *indlev* of x (which then becomes 1) resulting in a reduced GPU $\boldsymbol{r}: w \xrightarrow{1|(2-1+0)} x$, i.e. $\boldsymbol{r}: w \xrightarrow{1|1} x$.

We define similar operations for *indlist*s. A GPU composition is *valid* if the *indlist* of the pivot in GPU $\boldsymbol{p}$ is a prefix of the *indlist* of the pivot in GPU $\boldsymbol{c}$. For example, the *indlist* "$[*]$" is a prefix of the *indlist* "$[*, n]$". The addition $(+)$ of the difference $(-)$ in the *indlev*s of the pivot to the *indlev* of one of the other two nodes is replaced by the list-append operation denoted @.

Similarly computing the difference $(-)$ in the *indlev* of the pivot is replaced by a 'list-difference' or 'list-remainder' operation, $\mathsf{Rem} : \textit{indlist} \times \textit{indlist} \to \textit{indlist}$; this takes two *indlist*s as its arguments where the first is a prefix of the second and returns the suffix of the second *indlist* that remains after removing the first *indlist* from it. Given $il_2 = il_1 \,@\, il_3$, $\mathsf{Rem}(il_1, il_2) = il_3$. When $il_1 = il_2$, the remainder il_3 is an empty *indlist* (denoted $[\,]$). A GPU composition is *valid* only when il_1 is a prefix of il_2; $\mathsf{Rem}(il_1, il_2)$ is computed only for *valid* GPU compositions. This is again a natural generalization of the integer based *indlev* formulation earlier.

Example 65. Consider $\boldsymbol{c} : w \xrightarrow{[*]|[*,n]} y$ and $\boldsymbol{p} : y \xrightarrow{[*]|[\,]} x$ where y is the pivot (Figure 8.3). We find the list remainder of the *indlist*s of y in the two GPUs. This operation ($\mathsf{Rem}([*], [*, n])$) returns $[n]$ which is appended to the *indlist* of node x (which is $[\,]$) resulting in a new *indlist* $[\,] \mathbin{@} [n] = [n]$ and thus, we get a reduced GPU $w \xrightarrow{[*]|[n]} x$.

Figure 8.3 gives an analogy between the operations performed for GPU composition using *indlev*s and *indlist*s.

The formal definition of GPU composition using *indlist*s is similar to that using *indlev*s (Definition 3) and is given in Definition 13. Note that for *TS* and *SS* compositions in the equations, the pivot is x. Besides, for *SS* composition, the condition $il_6 \neq [\,]$ (generalizing the strict inequality '<' in Definition 3) ensures that the consumer GPU does not redefine the location defined by the producer GPU. Unlike the case of pointers to scalars, *TS* and *SS* compositions are not mutually exclusive for pointers to structures. For example, an assignment $x \to n = x$ could have both *TS* and *SS* compositions with a GPU $\boldsymbol{p}$ defining x. The two compositions are independent because *SS* composition resolves the source of a consumer GPU whereas *TS* composition resolves the target of the GPU. Hence, they can be performed in any order.

A GPU composition is *desirable* if the *indlev* of $\boldsymbol{r}$ does not exceed that of $\boldsymbol{c}$. Similarly, in the case of *indlist*s, a GPU composition is *desirable* if *indlist*s of $\boldsymbol{r}$ (say $il'_1|il'_2$) does not exceed that of $\boldsymbol{c}$ (say $il_1|il_2$), i.e. $|il'_1| \leq |il_1| \;\wedge\; |il'_2| \leq |il_2|$ where $|il|$ denotes the length of *indlist* il. Note that, for *desirability*, we only need a smaller length and not a prefix relation between *indlist*s. In fact, the *indlist* in $\boldsymbol{r}$ is always a suffix of the *indlist* in $\boldsymbol{c}$ as illustrated by the following example.

Example 66. Consider the code snippet on right. The effect of statement 22 in the context of statement 21 can be seen as an assignment $z = y.n$. The composition of GPUs $\boldsymbol{c}{:}\, z \xrightarrow[22]{[*]|[*,n]} x$ and $\boldsymbol{p}{:}\, x \xrightarrow[21]{[*]|[\,]} y$ results in the GPU $\boldsymbol{r}{:}\, z \xrightarrow[22]{[*]|[n]} y$. The *indlist* of the target (y) of $\boldsymbol{r}$ is not a prefix of that of target (x) of $\boldsymbol{c}$ but is a suffix.

```
21 : x = &y;
22 : z = x → n;
```

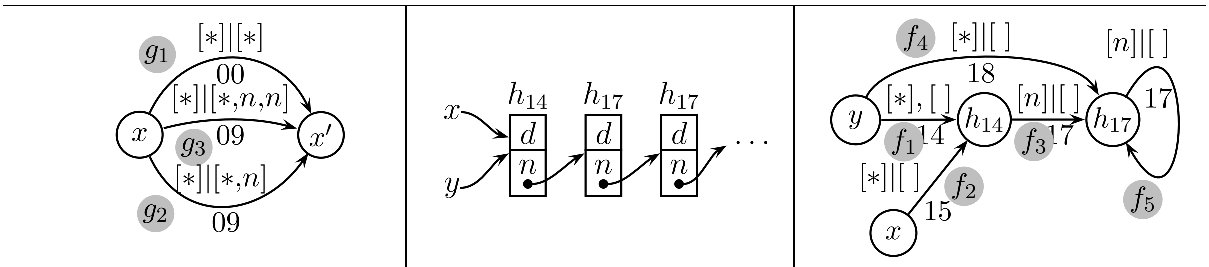
```
struct node ∗ x;

01  struct node {
02      struct node ∗ n;
03      int d;
04  };

05  void g() {
06      struct node ∗ y;
07      while (...) {
08          print x → d;
09          x = x → n;
10      }
11  }

12  void f() {
13      struct node ∗ y;
14      y = malloc(. . .);
15      x = y;
16      while (...) {
17          y → n = malloc(. . .);
18          y = y → n;
19      }
20      g();
21  }
```

(a) A program for creating a linked list and traversing it. We have omitted the null assignment for the last node of the list and the associated GPUs



(b) $\overline{\mathsf{RGOut}}_{11}$ (GPUs reaching the End of g for $k = 3$)

(c) Linked list created by procedure f

(d) $\overline{\mathsf{RGIn}}_{20}$ (GPUs reaching the call to g on line 20)

Figure 8.4: An example demonstrating the need of k-limiting summarization technique in addition to allocation-site-based abstraction for the heap. h_{14} and h_{17} are the heap nodes allocated on lines 14 and 17 respectively.

8.3 Summarization Using Allocation Sites

Under the allocation-site-based abstraction for the heap, the objects created by an allocation statement are collectively named by the allocation site and undergo weak update. Thus, a statement $x = \mathit{malloc}(\ldots)$ is represented by a GPU $x \xrightarrow[i]{[*]|[\,]} \mathrm{h}_i$ where h_i is the heap location created at the allocation site i. The example below illustrates how this bounds an unbounded heap in a GPG. For convenience, we identify GPUs using procedure names.

Example 67. For procedure f shown in Figure 8.4 we create heap objects h_{14} and h_{17} allocated at line numbers 14 and 17. The GPU set $\overline{\mathsf{RGIn}}_{20}$ in procedure f represents a linked list with x as its head pointer (Figure 8.4(d)) and h_{14} as its first node. The remaining nodes in the list are represented by the heap location h_{17} and are summarized by a self-loop over the node. This set of GPUs is computed as follows: The GPU $f_1\!:\!y \xrightarrow[14]{[*]|[\,]} h_{14}$ is created for allocation-site 14. The GPU $x \xrightarrow[15]{[*]|[*]} y$ composes with f_1 (under TS composition) to create a new GPU $f_2\!:\!x \xrightarrow[15]{[*]|[\,]} h_{14}$. When statement 17 is processed for the first time, GPU $y \xrightarrow[17]{[*,n]|[\,]} h_{17}$ composes with f_1 (under SS composition) to create a GPU $f_3\!:\!h_{14} \xrightarrow[17]{[n]|[\,]} h_{17}$. When statement 18 is processed for the first time, the GPU $y \xrightarrow[18]{[*]|[*,n]} y$ composes with f_1 (under TS composition) to create a GPU $y \xrightarrow[18]{[*]|[n]} h_{14}$ which is further composed with f_3 (under TS composition) to create a GPU $f_4\!:\!y \xrightarrow[18]{[*]|[\,]} h_{17}$. GPU f_4 kills GPU f_1 because y is redefined by statement 18. This completes the first iteration of the loop and the set of GPUs $\overline{\mathsf{RGOut}}_{19}$ is $\{f_2, f_3, f_4\}$ representing the following information:

- f_2 indicates that x points to the head of the linked list, i.e., heap location h_{14}.
- f_3 indicates that the field n of heap location h_{14} points to heap location h_{17}.
- f_4 indicates that y points to heap location h_{17}.

In the second iteration of the reaching GPUs analysis over the loop, $\overline{\mathsf{RGOut}}_{15}$ and $\overline{\mathsf{RGOut}}_{19}$ are merged to compute $\overline{\mathsf{RGIn}}_{16}$ as $\{f_1, f_2, f_3, f_4\}$. When statement 17 is processed for the second time, the GPU $y \xrightarrow[17]{[*,n]|[\,]} h_{17}$ composes with

- f_1 (under SS composition) to create f_3, and with
- f_4 (under SS composition) to create $f_5\!:\!h_{17} \xrightarrow[17]{[n]|[\,]} h_{17}$.

When statement 18 is processed for the second time, f_4 is recreated killing f_1. This completes the second iteration of the loop and the set of GPUs $\overline{\mathsf{RGIn}}_{20}$ is $\{f_1, f_2, f_3, f_4, f_5\}$. The new GPU f_5 implies that the field n of heap location h_{17} holds the address of heap location h_{17}. The self loop represents an unbounded list $\big(h_{17} \xrightarrow{n} h_{17} \xrightarrow{n} h_{17} \xrightarrow{n} h_{17} \ldots\big)$ under the allocation-site-based abstraction. The third iteration of reaching GPUs analysis over the loop does not add any new information and reaching GPUs analysis reaches a fixed point.

The following example discusses the absence of blocking in the procedures in Figure 8.4.

Example 68. The GPUs in $\overline{\mathsf{RGIn}}_{14}$ reach statement 17 unblocked because there is no barrier. Since the pointee of y is available, the set $\overline{\mathsf{RGGen}}_{14}$ does not contain any indirect GPUs and hence do not contribute to the blocking of any GPUs. If the allocation site at statement 14 was not available, then the GPU for statement 17 would not have been reduced and hence the set $\overline{\mathsf{RGGen}}_{17}$ would contain an indirect GPU $y \xrightarrow[17]{[*,n]|[]} h_{17}$. This GPU would block all GPUs in $\overline{\mathsf{RGIn}}_{18}$ and in turn would be blocked by the GPUs in $\overline{\mathsf{RGGen}}_{18}$ so that it cannot be used for reduction of any successive GPUs.

8.4 Summarization Using k-Limiting

This section shows why allocation-site-based abstraction is not sufficient for a bottom-up points-to analysis although it serves the purpose well in a top-down analysis.

8.4.1 The Need for k-Limiting

In some cases, the allocation site may not be available during the construction of the GPG of a procedure. In Figure 8.4, when the GPG is constructed for procedure g, we do not know the allocation site because the accesses to heap in procedure g refer to the data-structure created in procedure f. Thus allocation-site-based abstraction is not useful for constructing the GPG for procedure g and the indirection lists grow without bound.

In a top-down analysis, k-limiting is not required because allocation sites are propagated from callers to callees.

Example 69. When the GPG for procedure g in Figure 8.4 is constructed, we have a boundary definition $g_1 : x \xrightarrow[00]{[*]|[*]} x'$ at the start of the procedure. In the first iteration of the analysis over the loop, the GPU $x \xrightarrow[09]{[*]|[*,n]} x$ composes with g_1 (under TS composition) creating a reduced GPU $g_2 : x \xrightarrow[09]{[*]|[*,n]} x'$. The GPU g_2 kills GPU g_1

because x is redefined by statement at 09. However, the merge at the top of the loop reintroduces it. In the second iteration, the GPU $x \xrightarrow[09]{[*]||[*,n]} x$ composes with g_1 to recreate g_2, and with g_2 to create $g_3 : x \xrightarrow[09]{[*]||[*,n,n]} x'$. In the third iteration, we get an additional GPU $g_4 : x \xrightarrow[09]{[*]||[*,n,n,n]} x'$ apart from g_2 and g_3. This continues and the indirection lists of the GPUs between x and x' grow without bound leading to non-termination.

There are two ways of handling traversals of data structures created in some other (caller) procedure.

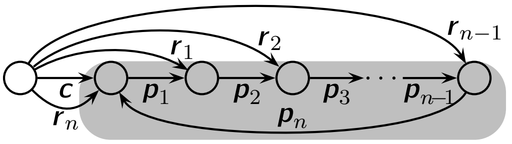
- As the above example illustrates, we perform compositions involving upwards exposed variables inspite of these compositions being *valid* but *undesirable*.
- Alternatively, we can postpone these compositions (as suggested before) until call inlining enables their reduction.

We use the first approach and bound the length of indirection lists using k-limiting. This limits the participation of the GPUs in the fixed-point computation for the procedures containing them. The second approach requires the GPUs to participate in the fixed-point computations for the callers as well. This could cause inefficiency.

While the use of k-limiting in a bottom-up approach seems like an additional restriction, unless the locations involved in a pointer chain are allocated by $m > k$ distinct allocation sites, there is no loss of precision compared to a top-down approach that uses allocation-site-based abstraction.

8.4.2 Incorporating k-Limiting

We limit the length of *indlist*s to k such that an *indlist* is exact up to $k - 1$ dereferences and approximate for k or more dereferences in terms of an unbounded number of dereferences. Besides, the dereferences are field-insensitive beyond k. This summarization is implemented by redefining the list concatenation operator @ such that for $il_1 \,@\, il_2$, the result is a k-limited prefix of the concatenation of il_1 and il_2.



- The shaded part shows the GPUs in $\overline{\mathsf{RGIn}}$.
- Let $\boldsymbol{r}_0 = \boldsymbol{c}$. Then $\boldsymbol{r}_i = \boldsymbol{r}_{i-1} \circ^{\tau} \boldsymbol{p}_i$, $i > 0$.
- For simplicity, the directions chosen in the GPUs illustrate only *TS* compositions.

Figure 8.5: Series of compositions and its consequence when the graph induced by the GPUs in $\overline{\mathsf{RGIn}}$ (shown by the shaded part) has a cycle. The compositions may happen more than the required number of times, resulting in a points-to edge.

Example 70. The set of GPUs $\overline{\mathsf{RGOut}}_{11}$ reaching the End of procedure g of Figure 8.4, for $k = 3$ is given in the Figure 8.4(b). A GPU between x and x' has an *indlist* $[*, n]$ of length 2 and all *indlist*s of length ≥ 3 are approximated by $[*, n, n]$.

GPU $g_1 : x \xrightarrow[00]{[*][*]} x'$ in the GPG for procedure g represents the effect of **while** loop not executed even once. GPU $g_2 : x \xrightarrow[09]{[*] | [*,n]} x'$ represents the effect of the first iteration of the **while** loop. The GPU $g_3 : x \xrightarrow[09]{[*] | [*,n,n]} x'$ represents the combined effect of the second and all subsequent iterations of the **while** loop. The GPG of procedure g ($\varDelta_g$) contains a single GPB which in turn contains a set of GPUs $\{g_2, g_3\}$.

Note that an explicit summarization is required only for heap locations and address-escaped stack locations in recursive procedures because the *indlist*s can grow without bound only in these cases[3].

The GPU composition defined in Section 8.2 (Definition 13) is extended to handle k-limited *indlist*s in the following way: The removal of a prefix from a k-limited *indlist* in the Rem operation is over-approximated by suffixing special field-insensitive dereferences denoted by "†" where † represents any field. For an operation $\mathsf{Rem}(il_1, il_2)$, il_1 must be a prefix of il_2 as explained in Section 8.2. Let $il_2 = il_1 \mathbin{@} il_3$ for $\mathsf{Rem}(il_1, il_2)$. We define a summarized list-remainder operation sRem : *indlist* × *indlist* → 2^{indlist} which takes two

[3] We also perform weak updates for address-escaped variables (Section 9.1) because they share many similarities with heap locations. Like heap locations, address-escaped variables could outlive the lifetime of the procedures that create them. They potentially represent multiple concrete locations because of multiple calls to procedures. Further, this number could be unbounded in the case of recursive calls.

*indlist*s as its arguments and computes a set of *indlist*s as shown below:

$$\mathsf{sRem}(il_1, il_2) = \begin{cases} \{il_3 \mid il_2 = il_1 \mathbin{@} il_3\} & |il_2| < k \\ \{il_3 \mathbin{@} \sigma \mid il_2 = il_1 \mathbin{@} il_3, \sigma \text{ is a seq. of } \dagger, 0 \leq |\sigma| \leq |il_1|\} & \text{otherwise} \end{cases}$$

Observe that sRem is a generalization of Rem defined in Section 8.2 because it computes a set of *indlist*s when its second argument is a k-limited *indlist*; for non k-limited *indlist*, sRem returns a singleton set. The longest *indlist* in the set computed by sRem represents a summary whereas the other *indlist*s are exact in length but approximate in terms of fields because of field insensitivity introduced by †.[4] This is illustrated in the example below.

Example 71. For $k = 3$, some examples of the sets of *indlist*s computed by the sRem operation are shown below:

$$\mathsf{sRem}([*], [*, n, n]) = \{[n, n], [n, n, \dagger]\}$$
$$\mathsf{sRem}([*, n], [*, n, n]) = \{[n], [n, \dagger], [n, \dagger, \dagger]\}$$
$$\mathsf{sRem}([*, n, n], [*, n, n]) = \{[\,], [\dagger], [\dagger, \dagger], [\dagger, \dagger, \dagger]\}$$

For the last case, the sRem operation can be viewed as an operation that creates an intermediate set $S = \{[*, n, n], [*, n, n, \dagger], [*, n, n, \dagger, \dagger], [*, n, n, \dagger, \dagger, \dagger]\}$ obtained by adding upto 3 occurrences of † (because $k = 3$). The sRem operation can then be viewed as a collection of $\mathsf{Rem}([*, n, n], \sigma)$ for each σ in this set:

$$\mathsf{sRem}([*, n, n], [*, n, n]) = \{\mathsf{Rem}([*, n, n], \sigma) \mid \sigma \in S\}$$

The first two cases in this example can also be explained in a similar manner.

GPU composition using *indlev*s (Section 4.3.2) or using *indlist*s (Section 8.2) is a partial operation defined to compute a single GPU as its result when it succeeds. Since we do not have a representation for an "invalid" GPU, we model failure by defining GPU composition as a partial function for GPUs containing *indlev*s or non-k-limited *indlist*s.

[4] This is somewhat similar to materialization [89] which extracts copies out of summary representation of an object to create some exact objects.

```
Input: $\boldsymbol{c}$         // The consumer GPU to be simplified.
       $R$         // The context (set of GPUs) in which $\boldsymbol{c}$ is
                   // to be simplified.
       Used        // The set of GPUs used for GPU reduction
                   // for a GPU.
Output: Red        // The set of simplified GPUs equivalent to $\boldsymbol{c}$.
GPU_reduction ($\boldsymbol{c}$, $R$, Used)
{  if $(R = \emptyset \wedge \boldsymbol{c} = \cdot \xrightarrow[\cdot]{1|0} \cdot)$  // $\boldsymbol{c}$ is a points-to edge
   {  Red = $\{\boldsymbol{c}\}$
      composed = true
   }
   else
   {  Red = $\emptyset$
      composed = false
   }
   for each $\gamma \in (R -$ Used$)$
   {  for each $r \in (\boldsymbol{c} \circ^{\text{ts}} \gamma)$
      {  Red = Red $\cup$ GPU_reduction $(r, R,$ Used $\cup$ $\{\gamma\})$
         composed = true
      }
      for each $r \in (\boldsymbol{c} \circ^{\text{ss}} \gamma)$
      {  Red = Red $\cup$ GPU_reduction $(r, R,$ Used $\cup$ $\{\gamma\})$
         composed = true
      }
   }
   if ($\neg$ composed )
      Red = Red $\cup$ $\{\boldsymbol{c}\}$
   return Red
}
```

Definition 14: *GPU Reduction* $\boldsymbol{c} \circ R$ *for handling heap.*

However, when *indlist*s are summarized using k-limiting, sRem naturally computes a set of *indlist*s (unlike Rem which computes a single *indlist*). This allows us to define GPU composition as a total function, since we can express the previous partiality simply by returning an empty set.

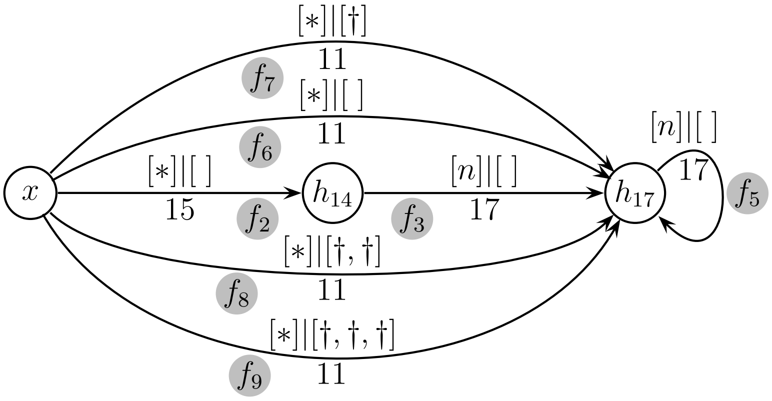


Figure 8.6: The set of GPUs $\overline{\mathsf{RGOut}}_{20}$ after the call to procedure g in procedure f of Figure 8.4. Local variable y has been eliminated.

8.5 Extending GPU Reduction to Handle Cycles in GPUs

In the presence of heap, the graph induced by the set of GPUs reaching a GPB can contain cycles of the following two kinds:

- Cycles arising out of creation of a recursive data structure in a procedure under allocation-site-based abstraction. This manifests itself in the form of a cycle involving heap nodes h_i as illustrated in Example 67 in Section 8.3. These cycles are closed form representations of acyclic unbounded paths in the memory.
- Cycles arising out of cyclic data structures. These cycles represent cycles in the memory graph.

Both these cases of cycles are handled by GPU composition using sRem operation over indirection lists. Definition 14 extends the algorithm for GPU reduction to use the new definition of GPU composition (which is a total function) which computes a set of GPUs instead of a single GPU.

We explain this in the context of reaching GPUs analysis with blocking ($\overline{\mathsf{RGIn}}$ and $\overline{\mathsf{RGOut}}$). The same explanation holds for reaching GPUs analysis without blocking (RGIn and RGOut). For GPU reduction $\boldsymbol{c} \circ R$, an ***admissible*** composition $\boldsymbol{r}_1 = \boldsymbol{c} \circ^{\tau} \boldsymbol{p}_1$ (where $\boldsymbol{p}_1 \in \overline{\mathsf{RGIn}}$) may lead to another composition $\boldsymbol{r}_2 = \boldsymbol{r}_1 \circ^{\tau} \boldsymbol{p}_2$ (where $\boldsymbol{p}_2 \in \overline{\mathsf{RGIn}}$). This in turn may lead to another composition thereby creating a chain of compositions. If the graph induced by the reaching GPUs (i.e. GPUs in $\overline{\mathsf{RGIn}}$) has a cycle (as illustrated in Example 67 in Section 8.3), some $\boldsymbol{p}_m$ must be adjacent to $\boldsymbol{p}_1$ with the length of the cycle being $m+1$ as illustrated in Figure 8.5. The lengths of *indlist*s in $\boldsymbol{r}_i$ would be smaller than (or equal to) those in $\boldsymbol{r}_{i-1}$ because of *admissibility*. If the length of an *indlist* in $\boldsymbol{c}$ exceeds m, the series of compositions would resume with $\boldsymbol{p}_1$ after the composition with $\boldsymbol{p}_m$. In other words, after computing $\boldsymbol{r}_{m-1}$ using the composition $\boldsymbol{r}_{m-2} \circ \boldsymbol{p}_m$, the next GPU $\boldsymbol{r}_m$ would be computed using the composition $\boldsymbol{r}_{m-1} \circ \boldsymbol{p}_1$ and the process will continue until some $\boldsymbol{r}_j$, $j \geq m$ is a points-to edge.[5] Thus, we will have more compositions than required and the result of GPU reduction may not represent the updates of locations that are updated by the original GPU $\boldsymbol{c}$. In order to prohibit this, we allow a GPU $\boldsymbol{p}$ to be used only once in a chain of compositions.

Hence, the new definition of GPU reduction (Definition 14) uses an additional argument, Used, which maintains a set of GPUs that have been used in a chain of GPU compositions. For the top level non-recursive call to $\mathsf{GPU\_reduction}$, $\mathsf{Used} = \emptyset$. In the case of pointers to scalars, a graph induced by a set of GPUs cannot have a cycle, hence a GPU $\boldsymbol{p}$ cannot be used multiple times in a series of GPU compositions. Therefore, we did not need set Used for defining GPU reduction in the case of pointers to scalars (Definition 4).

Example 72 illustrates GPU reduction with 3-limited *indlist*s.

Example 72. The GPU reduction with 3-limited *indlist*s using GPU g_3 of Δ_g shown in Figure 8.4(b) is as follows: At the call site 20 in procedure f of Figure 8.4(a), the upwards-exposed variable x' in Δ_g is substituted by x in Δ_f (see Chapter 6). All GPU compositions for this examples are *TS* compositions. The GPUs in $\overline{\mathsf{RGIn}}_{20}$ (Figure 8.4(d)) are used for composition.

The GPU composition $g_2 \circ f_2$ for $f_2 : x \xrightarrow[15]{[*]|[\,]} h_{14}$ and $g_2 : x \xrightarrow[11]{[*]|[*,n]} x$ (with x substituting for x') creates a reduced GPU $x \xrightarrow[11]{[*]|[n]} h_{14}$ which is further composed with $f_3 : h_{14} \xrightarrow[17]{[n]|[\,]} h_{17}$ to create a reduced GPU $f_6 : x \xrightarrow[11]{[*]|[\,]} h_{17}$ (Figure 8.6).

Now GPU g_3 must be composed with f_2, f_3 and f_5. The composition $g_3 \circ f_2$ for $g_3 : x \xrightarrow[11]{[*]|[*,n,n]} x$ creates two GPUs $x \xrightarrow[11]{[*]|[n,n]} h_{14}$ and $x \xrightarrow[11]{[*]|[n,n,\dagger]} h_{14}$. The newly cre-

[5] Note that this happens for reducing a single GPU $\boldsymbol{c}$ in the context of $\overline{\mathsf{RGIn}}$ and does not require a cycle in the GPG.

ated GPU $x \xrightarrow[11]{[*]\|[n,n]} h_{14}$ is further composed with f_3 to create GPU $x \xrightarrow[11]{[*]\|[n]} h_{17}$ which is further composed with f_5 to recreate GPU $f_6 : x \xrightarrow[11]{[*]\|[\,]} h_{17}$. The GPU composition between the other newly created GPU $x \xrightarrow[11]{[*]\|[n,n,\dagger]} h_{14}$ and f_3 creates GPUs $x \xrightarrow[11]{[*]\|[n,\dagger]} h_{17}$ and $x \xrightarrow[11]{[*]\|[n,\dagger,\dagger]} h_{17}$. The GPU $x \xrightarrow[11]{[*]\|[n,\dagger]} h_{17}$ further composes with f_5 creating a GPU $f_7 : x \xrightarrow[11]{[*]\|[\dagger]} h_{17}$ while the composition between GPUs $x \xrightarrow[11]{[*]\|[n,\dagger,\dagger]} h_{17}$ and f_5 creates two reduced GPUs $f_8 : x \xrightarrow[11]{[*]\|[\dagger,\dagger]} h_{17}$ and $f_9 : x \xrightarrow[11]{[*]\|[\dagger,\dagger,\dagger]} h_{17}$. Note that GPU f_5 is used only once in a series of compositions (Example 73 explains this).

The final reduced GPUs f_6, f_7, f_8, and, f_9 are members of the set $\overline{\mathsf{RGOut}}_{21}$ containing the GPUs reaching the End of procedure f (as shown in Figure 8.6). These reduced GPUs represent the following information:

- f_6 implies that x now points-to heap location h_{17}.
- f_7 implies that x points-to heap locations that are one dereference away from the heap location h_{17}.
- f_8 implies that x points-to heap locations that are two dereferences away from the heap location h_{17}.
- f_9 implies that x points-to heap locations that are beyond two dereferences from the heap location h_{17}.

Thus, x points to every node in the linked list.

Example 73 illustrates the need for restricting the use of a GPU only once in a chain of compositions (GPU reduction in Definition 14).

Example 73. Observe that GPUs f_7, f_8 and f_9 can be further composed with GPU f_5. The composition of f_7 with f_5 creates GPU f_6. Similarly, repetitive compositions of f_8 with f_5 also creates GPU f_6. This indicates that x points to only h_{17} and misses out the fact that x points to every location in the linked list which is represented by h_{17} and is represented by GPUs f_7, f_8 and f_9.

A cycle in a graph induced by a set of GPUs could also occur because of a cyclic data structure.

Example 74. Let an assignment $y \to n = x$ be inserted in procedure f after line 19 in Figure 8.4. This creates a circular linked list instead of a simple linked list. This will cause inclusion of the GPU $h_{17} \xrightarrow{[n]|[\,]} h_{14}$ in Figure 8.4(d), thereby creating a cycle between the nodes h_{14} and h_{17}.

8.6 Chapter Summary

In this chapter, we have generalized the concept of *indlev*s to indirection lists (*indlist*s) to handle structures and heap accesses field sensitively. Heap locations are abstracted using allocation sites. This approximation allows us to handle the unbounded nature of heap as if it were bounded. An additional summarization technique based on k-limiting is used to bound the accesses in a loop when the locations being accessed are not allocated within the procedure. Both these summarization techniques are required to create a decidable version of our method of constructing GPGs in the presence of heap which is unbounded.

We have extended the current definitions of GPU composition and GPU reduction to handle *indlist*s, k-limiting summarization, and cycles that may be present in the graph induced by a set of GPUs. The optimizations performed on GPGs and the required data flow analyses (reaching GPUs analyses and coalescing analysis) remain the same.

Chapter 9

Empirical Evaluation

The main motivation of our implementation was to evaluate the effectiveness of our optimizations in handling the following challenge for practical programs:

> A procedure summary for flow- and context-sensitive points-to analysis needs to model the accesses of pointees defined in the callers and needs to maintain control flow between memory updates when the data dependence between them is not known. Thus, the size of a summary can be potentially large. This effect is exacerbated by the transitive inlining of the summaries of the callee procedures which can increase the size of a summary exponentially thereby hampering the scalability of analysis.

Section 9.1 describes our implementation, Section 9.2 describes the metrics that we have used for our measurements, Section 9.3 describes our empirical observations, and Section 9.4 analyzes our observations and describes the lessons learnt.

9.1 Implementation and Experiments

We have implemented GPG-based points-to analysis in GCC 4.7.2 using the LTO framework and have carried out measurements on SPEC CPU2006 benchmarks on a machine with 16 GB RAM with eight 64-bit Intel i7-4770 CPUs running at 3.40GHz.

Our method eliminates non-address-taken local variables using the def-use chains explicated by the SSA-form. Although we construct GPUs involving such variables, they are used for computing the points-to information within the procedure and do not appear in the GPG of the procedure. If a GPU defining a global variable or a parameter reads a non-address-taken local variable, we identify the corresponding producer GPUs by traversing the def-use chains transitively. This eliminates the need for filtering out the local variables from the GPGs for inlining them in the callers. As a consequence, a GPG of a procedure consists of GPUs that involve global variables[1], parameters of the procedure, and the return variable which is visible in the scope of its callers. Since non-address-taken local variables have SSA versions, storing the GPUs that define them flow-insensitively results in no loss of precision.

All address-taken local variables in a procedure are treated as global variables because they can escape the scope of the procedure. However, these variables are not strongly updated because they could represent multiple locations.

We approximate the heap memory by maintaining k-limited indirection lists of field dereferences for $k = 3$ (see Chapter 8). An array is treated as a single variable in the following sense: accessing a particular element is seen as accessing every possible element and updates are treated as weak updates. This applies to both when arrays of pointers are manipulated, as well as when arrays are accessed through pointers. Since there is no kill owing to weak update, arrays are maintained flow-insensitively by our analysis.

For pointer arithmetic involving a pointer to an array, we approximate the pointer being defined to point to every element of the array. For pointer arithmetic involving other pointers, we approximate the pointer being defined to point to every possible location. Our current implementation handles only locally defined function pointers (Section 6.4) but can be easily extended to handle function pointers defined in the calling contexts too.

We have also implemented flow-insensitive points-to analysis by collecting the GPUs in a *GPG store* which differs from a GPB in that GPUs within a store can compose with each other whereas those in GPB cannot. This allowed us to implement the following:

- Flow- and context-insensitive (FICI) points-to analysis. For each benchmark program, we collected all GPUs across all procedures in a common store and performed all possible reductions. The resulting GPUs were classical points-to edges representing the flow- and context-insensitive points-to information.

- Flow-insensitive and context-sensitive (FICS) points-to analysis. For each procedure of a benchmark program, all GPUs within the procedure were collected in a store

[1] From now on we regard static, heap-summary nodes, and address-taken local variables as 'special global variables' that do not undergo strong updates.

Program	kLoC	# of pointer stmts	# of call sites	# of procs.	Proc. count for different buckets of # of calls				# of procs. requiring different no. of PTFs based on the no. of aliasing patterns					
					2-5	5-10	10-20	20+	2-5	6-10	11-15	15+	2-5	15+
	A	*B*	*C*	*D*	*E*				*F*				*G*	
lbm	0.9	370	30	19	5	0	0	0	8	0	0	0	13	0
mcf	1.6	480	29	23	11	0	0	0	0	0	0	0	4	0
libquantum	2.6	340	277	80	24	11	4	3	7	3	1	0	14	4
bzip2	5.7	1650	288	89	35	7	2	1	22	0	0	0	28	2
milc	9.5	2540	782	190	60	15	9	1	37	8	0	1	35	25
sjeng	10.5	700	726	133	46	20	5	6	14	3	1	3	10	14
hmmer	20.6	6790	1328	275	93	33	22	11	62	5	3	4	88	32
h264ref	36.1	17770	2393	566	171	60	22	16	85	17	5	3	102	46
gobmk	158.0	212830	9379	2699	317	110	99	134	206	30	9	10	210	121

Table 9.1: Benchmark characteristics relevant to our analysis.

for the procedure and all possible reductions were performed. The resulting store was used as a summary in the callers of the procedure giving context-sensitivity. In the process, the GPUs are reduced to classical points-to edges using the information from the calling context. This represents the flow-insensitive and context-sensitive points-to information for the procedure.

The third variant i.e., flow-sensitive and context-insensitive (FSCI) points-to analysis can be modelled by constructing a supergraph by joining the control flow graphs of all procedures such that calls and returns are replaced by gotos. This amounts to a top-down approach (or a bottom-up approach with a single summary for the entire program instead of separate summaries for each procedure). For practical programs, this initial GPG is too large for our analysis to scale. Our analysis achieves scalability by keeping the GPGs as small as possible at each stage. Therefore, we did not implement this variant of points-to analysis. Note that the FICI variant is also not a bottom-up approach because a separate summary is not constructed for every procedure. However, it was easy to implement because of a single store.

9.2 Measurements

We have measured the following for each benchmark program. Since the number of procedures varies significantly across the benchmark programs causing the number of GPUs and GPBs to vary across GPGs, we have plotted such data in terms of percentages. The actual procedure counts are given in Appendix A.

1) Characteristics of benchmark programs (Table 9.1).

2) Effectiveness of redundancy elimination optimizations (Figure 9.1):

   a) The number of dead GPUs for each procedure.

      The first plot in Figure 9.1 shows the points (u, v) that are computed as follows: Let x and y denote the number of GPUs before and after dead GPU elimination respectively. Then the number of dead GPUs is $d = x - y$ and percentage of dead GPUs is computed as $u = (d/x) \times 100$ (rounded to the nearest integer). Let t be the total number of procedures in a program and let p procedures have $u\%$ dead GPUs. Then the percentage of procedures is $v = (p/t) \times 100$.

   b) The number of empty GPBs for each procedure created by strength reduction, call inlining and dead GPU elimination. The impact of empty GPB elimination is shown by plotting the percentage of empty GPBs (similar to percentage of dead GPUs) on X-axis and the corresponding percentage of procedures that contain those empty GPBs on Y-axis.

   c) Reduction in the number of GPBs due to coalescing. The third plot gives the reduction in the number of GPBs because of coalescing. This information is computed in a similar manner as that of dead GPUs and empty GPBs.

   d) Reduction in the number of back edges due to coalescing. The third plot gives the reduction in the number of back edges because of coalescing. This information is computed in a similar manner as that of dead GPUs and empty GPBs.

3) The goodness metric of the optimized procedure summaries (Figure 9.2):

   a) Number of GPBs in the optimized GPGs. The first plot in Figure 9.2 shows the count of GPBs in an optimized GPG of a procedure on X-axis. Y-axis shows the percentage of procedures for a given GPB count.

b) Number of GPUs in the optimized GPGs. The second plot in Figure 9.2 shows the count of GPUs in an optimized GPG of a procedure on X-axis. Y-axis shows the percentage of procedures for a given GPU count.

c) Number of GPUs that are dependent on locally defined pointers alone. The third plot in Figure 9.2 shows the percentage of context-independent information in terms of points (u, v) that are computed as follows: Let x and y denote the number of GPUs with *indlev* "1|0" and total number of GPUs respectively in an optimized GPG. Then $u = (x/y) \times 100$ (rounded to the nearest integer). For a given value of u, v is a percentage of procedures.

4) The number of GPBs in a GPG (Figure 9.3):

The first plot in Figure 9.3 shows the points (u, v). The value $u = (x/y) \times 100$ (rounded to the nearest integer) and v is the percentage of procedures where x and y represent the following:

a) x is the number of GPBs in a GPG obtained after call inlining and y is the number of basic blocks in the CFG[2].

b) x is the number of GPBs in a GPG obtained after all optimizations and y is the number of basic blocks in the CFG.

c) x is the number of GPBs in a GPG obtained after all optimizations and y is the number of GPBs in a GPG obtained after call inlining.

5) The number of GPUs in a GPG (Figure 9.4): The second plot shows the number of GPUs relative to the number of pointer assignments. This is computed in a manner similar to the number of GPBs relative to the number of basic blocks as explained in the item above.

6) The number of control flow edges in a GPG (Figure 9.5): The third plot shows the number of control flow edges in a GPG relative to those in a CFG. This is also

[2] Since GPGs have callee GPGs inlined within them, for a fair comparison, the CFG size must be counted by accumulating the sizes of the CFGs of the callee procedures. This is easy for non-recursive procedures. For recursive procedures, we accumulate the size of a CFG as many times as the number of inlinings of the corresponding GPG (Section 6.3).

Program	# of Proc. which have 0 GPUs	# of Proc. which have $\varDelta_\top$ as GPG	# of Proc. in which back edges are present in a CFG	# of Proc. in which back edges are present in a GPG	Exported Definitions	Imported Uses	# Queued GPUs	# Soundness Alerts
lbm	15	0	10	0	1.68	16.63	0	0
mcf	12	0	20	1	12.30	29.26	117	0
libquantum	38	0	36	0	1.54	1.89	0	0
bzip2	78	8	43	1	1.21	17.37	0	0
milc	184	3	94	0	0.70	6.14	0	0
sjeng	101	2	65	0	0.81	1.77	0	0
hmmer	242	5	153	0	2.26	13.02	19	0
h264ref	434	3	308	5	1.60	26.75	13	0
gobmk	1436	2	464	8	0.39	1.36	6	0

Table 9.2: Miscellaneous data about the GPGs.

computed in a manner similar to the number of GPBs and the number of GPUs as explained above.

7) Miscellaneous data about GPGs (Table 9.2).

8) Time measurements (Figure 9.6):

   a) FSCS (with and without blocking), FICI, and FICS variants of points-to analyses (second plot).

   b) Time for different optimizations without blocking (third plot).

   c) Time for different optimizations with blocking (fourth plot).

9) Average points-to pairs per procedure in FSCS, FICI, and FICS variants of points-to analyses. This data is plotted in the first plot of Figure 9.6.

9.3 Observations

We describe our observations about the sizes of GPGs, GPG optimizations, and performance of the analysis. Observations related to the time measurements are presented in the end. Section 9.4 discusses these observations by analyzing them.

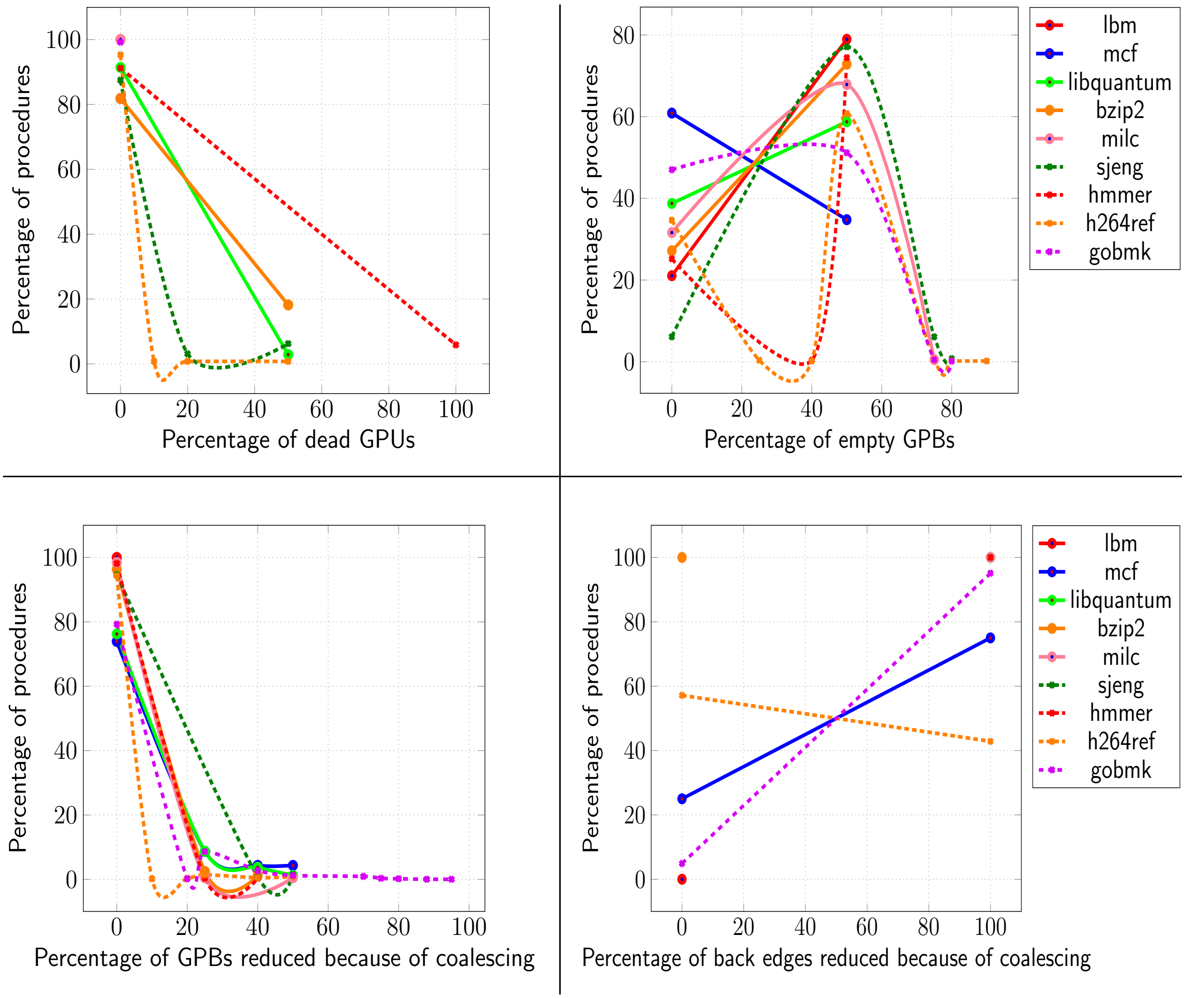


Figure 9.1: Effectiveness of redundancy elimination optimizations. Benchmarks libquantum, milc, sjeng, and hmmer have all procedures whose all back edges are eliminated because of coalescing shown by the same point (100, 100) in the fourth plot. Since the points are overlaid on each other, they are not visible separately.

9.3.1 Effectiveness of Redundancy Elimination Optimizations

We observe that:

(a) The percentage of dead GPUs is very small and the dead GPU elimination optimization is the least effective of all optimizations. Also, this optimization requires very little time compared to other optimizations (see Figure 9.6). Hence, disabling the optimization will neither improve the efficiency or scalability of the analysis nor will it affect the compactness of the GPGs.

(b) The transformations performed by call inlining, strength reduction, and dead GPU

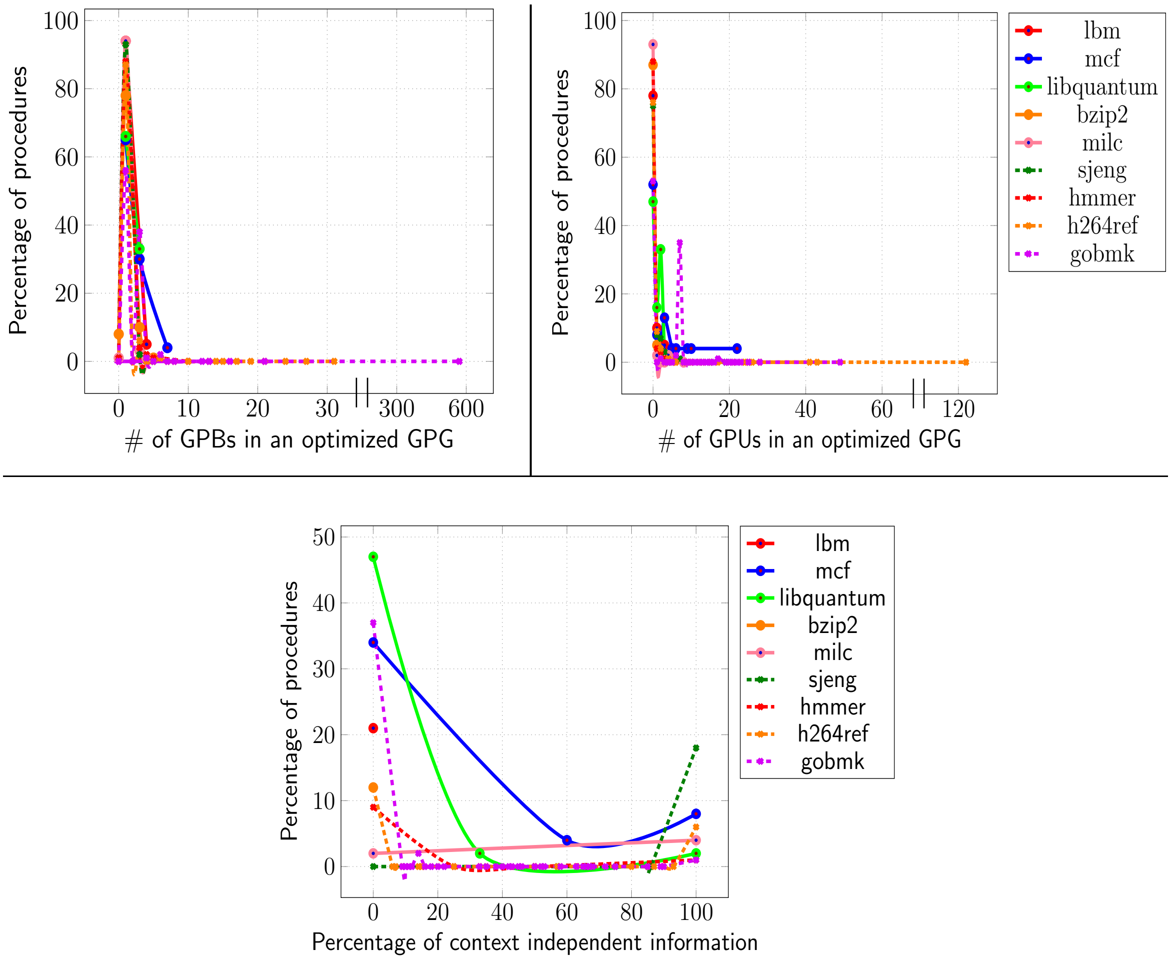


Figure 9.2: Goodness measure of procedure summaries. A break in X-axis shown by two parallel lines is a discontinuity necessitated by wide variation in the number of GPUs and GPBs across benchmarks.

elimination create empty GPBs which are removed by empty GPB elimination. For most procedures, 0%-5% or close to 50% of GPBs are empty.

(c) The last optimization among the redundancy elimination optimizations, coalesces the adjacent GPBs that do not require control flow between them. In our experience, many benchmarks had some very large GPGs in the presence of recursion. GPGs for recursive procedures are constructed by repeated inlinings of recursive calls. Coalescing was most effective for such procedures. Once these GPGs were optimized, the GPGs of the caller procedures did not have much scope for coalescing. In other words, coalescing did not cause uniform reduction across all GPGs but helped the most critical GPGs. Hence we observe a reduction of 20% to 50% of GPBs for some

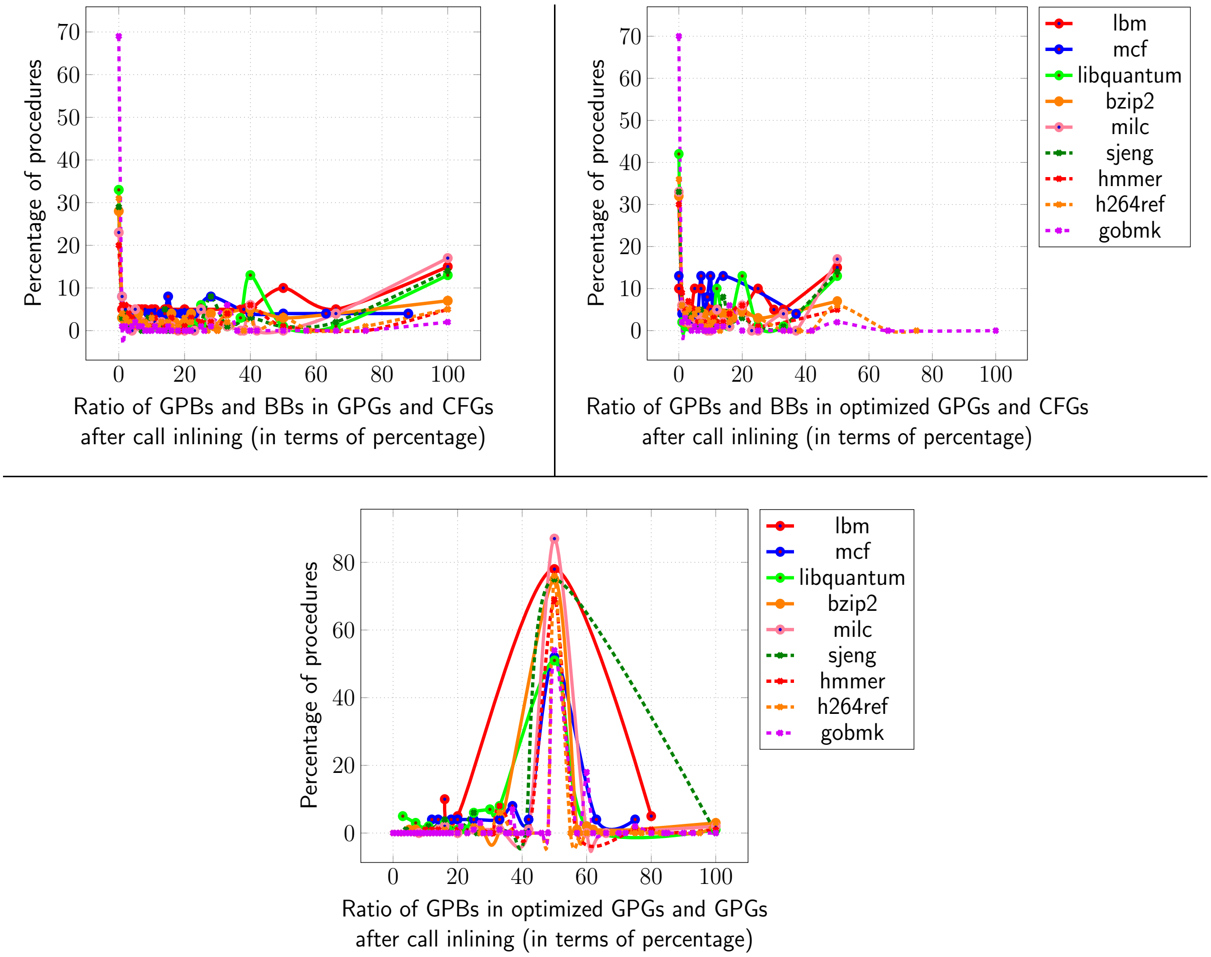


Figure 9.3: Size of GPGs relative to the size of corresponding procedures in terms of GPBs and basic blocks.

but not majority of procedures.

Even if coalescing did not reduce the number of GPBs uniformly, it eliminated almost all back edges as shown in fourth plot in Figure 9.1. This is significant because most of the inlined GPGs are acyclic and hence analyzing the GPGs of the callers does not require additional iterations in a fixed-point computation.

9.3.2 Goodness of Procedure Summaries

This data is presented in Tables 9.1, 9.2, and Figure 9.2. We use the following goodness metrics on procedure summaries:

(a) Reusability. The number of calls to a procedure is a measure for the reusability of

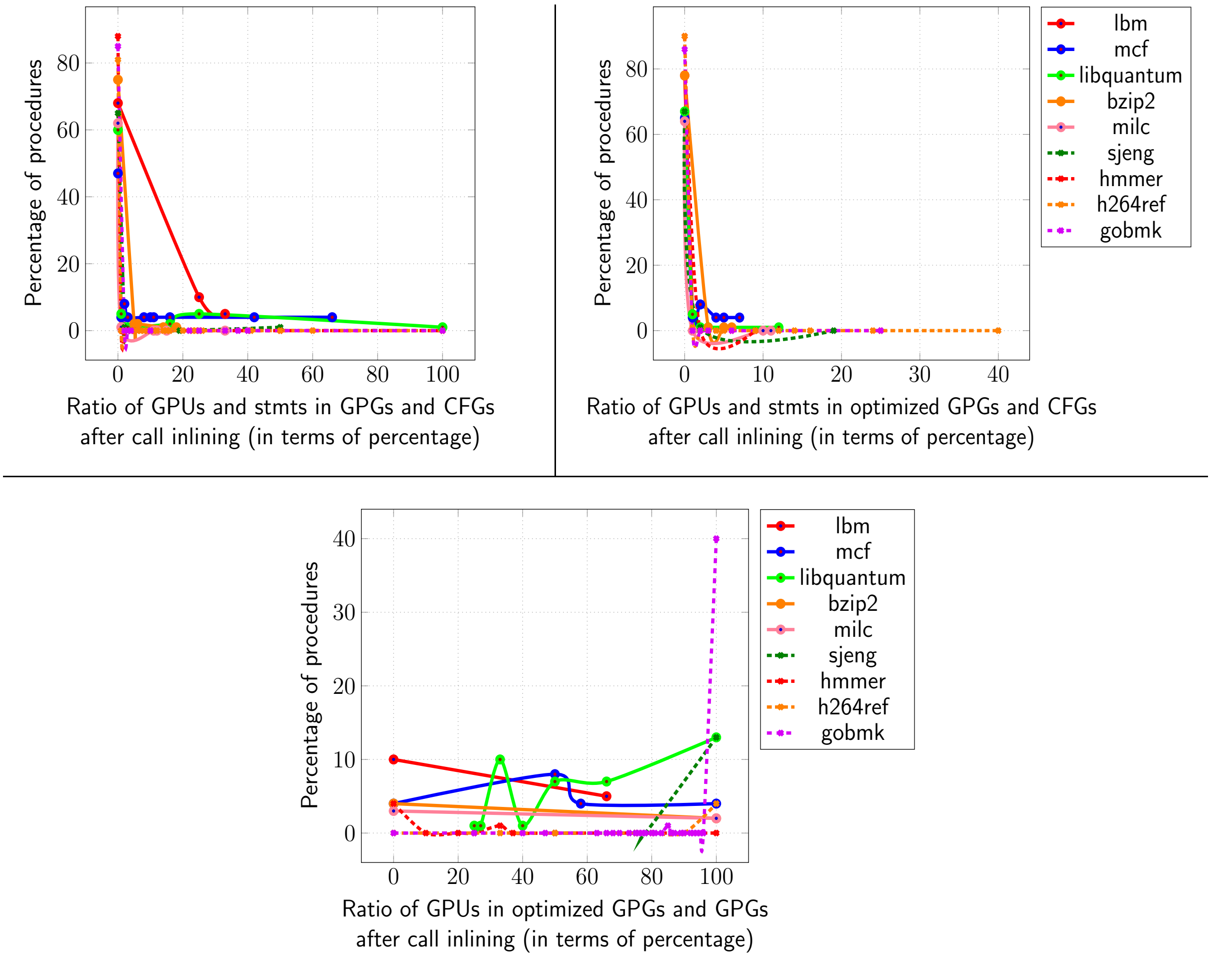


Figure 9.4: Size of GPGs relative to the size of procedures in terms of GPUs and pointer assignments.

its summary. The construction of a procedure summary is meaningful only if it is use multiple times. From column E in Table 9.1, it is clear that most procedures are called from many call sites. This indicates a high reusability of procedure summaries.

(b) Compactness of a procedure summary. For scalability of a bottom-up approach, a procedure summary should be as compact as possible. In the worst case, a procedure summary may be same as the procedure. In such a case, the application of a procedure summary at the call sites in its callers is meaningless because it is as good as visiting the procedure multiple times which is similar to a top-down approach.

Figure 9.2 and Table 9.2 show that the procedure summaries are indeed small in terms of number of GPBs and GPUs. GPGs for a large number of procedures have 0 GPUs

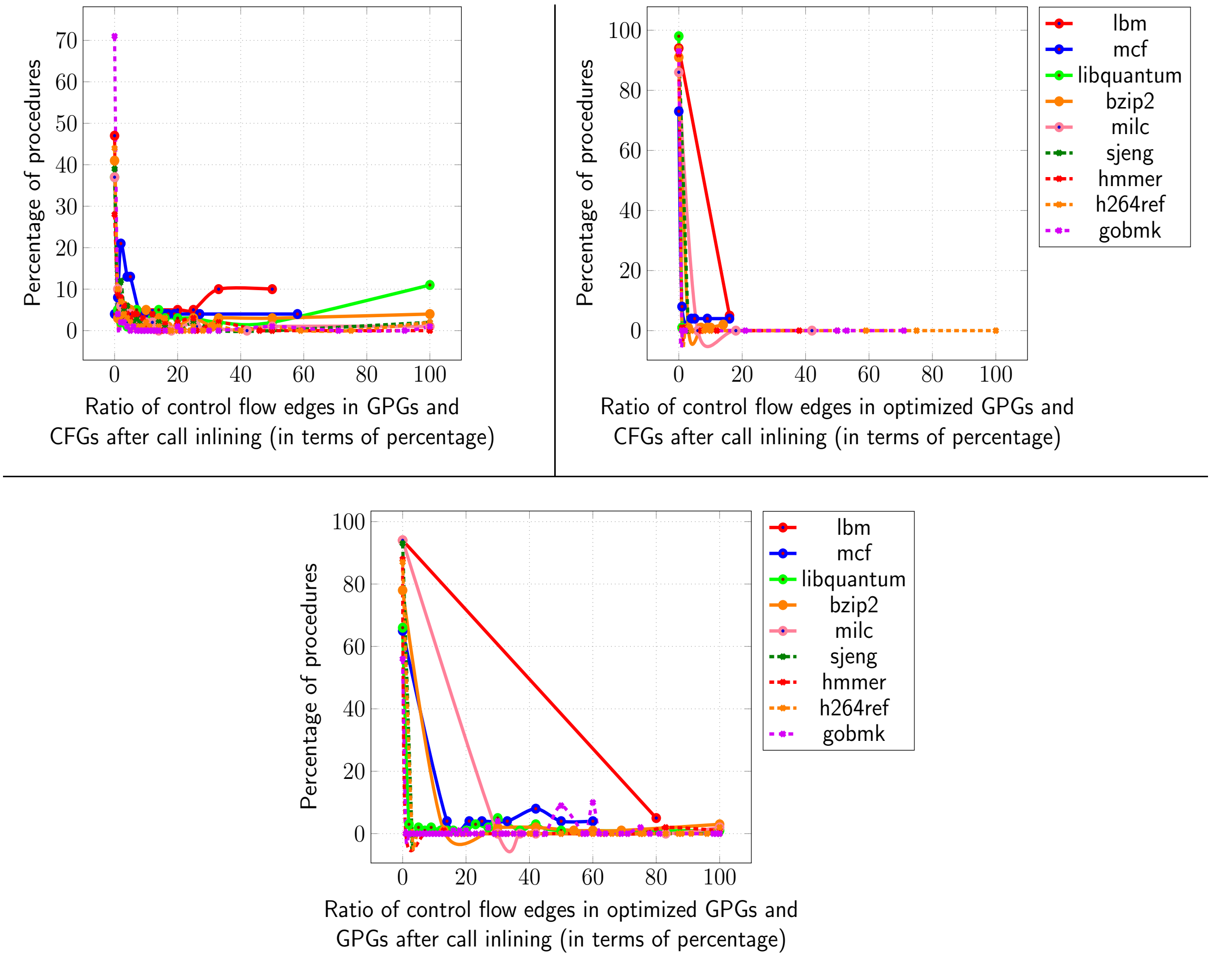


Figure 9.5: Size of GPGs relative to the size of corresponding procedures in terms of control flow edges.

because they do not manipulate global pointers (and thereby represent the identity flow function). Further, the majority of GPGs have 1 to 3 GPBs.

Note that this is an absolute size of GPGs. Observations about the relative size of GPGs with respect to their CFGs are presented in Section 9.3.3 below.

(c) Percentage of context-independent information (GPUs with *indlev* “1|0”). A procedure summary is very useful if it contains high percentage of context-independent information. We observe that the number of procedures with a high amount of context-independent information is larger in the larger benchmarks. Thus, a bottom-up approach is particularly useful for large programs.

Program	# of Proc.	# of Stmts.	FSCS			FICI	FICS
			FS	FI	FS+FI		
			Avg (per stmt)	Avg (per proc)	Avg (per proc)	Avg (per proc)	Avg (per proc)
lbm	19	367	1.99	0.79	0.63	19.26	17.11
mcf	23	484	4.12	9.30	2.30	82.13	77.39
libquantum	80	342	0.58	0.57	0.95	3.46	2.01
bzip2	89	1645	2.18	0.65	0.48	14.72	12.96
milc	196	2504	1.18	3.10	0.09	13.21	8.71
sjeng	133	684	1.44	1.83	0.32	10.04	8.17
hmmer	275	6719	1.28	1.14	0.44	25.12	19.01
h264ref	566	17253	2.35	12.02	0.82	35.04	30.75
gobmk	2699	10557	0.74	6.36	0.08	2.95	1.59

Table 9.3: Final points-to information. FSCS (flow- and context-sensitive), FICI (flow- and context-insensitive), FICS (flow-insensitive and context-sensitive).

9.3.3 Relative Size of GPGs with respect to the Size of Corresponding Procedures

For an exhaustive study, we compare three representations of a procedure with each other: (I) the CFG of a procedure, (II) the initial GPG obtained after call inlining, and (III) the final optimized GPG. Since GPGs have callee GPGs inlined within them, for a fair comparison, the CFG size must be counted by accumulating the sizes of the CFGs of the callee procedures. This is easy for non-recursive procedures. For recursive procedures, we accumulate the size of a CFG as many times as the number of inlinings of the corresponding GPG (Section 6.3). Further, the number of statements in a CFG is measured only in terms of the pointer assignments. This data is presented in Figures 9.3, 9.4, and 9.5.

(a) The first plot in these figures gives the size of the initial GPG (i.e. II) relative to that of the corresponding CFG (i.e. I). It is easy to see that the reduction is immense: a large number of procedures are in the range 0%-20% indicating more reduction in terms of GPBs, GPUs, and control flow edges in GPGs.

(b) The second plot in these figures gives the size of the optimized GPG (i.e. III) relative to that of the corresponding CFG (i.e. I). The number of procedures in the range of 0%-20% is larger here than in the first plot indicating more reduction because of

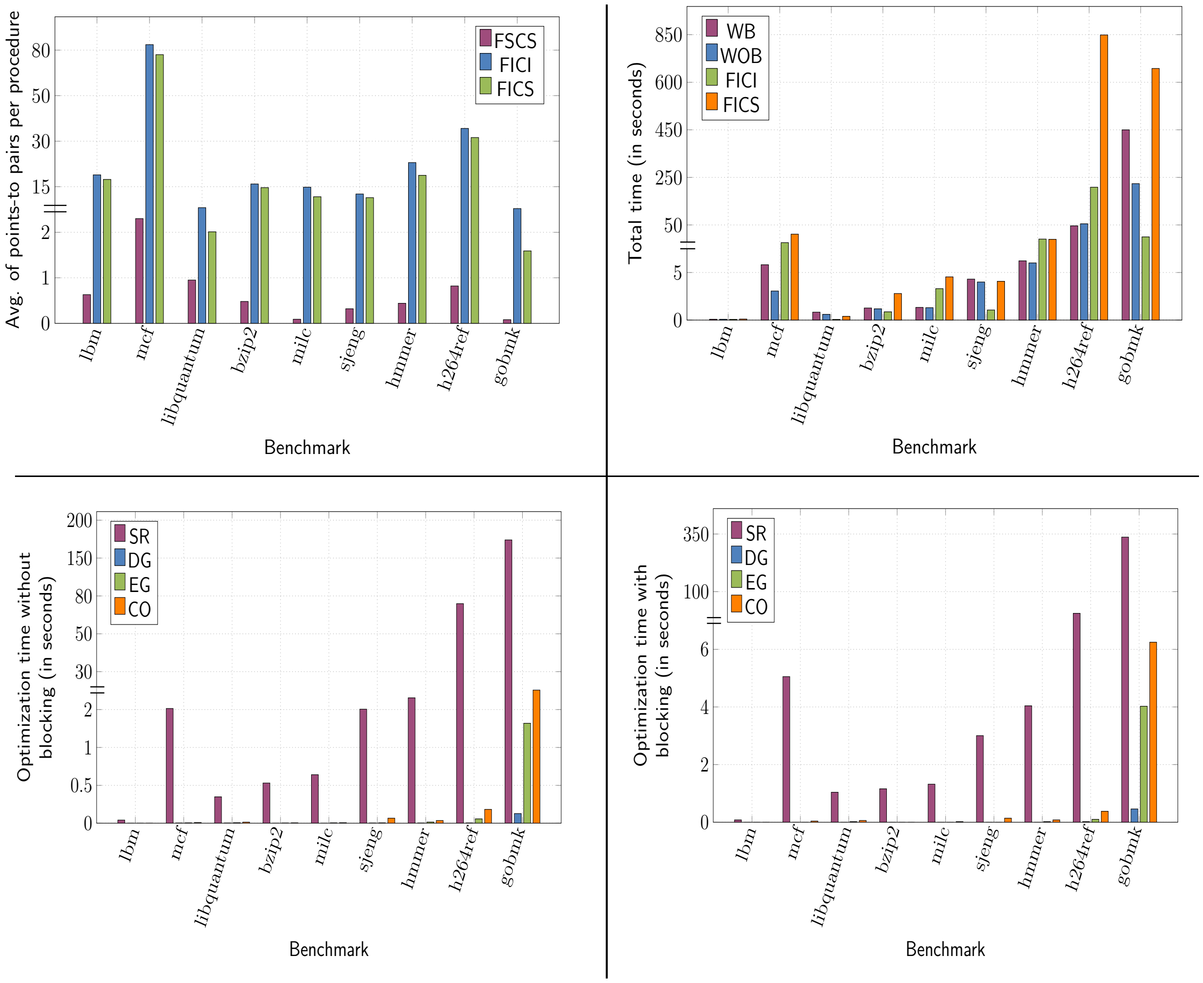


Figure 9.6: Final points-to information measurements (first plot) and time measurements (the remaining three plots). FSCS (flow- and context-sensitive), FICI (flow- and context-insensitive), FICS (flow-insensitive and context-sensitive), WOB (our analysis without blocking), WB (our analysis with blocking), SR (strength reduction optimization), DG (dead GPU elimination), EG (empty GPB elimination), CO (coalescing). The time taken by dead GPU elimination, empty GPB elimination, and coalescing is negligible for small benchmarks and hence the corresponding bars are not visible.

GPG optimizations.

(c) The third plot in these figures gives the size of the optimized GPG (i.e. III) relative to that of the initial GPG (i.e. I). Here the distribution of procedures is different for GPBs, GPUs, and control flow edges. In the case of GPBs, the reduction factor is 50%. For GPUs, the reduction varies widely. The largest reduction is found for control flow: a large number of procedures fall in the range 0%-20%. The number of procedures in this range is larger than in the case of GPBs or GPUs indicating that the control flow is optimized the most.

(d) As a special case of control flow reduction, we have measured the effect of our optimizations on back edges. This is because the presence of back edges increases the number of iterations required for fixed-point computation in an analysis. If a procedure summary needs to encode control flow, it is desirable to eliminate back edges to the extent possible. The data in Table 9.2 shows that most of the GPGs are acyclic in spite of the fact that the number of procedures with back edges in CFG is large.

9.3.4 Final Points-to Information

We compared the amount of points-to information computed by our approach with flow- and context-insensitive (FICI) and flow-insensitive and context-sensitive (FICS) methods (first plot of Figure 9.6 and Table 9.3). For this purpose, we computed number of points-to pairs per procedure in all the three approaches by dividing the total number of unique points-to pairs across all procedures by the total number of procedures. Predictably, this number is smallest for our analysis (FSCS) and largest for FICI method.

9.3.5 Time Measurements

We have measured the overall time as well as the time taken by each of the optimizations (Figure 9.6). We have also measured the time taken by the FICI and FICS variants of points-to analysis. Our observations are:

(a) Our analysis takes less than 8 minutes on gobmk.445 which is a large benchmark with 158 kLoC. Our current implementation does not scale beyond that. We imposed a time limit of 2 hours and the larger benchmarks timed out.

(b) Strength reduction is the most expensive optimization followed by coalescing which is the most expensive among the redundancy elimination optimizations.

(c) We introduced reaching GPUs analysis with blocking to ensure soundness of strength reduction so that a barrier GPU does not cause a side-effect invalidating strength reduction. However, our intuition was that very few of us write programs where a pointer is manipulated in such a manner. Hence we identified possible soundness alerts. The soundness alerts arise when a GPU whose composition was postponed, is updated by a GPU within the same GPG after inlining in a caller GPG. This is identified by checking if a GPU in the set Queued of a GPG is killed by the GPU of the same GPG when it is inlined in a caller.

We also measured the number of GPUs that were queued (i.e. not used as producer GPUs). Our measurements show that the number of GPUs in the Queued set is relatively small (see Table 9.2). We did not find a single instance of a soundness alert that was valid; we did find a very small number of false positives that were manually examined and rejected.

(d) FICI variant is consistently faster than the FICS variant, and faster than FSCS in most programs. Further, FSCS is faster than FICS in most cases. A flow-sensitive version being faster than a flow-insensitive version might look counter intuitive. However, it is not the case because of context sensitivity—the number of calling contexts in FICS is much larger than that in FSCS owing to flow insensitivity.

9.4 Discussion: Lessons From Our Empirical Measurements

Our experiments and empirical data leads us to the following important learnings:

1. The real killer of scalability in program analysis is not the amount of data but the amount of control flow that the data propagation may be subjected to, in search of precision.

2. For scalability, the bottom-up summaries must be kept as small as possible at each stage of summary construction.

3. Some amount of top-down flow is very useful for achieving scalability.

4. Type-based non-aliasing aids scalability significantly.

5. The indirect effects for which we devised blocking to postpone GPU compositions are extremely rare in practical programs. We did not find a single instance in our benchmarks.

6. Not all information is flow-sensitive.

We learnt these lessons the hard way as described in the rest of this section.

9.4.1 Handling Recursion

In our first attempt of handling recursion, we converted indirect recursion to self recursion, and repeatedly inlined the recursive calls to optimize them. This failed because in some cases, the size of GPG after inlining calls became too big and our analyses and optimizations did not scale. Hence, instead of first creating a naively large GPG and then optimizing it to bring down the size, we decided to keep the GPGs small at every step in a fixed-point computation, starting from $\varDelta_\top$.

9.4.2 Handling Large Size of Context-Dependent Information

Some GPGs had a large amount of context-dependent information (i.e. GPUs with upwards-exposed versions of variables) and the GPGs could not be optimized much. This caused the size of the caller GPGs to grow significantly, threatening the scalability of our analysis. Hence, we devised a heuristic threshold t representing the number of GPUs containing upwards-exposed versions of variables. This threshold is used as follows: Let a GPG contain x GPUs containing upwards-exposed versions.

- If $x < t$ for a GPG, then the GPG is inlined in its callers.
- if $x \geq t$ for a GPG, then the GPG is not inlined in its callers. Instead its calls are represented symbolically with the GPUs containing upwards-exposed versions. As the analysis proceeds, these GPUs are reduced decreasing the count of x after which the GPG is inlined.

This keeps the size of the caller GPG small and at the same time, allows reduction of the context-dependent GPUs in the calling context. Once all GPUs are reduced to classical points-to edge, we effectively get the procedure summary of the original callee procedure for that call chain. Since the reduction of context-dependent GPUs is different for different calling contexts, the process needs to be repeated for each call chain. This is similar to the top-down approach where we analyze a procedure multiple times. We used a threshold of 80% context-dependent GPUs in a GPG containing more than 10 GPUs. Thus, 8 context-dependent GPUs from a total of 11 GPUs was below our threshold as was 9 context-dependent GPUs from a total of 9 GPUs.

Note that in our implementation, we discovered very few cases (in the order of single digits and only in large benchmarks) where the threshold actually exceeded. The number of call chains that required multiple traversals are in single digits and they are not very long. The important point to note is that we got the desired scalability only when we introduced this small twist of using symbolic GPG.

9.4.3 Handling Function Pointers

Function pointers used in a procedure but defined in its callers is another case where we had to inline unoptimized GPGs in the callers because the GPGs of the procedure's callees were not known and hence their flow function was $\Delta_{\top}$. This hampered scalability. Since our primary goal was to evaluate the effectiveness of our optimizations, our current implementation handles only locally defined function pointers (Section 6.4) Our implementation can be easily extended to handle function pointers defined in the calling contexts. We can handle such function pointers by using a symbolic $\Delta_{\top}$ GPG and introducing a small touch of top-down analysis as was done above when handling a large number of context-dependent GPUs. We leave this as future work.

9.4.4 Handling Arrays and SSA Form

Pointers to arrays were weakly updated, hence we realized early on that maintaining this information flow sensitively prohibited scalability. This was particularly true for large arrays with static initializations. Similarly, GPUs involving SSA versions of variables were not required to be maintained flow sensitively. This allowed us to reduce the propagation of data across control flow without any loss in precision.

9.4.5 Making Coalescing More Effective

Unlike dead GPU elimination, coalescing proved to be a very significant optimization for boosting the scalability of the analysis. The points-to analysis failed to scale in the absence of this optimization. However, this optimization was effective (i.e. coalesced many GPBs) only when we brought in the concept of types: in cases where the data dependence between the GPUs was unknown because of the dependency on the context information, we used type-based non-aliasing to enable coalescing.

9.4.6 Estimating the Number of Context-Dependent Summaries

Constructing context-dependent procedure summaries (i.e., partial transfer functions in an MTF approach, see Sections 1.3.3 and 2.4.1) using the aliases or points-to information from calling contexts obviates the need of control flow. Since control flow is the real bottleneck as per our findings, we computed the number of aliases after computing the final points-to information to estimate the number of context-dependent summaries that may be required for real program. This number (column F in Table 9.1) is large suggesting that it is undesirable to construct multiple PTFs for a procedure using the aliases from the calling contexts.

9.5 Chapter Summary

A procedure summary is useful if it is (a) reusable, and (b) compact. Our measurements show that the procedure summaries are highly reusable and indeed small in terms of number of GPBs and GPUs.

Coalescing proved to be a very significant optimization for boosting the scalability of the analysis. Coalescing did not cause uniform reduction across all GPGs but helped in the most critical GPGs (GPGs may correspond to recursive procedures). The points-to analysis failed to scale in the absence of this optimization. However, this optimization was effective (i.e. coalesced many GPBs) only when we brought in the concept of types.

The need for scalability and precision of points-to analysis demands an hybrid approach which explores the pros and cons of each of the approaches. A good study of the nature of procedures and their interaction with other procedures is needed to determine whether a procedure should be visited in a top-down approach fashion or its procedure summary should be constructed which could be used at the call sites in all its callers.

Chapter 10

Conclusions and Future Work

We conclude the thesis by reflecting on our ideas and envisioning future possibilities.

10.1 Reflections

Constructing compact procedure summaries for flow- and context-sensitive points-to analysis seems hard because it

(a) needs to model the indirect accesses of pointees that are defined in callers without examining their code,

(b) needs to preserve data dependence between memory updates, and

(c) needs to incorporate the effect of summaries of the callee procedures transitively.

In past, the first issue has been handled by modelling accesses of unknown pointees using placeholders. However, it may require a large number of placeholders. The second issue has been handled by constructing multiple versions of a procedure summary for different aliases in the calling contexts. The third issue can only be handled by inlining the summaries of the callees. However, it can increase the size of a summary exponentially thereby hampering the scalability of analysis.

We have handled the first issue by proposing the concept of generalized points-to updates (GPUs) which track indirection levels. Simple arithmetic on indirection levels allows composition of GPUs to create new GPUs with smaller indirection levels; this reduces them progressively to classical points-to edges.

In order to handle the second issue, we maintain control flow within a GPG and perform optimizations of strength reduction and redundancy elimination. Together, these optimizations reduce the indirection levels of GPUs, eliminate data dependences between GPUs, and minimize control flow significantly. These optimizations also mitigate the impact of the third issue.

In order to achieve the above, we have devised novel data flow analyses such as reaching GPUs analysis (with and without blocking) and coalescing analysis which is a bidirectional analysis. Interleaved call inlining and strength reduction of GPGs facilitated a novel optimization that computes flow- and context-sensitive points-to information in the first phase of a bottom-up approach. This obviates the need for the second phase.

An interesting aspect of our design is that our method is a hybrid approach that can be tuned to combine top-down and bottom-up approaches in a flexible manner for an individual GPG as desired: After constructing an initial GPG, we use a threshold of context-dependent information (i.e. GPUs with upwards-exposed versions of variables) in the GPG to decide whether to optimize it and then process the caller GPGs in a bottom-up manner or to process the caller GPGs (without inlining the callee GPGs) to propagate GPUs top-down. If the threshold is kept very small, we have a largely top-down approach, if it is kept very large, we have a largely bottom-up approach.

Our measurements on SPEC benchmarks show that GPGs are small enough to scale fully flow- and context-sensitive exhaustive points-to analysis to C programs as large as 158 kLoC. Further, most the GPGs are acyclic even if they represent procedures that have loops or are recursive.

Some important takeaways from our empirical evaluation are:

(a) Flow- and context-sensitive points-to information is small and sparse.

(b) 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 propagation of data may be subjected to in search of precision. This observation supports the concept of sparse analysis. However, the construction of a full interprocedural SSA even for scalar global variables is hard because of the side-effects that a procedure call may cause; for pointers it seems harder. Our analysis achieves scalability by minimizing the control flow significantly.

(c) Although a bottom-up approach is much more efficient than a top-down approach, in practice, a hybrid approach is more scalable than the individual approaches. Our measurements found very few cases (in the order of single digits and only in large benchmarks) where our threshold was actually exceeded.

10.2 Possible Directions of Future Work

We feel that this work can be taken further in many ways. We have grouped the ideas of possible directions under the following categories:

- Ideas influencing the scalability of points-to analysis using GPGs.
- Ideas influencing the precision of points-to analysis using GPGs.
- Ideas related to the formal aspects of GPGs.
- Use of GPGs for other analyses.
- Use of GPGs for other paradigms.

Some of these ideas extend or consolidate the GPG-based points-to analysis whereas some others explore whether these concepts could be used in other situations.

Ideas influencing the scalability of points-to analysis using GPGs:

- It would be useful to explore the possibility of scaling the implementation to larger programs; we suspect that this would be centered around examining the control flow in the GPGs and optimizing it still further.
- In liveness based points-to analysis [53], points-to information is computed only for live pointer variables. It would be interesting to explore the possibility of restricting the GPG construction to live pointer variables for scalability. For points-to analysis, the GPG construction is a forward flow problem whereas liveness analysis is a backward flow problem. Thus for liveness analysis of pointers, we may need to construct "backward" GPGs. It would be interesting to study how the interaction between forward and backward GPGs can be used for computing liveness based points-to information.

- An interesting idea of scaling points-to analysis is to perform analysis only for the "top-level" pointers (i.e., whose addresses are not taken) and use the points-to information to eliminate some pointer indirections thereby exposing the next-level pointers as top-level pointers [116]. This analysis constructs a procedure summary for each level (starting with the highest level) and uses the points-to information from the previous levels. It would be interesting to explore the possibility of using this technique for GPG construction and analyze the impact of this divide and conquer technique on scalability.

Ideas influencing the precision of points-to analysis using GPGs:

- Currently GPGs use allocation-site-based abstraction for dynamically allocated memory locations. Currently, we use a simple abstraction that does not distinguish between the calling contexts of procedures allocating memory. GPGs can also take the advantage of the idea of heap cloning [57, 72, 111] to distinguish the memory allocated in distinct calling contexts. The reason why this seems eminently feasible is that callee GPGs are inlined in the callers and the consumer GPUs are propagated bottom-up to the calling contexts. Thus all we need for heap cloning is a consistent way of naming the allocations sites occurring in producer GPUs when they are used in compositions with a consumer GPG. We believe that this need-based cloning is an inherent advantage of a bottom-up method over a top-down method.

- Currently GPGs treat an entire array as a single variable that undergoes weak updates. One possible future work would be to extend GPGs to arrays using array SSA form [54] thereby distinguishing between different array elements and perform strong updates. We can also explore the benefits of using heap SSA form [120] (which models each field as a distinct logical "heap array") in the current implementation of GPG-based points-to analysis.

Ideas related to the formal aspects of GPGs:

- It would be interesting to work out formal proofs of correctness and analyze theoretical complexity of GPG-based points-to analysis.

Use of GPGs for other analyses:

- The concept of GPG provides a useful abstraction of memory and memory transformers involving pointers by directly modelling load, store, and copy of memory addresses. It would be interesting to combine GPGs with the abstractions of a client analysis, say property proving application for verification. This direction can also be explored in future.

Use of GPGs for other paradigms:

- It would be useful to extend the scope of the implementation of GPG-based points-to analysis to C++ and Java programs. It would also be interesting to extend our work to concurrent programs such as Java programs containing threads.

10.3 Final Thoughts

We believe that the ideas presented in this thesis can go far beyond what has been achieved in this work and GPGs hold a promise for future research in analysis of programs containing pointers.

References

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools (2Nd Edition).* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2006.

[2] Lars Ole Andersen. *Program Analysis and Specialization for the C Programming Language.* PhD thesis, DIKU, University of Copenhagen, May 1994. (DIKU report 94/19).

[3] Thomas Ball and Sriram K. Rajamani. The slam project: Debugging system software via static analysis. In *Proceedings of the 29th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '02, pages 1–3, New York, NY, USA, 2002. ACM.

[4] Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In *Proceedings of the ACM SIGPLAN 2003 Conference on Programming Language Design and Implementation*, PLDI '03, pages 103–114, New York, NY, USA, 2003. ACM.

[5] Dirk Beyer, Thomas A. Henzinger, and Grégory Théoduloz. Configurable software verification: Concretizing the convergence of model checking and program analysis. In *Proceedings of the 19th International Conference on Computer Aided Verification*, CAV'07, pages 504–518, Berlin, Heidelberg, 2007. Springer-Verlag.

[6] Randal E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Trans. Comput.*, 35(8):677–691, August 1986.

[7] J. R. Burch, E. M. Clarke, D. E. Long, K. L. McMillan, and D. L. Dill. Symbolic model checking for sequential circuit verification. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 13(4):401–424, Apr 1994.

[8] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation*, PLDI '90, pages 296–310, New York, NY, USA, 1990. ACM.

[9] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. Relevant context inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '99, pages 133–146, New York, NY, USA, 1999. ACM.

[10] Ben-Chung Cheng and Wen-Mei W. Hwu. Modular interprocedural pointer analysis using access paths: Design, implementation, and evaluation. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 57–69, New York, NY, USA, 2000. ACM.

[11] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '93, pages 232–245, New York, NY, USA, 1993. ACM.

[12] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '91, pages 55–66, New York, NY, USA, 1991. ACM.

[13] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *In Tools and Algorithms for the Construction and Analysis of Systems*, pages 168–176. Springer, 2004.

[14] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Trans. Program. Lang. Syst.*, 13(4):451–490, October 1991.

[15] Camil Demetrescu, David Eppstein, Zvi Galil, and Giuseppe F. Italiano. Dynamic graph algorithms. In Mikhail J. Atallah and Marina Blanton, editors, *Algorithms and Theory of Computation Handbook*, pages 9.1–9.14. Chapman & Hall/CRC, 2010.

[16] Camil Demetrescu and Giuseppe F. Italiano. Mantaining dynamic matrices for fully dynamic transitive closure. *Algorithmica*, 51(4):387–427, May 2008.

[17] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, New York, NY, USA, 2008. ACM.

[18] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Reducing the cost of data flow analysis by congruence partitioning. In *In International Conference on Compiler Construction*, pages 357–373. Springer-Verlag, 1994.

[19] Marcus Edvinsson, Jonas Lundberg, and Welf Löwe. Parallel points-to analysis for multi-core machines. In *Proceedings of the 6th International Conference on High Performance and Embedded Architectures and Compilers*, HiPEAC '11, pages 45–54, New York, NY, USA, 2011. ACM.

[20] Maryam Emami. *A Practical Interprocedural Alias Analysis for An Optimizing/Parallelizing C Compiler.* PhD thesis, Montreal, Canada, 1993.

[21] Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 85–96, New York, NY, USA, 1998. ACM.

[22] Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Proceedings of the ACM SIGPLAN 2000 Conference on Programming Language Design and Implementation*, PLDI '00, pages 253–263, New York, NY, USA, 2000. ACM.

[23] Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. Bottom-up context-sensitive pointer analysis for Java. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*, 2015.

[24] Jeffrey Fischer, Ranjit Jhala, and Rupak Majumdar. Joining dataflow with predicates. In *Proceedings of the 10th European Software Engineering Conference Held*

Jointly with 13th ACM SIGSOFT International Symposium on Foundations of Software Engineering, ESEC/FSE-13, pages 227–236, New York, NY, USA, 2005. ACM.

[25] Pritam M. Gharat, Uday P. Khedker, and Alan Mycroft. Flow- and context-sensitive points-to analysis using generalized points-to graphs. In *Proceedings of the 23rd Static Analysis Symposium*, SAS'16, Berlin, Heidelberg, 2016. Springer-Verlag.

[26] Pritam M. Gharat, Uday P. Khedker, and Alan Mycroft. Generalized points-to graphs: A new abstraction of memory in presence of pointers. Technical report, IIT Bombay, 2018. (Under submission).

[27] T. Gutzmann, J. Lundberg, and W. Lowe. Towards path-sensitive points-to analysis. In *Seventh IEEE International Working Conference on Source Code Analysis and Manipulation (SCAM 2007)*, pages 59–68, Sept 2007.

[28] Samuel Z. Guyer and Calvin Lin. Client-driven pointer analysis. In *Proceedings of the 10th International Conference on Static Analysis*, SAS'03, pages 214–236, Berlin, Heidelberg, 2003. Springer-Verlag.

[29] Samuel Z. Guyer and Calvin Lin. Error checking with client-driven pointer analysis. *Sci. Comput. Program.*, 58(1-2):83–114, October 2005.

[30] Brian Hackett and Alex Aiken. How is aliasing used in systems software? In *Proceedings of the 14th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, SIGSOFT '06/FSE-14, New York, NY, USA, 2006. ACM.

[31] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: Fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '07, pages 290–299, New York, NY, USA, 2007. ACM.

[32] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2009, Savannah, GA, USA, January 21-23, 2009*, pages 226–238, 2009.

[33] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the CGO 2011, The 9th International Symposium on Code Generation and Optimization, Chamonix, France, April 2-6, 2011*, pages 289–298, 2011.

[34] Rebecca Hasti and Susan Horwitz. Using static single assignment form to improve flow-insensitive pointer analysis. In *Proceedings of the ACM SIGPLAN 1998 Conference on Programming Language Design and Implementation*, PLDI '98, pages 97–105, New York, NY, USA, 1998. ACM.

[35] M. S. Hecht. *Flow Analysis of Computer Programs.* Elsevier North-Holland Inc., 1977.

[36] M. S. Hecht and J. D. Ullman. Flow graph reducibility. In *Proceedings of the ACM Symposium on Theory of Computing*, pages 238–250, 1972.

[37] M. S. Hecht and J. D. Ullman. Characterization of reducible flow graphs. *Journal of ACM*, 21(3):367–375, 1974.

[38] Nevin Heintze and Olivier Tardieu. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 24–34, New York, NY, USA, 2001. ACM.

[39] Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using cla: A million lines of C code in a second. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 254–263, New York, NY, USA, 2001. ACM.

[40] Michael Hind. Pointer analysis: Haven't we solved this problem yet? In *PASTE'01*, pages 54–61. ACM Press, 2001.

[41] Michael Hind, Michael Burke, Paul Carini, and Jong-Deok Choi. Interprocedural pointer alias analysis. *ACM Trans. Program. Lang. Syst.*, 21(4):848–894, July 1999.

[42] Michael Hind and Anthony Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings*, pages 57–81, 1998.

[43] Michael Hind and Anthony Pioli. Which pointer analysis should I use? In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '00, pages 113–123, New York, NY, USA, 2000. ACM.

[44] Susan Horwitz. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Trans. Program. Lang. Syst.*, 19(1):1–6, January 1997.

[45] Franjo Ivančić, Ilya Shlyakhter, Aarti Gupta, Malay K Ganai, Vineet Kahlon, Chao Wang, and Zijiang Yang. Model checking C programs using F-Soft. In *IEEE International Conference on Computer Design*, pages 297–308. IEEE, 2005.

[46] Ranjit Jhala and Rupak Majumdar. Software model checking. *ACM Comput. Surv.*, 41(4):21:1–21:54, October 2009.

[47] Vineet Kahlon. Bootstrapping: A technique for scalable flow and context-sensitive pointer alias analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '08, pages 249–259, New York, NY, USA, 2008. ACM.

[48] Vini Kanvar and Uday P. Khedker. Heap abstractions for static analysis. *ACM Comput. Surv.*, 49(2):29:1–29:47, June 2016.

[49] Owen Kaser, C. R. Ramakrishnan, and Shaunak Pawagi. On the conversion of indirect to direct recursion. *ACM Lett. Program. Lang. Syst.*, 2(1-4):151–164, March 1993.

[50] George Kastrinis and Yannis Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '13, pages 423–434, New York, NY, USA, 2013. ACM.

[51] U. P. Khedker, A. Sanyal, and B. Sathe. *Data Flow Analysis: Theory and Practice.* Taylor & Francis (CRC Press, Inc.), Boca Raton, FL, USA, 2009.

[52] Uday P. Khedker and Bageshri Karkare. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: resurrecting the classical call strings method. In *Proceedings of the Joint European Conferences on Theory*

and Practice of Software 17th international conference on Compiler construction, CC'08/ETAPS'08, 2008.

[53] Uday P. Khedker, Alan Mycroft, and Prashant Singh Rawat. Liveness-based pointer analysis. In *Proceedings of the 19th International Static Analysis Symposium*, SAS'12, Berlin, Heidelberg, 2012. Springer-Verlag.

[54] Kathleen Knobe and Vivek Sarkar. Array SSA form and its use in parallelization. In *Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '98, pages 107–120, New York, NY, USA, 1998. ACM.

[55] William Landi. Undecidability of static analysis. *ACM Lett. Program. Lang. Syst.*, 1(4):323–337, December 1992.

[56] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of the ACM SIGPLAN 1992 Conference on Programming Language Design and Implementation*, PLDI '92, pages 235–248, New York, NY, USA, 1992. ACM.

[57] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, June 2007.

[58] Ondrej Lhoták and Kwok-Chiang Andrew Chung. Points-to analysis with efficient strong updates. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 3–16, New York, NY, USA, 2011. ACM.

[59] Ondřej Lhoták and Laurie Hendren. Scaling Java points-to analysis using spark. In Görel Hedin, editor, *Compiler Construction*, pages 153–169, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[60] Ondrej Lhoták, Yannis Smaragdakis, and Manu Sridharan. Pointer Analysis (Dagstuhl Seminar 13162). *Dagstuhl Reports*, 3(4):91–113, 2013.

[61] Ondřej Lhoták and Laurie Hendren. Evaluating the benefits of context-sensitive points-to analysis using a BDD-based implementation. *ACM Trans. Softw. Eng. Methodol.*, 18(1):3:1–3:53, October 2008.

[62] Lian Li, Cristina Cifuentes, and Nathan Keynes. Precise and scalable context-sensitive pointer analysis via value flow graph. In *Proceedings of the 2013 International Symposium on Memory Management*, ISMM '13, New York, NY, USA, 2013. ACM.

[63] Donglin Liang and Mary Jean Harrold. Efficient points-to analysis for whole-program analysis. In *Proceedings of the 7th European Software Engineering Conference Held Jointly with the 7th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, ESEC/FSE-7, pages 199–215, London, UK, UK, 1999. Springer-Verlag.

[64] Jonas Lundberg and Welf Löwe. A scalable flow-sensitive points-to analysis, 2007.

[65] Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani. Modular heap analysis for higher-order programs. In *Proceedings of the 19th International Conference on Static Analysis*, SAS'12, Berlin, Heidelberg, 2012. Springer-Verlag.

[66] R. Manevich, G. Ramalingam, J. Field, D. Goyal, and M. Sagiv. Compactly representing first-order structures for static analysis. In Manuel V. Hermenegildo and Germán Puebla, editors, *Static Analysis*, pages 196–212, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[67] Mario Méndez-Lojo, Augustine Mathew, and Keshav Pingali. Parallel inclusion-based points-to analysis. In *Proceedings of the ACM International Conference on Object Oriented Programming Systems Languages and Applications*, OOPSLA '10, pages 428–443, New York, NY, USA, 2010. ACM.

[68] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Parameterized object sensitivity for points-to analysis for java. *ACM Trans. Softw. Eng. Methodol.*, 14(1):1–41, January 2005.

[69] Rupesh Nasre. Approximating inclusion-based points-to analysis. In *Proceedings of the 2011 ACM SIGPLAN Workshop on Memory Systems Performance and Correctness*, MSPC '11, pages 66–73, New York, NY, USA, 2011. ACM.

[70] Rupesh Nasre, Kaushik Rajan, Ramaswamy Govindarajan, and Uday P. Khedker. Scalable context-sensitive points-to analysis using multi-dimensional bloom filters. In *Programming Languages and Systems, 7th Asian Symposium, APLAS 2009, Seoul, Korea, December 14-16, 2009. Proceedings*, pages 47–62, 2009.

[71] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. Bottom-up and top-down context-sensitive summary-based pointer analysis. In *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*, 2004.

[72] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. Importance of heap specialization in pointer analysis. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering*, PASTE '04, pages 43–48, New York, NY, USA, 2004. ACM.

[73] Hakjoo Oh, Kihong Heo, Wonchan Lee, Woosuk Lee, and Kwangkeun Yi. Design and implementation of sparse global analyses for c-like languages. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '12, Beijing, China - June 11 - 16, 2012*, 2012.

[74] Hakjoo Oh, Wonchan Lee, Kihong Heo, Hongseok Yang, and Kwangkeun Yi. Selective context-sensitivity guided by impact pre-analysis. In *ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI '14, Edinburgh, United Kingdom - June 09 - 11, 2014*, 2014.

[75] Rohan Padhye and Uday P. Khedker. Interprocedural data flow analysis in SOOT using value contexts. In *Proceedings of the 2Nd ACM SIGPLAN International Workshop on State Of the Art in Java Program Analysis*, SOAP '13, New York, NY, USA, 2013. ACM.

[76] David J. Pearce, Paul H. J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis for c. In *Proceedings of the 5th ACM SIGPLAN-SIGSOFT Workshop on*

Program Analysis for Software Tools and Engineering, PASTE '04, pages 37–42, New York, NY, USA, 2004. ACM.

[77] David J. Pearce, Paul H.J. Kelly, and Chris Hankin. Efficient field-sensitive pointer analysis of c. *ACM Trans. Program. Lang. Syst.*, 30(1), November 2007.

[78] Sandeep Putta and Rupesh Nasre. Parallel replication-based points-to analysis. In *Proceedings of the 21st International Conference on Compiler Construction*, CC'12, pages 61–80, Berlin, Heidelberg, 2012. Springer-Verlag.

[79] G. Ramalingam. The undecidability of aliasing. *ACM Trans. Program. Lang. Syst.*, 16(5):1467–1471, September 1994.

[80] G. Ramalingam. On sparse evaluation representations. *Theor. Comput. Sci.*, 277(1-2):119–147, April 2002.

[81] Thomas Reps. Undecidability of context-sensitive data-dependence analysis. *ACM Trans. Program. Lang. Syst.*, 22(1):162–186, January 2000.

[82] Thomas Reps, Susan Horwitz, and Mooly Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22Nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '95, New York, NY, USA, 1995. ACM.

[83] Atanas Rountev, Ana Milanova, and Barbara G. Ryder. Points-to analysis for Java using annotated constraints. In *Proceedings of the 16th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '01, pages 43–55, New York, NY, USA, 2001. ACM.

[84] Erik Ruf. Partitioning dataflow analyses using types. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 15–26, New York, NY, USA, 1997. ACM.

[85] Radu Rugina and Martin Rinard. Pointer analysis for multithreaded programs. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 77–90, New York, NY, USA, 1999. ACM.

[86] Radu Rugina and Martin C. Rinard. Pointer analysis for structured parallel programs. *ACM Trans. Program. Lang. Syst.*, 25(1):70–116, January 2003.

[87] Barbara G. Ryder. Dimensions of precision in reference analysis of object-oriented programming languages. In *Proceedings of the 12th International Conference on Compiler Construction*, CC'03, pages 126–137, Berlin, Heidelberg, 2003. Springer-Verlag.

[88] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. In *Selected Papers from the 6th International Joint Conference on Theory and Practice of Software Development*, TAPSOFT '95, Amsterdam, The Netherlands, The Netherlands, 1996. Elsevier Science Publishers B. V.

[89] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Trans. Program. Lang. Syst.*, 20(1):1–50, January 1998.

[90] Diptikalyan Saha and C. R. Ramakrishnan. Incremental and demand-driven points-to analysis using logic programming. In *Proceedings of the 7th ACM SIGPLAN International Conference on Principles and Practice of Declarative Programming*, PPDP '05, pages 117–128, New York, NY, USA, 2005. ACM.

[91] Alexandru Salcianu and Martin Rinard. Pointer and escape analysis for multithreaded programs. In *Proceedings of the Eighth ACM SIGPLAN Symposium on Principles and Practices of Parallel Programming*, PPoPP '01, pages 12–23, New York, NY, USA, 2001. ACM.

[92] Lei Shang, Xinwei Xie, and Jingling Xue. On-demand dynamic summary-based points-to analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization*, CGO '12, New York, NY, USA, 2012. ACM.

[93] Marc Shapiro and Susan Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proceedings of the 24th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '97, pages 1–14, New York, NY, USA, 1997. ACM.

[94] A. Sharir M., Pnueli. Two approaches to interprocedural data flow analysis. *S.S., Jones, N.D. (eds.) Program Flow Analysis: Theory and Applications, (ch. 7)*, 1981.

[95] Yannis Smaragdakis and George Balatsouras. Pointer analysis. *Foundations and Trends® in Programming Languages*, 2(1):1–69, 2015.

[96] Yannis Smaragdakis, Martin Bravenboer, and Ondrej Lhoták. Pick your contexts well: Understanding object-sensitivity. In *Proceedings of the 38th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '11, pages 17–30, New York, NY, USA, 2011. ACM.

[97] Johannes Späth, Lisa Nguyen, Karim Ali, and Eric Bodden. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for Java. In *European Conference on Object-Oriented Programming (ECOOP)*, 17 - 22 July 2016.

[98] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for Java. In *Proceedings of the 27th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '06, pages 387–400, New York, NY, USA, 2006. ACM.

[99] Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. Aliasing in object-oriented programming. chapter Alias Analysis for Object-oriented Programs, pages 196–232. Springer-Verlag, Berlin, Heidelberg, 2013.

[100] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. Demand-driven points-to analysis for Java. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '05, pages 59–76, New York, NY, USA, 2005. ACM.

[101] Stefan Staiger-Stöhr. Practical integrated analysis of pointers, dataflow and control flow. *ACM Trans. Program. Lang. Syst.*, 35(1):5:1–5:48, 2013.

[102] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Proceedings of the 23rd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '96, pages 32–41, New York, NY, USA, 1996. ACM.

[103] Alexandru Sălcianu and Martin Rinard. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation*, VMCAI'05, Berlin, Heidelberg, 2005. Springer-Verlag.

[104] Yulei Sui, Sen Ye, Jingling Xue, and Pen-Chung Yew. Spas: Scalable path-sensitive pointer analysis on full-sparse ssa. In *Proceedings of the 9th Asian Conference on Programming Languages and Systems*, APLAS'11, pages 155–171, Berlin, Heidelberg, 2011. Springer-Verlag.

[105] Teck Bok Tok. *Removing Unimportant Computations in Interprocedural Program Analysis*. PhD thesis, Austin, TX, USA, 2007. AAI3290942.

[106] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *Proceedings of the 15th International Conference on Compiler Construction*, CC'06, pages 17–31, Berlin, Heidelberg, 2006. Springer-Verlag.

[107] John Whaley and Monica S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In *Proceedings of the 9th International Symposium on Static Analysis*, SAS '02, pages 180–195, London, UK, UK, 2002. Springer-Verlag.

[108] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 131–144, New York, NY, USA, 2004. ACM.

[109] John Whaley and Martin Rinard. Compositional pointer and escape analysis for Java programs. In *Proceedings of the 14th ACM SIGPLAN Conference on Object-oriented Programming, Systems, Languages, and Applications*, OOPSLA '99, New York, NY, USA, 1999. ACM.

[110] R. P. Wilson and M. S. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '95, 1995.

[111] Guoqing Xu and Atanas Rountev. Merging equivalent contexts for scalable heap-cloning-based context-sensitive points-to analysis. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis*, ISSTA '08, pages 225–236, New York, NY, USA, 2008. ACM.

[112] Dacong Yan, Guoqing Xu, and Atanas Rountev. Rethinking SOOT for summary-based whole-program analysis. In *Proceedings of the ACM SIGPLAN International Workshop on State of the Art in Java Program Analysis*, SOAP '12, New York, NY, USA, 2012. ACM.

[113] Sen Ye, Yulei Sui, and Jingling Xue. Region-based selective flow-sensitive pointer analysis. In Markus Müller-Olm and Helmut Seidl, editors, *Static Analysis*, pages 319–336, Cham, 2014. Springer International Publishing.

[114] Suan Hsi Yong, Susan Horwitz, and Thomas Reps. Pointer analysis for programs with structures and casting. In *Proceedings of the ACM SIGPLAN 1999 Conference on Programming Language Design and Implementation*, PLDI '99, pages 91–103, New York, NY, USA, 1999. ACM.

[115] Greta Yorsh, Eran Yahav, and Satish Chandra. Generating precise and concise procedure summaries. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, New York, NY, USA, 2008. ACM.

[116] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. Level by level: Making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th Annual IEEE/ACM International Symposium on Code Generation and Optimization*, CGO '10, pages 218–229, New York, NY, USA, 2010. ACM.

[117] Jyh-shiarn Yur, Barbara G. Ryder, and William A. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *Proceedings of the 21st International Conference on Software Engineering*, ICSE '99, pages 442–451, New York, NY, USA, 1999. ACM.

[118] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *Proceedings of the 4th ACM SIGSOFT Symposium on Foundations of Software Engineering*, SIGSOFT '96, pages 81–92, New York, NY, USA, 1996. ACM.

[119] Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. Hybrid top-down and bottom-up interprocedural analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI '14, New York, NY, USA, 2014. ACM.

[120] Jisheng Zhao, Michael G. Burke, and Vivek Sarkar. Parallel sparse flow-sensitive points-to analysis. In *Proceedings of the 27th International Conference on Compiler Construction*, CC 2018, pages 59–70, New York, NY, USA, 2018. ACM.

[121] Xin Zheng and Radu Rugina. Demand-driven alias analysis for c. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, POPL '08, pages 197–208, New York, NY, USA, 2008. ACM.

[122] Jianwen Zhu. Symbolic pointer analysis. In *Proceedings of the 2002 IEEE/ACM International Conference on Computer-aided Design*, ICCAD '02, pages 150–157, New York, NY, USA, 2002. ACM.

[123] Jianwen Zhu and Silvian Calman. Symbolic pointer analysis revisited. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation*, PLDI '04, pages 145–157, New York, NY, USA, 2004. ACM.

Appendix A

Additional Data

We have evaluated our GPG-based points-to analysis implementation on SPEC CPU 2006 benchmarks. The data is presented in the form of graphs. However, the number of procedures varies significantly across the benchmark programs. Besides, the number of GPUs and GPBs varies across GPGs. Hence we have plotted such data in terms of percentages. The actual procedure counts are given in the tables below.

Program	0 GPUs	0 GPBs	% of Dead GPUs					% of empty GPBs eliminated					% of GPBs reduced because of coalescing					% of back edges reduced because of coalescing			
			0-20	20-40	40-60	60-80	80-100	0-20	20-40	40-60	60-80	80-100	0-20	20-40	40-60	60-80	80-100	0-20	20-40	40-80	80-100
lbm	15	0	4	0	0	0	0	4	0	15	0	0	19	0	0	0	0	0	0	0	0
mcf	12	0	10	1	0	0	0	14	0	8	0	1	17	2	3	1	0	1	0	0	3
libquantum	45	0	32	2	1	0	0	31	0	47	2	0	61	7	4	5	3	0	0	0	6
bzip2	78	8	9	0	2	0	0	22	0	59	0	0	78	2	1	0	0	1	0	0	0
milc	184	3	12	0	0	0	0	61	0	131	1	0	190	2	1	0	0	0	0	0	1
sjeng	101	2	29	1	2	0	0	8	0	101	18	4	124	1	3	3	0	0	0	0	9
hmmer	241	5	31	0	1	0	2	68	0	202	0	0	265	2	1	1	1	1	0	0	1
h264ref	439	3	123	2	1	1	0	195	2	343	18	5	534	13	10	4	2	5	0	0	3
gobmk	1437	2	1260	2	0	0	0	1268	0	1380	45	4	2144	241	164	89	59	7	1	0	97

Table A.1: Effectiveness of redundancy elimination optimizations.

Program	Proc. count for different buckets of # of GPBs						Proc. count for different buckets of # of GPUs								Proc. count for different buckets of % of CI				
	0	1-3	4-10	11-25	26-35	>35	0	1-3	4-6	7-10	11-30	31-50	51-70	>70	0-20	20-40	40-60	60-80	80-100
lbm	0	18	1	0	0	0	15	4	0	0	0	0	0	0	4	0	0	0	0
mcf	0	22	1	0	0	0	12	6	2	2	1	0	0	0	8	0	0	1	2
libquantum	0	80	0	0	0	0	38	42	0	0	0	0	0	0	38	2	0	0	2
bzip2	8	79	2	0	0	0	78	10	1	0	0	0	0	0	11	0	0	0	0
milc	3	191	2	0	0	0	184	6	5	1	0	0	0	0	4	0	0	0	8
sjeng	2	128	3	0	0	0	101	26	1	3	2	0	0	0	1	0	2	3	26
hmmer	5	254	15	1	0	0	242	27	5	1	0	0	0	0	26	2	1	0	4
h264ref	3	531	23	7	2	0	434	81	20	8	18	3	1	1	78	3	4	0	47
gobmk	2	2568	120	3	0	6	1436	83	87	972	120	1	0	0	1077	33	60	31	62

Table A.2: Measurment of the goodness of procedure summaries.

Program	Time (in seconds)													
	FSCS with blocking						FSCS without blocking						FICI	FICS
	Total	Streng. Reduc.	Redundancy Elim				Total	Stren. Reduc.	Redundancy Elim					
			Total	Dead GPU Elim	Empty GPB Elim	Coalesce			Total	Dead GPU Elim	Empty GPB Elim	Coalesce		
lbm	0.085	0.043	0.001	0.000	0.000	0.000	0.084	0.041	0.001	0.000	0.000	0.000	0.072	0.108
mcf	5.815	5.052	0.032	0.002	0.003	0.017	3.053	2.529	0.021	0.001	0.003	0.008	13.224	47.088
libquantum	0.830	0.518	0.066	0.001	0.012	0.030	0.601	0.348	0.033	0.001	0.006	0.014	0.067	0.393
bzip2	1.267	0.582	0.010	0.000	0.002	0.003	1.183	0.530	0.009	0.000	0.001	0.003	0.867	2.787
milc	1.329	0.664	0.017	0.001	0.003	0.006	1.301	0.640	0.016	0.000	0.003	0.006	3.301	4.538
sjeng	4.304	2.184	0.081	0.001	0.004	0.068	4.001	1.861	0.077	0.001	0.003	0.066	1.057	4.073
hmmer	6.225	4.344	0.087	0.002	0.015	0.037	6.006	4.117	0.081	0.001	0.015	0.035	27.453	26.521
h264ref	80.319	64.542	0.388	0.005	0.046	0.188	88.464	76.217	0.397	0.002	0.057	0.182	234.175	841.067
gobmk	462.875	353.583	49.372	0.232	3.463	40.522	248.393	169.199	24.784	0.127	1.319	20.773	36.193	707.428

Table A.3: Time measurements.

Program	Proc. count for different buckets of ratio of GPBs/BBs in CFG and GPG after inlining					Proc. count for different buckets of ratio of GPBs/BBs in CFG and optimized GPG					Proc. count for different buckets of ratio of GPBs in GPG after inlining and optimized GPG				
	0-20	20-40	40-60	60-80	80-100	0-20	20-40	40-60	60-80	80-100	0-20	20-40	40-60	60-80	80-100
lbm	9	3	3	1	3	11	5	3	0	0	2	1	15	0	1
mcf	14	5	2	1	1	22	1	0	0	0	3	5	13	2	0
libquantum	40	16	12	1	11	56	13	11	0	0	15	21	41	2	1
bzip2	55	15	8	4	7	70	12	7	0	0	12	2	68	4	3
milc	118	21	14	9	34	138	24	34	0	0	10	6	174	2	4
sjeng	85	19	7	3	19	105	9	19	0	0	16	14	102	0	1
hmmer	194	41	22	1	17	235	23	16	0	1	29	37	194	4	11
h264ref	401	69	49	10	37	474	50	40	2	0	26	65	437	13	25
gobmk	2315	279	29	9	67	2594	37	66	1	1	141	464	1494	579	21

Table A.4: Relative size of GPGs with respect to the size of corresponding procedures in terms of GPBs and basic blocks.

Program	Proc. count for different buckets of ratio of GPUs/stmts in CFG and GPG after inlining					Proc. count for different buckets of ratio of GPUs/stmts in CFG and optimized GPG					Proc. count for different buckets of ratio of GPUs in GPG after inlining and optimized GPG				
	0-20	20-40	40-60	60-80	80-100	0-20	20-40	40-60	60-80	80-100	0-20	20-40	40-60	60-80	80-100
lbm	16	3	0	0	0	19	0	0	0	0	18	0	0	1	0
mcf	21	0	1	1	0	23	0	0	0	0	17	0	3	0	3
libquantum	75	4	0	0	1	80	0	0	0	0	45	10	7	6	12
bzip2	89	0	0	0	0	89	0	0	0	0	85	0	0	0	4
milc	195	1	0	0	0	196	0	0	0	0	191	0	0	0	5
sjeng	131	0	2	0	0	133	0	0	0	0	105	0	1	2	25
hmmer	273	0	1	0	1	275	0	0	0	0	266	7	1	0	1
h264ref	540	12	10	1	3	563	2	1	0	0	505	3	1	1	56
gobmk	2690	4	1	0	4	2698	1	0	0	0	1460	1	5	21	1212

Table A.5: Relative size of GPGs with respect to the size of corresponding procedures in terms of GPUs and pointer assignments.

Program	Proc. count for different buckets of ratio of CF edges in CFG and GPG after inlining					Proc. count for different buckets of ratio of CF edges in CFG and optimized GPG					Proc. count for different buckets of ratio of CF edges in GPG after inlining and optimized GPG				
	0-20	20-40	40-60	60-80	80-100	0-20	20-40	40-60	60-80	80-100	0-20	20-40	40-60	60-80	80-100
lbm	13	4	2	0	0	19	0	0	0	0	18	0	0	0	1
mcf	21	1	1	0	0	23	0	0	0	0	16	3	3	1	0
libquantum	61	8	2	0	9	80	0	0	0	0	64	11	4	0	1
bzip2	74	8	3	0	4	89	0	0	0	0	79	2	3	2	3
milc	183	6	4	0	3	195	0	1	0	0	188	2	1	0	5
sjeng	124	5	1	0	3	133	0	0	0	0	130	1	1	0	1
hmmer	246	24	3	0	2	274	1	0	0	0	254	3	4	3	11
h264ref	508	27	13	1	17	560	0	2	1	3	506	13	9	12	26
gobmk	2560	78	31	1	29	2695	1	2	1	0	1773	278	265	353	30

Table A.6: Relative size of GPGs with respect to the size of corresponding procedures in terms of control flow edges.