Generalized Points-to Graphs: A Precise and Scalable Abstraction for Points-to Analysis

PRITAM M. GHARAT and UDAY P. KHEDKER, Indian Institute of Technology Bombay
ALAN MYCROFT, University of Cambridge

Computing precise (fully flow- and context-sensitive) and exhaustive (as against demand-driven) points-to information is known to be expensive. Top-down approaches require repeated analysis of a procedure for separate contexts. Bottom-up approaches need to model unknown pointees accessed indirectly through pointers that may be defined in the callers and hence do not scale while preserving precision. Therefore, most approaches to precise points-to analysis begin with a scalable but imprecise method and then seek to increase its precision. We take the opposite approach in that we begin with a precise method and increase its scalability. In a nutshell, we create naive but possibly non-scalable procedure summaries and then use novel optimizations to compact them while retaining their soundness and precision.

For this purpose, 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 strength reduction (which reduces the indirection levels), control flow minimization (which removes control flow edges while preserving soundness and precision), and call inlining (which enhances the opportunities of these optimizations).

The effectiveness of GPGs lies in the fact that they discard as much control flow as possible without losing precision. This is the reason GPGs are very small even for main procedures that contain the effect of the entire program. This allows our implementation to scale to 158 kLoC for C programs.

At a more general level, GPGs provide a convenient abstraction to represent and transform memory 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.

CCS Concepts: • **Theory of computation** → **Program analysis**; • **Software and its engineering** → **Imperative languages**; **Compilers**; *Software verification and validation*;

Additional Key Words and Phrases: Flow- and context-sensitive interprocedural analysis, bottom-up interprocedural analysis, points-to analysis, procedure summaries

ACM Reference format:
Pritam M. Gharat, Uday P. Khedker, and Alan Mycroft. 2020. Generalized Points-to Graphs: A Precise and Scalable Abstraction for Points-to Analysis. *ACM Trans. Program. Lang. Syst.* 42, 2, Article 8 (May 2020), 78 pages.
https://doi.org/10.1145/3382092

P. Gharat was partially supported by TCS Research Fellowship.

Authors' addresses: P. M. Gharat and U. P. Khedker (corresponding author), Indian Institute of Technology Bombay, India; emails: {pritamg, uday}@cse.iitb.ac.in; A. Mycroft, University of Cambridge, UK; email: Alan.Mycroft@cl.cam.ac.uk.

0164-0925/2020/05-ART8 $15.00
https://doi.org/10.1145/3382092

1 INTRODUCTION

Points-to analysis discovers information about indirect accesses in a program. Its precision influences the precision and scalability of client program analyses significantly. Computationally intensive analyses such as model checking are noted as being ineffective on programs containing pointers, partly because of imprecision of points-to analysis [2].

1.1 The Context of This Work

We focus on exhaustive as against demand-driven [7, 13, 36, 37] points-to analysis. A demand-driven points-to analysis 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, however, computes all points-to information that can be queried later by a client analysis; multiple queries do not require points-to analysis to be repeated. For precision of points-to information, we are interested in full flow- and context-sensitive points-to analysis. A flow-sensitive analysis respects the control flow and computes separate dataflow information at each program point. This matters because a pointer could have different pointees at different program points because of redefinitions. Hence, a flow-sensitive analysis provides more precise results than a flow-insensitive analysis but can become inefficient at the interprocedural level. 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 fully context-sensitive analysis does not lose precision even in the presence of recursion. Both flow- and context-sensitivity enhance precision, and we aim to achieve this without compromising efficiency.

A top-down approach to interprocedural context-sensitive analysis propagates information from callers to callees [47] effectively traversing the call graph top-down. In the process, it analyzes a procedure each time a new dataflow value reaches it from some call. Several popular approaches fall in this category: the call-strings method [34], its value-based variants [20, 29], and the tabulation-based functional method [30, 34]. By contrast, bottom-up approaches [5, 9, 12, 16, 26, 32, 39, 42–47] avoid analyzing a procedure multiple times by constructing its *procedure summary*, which is used to incorporate the effect of calls to the procedure. Effectively, this approach traverses the call graph bottom-up.[1] 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 them to represent the effect of the calls at the call sites.

For points-to analysis, an additional dimension of context sensitivity arises because heap locations are typically abstracted using allocation sites—all locations allocated by the same statement are treated alike. These allocation sites could be created context insensitively or could be cloned based on the contexts. We summarize various methods of points-to analysis using the metric described in Figure 20 and use it to position our work in Section 11.

1.2 Our Contributions

Most approaches to precise points-to analysis begin with a scalable but imprecise method and then seek to increase its precision. We take the opposite approach in that we begin with a precise method and increase its scalability. We create naive, possibly non-scalable, procedure summaries and then

[1] 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*. At the interprocedural level, a forward dataflow analysis (e.g., available expressions analysis) could be top-down or bottom-up and thus can be a backward dataflow analysis (e.g., live variables analysis).

use novel optimizations to compact them while retaining their soundness and precision. More specifically, we advocate a new form of bottom-up procedure summaries, called the *generalized points-to graphs* (GPGs) for flow- and context-sensitive points-to analysis. GPGs represent memory transformers (summarizing the effect of a procedure) and contain *generalized points-to updates* (GPUs) representing individual memory updates along with the control flow between them. GPGs are compact—their compactness is achieved by a careful choice of a suitable representation and a series of optimizations as described next:

(1) Our representation of memory updates—the GPU, denoted γ—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. They are governed by the following possibilities of data dependence of a GPU γ_2 on another GPU γ_1 (illustrated in Section 2.2).
   - ***Case A*:** The dependence of γ_2 on γ_1 can be determined. Then, there are two possibilities:
     (i) GPU γ_2 follows γ_1 on some control flow path and has the following kind of dependence on γ_1: (a) a read-after-write (RaW) dependence, (b) a write-after-read (WaR) dependence, or (c) a write-after-write (WaW) dependence. A read-after-read (RaR) dependence is irrelevant.
     (ii) GPU γ_2 does not have a dependence on γ_1.
   - ***Case B*:** More information is needed to determine whether or not γ_2 has a dependence on γ_1. Then, γ_2 has a *potential* dependence on γ_1. We use source-language type information ubiquitously to rule out potential dependence following C-style rules on indirect accesses via type-casted pointers. This resolves some instances of case B into case A.ii. These cases are exploited by three classes of optimizations as described next:
   - *Elimination of data dependence*: These optimizations attempt to eliminate data dependences between GPUs so that the control flow can be minimized. The *strength reduction* optimization exploits the RaW dependence (case A.i.a) of GPU γ_2 on GPU γ_1. It simplifies GPU γ_2 by reducing the pointer indirection levels in it by using the pointer information from γ_1 to eliminate the data dependence between them. The *dead GPU elimination* optimization exploits the WaW dependence (case A.i.b) between GPU γ_1 and γ_2. If the locations written by γ_1 are rewritten by γ_2 along every path reaching the end of the procedure, γ_1 does not have any effect on the callers. Deleting it eliminates the WaW dependence between γ_1 and γ_2.
   - *Control flow minimization*: These optimizations exploit the WaR dependence (case A.i.c) and the absence of data dependence (case A.ii). They discard control flow selectively by converting some sequentially ordered GPUs into parallel GPUs when there is WaR dependence or no dependence between them—since all reads precede any write in parallel assignments, they preserve WaR dependences inherently. When there are RaW or WaW dependences between GPUs, we preserve control flow between them.
   - *Call inlining*: This optimization handles case B by progressively providing more information. It inlines the summaries of the callees of a procedure enhancing the opportunities of strength reduction and control flow optimization and enabling context-sensitive analyses. Recursive calls are handled by refining the GPGs through a fixed-point computation. Calls through function pointers are handled through delayed inlining.

   Our measurements suggest that 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. Our optimizations are effective because they eliminate data dependence wherever possible and discard irrelevant control flow. This aids the scalability of points-to analysis without violating soundness or causing imprecision.

(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 of a bottom-up analysis.

These optimizations are based on the following novel operations and analyses:

- We define operations of *GPU composition* and *GPU reduction* to simplify a GPU using the information from RaW dependences, thereby eliminating the dependences.
- We perform *reaching GPUs analysis* (to identify the GPUs reaching a given statement) and *coalescing analysis* (to remove control flow edges while preserving soundness and precision).

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 before [11]. This article provides a complete treatment and enhances the core ideas significantly. We describe our formulations for a C-like language.

1.3 Organization of the Article

Section 2 describes the limitations of past approaches. Section 3 introduces the concept of GPUs that form the basis of GPGs and provides an overview of GPG construction through a motivating example. Section 4 describes the strength reduction optimization performed on GPGs. Section 5 explains dead GPU elimination. Section 6 describes control flow minimization optimizations performed on GPGs. Section 7 explains the interprocedural use of GPGs by defining call inlining and shows how recursion is handled. Section 8 shows how GPGs are used for performing points-to analysis. Section 9 proves soundness and precision of our method by showing its equivalence with a top-down flow- and context-sensitive classical points-to analysis. Section 10 presents empirical evaluation on SPEC benchmarks, and Section 11 describes related work. Section 12 concludes the article.

Some details (handling fields of structures and union, heap memory, function pointers, etc.) are available in an appendix available electronically.[2] We have included cross references to the material in the appendix where relevant.

2 EXISTING APPROACHES AND THEIR LIMITATIONS

This section reviews some basic concepts and describes the challenges in constructing procedure summaries for efficient points-to analysis. It concludes by describing the limitations of the past approaches and outlining our key ideas. For further details of related work, see Section 11.

2.1 Basic Concepts

In this section, we describe the nature of memory, memory updates, and memory transformers.

2.1.1 Abstract and Concrete Memory. There are two views of memory and operations on it. First, we have the concrete memory view corresponding to runtime operations where a memory location representing a pointer always points to exactly one memory location or NULL (which is a distinguished memory location). Unfortunately, this is, in general, statically uncomputable.

[2] https://github.com/PritamMG/GPG-based-Points-to-Analysis.

Second, 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 overapproximate concrete operations to ensure soundness. Formally, let L and $L_P \subseteq L$ denote the sets of locations[3] and pointers, respectively. The *concrete memory* after a pointer assignment is a function $M : L_P \rightarrow L$. The *abstract memory* after a pointer assignment is a relation $M \subseteq L_P \times L$. In either case, we view M as a graph with L as the set of nodes. An edge $x \rightarrow y$ in M is a *points-to edge* indicating that $x \in L_P$ contains the address of $y \in L$. The abstract memory associated with a statement is an overapproximation of the concrete memory associated with every occurrence of the statement in the same or different control flow paths. Unless noted explicitly, all subsequent references to memory and its transformations refer to the abstract view.

2.1.2 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. We call it a *memory transformer* because it computes the memory after a call to a procedure based on the memory before the call. Given a memory M and a memory transformer Δ, the updated memory M' is computed by $M' = \Delta(M)$ as illustrated in Example 2 (Section 2.3).

2.1.3 Strong and Weak Updates. In concrete memory, every assignment overwrites the contents of the (single) 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 these locations but merely *adds* x to their 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 that we return to later: prior to the preceding 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* (i.e., 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 4.4). A direct assignment to a variable (e.g., $p = \&x$) is special case of 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 and make the distinction only where required.

2.2 Challenges in Constructing Procedure Summaries for Points-to Analysis

In the absence of indirect assignments involving pointers, data dependence between memory updates within a procedure can be inferred by using variable names without requiring any information from the callers. In such a situation, procedure summaries for some analyses, including various bit-vector dataflow analyses (e.g., live variables analysis), can be precisely represented by constant *gen* and *kill* sets [1, 22] or graph paths discovered using reachability [30].

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. Often, and particularly for points-to analysis, we have a situation

[3] Here we talk about non-heap locations. Heap locations are handled as explained in the electronic appendix.

where a procedure summary must either lose information or retain internal details that can only be resolved when its caller is known.

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

Example 1. For many calls, procedure $Q()$ on the right simply returns $\&a$, but until we are certain that $*p$ does not alias with x, we cannot perform this constant-propagation optimization; assignment 04 *blocks* it. If it is known that $*p$ and x *always* alias , then we can optimize Q to return $\&b$ (WaW dependence of statement 04 on statement 03 and RaW dependence of statement 05 on statement 04). If it is known that they *never* alias , we can optimize this code to return $\&a$ (no dependence between statements 04 and 03 but RaW dependence of statement 05 on statement 03). If nothing is known about the alias information, then we must retain assignment 04 in the procedure summary for Q (potential dependence of statement 04 on statement 03 and of statement 05 on statements 04 and 03). The key idea is that information from the calling context(s) can determine whether a potentially blocking assignment really blocks an optimization or not.

The preceding 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 overapproximated.

Thus, a flow-sensitive memory transformer for points-to analysis requires a compact representation for memory updates that captures the minimal control flow between them succinctly.

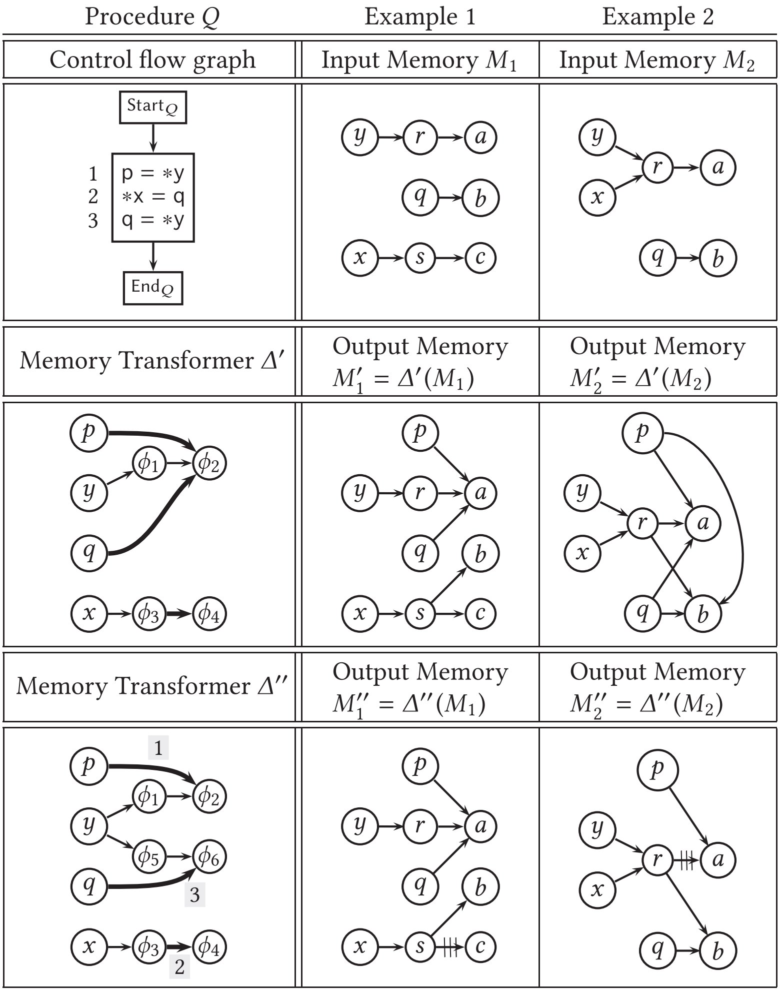
2.3 Limitations of Existing Procedure Summaries for Points-to Analysis

A common solution for modeling indirect accesses of unknown pointees in a memory transformer is to use *placeholders* (also known as external variables [26, 39, 42] and extended parameters [43]). They are pattern-matched against the input memory to compute the output memory. Here we describe two broad approaches that use placeholders.

The first approach, which we call a *multiple transfer functions* (MTFs) approach, proposed a precise representation of a procedure summary for points-to analysis as a collection of (conditional) *partial transfer functions* (PTFs) [5, 16, 43, 46]. 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 [4, 6, 23, 26, 27, 39, 42]. 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. 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 2 and that of Δ'' in Example 3.

Example 2. Transformer Δ' in Figure 1 is constructed by the STF approach. It is an abstract points-to graph containing placeholders ϕ_i for modeling unknown pointees. For example, ϕ_1 represents the pointees of y, and ϕ_2 represents the pointees of pointees of y. Note that 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).

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


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

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.

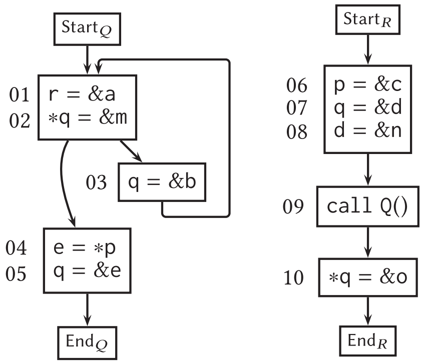
Memory transformer Δ'' is a flow-sensitive version of δ'. It shows that precision can be improved by using a separate placeholder for every access of a pointee. However, the size of the memory transformer increases. Labels on the edges indicate their sequencing. Edges killed in the memory are shown struck-through.

Fig. 1. STF-style memory transformers and associated transformations. Unknown pointees are denoted by placeholders ϕ_i. Thick edges in a memory transformer represent the points-to edges *generated* by it, whereas other edges are carried forward from the input memory.

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 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 3. In Figure 1, memory transformer Δ'' differs from Δ' in two ways. First, it uses a separate placeholder for every access of a pointee to avoid an overapproximation 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 $r \rightarrow a$



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

All variables are global

Fig. 2. A motivating example. Procedures are represented by their control flow graphs.

and hence q does not point to a (as shown in M_2''). Second, 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 that 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. This 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.

2.4 Our Key Ideas

We propose a GPG as a representation for a memory transformer of a procedure; special cases of GPGs also represent memory as a points-to relation. A GPG is characterized by the following key ideas that overcome the two limitations described in Section 2.3:

- A GPG leaves the placeholders implicit by using the counts of indirection levels. Simple arithmetic on the counts allows us to combine the effects of multiple memory updates.
- A GPG uses a flow relation to order memory updates. Interestingly, it can be compressed dramatically without losing precision and can be optimized into a compact acyclic flow relation in most cases, even if the procedure it represents has loops or recursive calls.

Section 3 illustrates them using a motivating example and gives a big-picture view.

3 THE GPGS AND AN OVERVIEW OF THEIR CONSTRUCTION

In this section, we define a GPG that serves as our memory transformer. It is a graph with *generalized points-to blocks* (GPBs) as nodes that contain GPUs. We provide an overview of our the ideas and algorithms in a limited setting of our motivating example of Figure 2. Toward the end of this section, Figure 6 summarizes them as a collection of abstractions, operations, dataflow analyses, and optimizations.

3.1 Defining a GPG

We model the effect of a pointer assignment on an abstract memory by defining the concept of GPU in Definition 1, which gives the abstract semantics of a GPU. The concrete semantics of a GPU $x \xrightarrow[s]{i|j} y$ can be viewed as the following C-style pointer assignment with $i-1$ dereferences of x (or i dereferences of $\&x$) and j dereferences of $\&y$.

$$s : \underbrace{* * \ldots *}_{(i-1)} \mathsf{x} = \underbrace{* * \ldots *}_{j} \&\mathsf{y}$$

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 pair (x, i) is called the *source* and the pair (y, j) is called the *target* of the GPU. The letter γ is used to denote a GPU unless named otherwise.

Definition 1. Generalized points-to update.

This conceptual understanding of a GPU is central to the development of this work. However, most compiler intermediate languages are at a lower level of abstraction and instead represent this GPU using (placeholder) temporaries l_k $(0 \leq k < i)$ and r_k $(0 \leq k \leq j)$ as a sequence of C-style assignments (illustrated in Figure 3):[4]

$$\begin{array}{lllll} r_0 = \&y; & r_1 = *r_0; & \ldots & r_{j-1} = *r_{j-2}; & r_j = *r_{j-1}; \\ l_0 = \&x; & l_1 = *l_0; & \ldots & l_{i-1} = *l_{i-2}; & \\ *\, l_{i-1} = r_j; & & & & \end{array} \tag{1}$$

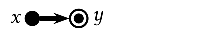
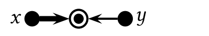
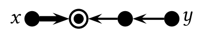
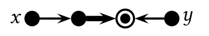
Statement labels, s, in GPUs are unique across procedures to distinguish between the statements of different procedures after call inlining. They facilitate distinguishing between strong and weak updates by identifying may-defined pointers (Section 3.1.1). Further, since GPUs are simplified in the calling contexts, statement labels allow back-annotation of points-to information within the procedure to which they belong. For simplicity, we omit the statement labels from GPUs when they are not required.

A GPU $\gamma : x \xrightarrow[s]{i|j} y$ generalizes a points-to edge[5] from x to y with the following properties:

- The direction indicates that the source x with *indlev* i identifies the locations being defined and the target y with *indlev* j identifies the locations whose addresses are read. We often refer to (x, i) as the source of γ and (y, j) as its target.

[4] Section 3.3.1 explains how this transformation is effectively reversed when transliterating intermediate code instructions for the "Initial GPG."

[5] Although a GPU is a generalization of a points-to edge, we reserve the term *edge* for a "flow edge" in a GPG.

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

Relevant memory graph for the GPU $x \xrightarrow[s]{i|j} y$ represented as in Equation (1)

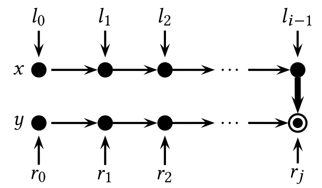


Fig. 3. GPUs and their memory graphs for basic pointer assignments in C (left) and for a general GPU (right). Solid circles represent memory locations, some of which are unknown. A double circle indicates the location whose address is being assigned, and a thick arrow shows the generated edge representing the effect of the assignment. In abstract memory, the circles may represent multiple locations.

- 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 4. The pointer assignment in statement 01 in Figure 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 presents the GPUs for basic pointer assignments in C and for the general GPU $x \xrightarrow[s]{i|j} y$. (To deal with C structs and unions, GPUs are extended to encode lists of field names—for details see Figure B.1 in Appendix B).

GPUs are useful rubrics of our abstractions because they can be composed to construct new simplified GPUs (i.e., GPUs with smaller indirection levels) whenever possible, thereby converting them progressively to classical points-to edges. The composition between GPUs eliminates RaW dependence between them and thereby the need for control flow ordering between them.

A GPU can be seen as a primitive memory transformer that is used as a building block for the GPG as a memory transformer for a procedure (Definition 2). The optimized GPG for a procedure differs from its control flow graph (CFG) in the following way:

- The CFG could have procedure calls, but an optimized GPG does not. We observe that an optimized GPG is acyclic in almost all cases, even if a procedure has loops or recursive calls.
- The GPBs that form the nodes in a GPG are analogous to the basic blocks of a CFG except the basic blocks are sequences of statements but GPBs are (unordered) sets of GPUs.

3.1.1 Abstract Semantics of GPBs. Abstract semantics of GPBs is a generalization of the semantics of pointer assignment in two ways. The first generalization is from a pointer assignment to a GPU, and the second generalization is from a single statement to multiple statements.

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 Δ, is a graph (N, E) whose edges in E abstract the control flow of the procedure. Nodes n in N are labelled both with GPBs, δ_n, and with may-definition sets, μ_n, containing GPU sources that are *may defined.* The latter set is empty initially, but is populated during GPG optimizations.
Finally, we use standard terminology from CFGs for GPGs: distinguished Start and End nodes, the notions of control flow path, dominance, the function *pred* giving the control flow predecessors (and its transitively closed version $pred^+$) and similarly *succ* and $succ^+$ for successors. By common abuse of notation, we often conflate nodes and their GPB labellings.

Definition 2. Generalized points-to blocks and generalized points-to graphs.

The semantics of a GPU (Definition 1) forms the basis of the semantics of a GPB. However, since a GPB has no control flow ordering on its GPUs and may contain multiple (simplified forms of) GPUs for a single source-language statement, or GPUs for multiple statements, we need to specify the combined effect of these multiple GPUs . In particular, differing concrete runs may execute a only a subset of the GPUs in some order. Let δ be a GPB and μ be its associated may-definition set, and let S be the set of source-language labels s occurring as labels of GPUs in δ. Now write $\delta|_{\text{s}}$ for $\{x \xrightarrow[s]{i|j} y \in \delta\}$. The abstract execution of δ is characterized by the following two features:

(1) All GPUs in $\delta|_{\text{s}}$ for every $s \in S$ are executed as parallel assignments so that all reads precede all writes, noting that the writes take place in a non-deterministic order.
(2) Due to abstract execution, some writes may not cause the previous pointees to be overwritten. The updates performed by a GPU $\gamma \in \delta|_{\text{s}}$ for some $s \in S$ are weak whenever the source of γ is a member of μ; otherwise, they are strong updates.

    In the simplest case, when the GPUs in $\delta|_{\text{s}}$ define multiple sources, all sources are included in μ—since each concrete execution of statement s defines only one source, there is a concrete run for every source that does not define the source. In other cases, the source of a GPU $\gamma \in \delta|_{\text{s}}$ may be included in μ if there is concrete run of δ that does not execute statement s.

Example 5. Consider a GPB $\delta = \{\gamma_1 : x \xrightarrow[11]{1|0} a, \gamma_2 : x \xrightarrow[11]{1|0} b, \gamma_3 : y \xrightarrow[12]{1|0} c, \gamma_4 : z \xrightarrow[13]{1|0} d, \gamma_5 : t \xrightarrow[13]{1|0} d, \}$ and its associated may-definition set $\mu = \{(z, 1), (t, 1)\}$ because γ_4 and γ_5 correspond to a single statement (statement 13) but define multiple sources. Note that γ_1 and γ_2 also correspond to a single statement (statement 11), but they define a single source $(x, 1)$. Then, after executing δ abstractly, we know that the points-to set of x is overwritten to become $\{a, b\}$ (i.e., x definitely points to one of a and b). Similarly, the points-to set of y is overwritten to become $\{c\}$ because γ_3 defines a single location c in statement 12. However, δ causes the points-to sets of z and t to *include* $\{d\}$ (without removing the existing pointees) because their sources are members of μ. Thus, x and y are strongly updated (their previous pointees are removed), but z and t are weakly updated (their previous pointees are augmented).

3.1.2 *Data Dependence Between GPUs.*

We use the usual notion of data dependence based on Bernstein's conditions [3]: two statements have a data dependence between them if they access the same memory location and at least one of them writes into the location [19, 28]. However, we restrict ourselves to locations that are pointers and use more intuitive names such as *read-after-write*, *write-after-write*, and *write-after-read* for *flow*, *output*, and *anti* dependence, respectively.

Formally, suppose $\gamma_1 : x \xrightarrow[s]{i|j} y$ is followed by $\gamma_2 : w \xrightarrow[t]{k|l} z$ on some control flow path. Then, γ_2 has the following dependence on γ_1 in the following cases (note that $i, k > 0$ and $j, l \geq 0$ in all cases):

WaW: $(w = x \wedge k = i)$
WaR: $(w = x \wedge k < i) \vee (w = y \wedge k \leq j)$
RaW: $(w = x \wedge k > i) \vee (z = x \wedge l \geq i)$.

Note that putting $i = j = k = l = 1$ reduces to the classical definitions of these dependences.

We call these dependences *definite* dependences. They correspond to case **A**.i in Section 1.2. In addition, if γ_2 postdominates γ_1 (i.e., follows γ_1 on every control flow path), we call the dependence *strict*. As illustrated in Example 1, γ_1 and γ_2 can have a dependence even when they do not have a common variable. Such a dependence is called a *potential* dependence (case **B** in Section 1.2).

Two GPUs on a control flow path cannot be placed within a single GPB if there is a definite or potential RaW or WaW dependence between them. However, it is safe to include them in the case of WaR dependence because of the "all reads precede all writes" semantics of GPBs.

Example 6. Consider the code snippet on the right. There is a WaR data dependence between statements 01 and 02. If the control flow were simply ignored, the statements could be executed in the reverse order, causing y to erroneously point to a. We construct a GPB $\{y \xrightarrow[01]{1|1} x, x \xrightarrow[02]{} 1|0^{a}\}$ for the code snippet. Since all reads precede any write, the execution of this GPB in the context of the memory represented by GPU $x \xrightarrow[12]{1|0} b$, computes the points-to information $\{y \rightarrow b, x \rightarrow a\}$ and excludes $y \rightarrow a$ thereby preserving the WaR dependence.

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

3.1.3 Finiteness of the Sets of GPUs. 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 and number of variables are 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, Appendix B replaces *indlev*s by indirection lists (*indlist*s) and shows how they are summarized ensuring the finiteness of the number of possible GPUs.

3.2 An Overview of GPG Operations

In this section, we intuitively describe GPU composition and GPU reduction.

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 $p = \&a; q = \&a$. GPU composition facilitates similar simplifications: Suppose a GPU γ_1 precedes γ_2 on some control flow path. If γ_2 has a RaW dependence on γ_1, then γ_2 is a *consumer* of the pointer information represented by the *producer* γ_1. In such a situation, a GPU composition $\gamma_3 = \gamma_2 \circ \gamma_1$ computes a new GPU γ_3 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, and although we might prefer γ_3, we cannot delete γ_2 until we have considered *all* control flow paths (see Section 3.2.2). GPU composition is a partial function—either succeeding with a simplified

GPU or signaling failure. A comparison of *indlev*s allows us to determine whether a composition is possible; if so, simple arithmetic on *indlev*s computes the *indlev* of the resulting GPU.

Example 7. 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 preceding example illustrates that 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).

3.2.2 GPU Reduction. We generalize the operation of composition as follows. If, instead of a single producer GPU as defined in Section 3.2.1, we have a set $\mathcal{R}$ of GPUs (representing generalized-points-to knowledge from all control flow paths to node n and obtained from the *reaching GPUs analyses* of Sections 4.5 and 4.6) and a single GPU $\gamma \in \delta_n$ corresponding to statement s, then *GPU reduction* $\gamma \circ \mathcal{R}$ constructs a set of one or more GPUs, all of which correspond to statement s. Taking the union of all such sets, as γ varies over δ_n, is considered as the information generated for node n and is semantically equivalent to δ in the context of $\mathcal{R}$ and, as suggested earlier, may beneficially replace δ.

GPU reduction $\gamma \circ \mathcal{R}$ eliminates the RaW data dependence of γ on the GPUs in $\mathcal{R}$, wherever possible, thereby eliminating the need for control flow between γ and the GPUs in $\mathcal{R}$.

3.3 An Overview of GPG Construction

The GPG of procedure R (denoted Δ_R) is constructed by traversing a spanning tree of the call graph starting with its leaf nodes. It involves the following steps:

(1) *creation* of the initial GPG and *inlining* optimized GPGs of called procedures within Δ_R,
(2) *strength reduction* optimization to simplify the GPUs in Δ_R by performing *reaching GPUs analyses* and transforming GPBs using *GPU reduction*,
(3) *dead GPU elimination* to remove redundant GPUs in a GPG (their presence may hinder control flow minimization because of WaW dependences), and
(4) *control flow minimization* to improve the compactness of Δ_R.

We illustrate these steps intuitively using the motivating example in Figure 2.

3.3.1 Creating a GPG and Call Inlining. 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 by treating the former as empty statements and the latter as non-deterministic control flow.
- Each pointer assignment, labeled s, in the CFG is transliterated to a GPU $x \xrightarrow[s]{i|j} y$, following Figure 3. If earlier compiler stages have broken compound C assignments such as $** p = ***q$; into a sequence of simpler SSA-form intermediate-language assignments using temporaries as in Equation (1), then compound statements are reconstructed by following def-use chains to eliminate such temporaries.
- A GPG node n is created for each such assignment with its GPB, δ_n, being the singleton set containing this GPU and with its associated may-definition set, μ_n, being empty.

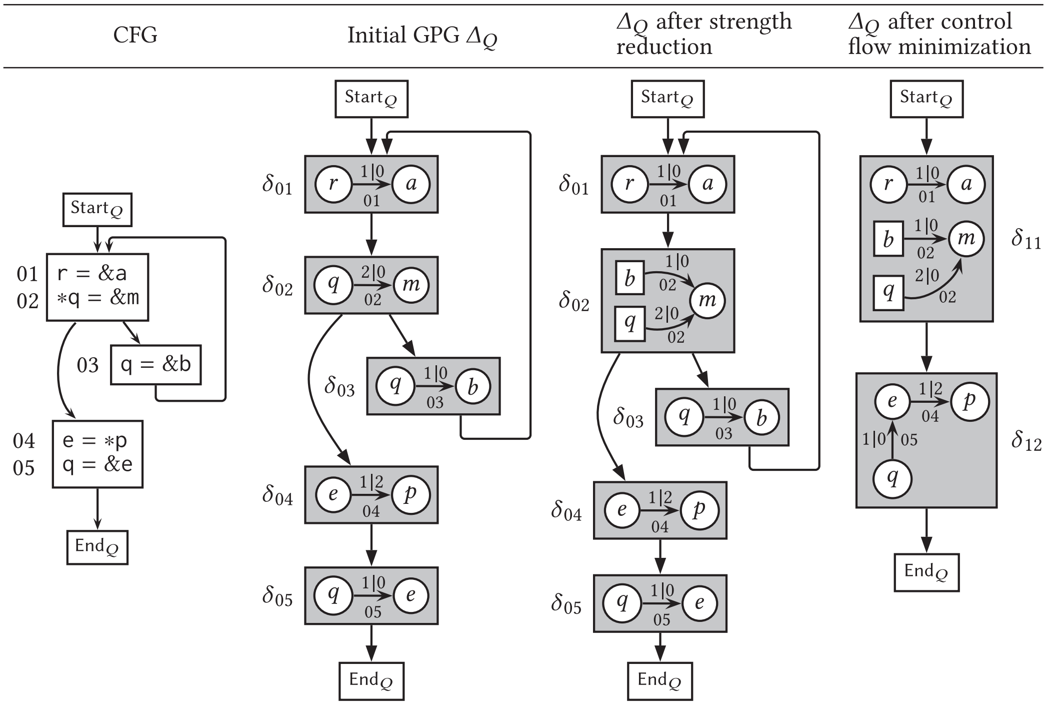

CFG
Initial GPG Δ_Q
Δ_Q after strength reduction
Δ_Q after control flow minimization
Start$_Q$
01 r = &a
02 *q = &m
03 q = &b
04 e = *p
05 q = &e
End$_Q$
δ_{01}
δ_{02}
δ_{03}
δ_{04}
δ_{05}
δ_{11}
δ_{12}
$r \xrightarrow[01]{1|0} a$
$q \xrightarrow[02]{2|0} m$
$b \xrightarrow[02]{1|0} m$
$q \xrightarrow[03]{1|0} b$
$e \xrightarrow[04]{1|2} p$
$q \xrightarrow[05]{1|0} e$

Fig. 4. Constructing the GPG for procedure Q (see Figure 2). Strength reduction of GPB δ_{02} causes a weak update by defining two sources $(b, 1)$ and $(q, 2)$. This is captured by the may-definition sets μ_{02} (after strength reduction) and μ_{11} (after control flow minimization) being $\{(b, 1), (q, 2)\}$. Pictorially, we use rectangles rather than circles to mark may-defined sources.

- The control flow between GPBs is induced from their order within a basic block in the CFG and from the control flow edges of the CFG.
- The procedure calls are replaced by the optimized GPGs of the callees. Every time we inline a GPG, we must take a fresh copy of its nodes, here achieved by simple renumbering. Note that the statement labels s appearing within GPUs are not renumbered.

Example 8. The initial GPG for procedure Q of Figure 2 is given in Figure 4. Each assignment is replaced by its corresponding GPU. The initial GPG for procedure R is shown in Figure 5 with the call to procedure Q on line 09 replaced by its optimized GPG.

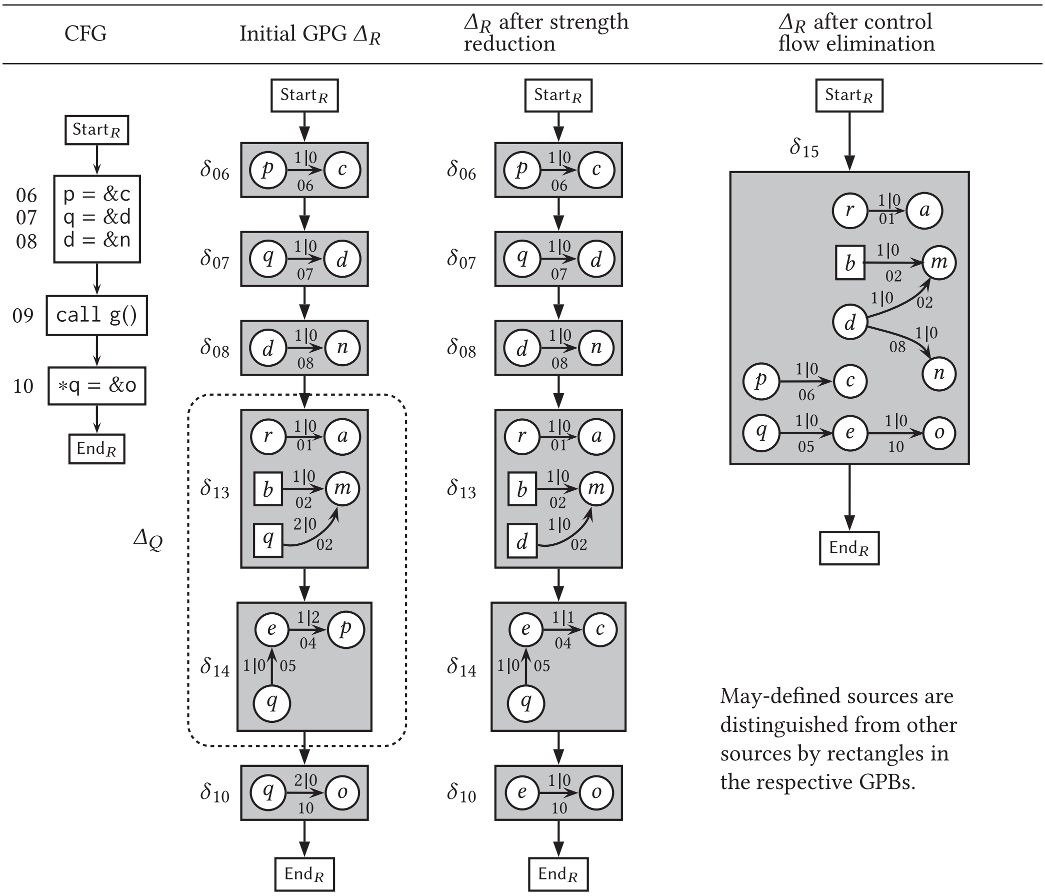
Examples 9 through 12 explain the analyses and optimizations over Δ_Q and Δ_R (GPGs for procedures Q and R) at an intuitive level.

3.3.2 *Strength Reduction Optimization.*

This step simplifies GPB δ_n for each node n by

- performing reaching GPUs analysis, which performs GPU reduction for each $\gamma \in \delta_n$ to compute a set of GPUs that are equivalent to δ_n, and
- replacing δ_n by the resulting GPUs and updating the associated μ_n as necessary.

Effectively, strength reduction simplifies each GPB as much as possible without needing the knowledge of aliasing in the caller. In the process, data dependences are eliminated to the extent


CFG
Initial GPG Δ_R
Δ_R after strength reduction
Δ_R after control flow elimination
Start$_R$
06 p = &c
07 q = &d
08 d = &n
09 call g()
10 *q = &o
End$_R$
δ_{06} p 1|0 06 c
δ_{07} q 1|0 07 d
δ_{08} d 1|0 08 n
Δ_Q
δ_{13} r 1|0 01 a; b 1|0 02 m; q 2|0 02
δ_{14} e 1|2 04 p; q 1|0 05
δ_{10} q 2|0 10 o
δ_{13} r 1|0 01 a; b 1|0 02 m; d 1|0 02
δ_{14} e 1|1 04 c; q 1|0 05
δ_{10} e 1|0 10 o
δ_{15} r 1|0 01 a; b 1|0 02 m; d 1|0 02; d 1|0 08 n; p 1|0 06 c; q 1|0 05 e 1|0 10 o

May-defined sources are distinguished from other sources by rectangles in the respective GPBs.

Fig. 5. Constructing the GPG for procedure R (see Figures 2 and 4). GPBs δ_{13} through δ_{14} in the GPG are the (renumbered) GPBs representing the inlined optimized GPG of procedure Q.

possible, facilitating dead GPU elimination and control flow minimization. Note that strength reduction does not create new GPBs; it only creates new (equivalent) GPUs within the same GPB. The statement labels in GPUs remain unchanged because the simplified GPUs of a statement continue to represent the same statement.

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 dataflow analysis in the spirit of the classical reaching definitions analysis except it computes sets of GPUs. It identifies the simplified GPUs for a GPB in the context of the GPUs reaching the GPB. By construction, all resulting GPUs are equivalent to the original GPUs of the GPB and have indirection levels that do not exceed that of the original GPUs. This process requires 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 the analysis is intraprocedural.[6]

The following two issues in reaching GPUs analysis are not illustrated in this section:

[6] In the presence of recursion and function pointers, the effect of calls gets progressively refined through repeated analyses of a procedure and its callees, but the analysis still remains intraprocedural (see Section 7.2 and Appendix C).

- In some cases, the reaching GPUs analysis needs to *block* certain GPUs from participating in GPU reduction (as in Example 1 in Section 2.2). There is no such instance in our example.
- The start GPB of a GPG contains the GPUs representing the *boundary definitions* (Section 4.4) representing the boundary conditions [1].

Example 9. We intuitively explain the reaching GPUs analysis for procedure Q over its initial GPG (Figure 4). The final result is shown later in Figure 8. GPU $r \xrightarrow[01]{1|0} a$ representing statement 01 reaches δ_{02} in the first iteration. However, it does not simplify GPU $q \xrightarrow[02]{2|0} m$ in δ_{02}. The GPUs $\{r \xrightarrow[01]{1|0} a, q \xrightarrow[02]{2|0} m\}$ reach the GPB δ_{03}. GPU $q \xrightarrow[03]{1|0} b$ cannot be simplified any further. In the second iteration, GPUs $\{r \xrightarrow[01]{1|0} a, q \xrightarrow[02]{2|0} m, q \xrightarrow[03]{1|0} b\}$ reach δ_{02} and δ_{03}. Composing $q \xrightarrow[02]{2|0} m$ with $q \xrightarrow[03]{1|0} b$ results in $b \xrightarrow[02]{1|0} m$. In addition, the pointee information of q is available only along one path (identified with the help of boundary definitions not shown here). Hence, the assignment causes a weak update and GPU $q \xrightarrow[02]{2|0} m$ is also retained. Thus, GPB δ_{02} contains two GPUs, $b \xrightarrow[02]{1|0} m$ and $q \xrightarrow[02]{2|0} m$, after simplification, and sources $(b, 1)$ and $(q, 2)$ are both included in μ_{02}. This process continues until the least fixed point is reached. Strength reduction optimization based on these results gives the GPG shown in the third column of Figure 4.

3.3.3 *Dead GPU Elimination.*

The following example illustrates dead GPU elimination in our motivating example. This optimization removes the WaW dependences where possible.

Example 10. In procedure Q of Figure 4, pointer q is defined in δ_{03} but is redefined in δ_{05} and hence GPU $q \xrightarrow[03]{1|0} b$ is eliminated. Therefore, GPB δ_{03} becomes empty and is removed from Δ_Q. Since GPU $q \xrightarrow[02]{2|0} m$ does not define q but its pointee, it is not killed by δ_{05} and is not eliminated from Δ_Q.

For procedure R in Figure 5, GPU $q \xrightarrow[07]{1|0} d$ in δ_{07} is killed by GPU $q \xrightarrow[05]{1|0} e$ in δ_{14}. Hence, GPU $q \xrightarrow[07]{1|0} d$ is eliminated from GPB δ_{07}. Similarly, GPU $e \xrightarrow[04]{1|1} c$ in GPB δ_{14} is removed because e is redefined by GPU $e \xrightarrow[10]{1|0} o$ in GPB δ_{10} (after strength reduction in Δ_R). 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 GPU $d \xrightarrow[02]{1|0} m$. This is because δ_{13} also contains GPU $b \xrightarrow[02]{1|0} m$, which defines b. Since statement 02 defines two sources, both of them are may-defined in δ_{13} (i.e., are included in μ_{13}). Thus, the previous definition of d cannot be killed—giving a weak update.

3.3.4 *Control Flow Minimization.*

This step improves the compactness of a GPG by eliminating empty GPBs from a GPG and then minimizing control flow by coalescing adjacent GPBs into a single GPB wherever there is no RaW or WaW dependences between them.

Example 11. After eliminating GPU $q \xrightarrow[07]{1|0} d$ from the GPG of procedure R in Figure 5 (because it is dead), GPB δ_{07} becomes empty and is removed from the optimized GPG.

We eliminate control flow in the GPG by performing coalescing analysis (Section 6). It partitions the nodes of a GPG (into *parts*) such that all GPBs in a part are coalesced (i.e., the GPB of the coalesced node contains the 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_n in a part, a control flow successor δ_m can appear in the same part only if the control flow between them is redundant. This requires that the GPUs in δ_m do not have RaW or WaW dependence on the other GPUs in the part.

A GPB obtained after coalescing may contain GPUs belonging to multiple statements, and not all of them may be executed in a concrete run of the GPB. This requires determining the associated may-definition set for the coalesced node that identifies the sources that are may-defined to maintain the abstract semantics of a GPB (Section 3.1.1).

Example 12. For procedure Q in Figure 4, the GPBs δ_1 and δ_2 can be coalesced: there is no data dependence between their GPUs because GPU $r \xrightarrow[01]{1|0} a$ in δ_1 defines r whose type is "`int **`," whereas the GPUs in δ_2 read the address of m, pointer b, and pointee of q. The type of latter two is "`int *`." Thus, a potential dependence between the GPUs in δ_1 and δ_2 is ruled out using types. However, GPUs $q \xrightarrow[02]{2|0} m$ in δ_2 and $e \xrightarrow[04]{1|2} p$ in δ_4 have a potential RaW dependence (p and q could be aliased in the caller) that is not ruled out by type information. Thus, we do not coalesce GPBs δ_2 and δ_4. Since there is no RaW dependence between the GPUs in the GPBs δ_4 and δ_5, we coalesce them (potential WaR dependence does not matter because all reads precede any write).

The GPB resulting from coalescing GPBs δ_1 and δ_2 is labeled δ_{11}. Similarly, δ_{12} is the result of coalescing GPBs δ_4 and δ_5. The loop formed by the back edge $\delta_2 \rightarrow \delta_1$ in the GPG before coalescing now becomes a self-loop over δ_{11}. Since, by definition, the GPUs in a GPB can never have a dependence between each other, the self-loop $\delta_{11} \rightarrow \delta_{11}$ is redundant and is hence removed.

For procedure R in Figure 5, after performing dead GPU elimination, the remaining GPBs in the GPG of procedure R are all coalesced into a single GPB δ_{15} because there is no data dependence between the GPUs of its GPBs.

As shown in Example 10, the GPUs $b \xrightarrow[02]{1|0} m$ and $q \xrightarrow[02]{2|0} m$ in procedure Q cause inclusion of the sources $(b, 1)$ and $(q, 2)$ in μ_{02}, leading further to their inclusion in μ_{11} for the coalesced GPB δ_{11}. Similarly, for procedure R, $(b, 1)$ is may-defined in GPB δ_{15} but not $(d, 1)$ because the latter is defined along all paths through procedure R but not the former as shown Figure 5.

3.4 The Big Picture

Figure 6 provides the big picture of GPG construction by listing specific abstractions, operations, dataflow analyses, and optimizations and shows dependences between them, along with the section that define them. The optimizations use the results of dataflow analyses. The reaching GPUs analysis uses the GPU operations that are defined in terms of key abstractions. The abstractions of allocation sites, indirection lists (*indlists*), and k-limiting (required for extending the analysis to structures, unions, and heap) are left to the appendix.

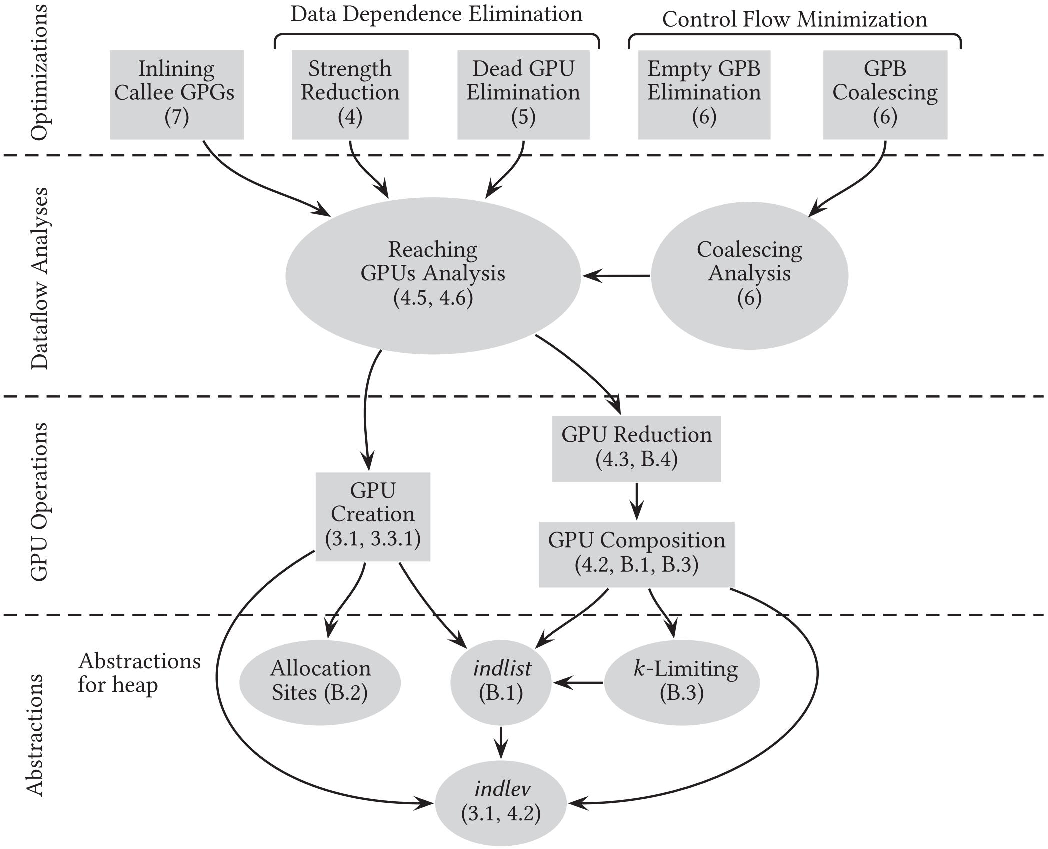


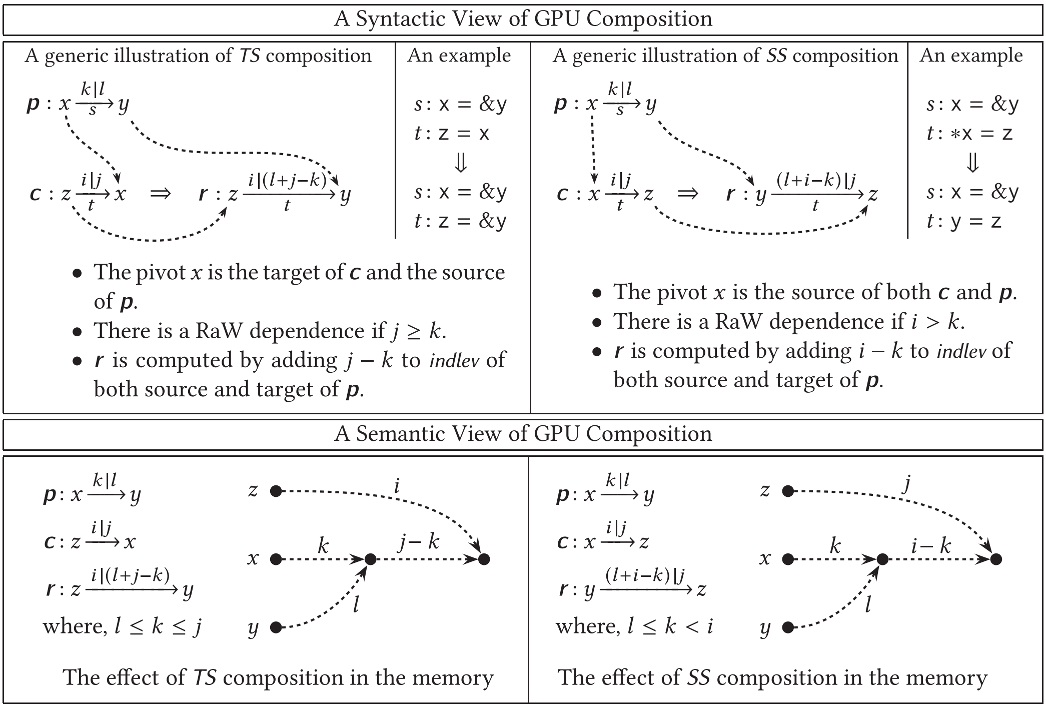
Fig. 6. The big picture of GPG construction. The arrows show the dependence between specific instances of optimizations, analyses, operations, and abstractions. The labels in parentheses refer to relevant sections.

4 STRENGTH REDUCTION OPTIMIZATION

This section begins with a motivation in Section 4.1. Section 4.2 defines GPU composition as a family of partial operations. Section 4.3 defines GPU reduction. Sections 4.5 presents the reaching GPUs analysis without blocking, and Section 4.6 extends it to include blocking.

4.1 Overview of Strength Reduction Optimization

Strength reduction optimization uses the knowledge of a *producer* GPU $\boldsymbol{p}$, to simplify a *consumer* GPU $\boldsymbol{c}$ (on a control flow path from $\boldsymbol{p}$) through an operation called *GPU composition* denoted $\boldsymbol{c} \circ \boldsymbol{p}$ (Section 4.2). 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* $\boldsymbol{c} \circ \mathcal{R}$ that involves a series of GPU compositions with appropriate producer GPUs in $\mathcal{R}$ to simplify the consumer GPU $\boldsymbol{c}$ maximally. The set $\mathcal{R}$ of GPUs used for simplification provides a context for $\boldsymbol{c}$ and represents generalized points-to knowledge from previous GPBs. It is obtained by performing a dataflow analysis called the *reaching GPUs analysis* (Sections 4.5, and 4.6), which computes the sets RGIn_n, RGOut_n, RGGen_n, and RGKill_n for every GPB δ_n. These dataflow variables represent the GPUs reaching the entry of GPB δ_n, its exit, the GPUs obtained through GPU reduction, and the GPUs whose propagation is killed by δ_n, respectively. The set RGGen_n is semantically equivalent to δ_n in the context of RGIn_n and may beneficially replace δ_n.


A Syntactic View of GPU Composition
A generic illustration of TS composition
$p : x \xrightarrow[s]{k|l} y$
$c : z \xrightarrow[t]{i|j} x \Rightarrow r : z \xrightarrow[t]{i|(l+j-k)} y$
An example
s: x = &y
t: z = x
⇓
s: x = &y
t: z = &y
A generic illustration of SS composition
$p : x \xrightarrow[s]{k|l} y$
$c : x \xrightarrow[t]{i|j} z \Rightarrow r : y \xrightarrow[t]{(l+i-k)|j} z$
An example
s: x = &y
t: *x = z
⇓
s: x = &y
t: y = z
A Semantic View of GPU Composition
$p: x \xrightarrow{k|l} y$
$c: z \xrightarrow{i|j} x$
$r: z \xrightarrow{i|(l+j-k)} y$
where, $l \leq k \leq j$
z
i
x
k
j − k
y
l
$p: x \xrightarrow{k|l} y$
$c: x \xrightarrow{i|j} z$
$r: y \xrightarrow{(l+i-k)|j} z$
where, $l \leq k < i$
z
j
x
k
i − k
y
l

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

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

The effect of *TS* composition in the memory

The effect of *SS* composition in the memory

Definition 3. Composing a consumer GPU $\boldsymbol{c}$ with a producer GPU $\boldsymbol{p}$ to compute a new GPU $\boldsymbol{r}$ that is equivalent to $\boldsymbol{c}$ in the context of $\boldsymbol{p}$. Both *SS* and *TS* compositions exploit a RaW dependence of statement labeled t on the statement labeled s because the pointer defined in $\boldsymbol{p}$ is used to simplify a pointer used in $\boldsymbol{c}$.

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. This could happen because of a potential dependence between $\boldsymbol{p}$ and $\boldsymbol{b}$. (Section 2.2). In such a situation (characterized formally in Section 4.6.1), replacing δ_{n} by RGGen$_n$ during strength reduction may be unsound. Hence we *postpone* the composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ explicitly by eliminating those GPUs from $\mathcal{R}$ that are blocked by a barrier. After inlining, the knowledge of the calling context may allow a barrier GPU to be reduced so that it no longer blocks a postponed reduction.

4.2 GPU Composition

We first present the intuition behind GPU composition before defining it formally.

4.2.1 The Intuition Behind GPU Composition. The composition of a consumer GPU $\boldsymbol{c}$ and a producer GPU $\boldsymbol{p}$ is possible when $\boldsymbol{c}$ has a RaW dependence on $\boldsymbol{p}$ through a common variable called the *pivot* of composition. It is the source of $\boldsymbol{p}$ but may be the source or the target of $\boldsymbol{c}$.

The type τ of composition $\boldsymbol{r} = \boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ indicates the name of the composition, which is *TS* or *SS*, where the first letter indicates the role of the pivot in $\boldsymbol{c}$ and second letter indicates its role in $\boldsymbol{p}$. For a *TS* composition, the pivot is the target of $\boldsymbol{c}$ (*T* for target) and the source of $\boldsymbol{p}$ (*S* for source), whereas for *SS* composition, pivot is the source of both $\boldsymbol{c}$ and $\boldsymbol{p}$. Note that *TS* and *SS* compositions are mutually exclusive for a given pair of $\boldsymbol{c}$ and $\boldsymbol{p}$ because the same variable cannot occur both in the RHS and LHS of an assignment in the case of pointers to scalars.[7]

[7] Since our language is modeled 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*

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

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

Fig. 7. GPU composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$.

Figure 7 illustrates these compositions. For *TS* composition, consider $\boldsymbol{c} : z \xrightarrow[t]{i|j} x$ and $\boldsymbol{p} : x \xrightarrow[s]{k|l} y$ with pivot x, which is the target of $\boldsymbol{c}$ and the source of $\boldsymbol{p}$. The goal of the 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 $x \; j|(l + \overrightarrow{j} - k) \; y$. This *balances* the *indlev*s of x in the two GPUs, allowing us to create a simplified GPU $\boldsymbol{r} : z \; i|(l + \overrightarrow{j} - k) \; y$.

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

(a) A composition $\boldsymbol{r} = \boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ is *valid* only if $\boldsymbol{c}$ has a RaW dependence on $\boldsymbol{p}$ through the pivot of the composition.
(b) A composition $\boldsymbol{r} = \boldsymbol{c} \circ^{\tau} \boldsymbol{p}$ is *desirable* only if the *indlev* of $\boldsymbol{r}$ does not exceed the *indlev* of $\boldsymbol{c}$.

Validity requires the *indlev* of the pivot in $\boldsymbol{c}$ to be greater than the *indlev* of pivot in $\boldsymbol{p}$. For the generic *indlev*s used in Figure 7, this requirement translates to the following constraints:

$$j \geq k \qquad (\textit{TS} \text{ composition}) \tag{2}$$

$$i > k \qquad (\textit{SS} \text{ composition}) \tag{3}$$

Observe that *SS* composition condition (3) prohibits equality because it involves the source nodes of both the GPUs. When $i = k$, $\boldsymbol{c}$ has WaW dependence on $\boldsymbol{p}$ instead of RaW dependence (Section 3.1.2) because $\boldsymbol{c}$ overwrites the location written by $\boldsymbol{p}$.

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 $\boldsymbol{r}$ does not exceed the corresponding *indlev* in the consumer GPU $\boldsymbol{c}$. This requires the *indlev* in the simplified GPU $\boldsymbol{r}$ and the consumer GPU $\boldsymbol{c}$ 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 $\boldsymbol{r}$, whereas the second term compares those of the targets (see Figure 7):

$$(i \leq i) \;\wedge\; (l + j - k \leq j) \quad \text{or equivalently} \quad l \leq k \qquad (\textit{TS} \text{ composition}), \tag{4}$$

$$(l + i - k \leq i) \;\wedge\; (j \leq j) \quad \text{or equivalently} \quad l \leq k \qquad (\textit{SS} \text{ composition}). \tag{5}$$

behavior that the programmer has not abused type casting to simulate such prohibited statements. Appendix B considers the richer situation with structs and unions where we can have an assignment $x \to n = x$ that might have both *TS* and *SS* compositions with a GPU $\boldsymbol{p}$ that defines x.

Example 13. Consider the statement sequence $x = *y; z = x$. A *TS* composition of the corresponding GPUs $\boldsymbol{p} : x \overset{\rightarrow}{1|2} y$ and $\boldsymbol{c} : z \overset{\rightarrow}{1|1} x$ is *valid* because $j = k = 1$, satisfying Constraint 2. However, if we perform this composition, we get $\boldsymbol{r} : z \overset{\rightarrow}{1|2} y$. 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.

We take a conjunction of the constraints of *validity* (2 and 3) and *desirability* (4 and 5) to characterize *admissible* GPU compositions:

$$l \leq k \leq j \qquad\qquad (TS\text{composition}), \tag{6}$$

$$l \leq k < i \qquad\qquad (SS\text{composition}). \tag{7}$$

Note that an *undesirable* GPU composition in a GPG 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.

4.3 GPU Reduction

GPU reduction Red $= \boldsymbol{c} \circ \mathcal{R}$ uses the GPUs in $\mathcal{R}$ (a set of data-dependence-free GPUs) to compute a set of GPUs whose *indlev*s do not exceed that of $\boldsymbol{c}$. During reduction, the *indlev* of $\boldsymbol{c}$ is reduced progressively using the GPUs from $\mathcal{R}$ through a sequence of *admissible* GPU compositions. A GPU resulting from GPU reduction is called a *simplified* GPU.

Formally, Red is the fixed point of the equation Red = GPU_reduction(Red, $\mathcal{R}$) with the initialization Red = $\{\boldsymbol{c}\}$. Function GPU_reduction (Definition 4) simplifies the GPUs in Red by composing them with those in $\mathcal{R}$. The resulting GPUs are accumulated in Red′, which is initially $\emptyset$. If a GPU $\gamma_1 \in$ Red is simplified, its simplified GPU r is included in *temp,* which is then added to Red′. However, if γ_1 cannot compose with any GPU in $\mathcal{R}$, then γ_1 is then added to Red′. The GPUs in Red′ are then simplified in the next iteration of the fixed-point computation. The fixed point is achieved when no GPU in Red′ can be simplified any further.

Example 14. Consider $\boldsymbol{c} : x \xrightarrow[23]{1|2} y$ with $\mathcal{R} = \{y \xrightarrow[21]{1|0} a, a \xrightarrow[22]{1|0} b\}$. The reduction $\boldsymbol{c} \circ \mathcal{R}$ involves two consecutive *TS* compositions. The first step with $y \xrightarrow[21]{1|0} a$ as $\boldsymbol{p}$ computes Red′ $= \{x \xrightarrow[23]{1|1} a\}$. Then, the reduced GPU $x \xrightarrow[23]{1|1} a$ becomes the consumer GPU and is composed with $a \xrightarrow[22]{1|0} b$ from $\mathcal{R}$, which results in Red′ $= \{x \xrightarrow[23]{1|0} b\}$. It cannot be reduced further, as it is already in the classical points-to form and the computation has reached the fixed point.

GPU reduction requires the set $\mathcal{R}$ to satisfy following properties:

- *Reduced form*: GPUs in $\mathcal{R}$ must be in their reduced form. Consider the GPU $x \xrightarrow{i|j} y$. Then,

```
Input: Red              // GPUs to be simplified
       $\mathcal{R}$              // The context in which the GPUs are to be simplified
Output: Red$'$          // The set of simplified GPUs equivalent to $\boldsymbol{c}$
GPU_reduction (Red, $\mathcal{R}$)       // One step in fixed point computation
{   Red$' = \emptyset$
    for each $\gamma_1 \in$ Red
    {   $temp = \emptyset$           // The set of simplified GPUs equivalent to $\gamma_1$
        for each $\gamma_2 \in \mathcal{R}$
        {   if ($r = \gamma_1 \circ^{\text{ts}} \gamma_2$) succeeds  then $temp = temp \cup \{r\}$
            else if ($r = \gamma_1 \circ^{\text{ss}} \gamma_2$) succeeds  then $temp = temp \cup \{r\}$
        }
        if $temp \neq \emptyset$ then Red$'$ = Red$' \cup$ $temp$
        else Red$'$ = Red$' \cup \{\gamma_1\}$
    }
    return Red$'$
}
```

Definition 4. GPU reduction $\boldsymbol{c} \circ \mathcal{R}$.

  - either $i = 1$ or x is live on entry (represented by a special variable x'), and
  - either $j = 0$ or y is live on entry (represented by a special variable y').

  The special variables are explained in Section 4.4.
- *Acyclicity:* The graph induced by the GPUs in $\mathcal{R}$ should be acyclic. $\mathcal{R}$ cannot have a subset like $\{x \xrightarrow{1|1} y, y \xrightarrow{1|1} x\}$. Taking the reduced form serves to replace this subset with $\{x \xrightarrow{1|1} y', y \xrightarrow{1|1} x'\}$, thereby ruling out the cycles.[8]
- *Completeness:* If $\mathcal{R}$ contains a GPU $x \xrightarrow{i|j} y$, then the source (x, i) must be defined along all paths reaching the GPB that is undergoing reduction. If x is the pivot of composition with $\boldsymbol{c}$ but (x, i) is not defined along some path, it means that the simplification of $\boldsymbol{c}$ is not complete and Red cannot replace $\boldsymbol{c}$. This is ensured by the introduction of boundary definitions in Section 4.4. Example 21 in Section 4.6 illustrates why completeness of $\mathcal{R}$ is required.
- *Absence of data dependence:* GPUs in $\mathcal{R}$ should not have a data dependence between them. This is preserved by reaching GPUs analyses (Sections 4.5 and 4.6). If GPU $\gamma_2 \in \mathcal{R}$ had a RaW dependence on GPU $\gamma_1 \in \mathcal{R}$, then γ_1 would have been simplified during reaching GPUs analysis; if γ_2 had a WaW dependence on γ_1 along a control flow path, γ_2 would be killed during reaching GPUs analysis along the path. Finally, if γ_2 had a potential dependence on γ_1, γ_2 would have been blocked and not included in $\mathcal{R}$.

These properties also hold for Red and are preserved by both variants of reaching GPUs analyses (Sections 4.5 and 4.6) both before and after coalescing (Section 6).

The convergence of reduction $\boldsymbol{c} \circ \mathcal{R}$ on a unique solution is guaranteed by the following:

- The *indlev*s in the GPUs in Red in step $i + 1$ are smaller than the *indlev*s in the GPUs in step i and the number of GPUs is finite (Section 3.1.3). Since there are no cycles in $\mathcal{R}$, once a GPU γ is simplified, further simplifications cannot recreate γ again; hence, there is no

[8] In the presence of structures, cycles may occur via fields of structures; Appendix B.4 shows how they are handled.

oscillation across the iterations of fixed-point computation, ensuring the termination of GPU reduction.

- The order in which GPU γ_2 is selected from $\mathcal{R}$ for composition with γ_1 does not matter because the GPUs in $\mathcal{R}$ do not have a data dependence between them.

4.4 Boundary Definitions

Recall that for an indirect assignment ($*p = \&x$ say) as a consumer, GPU reduction typically returns a set of GPUs that define multiple abstract locations, leading to a weak update. Sometimes, however, we may discover that p has a single pointee within the procedure and the assignment defines only one abstract location. In this case, we may, in general, perform a strong update. However, this condition, although necessary, is not sufficient for a strong update because the source of p may not be defined along all paths—there may be a path along which the source of p is not defined within the procedure (i.e., is live on entry to the procedure) and is defined in a caller. 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.

A control flow path $n_1, n_2, \ldots n_k$ in Δ is a definition-free path for source (x, i) if

- no node n_i, $1 \le i \le k$, kills (Definition 5) or blocks (Definition 7) GPUs with source (x, i),
- node n_1 is either Start or has a predecessor that kills or blocks (x, i), and
- node n_k is either End or has a successor that kills or blocks (x, i).

We identify the definition-free paths by introducing *boundary definitions* (explained in the following)

- in RGIn of Start for global variables and formal parameters, and
- in RGOut of the nodes that block some GPUs for the sources of the blocked GPUs.

This ensures the property of completeness of reaching GPUs (Section 4.3) that guarantees that some definition of every source (x, i) reaches every node, thereby enabling strength reduction and distinguishing between strong and weak updates.

The boundary definitions are of the form $x \xrightarrow[0]{i|i} x'$, where x' is a symbolic representation of the initial value of x at the start of the procedure and i ranges from 1 to the maximum depth of the indirection level that depends on the type of x (e.g., for type (int $**$), i ranges from 1 to 2). Variable x' is called the *upwards-exposed* [22] version of x. This is similar to Hoare-logic style specifications in which postconditions use (immutable) *auxiliary variables* x' to denote the original value of variable x (which may have since changed). Our upwards-exposed versions serve a similar purpose; logically on entry to each procedure, the statement $x = x'$ provides a definition of x. The rationale behind the label 0 in the boundary definitions is explained after the following example.

Example 15. Consider a GPB $\delta_{\text{n}} = \{p \xrightarrow[s]{2|0} a\}$ for statement $*p = \&a$. After the introduction of a boundary definition $p \xrightarrow[0]{1|1} p'$, if there is a definition-free path from Start to δ_{n}, then the boundary definition will reach δ_{n}; otherwise, it will not. Then there are three cases to consider:

- *GPUs* $p \xrightarrow[t]{1|0} q$ *and* $p \xrightarrow[0]{1|1} p'$ *reach* δ_{n}.: Then, δ_{n} will be replaced by Red containing both $q \xrightarrow[s]{1|0} a$ and $p' \xrightarrow[s]{2|0} a$. Thus, sources $(q, 1)$ and $(p', 2)$ are included in μ_{n}, causing a weak update (because both of them are defined by the same source).

$$
\begin{aligned}
\mathsf{RGIn}_n &:= \begin{cases} \left\{ x \xrightarrow[0]{i|i} x' \mid x \in L_P, 0 < i \leq \kappa \right\} & n = \text{Start}, \kappa \text{ is the largest } \textit{indlev} \\ \displaystyle\bigcup_{m \,\in\, pred(n)} \mathsf{RGOut}_m & \text{otherwise} \end{cases} \\
\mathsf{RGOut}_n &:= (\mathsf{RGIn}_n - \mathsf{RGKill}_n) \cup \mathsf{RGGen}_n \\
\mathsf{RGGen}_n &:= \mathsf{Gen}\,(\delta_n,\ \mathsf{RGIn}_n) \\
\mathsf{RGKill}_n &:= \mathsf{Kill}\,(\mathsf{RGGen}_n,\ \mathsf{RGIn}_n) \\
\mathsf{Gen}(X, \mathcal{R}) &:= \bigcup_{\gamma\, \in X} \gamma \circ \mathcal{R} \\
\mathsf{Kill}(X, \mathcal{R}) &:= \bigcup_{\gamma \in X,\, |\mathsf{Def}(X, \gamma)| = 1} \mathsf{Match}(\gamma, \mathcal{R}) \\
\mathsf{Match}(x \xrightarrow[s]{i|j} y, \mathcal{R}) &:= \left\{ \gamma \in \mathcal{R} \mid \gamma = w \xrightarrow[t]{k|l} z,\ x = w,\ i = k,\ \neg\mathsf{Upex}(x),\ (x, i) \notin \mu_n \right\} \\
\mathsf{Def}\Big(X, w \xrightarrow[s]{k|l} z\Big) &:= \left\{ (x, i) \mid x \xrightarrow[s]{i|j} y \in X \right\} \\
\mathsf{Upex}(x) &:= \begin{cases} \textit{true} & x \text{ is an upwards-exposed version of some variable} \\ \textit{false} & \text{otherwise} \end{cases}
\end{aligned}
$$

Definition 5. Dataflow equations for reaching GPUs analysis without blocking.

- *Only* $p \xrightarrow[t]{1|0} q$ *reaches* δ_{n}: Then, Red contains only $q \xrightarrow[s]{1|0} a$, causing a strong update because statement s defines a singe source $(q, 1)$.
- *Only* $p \xrightarrow[0]{1|1} p'$ *reaches* δ_{n}: Then, Red contains only $p' \xrightarrow[s]{2|0} a$. Since p' could have multiple pointees in the callers, we perform a weak update.

The boundary definitions are symbolic in that they are never contained in any GPB but are only contained in the set of producer GPUs that reach the GPBs. This allows us to use a synthetic label 0 in them because only the labels of consumer GPUs matter (because they identify a source-language statement); the labels of producer GPUs are irrelevant because they only provide information that is used for simplifying consumer GPUs labeled s into one or more GPUs all labeled s. The boundary definitions participate in GPU reduction algorithm (without requiring any change in GPU composition) like any other producer GPU. After GPU reduction, upwards-exposed versions of variables can appear in simplified GPUs.

4.5 Reaching GPUs Analysis Without Blocking

In this section, we define reaching GPUs analysis ignoring the effect of barriers.

The reaching GPUs analysis is an intraprocedural forward dataflow analysis in the spirit of the classical reaching definitions analysis. Its dataflow equations are presented in Definition 5. They compute set RGIn_n of GPUs reaching a given GPB δ_n by combining the GPUs in RGOut_m of the predecessor GPBs δ_m. RGIn_n is then used to reduce the GPUs in δ_n to compute RGGen_n, which is semantically equivalent to δ_n (except for the effect of blocking) but with the additional property that the *indlev*s of GPUs in RGGen_n do not exceed those of the corresponding GPUs in δ_n.

$\mathsf{RGKill}\ _n$ contains the GPUs that are to be excluded from RGOut_n because of a strong update. A GPU in γ' from RGIn_n is included in RGKill_n if its source (x, i) matches the source of a reduced GPU $\gamma \in \delta|_{\mathrm{s}} \subseteq \mathsf{RGGen}_n$ corresponding to some statement s (identified by $\mathsf{Match}(\gamma, \mathsf{RGIn}_n)$) provided:

(1) All GPUs in $\delta|_{\mathrm{s}}$ define the same source (x, i). Condition $|\mathsf{Def}(X, \gamma)| = 1$ for $\mathsf{Kill}(X, \mathcal{R})$ in Definition 5 ensures this where $\mathsf{Def}(X, \gamma)$ extracts the sources of GPUs of the same statement.
(2) Variable x in (x, i) is not an upwards-exposed version z' because then $i > 1$ and z' could point to multiple pointees in the caller (note that an upwards-exposed version can appear in the source of a reduced GPU only if the GPU represents an indirect assignment).
(3) Source (x, i) is not in μ_{n}.

If any of these conditions is violated, then γ' is excluded from RGKill_n, leading to weak update. Note that the GPUs that are killed are determined by the GPUs in RGGen_n and not those in δ_n.

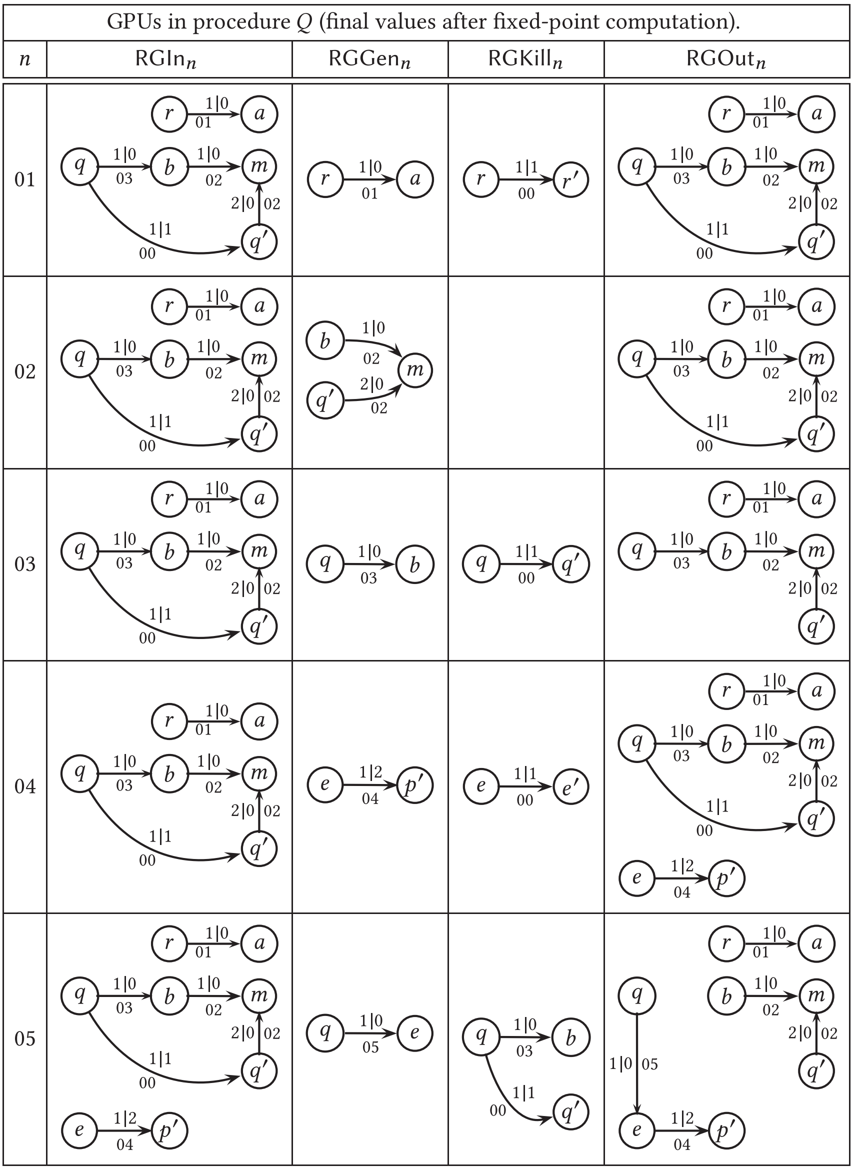
Example 16. Figure 8 gives the final result of reaching GPUs analysis for procedure Q of our motivating example. We have shown the boundary GPU $q \xrightarrow[00]{1|1} q'$ for q. Other boundary GPUs are not required for strong updates in this example and have been omitted. This result has been used to construct GPG Δ_Q shown in Figure 4. For procedure R, 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 e is being defined by the assignment and source $(e, 1)$ is not may-defined (i.e. not in μ_{10}), this is a 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 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, $q \xrightarrow[10]{2|0} o$).

4.6 Reaching GPUs Analysis with Blocking

This section extends the reaching GPUs analysis to incorporate the effect of blocking by defining a dataflow analysis that computes $\overline{\mathsf{RGIn}}$ and $\overline{\mathsf{RGOut}}$ for the purpose.

4.6.1 The Need of Blocking. Consider the possibility of the composition of a consumer GPU $\boldsymbol{c}$ that appears to have a RaW dependence on a producer GPU $\boldsymbol{p}$ because they have a pivot but there is a barrier GPU $\boldsymbol{b}$ (Sections 2.2 and 4.1) between the two such that $\boldsymbol{b}$ has a potential WaW dependence on $\boldsymbol{p}$. This possible if the *indlev* of the source of $\boldsymbol{b}$ or $\boldsymbol{p}$ is greater than 1. We call such a GPU an *indirect* GPU. The execution of $\boldsymbol{b}$ may alter the apparent dependence between $\boldsymbol{c}$ and $\boldsymbol{p}$, and hence the composition of $\boldsymbol{c}$ with $\boldsymbol{p}$ may be unsound.

Since this potential dependence between $\boldsymbol{p}$ and $\boldsymbol{b}$ cannot be resolved without the alias information in the calling context, we *block* such producer GPUs so that such GPU compositions leading to potentially unsound strength reduction optimization are *postponed*. Wherever possible, we use the type information to rule out some GPUs as barriers. After inlining the GPG in a caller, more information may become available. Thus, it may resolve potential data dependence of a barrier with a producer. Then, if a consumer still has a RaW dependence on a producer, the composition that was earlier postponed may now safely be performed.


GPUs in procedure Q (final values after fixed-point computation).
n
RGIn$_n$
RGGen$_n$
RGKill$_n$
RGOut$_n$
01
02
03
04
05

We show only one boundary definition $q \xrightarrow[00]{1|1} q'$ in RGIn and RGOut because other boundary definitions do not participate in GPU reduction for this example.

However, the boundary definitions that are removed are shown in RGKill.

Fig. 8. The dataflow information computed by reaching GPUs analysis for procedure Q of Figure 2.

Example 17. Consider the procedure in Figure 9(a). The composition of the GPUs for statements 02 and 04 is *admissible*. However, statement 03 may cause a side effect by indirectly defining y (if x points to y in the calling context). Thus, q in statement 04 would point to b if x points to y; otherwise, it would point to a. If we replace the GPU $q \xrightarrow[04]{1|1} y$ by $q \xrightarrow[04]{1|0} a$ (which is the result of composing $q \xrightarrow[04]{1|1} y$ with $y \xrightarrow[02]{1|0} a$), then we would miss the GPU $q \xrightarrow[04]{1|0} y$ if x points to y in the calling context—leading to unsoundness. Since the calling context is not available during

```
    int a, b, *y, *q, **x;
01  void P()
02  {  y = &a;      /* GPU p */
03     *x = &b;     /* GPU b */
04     q = y;       /* GPU c */
05  }
```

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

(a) Barrier is in indirect GPU, producer is not

```
    int a, b, *y, *q, **x;
01  void P()
02  {  *x = &a;     /* GPU p */
03     y = &b;      /* GPU b */
04     q = *x;      /* GPU c */
05  }
```

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

(b) Producer is an indirect GPU, barrier is not

Fig. 9. Risk of unsoundness in GPU reduction caused by a barrier GPU.

GPG construction and optimization, we postpone this composition to eliminate the possibility of unsoundness. Reaching GPUs analysis with blocking blocks the GPU $y \xrightarrow[02]{1|0} a$ by a barrier $x \xrightarrow[03]{2|0} b$. This corresponds to the first case described earlier.

For the second case, consider statement 02 of the procedure in Figure 9(b), which may indirectly define y (if x points to y). Statement 03 directly defines y. Thus, q in statement 04 would point to b if x points to y; 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 $\boldsymbol{p}$ (here, the GPU $y \xrightarrow[03]{1|0} b$ acts as a barrier).

Consider a GPU $\boldsymbol{p}$ originally blocked by a barrier $\boldsymbol{b}$. After inlining the GPG in its callers and performing reductions 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 eliminating the potential dependence. In this case, $\boldsymbol{b}$ ceases being a barrier and thus no longer blocks $\boldsymbol{p}$, leading to the following two situations:
   (a) $\boldsymbol{b}$ has a WaW dependence on $\boldsymbol{p}$ and therefore redefines the pointer defined by $\boldsymbol{p}$, killing $\boldsymbol{p}$, thereby obviating the composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$.
   (b) $\boldsymbol{b}$ does not have a dependence on $\boldsymbol{p}$, thereby allowing the composition $\boldsymbol{c} \circ^{\tau} \boldsymbol{p}$.

(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.

Example 18. The preceding Case 1(a) could arise if x points to p in the calling context of the procedure in Figure 9(a). As a result, GPU $y \xrightarrow[02]{1|0} a$ is killed by the barrier GPU $y \xrightarrow[03]{1|0} b$ (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 y in the calling context. In this case, the composition between $q \xrightarrow[04]{1|1} y$ and $y \xrightarrow[02]{1|0} a$ is sound, and q points to a for statement 04. Case 2 could arise if the 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 $y \xrightarrow[02]{1|0} a$.

Our measurements (Section 10) show that situation 1(a) rarely arises in practice because it amounts to defining the same pointer multiple times through different aliases in the same context.

$$\text{Blocked}\,(I,G) := \begin{cases} \emptyset & G = \emptyset & \text{(Case 1)} \\ \{\gamma \in I \mid \overline{\text{DDep}}(\text{IndGPUs}(G), \{\gamma\})\} & |\text{IndGPUs}(G)| > 1 & \text{(Case 2)} \\ \{\gamma \in \text{IndGPUs}(I) \mid \overline{\text{DDep}}(G, \{\gamma\})\} & \text{otherwise} & \text{(Case 3)} \end{cases}$$

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

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

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

$$\text{TRef}(X) := \left\{\text{typeof}(x, k) \mid 1 \le k < i, x \xrightarrow[s]{i|j} y \in X\right\} \cup \left\{\text{typeof}(y, k) \mid 1 \le k < j, x \xrightarrow[s]{i|j} y \in X\right\}$$

Here typeof(x, i) gives the type of the i^{th} pointee of x. For example, given a declaration of x such as '`int **x`', typeof$(x, 1)$ is '`int **`' and typeof$(x, 2)$ is '`int *`'. Note that typeof$(x, 0)$ is not a pointer and typeof$(x, 3)$ is undefined because x cannot be dereferenced thrice.

Definition 6. Blocking.

$$\overline{\text{RGIn}}_n := \begin{cases} \left\{x \xrightarrow[0]{i|i} x' \mid x \in L_P, 0 < i \le \kappa\right\} & n = \text{Start}, \kappa \text{ is the largest } \textit{indlev} \\ \bigcup\limits_{m \,\in\, pred(n)} \overline{\text{RGOut}}_m & \text{otherwise} \end{cases}$$

$$\overline{\text{RGOut}}_n := \left(\overline{\text{RGIn}}_n - \overline{\text{RGKill}}_n\right) \cup \overline{\text{RGGen}}_n \cup \{x \xrightarrow[0]{i|i} x' \mid x \xrightarrow[s]{i|j} y \in \text{Blocked}(\overline{\text{RGIn}}_n, \overline{\text{RGGen}}_n)\}$$

$$\overline{\text{RGGen}}_n := \text{Gen}\left(\delta_n,\ \overline{\text{RGIn}}_n\right)$$

$$\overline{\text{RGKill}}_n := \text{Kill}\left(\overline{\text{RGGen}}_n,\ \overline{\text{RGIn}}_n\right) \cup \text{Blocked}(\overline{\text{RGIn}}_n, \overline{\text{RGGen}}_n)$$

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

Definition 7. Dataflow equations for reaching GPUs analysis with blocking.

Example 19. To see how reaching GPUs analysis with blocking helps, consider the example in Figure 9(b). The set of GPUs reaching the statement 04 is $\text{RGIn}_{04} = \{x \xrightarrow[02]{2|0} a, y \xrightarrow[03]{1|0} b\}$. The GPU $x \xrightarrow[02]{2|0} a$ is blocked by the barrier GPU $y \xrightarrow[03]{1|0} b$, and hence $\overline{\text{RGIn}}_{04} = \{y \xrightarrow[03]{1|0} b\}$. Thus, GPU reduction for $\gamma_1 : q \xrightarrow[04]{1|2} x$ (in the context of $\overline{\text{RGIn}}_{04}$) computes Red as $\{\gamma_1\}$ because γ_1 cannot be reduced further within the GPG of the procedure. However, γ_1 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 γ_1 with $\boldsymbol{p} : x \xrightarrow[02]{2|0} a$ until $\boldsymbol{p}$ has been simplified.

4.6.2 Dataflow Equations for Computing $\overline{\text{RGIn}}$ *and* $\overline{\text{RGOut}}$. The following GPUs should be blocked as barriers:

- If $\overline{\text{RGGen}}_n$ contains a GPU $\boldsymbol{b}$, then all GPUs reaching δ_n that share a data dependence with $\boldsymbol{b}$ should be blocked regardless of the nature of other GPUs (if any) in $\overline{\text{RGGen}}_n$.

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

Additionally, we use the type information to minimize blocking. We define a predicate $\overline{\text{DDep}}(B, I)$ to check the presence of data dependence between the sets of GPUs B and I (Definition 6). When the types of $\boldsymbol{b} \in B$ and $\boldsymbol{p} \in I$ match, we assume the possibility of data dependence and hence $\boldsymbol{b}$ blocks $\boldsymbol{p}$. $\text{TDef}(B)$ is the set of types of locations being written by a barrier, whereas $(\text{TDef}(I) \cup \text{TRef}(I))$ represents the set of types of locations defined or read by the GPUs in I, thereby checking for a potential WaW and WaR dependence of the GPUs in B on those of I.

The dataflow equations in Definition 7 differ from those in Definition 5 as follows:

- $\overline{\text{RGKill}}_n$ additionally includes blocked GPUs computed using function $\text{Blocked}(I, G)$. The latter examines the GPUs in $\overline{\text{RGGen}}_n$ (argument "G" for generated) to identify the GPUs in $\overline{\text{RGIn}}_n$ (argument "I" for incoming) that should be blocked using three cases that exhaust all possibilities:
  —Case 1 corresponds to not blocking any GPU because $\overline{\text{RGGen}}_n$ is empty.
  —Case 2 corresponds to blocking some GPUs because $\overline{\text{RGGen}}_n$ contains an indirect GPU.
  —Case 3 corresponds to blocking indirect GPUs because $\overline{\text{RGGen}}_n$ does not contain an indirect GPU and is not $\emptyset$.
- $\overline{\text{RGOut}}_n$ explicitly introduces boundary definitions for the GPUs that are blocked. This is essential for ensuring the completeness of the set of reaching GPUs (Section 4.3) that is necessary for replacing δ_{n} by $\overline{\text{RGGen}}_n$. This is possible because of the following property of upwards-exposed versions: any read of x anywhere in the procedure can be replaced by x' without affecting soundness or precision because there is no GPU with (x', i) as its source. Hence, a statement $*x' = \&a$ does not have a RaW dependence on any statement, and the occurrences of x' cannot be simplified any further. After inlining in the caller, x' is replaced by x and hence the statement reverts to its original form.

Example 20. For the procedure in Figure 9(b), $\overline{\text{RGIn}}_{02} = \emptyset$ and $\overline{\text{RGGen}}_{02}$ is $\{x \xrightarrow[02]{2|0} a\}$. Although $\overline{\text{RGGen}}_{02}$ contains an indirect GPU, since no GPUs reach 02 (because it is the first statement), $\overline{\text{RGOut}}_{02}$ is $\{x \xrightarrow[02]{2|0} a\}$, indicating that no GPUs are blocked.

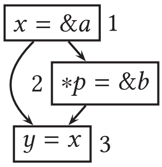
For statement 03, $\overline{\text{RGIn}}_{03} = \{x \xrightarrow[02]{2|0} a\}$ and $\overline{\text{RGGen}}_{03} = \{y \xrightarrow[03]{1|0} b\}$. $\overline{\text{RGGen}}_{03}$ is non-empty and does not contain an indirect GPU, and thus $\overline{\text{RGOut}}_{03} = \{y \xrightarrow[03]{1|0} b\}$ according to the third case in the Blocked equation in Definition 7, 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 $\overline{\text{RGIn}}_{03}$ is excluded from $\overline{\text{RGOut}}_{03}$. Note that the indirect GPU $x \xrightarrow[02]{2|0} a$ is blocked by the GPU $y \xrightarrow[03]{1|0} b$ because $\text{typeof}(x, 2)$ matches with $\text{typeof}(p, 1)$, indicating a possibility of WaW dependence.

For statement 04, $\overline{\text{RGIn}}_{04} = \{y \xrightarrow[03]{1|0} b\}$ and $\overline{\text{RGGen}}_{04}$ 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, $\overline{\text{RGGen}}_{04}$ does not contain an indirect GPU and $\overline{\text{RGOut}}_{04} = \{y \xrightarrow[03]{1|0} b, q \xrightarrow[04]{1|2} x\}$.

In Figure 9(a), the GPU $y \xrightarrow[02]{1|0} a$ is blocked by the barrier GPU $x \xrightarrow[03]{2|0} b$ because typeof$(y, 1)$ matches with typeof$(x, 2)$. Hence, the composition $(q \xrightarrow[04]{1|1} y \circ^{\text{ts}} y \xrightarrow[02]{1|0} a)$ is postponed.

In the GPG of procedure Q (of our motivating example) shown in Figure 4, 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 Q).

Example 21 shows the role of boundary definitions in ensuring completeness of reaching GPUs.



Example 21. Let the GPUs of the statements 1, 2, and 3 on the right be denoted by γ_1, γ_2, and γ_3, respectively. Then, γ_1 is blocked by γ_2 because of potential RaW dependence (if p points-to x in the caller). Thus, $\overline{\text{RGIn}_3} = \{\gamma_1, \gamma_2\}$. Then, the $\overline{\text{RGGen}_3} = \{y \xrightarrow[3]{1|0} a\}$. Replacing δ_3 by $\overline{\text{RGGen}_3}$ is unsound because if p points to x in the caller, then y should also point to b. The problem arises because $\overline{\text{RGIn}_3}$ does not have the source $(x, 1)$ defined along the path 1-2-3 because of blocking in node 2. This violates the completeness of $\overline{\text{RGIn}_3}$. Explicitly adding the boundary definition $x \xrightarrow[0]{1|1} x'$ in 2 ensures that $(x, 1)$ is defined along both the paths, leading to $\overline{\text{RGGen}_3} = \{y \xrightarrow[3]{1|0} a, y \xrightarrow[3]{1|1} x'\}$. When the resulting GPG is inlined in the caller, x' is replaced by x and the original consumer GPU $\boldsymbol{c}$ representing $y = x$ is recovered. Thus, if p points to x, then after node 3, y points to both a and b. However, if p does not point to x, then after node 3, y points to a as expected.

5 DEAD GPU ELIMINATION

For each node n, dead GPU elimination removes redundant GPUs—that is, those $\gamma \in \delta_n$ that are killed along every control flow path from n to the End node of the procedure. However, two kinds of GPUs should not be removed even if they do not reach the End node: GPUs that are blocked, or GPUs that are producer GPUs for compositions that have been postponed (Section 4.2.2).

For the first requirement, we check that a GPU considered for dead GPU elimination does not belong to $\text{RGOut}_{\text{End}}$ (the result of reaching GPUs analysis without blocking). However, this analysis also performs the compositions that should be blocked and hence may not contain the non-reduced forms of some GPUs that then may be considered for dead GPU elimination. We exclude such GPUs placing an additional condition that the GPUs considered for dead GPU elimination should not belong to $\overline{\text{RGOut}_{\text{End}}}$ (the result of reaching GPUs analysis with blocking, see Example 23). For the second requirement, we check that the GPU is not a producer GPU for a postponed composition. During the computation of RGOut, GPU reduction records such GPUs in the set Queued (Appendix A augments the GPU reduction for this). Thus, dead GPU elimination removes a GPU $\gamma \in \delta_n$ if $\gamma \notin (\text{RGOut}_{\text{End}} \cup \overline{\text{RGOut}_{\text{End}}} \cup \text{Queued})$.

Example 22. In procedure Q of Figure 4, 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 R (Figure 5) are eliminated from their corresponding GPBs.

Example 23. For the procedure in Figure 9(a), the GPU $y \xrightarrow[02]{1|0} a$ is blocked by the barrier $x \xrightarrow[03]{2|0} b$; hence, it is present in RGOut_{05} but not in $\overline{\text{RGOut}}_{05}$ (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 $y \xrightarrow[02]{1|0} a$). Thus, it is not removed by dead GPU elimination.

To see the need of $\overline{\text{RGOut}}_{\text{End}}$, observe that $q \xrightarrow[04]{1|1} y$ is reduced to $q \xrightarrow[04]{1|0} a$ in $\text{RGOut}_{\text{End}}$. Hence, $q \xrightarrow[04]{1|1} y$ is not contained in $\text{RGOut}_{\text{End}}$. However, it cannot be removed as dead code. It is contained in $\overline{\text{RGOut}}_{\text{End}}$, which should be additionally used for determining which GPUs are dead.

6 CONTROL FLOW MINIMIZATION

We minimize control flow by *empty GPB elimination* and *coalescing of GPBs*. They improve the compactness of a GPG and reduce the repeated re-analysis of GPBs after inlining. Empty GPBs are eliminated by connecting their predecessors to their successors.

Example 24. In the GPG of procedure Q of Figure 4, 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 R in Figure 5.

In the rest of this section, we explain coalescing of GPBs.

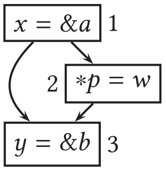
6.1 The Motivation Behind Coalescing

After strength reduction and dead GPU elimination, we *coalesce* multiple GPBs into a single GPB whenever possible to reduce the size of GPGs (in terms of control flow information). It relies on the elimination of data dependence by strength reduction and dead GPU elimination. This turns out to be the core idea for making GPGs a scalable technique for points-to analysis.

Strength reduction exploits and removes all definite RaW dependences, whereas dead GPU elimination removes all definite WaW dependences that are strict (Section 3.1.2). Only the potential dependences, definite WaR dependences, and definite non-strict WaW dependences remain. Recall that WaR dependences are preserved by GPBs; as we shall see in this section, definite non-strict WaW dependences are also preserved by coalesced GPBs. This make much of the control flow redundant.

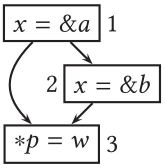
For a control flow edge $\delta_{n_1} \rightarrow \delta_{n_2}$, the decision to coalesce GPBs δ_{n_1} and δ_{n_2} is influenced not only by the dependence between the GPUs of δ_{n_1} and δ_{n_2} but also by the dependence of the GPUs of δ_{n_1} and δ_{n_2} with the GPUs in some other GPB as illustrated in the following example.

Example 25. Let the GPUs of the statements 1, 2, and 3 on the right be denoted by γ_1, γ_2, and γ_3, respectively. Then, γ_1 cannot be coalesced with γ_2 because of potential WaW dependence (if p points-to x in the caller). Similarly, γ_2 cannot be coalesced with γ_3 because of potential WaW dependence (if p points to y in the caller). There is no data dependence between γ_1 and γ_3. However, they cannot be coalesced together because doing so will create GPBs $\delta = \{\gamma_1, \gamma_3\}$ and $\delta' = \{\gamma_2\}$ with control flow edges $\delta \rightarrow \delta'$ and $\delta' \rightarrow \delta$, leading to spurious potential data dependences.



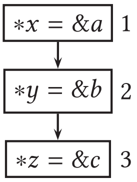
The next example illustrate that a non-strict WaW dependence does not constrain coalescing.

Example 26. Let the GPUs of the statements 1, 2, and 3 on the right be denoted by γ_1, γ_2, and γ_3, respectively. The WaW dependence between γ_1 and γ_2 is definite but not strict and is not removed by dead GPU elimination because γ_1 is not killed along the path 1,3. Thus, both γ_1 and γ_2 reach statement 3. Hence, although there is a WaW dependence between γ_1 and γ_2, they can be coalesced because the semantics of GPB allows both of them to the executed in parallel without any data dependence between them. This enables both of them to reach statement 3.



There is no "best" coalescing operation: given three sequenced GPUs γ_1, γ_2 γ_3, then γ_1 may coalesce with γ_2 and separately γ_2 may coalesce with γ_3, but the GPUs $\gamma_1, \gamma_2, \gamma_3$ do not all coalesce.

Example 27. Let the type of pointers x and z be "*int* $*$" and that of y be "*float* $*$." Then there is no data dependence between γ_1 and γ_2 because x and y are guaranteed to point to different locations based on types. Similarly, there is no data dependence between γ_2 and γ_3. However, there is a potential data dependence between γ_1 and γ_3. Thus, γ_1 and γ_2 can be coalesced and so can γ_2 and γ_3; however, all three of them cannot be coalesced.



Therefore, we formulate the coalescing operation on a GPG as a *partition* Π on its nodes (Section 6.2), set out the correctness conditions the partition must satisfy (Section 6.3), and describe how we select one of the maximally coalescing partitions satisfying the conditions (Section 6.4).

6.2 Creating a Coalesced GPG from a Partition

Recall that a partition Π of a set S is a collection of the non-empty subsets of S such that every element of S is a member of exactly one element of Π. We call the elements of Π *parts* and write $\Pi(x)$ for the part containing x. A partition induces an equivalence relation on S; thus, for example, $x \in \Pi(y)$ holds if and only if $y \in \Pi(x)$.

Following a practice common for CFGs, we have previously conflated the idea of a node n of a GPG with that of its GPB δ_{n} which is a set of GPUs. It is helpful to keep these separated when defining a partition, noting that, under coalescing, GPBs remain sets of GPUs while the definition of a node is changed.

Given a GPG Δ and a partition Π on its nodes, we obtain a *coalesced GPG*, written Δ/Π, in the following steps:

$$
\begin{aligned}
\mu_{\hat{n}} &:= \text{preservedSources}(\hat{n}) \cap \text{definedSources}(\hat{n}) \\
\text{preservedSources}(\hat{n}) &:= \left\{(x, i) \mid x \xrightarrow[s]{i|j} y \in \text{preservedGPUs}(\hat{n})\right\} \\
\text{preservedGPUs}(\hat{n}) &:= (\text{inGPUs}(\hat{n}) \cap \text{outGPUs}(\hat{n})) \; - \; \delta_{\hat{n}} \\
\text{inGPUs}(\hat{n}) &:= \bigcup_{n \in entry(\hat{n})} \overline{\text{RGIn}}_n \\
\text{outGPUs}(\hat{n}) &:= \bigcup_{n \in exit(\hat{n})} \overline{\text{RGOut}}_n \\
\text{definedSources}(\hat{n}) &:= \left\{(x, i) \mid x \xrightarrow[s]{i|j} y \in \delta_{\hat{n}}\right\}
\end{aligned}
$$

Definition 8. Computing the may-definition sets for the nodes in Δ/Π during coalescing.

(1) The nodes of Δ/Π (written $\hat{n}$) are sets of the nodes of Δ. More precisely, they are the members of Π and represent its equivalence classes.
   The *entry* and *exit* nodes of a part $\hat{n}$ are defined as follows:

$$
\begin{aligned}
entry(\hat{n}) &= \{n \in \hat{n} \mid \exists\, n' \in pred(n), n' \notin \hat{n}\} \\
exit(\hat{n}) &= \{n \in \hat{n} \mid \exists\, n' \in succ(n), n' \notin \hat{n}\}\,.
\end{aligned}
$$

(2) Δ/Π has an edge $\hat{n}_1 \longrightarrow \hat{n}_2$ if $\hat{n}_1 \neq \hat{n}_2$ and $\exists n_1 \in \hat{n}_1, n_2 \in \hat{n}_2$ such that $n_1 \longrightarrow n_2$ is an edge in Δ.

(3) The Start and End nodes of Δ/Π respectively are the parts containing the Start and End nodes of Δ.

(4) The GPB $\delta_{\hat{n}}$ of each node $\hat{n}$ (which represents the part $\Pi(n)$ for some n) is the union of the GPBs corresponding to the nodes in $\hat{n}$ (i.e., $\delta_{\hat{n}} = \bigcup_{n \in \hat{n}} \delta_{\text{n}}$).

(5) The may-definition set $\mu_{\hat{n}}$ of each node is computed using Definition 8 as follows:
- We identify the GPUs that are not modified in $\hat{n}$ by finding out the GPUs that reach the entry nodes of $\hat{n}$ (represented by the set inGPUs($\hat{n}$)) and the exit nodes of $\hat{n}$ (represented by the set outGPUs($\hat{n}$)) but are not generated within $\hat{n}$ (set $\delta_{\hat{n}}$).
- Set $\mu_{\hat{n}}$ contains the sources of the preceding GPUs (represented by the set preservedSources($\hat{n}$)) that are also defined within the GPB $\hat{n}$ (represented by the set definedSources($\hat{n}$)).
  In essence, we compute $\mu_{\hat{n}}$ in terms of μ_{n} ($n \in \hat{n}$).

This is the natural definition of quotient of a labeled graph, save that self-edges are removed as they serve no purpose. Due to strength reduction, a self-loop cannot represent a control flow edge with an unresolved data dependence between the GPUs across it. There are two possibilities for a self-loop: it exists in the original program or could result from empty GPB elimination and coalescing. In the former case, strength reduction, based on the fixed point of reaching GPUs analysis, ensures that the data dependence along the self-loop is eliminated (there is no blocking as the GPUs reached along the self-loop belong to an immediate successor). In the latter case, the reduction of a loop to a self-loop indicates that there are no indirect GPUs in the loop and hence no blocking. Thus, the data dependences in the loop are eliminated through strength reduction.

Observe that for every path in Δ, there is a corresponding path in Δ/Π. In the degenerate case, this path could well be a single node $\hat{n}$ if all nodes along a path are coalesced into the same part.

After finding a suitable partition, we revert to our previous abuse of notation and once again conflated nodes with their GPBs representing the sets of GPUs.

6.3 What Is a Valid Coalescing Partition?

A partition Π is valid for coalescing to construct Δ/Π if it preserves the semantic understandings of Δ. Validity is characterized by a set of conditions that ensures the following:

- *Soundness*: Every GPU that reaches the End GPB of Δ also reaches the End GPB of Δ/Π. This must hold for both variants of reaching GPUs analysis (Sections 4.5 and 4.6) and also for the GPUs representing the boundary definitions (Section 4.4).
- *Precision*: No GPU that does not reach the End GPB of Δ should reach the End GPB of Δ/Π. This must hold for both variants of reaching GPUs analysis and also for the GPUs representing boundary definitions.

Assuming that dead GPU elimination and empty GPB elimination have been performed before coalescing, the validity of a coalescing partition is formalized as the following sufficient conditions:

- *Soundness*:
  (S1) If there is a control flow path from n_1 to n_2 ($n_2 \in succ^+(n_1)$) and n_1 and n_2 are coalesced ($n_2 \in \Pi(n_1)$), then we require, for all GPUs $\gamma_1 \in \delta_{n_1}$ and $\gamma_2 \in \delta_{n_2}$, that γ_1 and γ_2 have no potential RaW dependence between them.
  (S2) Consider a definition-free path $\rho\colon n_1, n_2, \ldots n_k$ for source (x, i) in Δ. Then, Δ/Π must have a corresponding definition-free path $\hat{\rho}\colon \hat{n}_1, \hat{n}_2, \ldots \hat{n}_l$ such that
  — If n_1 is Start, then $\hat{n}_1 = \Pi(n_1)$; otherwise, $\hat{n}_1 = \Pi(n_{j+1})$ such that first j nodes in ρ belong to $\Pi(n_0)$ where $n_0 \in pred(n_1)$ and $(x, i) \notin \mu_{n_0)}$.
  — If n_k is End, then $\hat{n}_l = \Pi(n_k)$; otherwise, $\hat{n}_l = \Pi(n_{l-1})$ such that last l nodes in ρ belong to $\Pi(n_{k+1})$ where $n_{k+1} \in succ(n_k)$ and $(x, i) \notin \mu_{n_{k+1}}$.
- *Precision*:
  (P1) If there is a control flow path from n_1 to n_2 ($n_2 \in succ^+(n_1)$) and n_1 and n_2 are coalesced ($n_2 \in \Pi(n_1)$), then we require, for all GPUs $\gamma_1 \in \delta_{n_1}$ and $\gamma_2 \in \delta_{n_2}$, that γ_1 and γ_2 have no potential strict WaW dependence between them. Observe that definite strict WaW dependences have already been eliminated by dead GPU elimination.
  (P2) For every $\hat{n}$ in Δ/Π that may-defines source (x, i), there must be a control flow path $n_1, n_2, \ldots, n_{k-1}, n_k$ in Δ such that
  — n_1 belongs to a predecessor of $\hat{n}$, n_k belongs to a successor of $\hat{n}$, and
  — all nodes from n_2 to n_{k-1} belong to $\hat{n}$ and may-defines source (x, i).
  (P3) For every control flow edge $\hat{n}_1 \rightarrow \hat{n}_2$ in Δ/Π, Δ must have a control flow path from every $n_1 \in \hat{n}_1$ to every $n_2 \in \hat{n}_2$.

Condition (S1) ensures that no RaW dependence is missed in Δ/Π; condition (S2) ensures that no strict WaW dependence is spuriously included in Δ/Π. Together, they ensure that every GPU reaching the End node in Δ also reaches the End node of Δ/Π.

Conditions (P1) and (P2) ensure that killing is not underapproximated in Δ/Π by converting a strict WaW dependence into a non-strict dependence. Although definite strict WaW dependences with GPUs have been removed, we could still have a potential strict WaW dependence between GPUs or a definite strict WaW dependence with a boundary definition. Condition (P3) ensures that no spurious RaW dependence is included in Δ/Π. Together, they ensure that no GPU that does not reach the End node in Δ reaches the End node of Δ/Π.

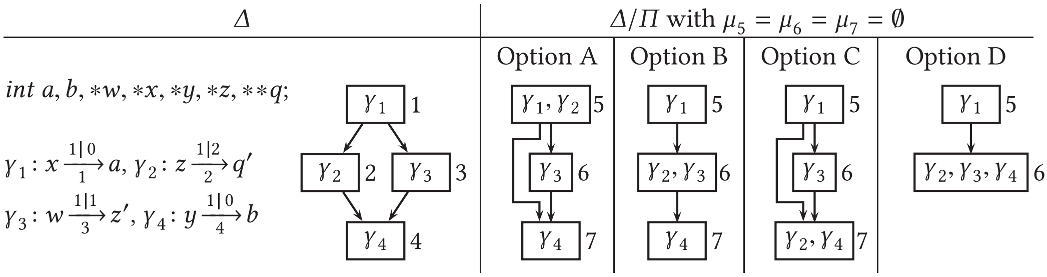

Δ
Δ/Π with $\mu_5 = \mu_6 = \mu_7 = \emptyset$
Option A
Option B
Option C
Option D
$int\ a, b, *w, *x, *y, *z, **q;$
$\gamma_1 : x \xrightarrow[1]{1|0} a,\ \gamma_2 : z \xrightarrow[2]{1|2} q'$
$\gamma_3 : w \xrightarrow[3]{1|1} z',\ \gamma_4 : y \xrightarrow[4]{1|0} b$
γ_1 1
γ_2 2
γ_3 3
γ_4 4
γ_1, γ_2 5
γ_3 6
γ_4 7
γ_1 5
γ_2, γ_3 6
γ_4 7
γ_1 5
γ_3 6
γ_2, γ_4 7
γ_1 5
$\gamma_2, \gamma_3, \gamma_4$ 6

Fig. 10. Illustrating soundness and precision of coalescing (Start and End nodes are not shown).

Note that coalescing only forbids nodes that have a potential RaW or WaW dependence from being coalesced if there is a control flow path between them; coalescing in the absence of data dependence (or in the presence of definite non-strict WaW dependence) is generally allowed.

Example 28. Consider the GPG in Figure 10 for coalescing and proposed partitions assuming that the may-definition sets are $\emptyset$. GPU γ_2 has a potential RaW dependence on γ_1. Option A violates both soundness and precision, whereas options B and D violate soundness, and only option C satisfies all conditions:

- Option A violates condition (S1) by coalescing nodes 1 and 2—if q points to x in the caller, the RaW dependence of γ_2 on γ_1 is missed. It also violates condition (P3) by creating edge 2 → 3, which creates a RaW dependence of γ_3 on γ_2 after inlining in the caller (because z' will be replaced by z). All other conditions are satisfied.
- Options B and D satisfy all conditions except (S2) because the definition-free paths for $(w, 1)$ and $(z, 1)$ are missed.

Two important characteristics of these conditions are the following:

- They are sufficiency conditions in that they are stronger than actual requirements—it is possible to create examples of Δ and Π such that Δ/Π do not satisfy these requirements and yet no data dependence is violated nor is a new data dependence created.
- They characterize the soundness and precision of coalescing but not its efficiency. Consider a trivial partitioning such that $\Pi(n) = \{n\}$ (i.e., every node is placed in a separate part). This partitioning is sound and precise but not efficient. However, there may be no potential data dependence between any of the GPUs of Δ; then, all nodes may be placed in a single part. Our empirical results show that a large number of GPGs nearly fall in this category, giving us the scalability.

Example 29. Consider the statement sequence $x = \&a;\ if(c)\ *y = \&b;$ in which there is a potential RaW dependence between the two pointers assignments (because x could point to y in a caller). This violates condition (S1), and yet coalescing these statements does not violate soundness or precision because no pointer-pointee association is missed, nor is a spurious association created by coalescing.

6.4 Honoring the Validity Conditions

This section describes how we ensure that the conditions of validity of partitioning are satisfied.

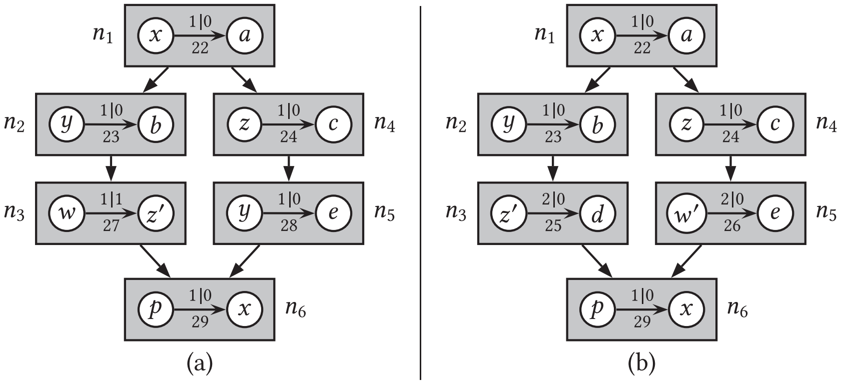


Fig. 11. Illustrating data dependence check and coherence in coalescing.

6.4.1 Ensuring Soundness. We honor the conditions for soundness in the following manner:

(1) Given a part $\Pi(n)$, a node $n_1 \notin \Pi(n)$ is considered for inclusion in $\Pi(n)$ only if some predecessor or some successor of n_1 is in $\Pi(n)$. This ensures that every part $\Pi(n)$ is a connected subgraph.[9]
(2) Node n_1 is included in $\Pi(n)$ only if there is no potential RaW or WaW dependences[10] between the GPUs n_1 and those of any node $n_2 \in \Pi(n) \cap pred^+(n_1)$.
(3) Definition-free paths are preserved by maintaining may-definition sets, with $\mu_{\hat{n}}$ containing the sources that are may-defined in $\hat{n}$.

Example 30. This example illustrates why the preceding step (2) only considers the dependence between n_1 and $n_2 \in (\Pi(n) \cap pred^+(n_1))$ rather than between n_1 and $n_2 \in \Pi(n)$. In Figure 11(a), nodes n_1, n_2, and n_4 can be included in the same part. Consider node n_3 for inclusion in this part: the GPU in n_3 appears to have RaW dependence with the GPU in node n_4 because variable z' will be replaced by z after inlining z'. However, there is no control flow from n_4 to n_3. Hence, the data dependence of the GPU in n_3 need only be checked with those in n_1 and n_2 and not with those in n_4. Thus, n_3 can also be included in the same part. Similarly, although it appears that there is a WaW dependence between n_2 and n_5, the latter can also be included in the same part.

6.4.2 Ensuring Precision. Define the *external predecessors* and *successors* of *entry* and *exit* nodes of a part $\hat{n}$ as follows:

$$xpred(n) = pred(n) - \Pi(n)$$
$$xsucc(n) = succ(n) - \Pi(n).$$

We now wish to demand that whenever n_1 and n_2 are entries of $\hat{n}$, they have the same set of external predecessors, and similarly for their exits and their external successors. Thus, we define

[9] Note that a part could be singleton too.
[10] We use a tighter condition and prohibit all potential WaW dependences and not just strict potential WaW dependences to avoid computing postdominance information after inlining calls within a procedure.

a partition Π to be *coherent* if for all nodes $\hat{n} \in \Delta/\Pi$, we have

$$\begin{aligned} n_1, n_2 \in \mathit{entry}(\hat{n}) &\Rightarrow \mathit{xpred}(n_1) = \mathit{xpred}(n_2) \wedge \\ n_1, n_2 \in \mathit{exit}(\hat{n}) &\Rightarrow \mathit{xsucc}(n_1) = \mathit{xsucc}(n_2). \end{aligned}$$

The identity partition, consisting entirely of single-entry and single-exit parts, is trivially coherent. Coherence guarantees that no "cross connection" results by merging all entries together (or by merging all exits together) in each node in Δ/Π. In other words, no spurious control flow is added, thereby ensuring precision. In addition, it allows combining GPUs reaching the entries and the GPUs reaching the exits of a part to compute the may-defined sources (Section 6.2).

Example 31. To see the role of coherence in precision, consider the GPG in Figure 11(b). Nodes n_1, n_2, and n_4 can be considered for inclusion in the same part. Nodes n_3 and n_5 have potential dependences with any other GPU. Assuming that the types rule out the possibility of potential dependences, the part $\{n_1, n_2, n_4\}$ violates coherence because it has two exits (n_2 and n_4) that have different external successors. If we form Δ/Π, we will have control flow from the GPU of n_4 to the GPU of n_3, creating a spurious RaW dependence between them because of variable z (the upwards-exposed version z' will be replaced by z after inlining). Some examples of coherent partitions are $\Pi_1 = \{\{n_1\}, \{n_2, n_3\}, \{n_4, n_5\}, \{n_6\}\}$, $\Pi_2 = \{\{n_1, n_2, n_3\}, \{n_4, n_5, n_6\}\}$, and $\Pi_3 = \{\{n_1\}, \{n_2, n_3, n_4, n_5\}, \{n_6\}\}$.

6.4.3 A Greedy Algorithm for Coalescing. Instead of exploring all possible partitions, we use the following greedy algorithm that implements the preceding heuristics in three steps:

(1) First, a dataflow analysis (described in Section 6.4.4) identifies whether a node can be merged into a partition containing its predecessors and its successors. It honors the constraints described in Sections 6.4.1 and 6.4.2, except the constraint for coherence, which is checked after the analysis (see the following step (2)).

As a heuristic, the dataflow analysis identifies partitions by accumulating nodes in a part in the "forward" direction. Consider a sequence of nodes n_1, n_2, n_3 such that we have control flow edges $n_1 \rightarrow n_2$ and $n_2 \rightarrow n_3$ such that there are two valid partitions: $\{\{n_1, n_2\}, \{n_3\}\}$ and $\{\{n_1\}, \{n_2, n_3\}\}$. Our algorithm constructs the first partition.

(2) The results of dataflow analysis are refined to ensure coherence of the partition obtained after dataflow analysis. If a part violates coherence, its entry and/or exit nodes are excluded from the part and the condition is applied recursively to the remaining part. The excluded nodes form independent parts.

(3) Actual partitions are created.

6.4.4 Dataflow Analysis for Coalescing. We define two interdependent dataflow analyses that

- construct part $\Pi(n)$ using data flow variables $\mathsf{ColIn}_n/\mathsf{ColOut}_n$ and
- compute the GPUs reaching node n (from within the nodes in $\Pi(n)$) in dataflow variables $\mathsf{GpuIn}_n/\mathsf{GpuOut}_n$. This information is used to identify the potential RaW or WaW data dependence between the GPUs in part $\Pi(n)$.

Unlike the usual dataflow variables that typically compute a set of facts, $\mathsf{ColIn}_n/\mathsf{ColOut}_n$ are predicates. Consider a control flow edge $\delta_{\mathrm{n}} \rightarrow \delta_{\mathrm{m}}$ in the GPG. Then, m and n belong to the same part if ColOut_n and ColIn_m are *true*. Thus, our analysis does not enumerate the parts as sets of GPBs explicitly; instead, parts are computed implicitly by setting predicates ColIn/ColOut of adjacent GPBs.

$$\mathsf{ColIn}_n := \begin{cases} \mathit{false} & n \text{ is Start} \\ \displaystyle\bigvee_{m \,\in\, pred(n)} \mathsf{coalesce}(m, n) & \text{otherwise} \end{cases}$$

$$\mathsf{ColOut}_n := \begin{cases} \mathit{false} & n \text{ is End} \\ \displaystyle\bigvee_{m \in succ(n)} \mathsf{ColIn}_m & \text{otherwise} \end{cases}$$

$$\mathsf{coalesce}(m, n) \Leftrightarrow \mathsf{ColOut}_m \wedge \Big(\mathsf{GpuOut}_m = \emptyset \vee \mathsf{gpuFlow}(m, n) \neq \emptyset\Big)$$

$$\mathsf{GpuIn}_n := \begin{cases} \emptyset & n \text{ is Start} \\ \displaystyle\bigcup_{m \,\in\, pred(n)} \mathsf{gpuFlow}(m, n) & \text{otherwise} \end{cases}$$

$$\mathsf{GpuOut}_n := \begin{cases} \mathsf{GpuIn}_n \cup \delta_n & \mathsf{ColIn}_n = \mathit{true} \\ \delta_n & \text{otherwise} \end{cases}$$

$$\mathsf{gpuFlow}(m, n) := \begin{cases} \emptyset & \neg\mathsf{ColIn}_n \wedge \mathsf{DDep}(\mathsf{GpuOut}_m, \delta_n) \\ \mathsf{GpuOut}_m & \text{otherwise} \end{cases}$$

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

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

Definition 9. Dataflow equations for coalescing analysis.

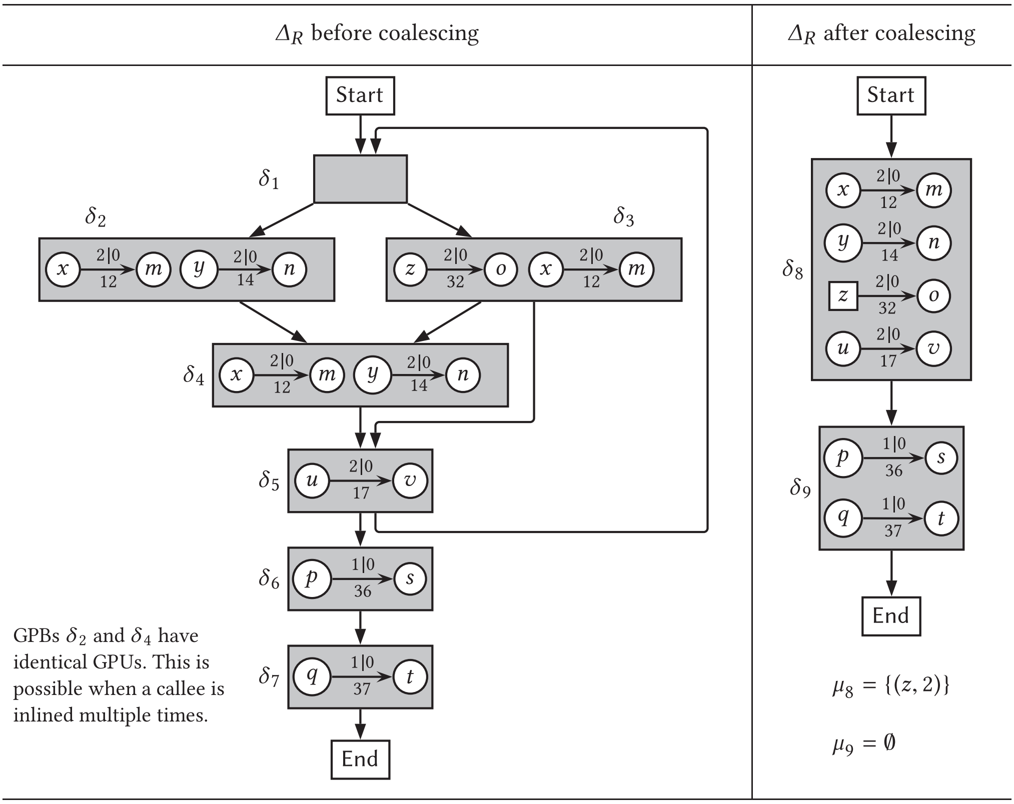
The dataflow equations to compute $\mathsf{ColIn}_n/\mathsf{ColOut}_n$ are given in Definition 9. The initialization is *false* for all GPBs. Predicate $\mathsf{coalesce}(P, n)$ uses $\mathsf{gpuFlow}(P, n)$ to check if GPUs reaching P from within $\Pi(P)$ are allowed to flow from P to n—if yes, then P and n belong to the same part. If GpuOut_P is $\emptyset$, they belong to the same part regardless of $\mathsf{gpuFlow}(P, n)$. The presence of ColOut_P in the equation of coalesce (Definition 9) ensures that GPB δ_P is considered for coalescing with δ_{n} only if δ_P has not been found to be an *exit* node of a part.

Unlike the usual dataflow equations, the dataflow variables ColIn_n and ColOut_n for GPB n are independent of each other—ColIn_n depends only on the ColOut of its predecessors, and ColOut_n depends only on the ColIn of its successors. Intuitively, this form of dataflow equations attempts to *melt* the boundaries of GPB n to explore fusing it with its successors and predecessors.

The incremental expansion of a part in a forward direction influences the flow of GPUs accumulated in a part leading to a forward dataflow analysis for computing the GPUs reaching node n in $\Pi(n)$ using dataflow variables $\mathsf{GpuIn}_n/\mathsf{GpuOut}_n$. The dataflow equations to compute them are given in Definition 9. Function $\mathsf{gpuFlow}(p, n)$ in the equation for GpuIn computes the set of GPUs reaching p in $\Pi(p)$ that flow from p to n (provided n can be included in $\Pi(p)$). It facilitates step (2) in Section 6.4.1. If no data dependence exists (i.e., predicate DDep is *false*), the GPUs accumulated in GpuOut_p are propagated to n. The presence of $\neg\mathsf{ColIn}_p$ in the equation for gpuFlow ensures that GPUs in GpuOut_p are propagated to δ_{n} only if δ_{n} has not been found to be an *entry* node of $\Pi(n)$.

Example 32. Figure 13 gives the dataflow information for the example of Figure 12. GPBs δ_1 and δ_2 can be coalesced because ColOut_1 is *true* and GpuOut_1 is $\emptyset$. Thus, $\mathsf{DDep}(\delta_1, \delta_2)$ 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,

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



Δ_R before coalescing
Δ_R after coalescing
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

GPBs δ_2 and δ_4 have identical GPUs. This is possible when a callee is inlined multiple times.

$\mu_8 = \{(z, 2)\}$

$\mu_9 = \emptyset$

Fig. 12. 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, which is redundant (Section 6.2) and is removed.

Names for GPUs. Statement labels play no part here											
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)	GpuIn$_n$	GpuOut$_n$	ColIn$_n$	ColOut$_n$
δ_1	$\emptyset$	$\emptyset$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	*false*	*true*
δ_2	{int∗, float∗}	{int∗∗, float∗∗}	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	*true*	*true*
δ_3	{short∗, int∗}	{short∗∗, int∗∗}	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	*true*	*true*
δ_4	{int∗, float∗}	{int∗∗, float∗∗}	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	*true*	*true*
δ_5	{int∗∗}	{int∗∗∗}	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	*true*	*false*
δ_6	{int∗}	$\emptyset$	$\emptyset$	$\{\gamma_5\}$	*false*	*true*
δ_7	{int∗}	$\emptyset$	$\{\gamma_5\}$	$\{\gamma_5, \gamma_6\}$	*true*	*false*

Fig. 13. The dataflow information computed by coalescing analysis for the example in Figure 12. The ColIn and ColOut 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 parts.

ColOut$_1$, ColIn$_2$, and ColIn$_3$ are *true*. We check the data dependence between the GPUs of GPBs δ_2 and δ_4 using the type information. However, DDep(δ_2, δ_4) returns *false* because the term (GpuOut$_2 - \delta_4$) 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 (GpuOut$_3 - \delta_4$) returns $z \xrightarrow[32]{2|0} o$ whose 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, DDep(δ_4, δ_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 DDep(δ_5, δ_6) returns *true* (type of source of the GPU $x \xrightarrow[12]{2|0} m \in$ GpuOut$_5$ 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 is $\delta_8 = \{\delta_1, \delta_2, \delta_3, \delta_4, \delta_5\}$. The loop $\delta_5 \rightarrow \delta_1$ before coalescing now reduces to the self-loop over GPB δ_8 after coalescing and is eliminated. GPB δ_6 becomes the entry of the new part. GPBs δ_6 and δ_7 can be coalesced as there is no data dependence between their GPUs. Note that the resulting partition is trivially coherent because each part is a single-entry and single-exit node.

GPU $z \xrightarrow[32]{2|0} o$ has a definition-free path in δ_8 because boundary definition $z \xrightarrow[0]{2|2} z$ reaches the exit of part δ_8 along the path $\delta_1 \rightarrow \delta_2 \rightarrow \delta_4 \rightarrow \delta_5$. No other GPU has a definition-free path.

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 accumulate the points-to information of a particular statement in all contexts.

Example 33. In Figure 4, GPBs δ_{01} and δ_{02} can be coalesced because DDep(δ_{01}, δ_{02}) returns *false*, indicating that there is no type matching and hence no possible data dependence between their GPUs. Thus, ColOut$_{01}$ and ColIn$_{02}$ are set to *true*. The loop formed by the back edge $\delta_{03} \rightarrow \delta_{01}$ reduces to a self-loop over GPB δ_{11} after coalescing. The self-loop is redundant, and hence it is eliminated. For GPBs δ_{02} and δ_{04}, DDep(δ_{02}, δ_{04}) returns *true* because typeof(q, 2) (for the GPU $q \xrightarrow[02]{2|0} m$ in δ_{02}) matches 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_{02} and δ_{04} (q and p could be aliased in the caller), and hence these GPBs cannot be coalesced. Thus, ColOut$_{02}$ and ColIn$_{04}$ are set to *false*. For GPBs δ_{04} and δ_5, DDep(δ_{04}, δ_{05}) returns *false* because there is no possible data dependence. Hence, ColOut$_{04}$ and ColIn$_{05}$ are set to *true*, and the two GPBs can be coalesced.

Example 34. In Figure 4, $\mu_i = \emptyset$ for all nodes i in the initial GPG. Strength reduction reduces the GPUs in δ_{02} and correspondingly updates μ_{02} to $\{(b, 1), (q, 2)\}$. After coalescing, the may-definition sets are computed to obtain $\mu_{11} = \{(b, 1), (q, 2)\}$ (because these sources have a definition-free path from the entry of $\delta_{01} \in entry(\delta_{11})$ to exit of $\delta_{02} \in exit(\delta_{11})$) and $\mu_{12} = \emptyset$.

```
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 ($P, \Delta_P^1, \Delta_P^i$)
{
    $Rprev = \text{RGOut}_{\text{End}}(\Delta_P^i)$
    $\overline{R}prev = \overline{\text{RGOut}}_{\text{End}}(\Delta_P^i)$
    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 $\Delta_P^{i+1}$
    $Rcurr = \text{RGOut}_{\text{End}}(\Delta_P^{i+1})$
    $\overline{R}curr = \overline{\text{RGOut}}_{\text{End}}(\Delta_P^{i+1})$
    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 control flow elimination optimizations over $\Delta_P^{i+1}$
    return $\Delta_P^{i+1}$
}
```

Definition 10. Computing GPGs for recursive procedures by successive refinement.

For procedure R (Figure 5), the boundary definition $b \xrightarrow[00]{1|1} b'$ reaches the exit of Δ_R, indicating that b is *may*-defined. Hence, $\mu_{15} = \{(b, 1)\}$. The GPU $q \xrightarrow[02]{2|0} m$ reduces to $d \xrightarrow[02]{1|0} m$ in δ_{13} in Δ_R. Note that d is defined in δ_{08} also, and hence neither $(q, 2)$ nor $(d, 1)$ is contained in μ_{15}.

7 CALL INLINING

We explain call inlining by classifying calls into three categories: (a) callee is known and the call is non-recursive, (b) callee is known and the call is recursive, and (c) callee is not known.

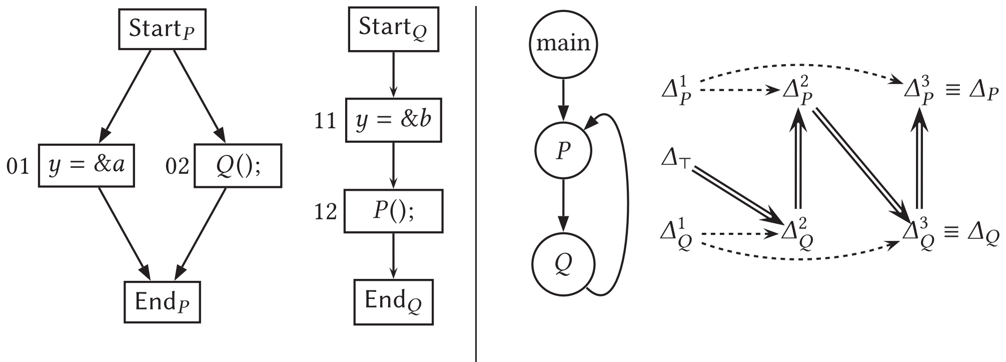
7.1 Callee Is Known and the Call Is Non-Recursive

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.

We inline the optimized GPGs of the callees at the call sites in the caller procedures by renumbering the GPB nodes and each inlining of a callee gives fresh numbering to the nodes. This process does not change the statement labels within the GPUs. In addition, the upwards-exposed variable x' occurring in a callee's GPU inlined in the caller is substituted by the original variable x.

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 respectively contain the following:

- GPUs that correspond to the actual-to-formal-parameter mapping.
- 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).



(a) Mutually recursive procedures

(b) Call graph and the order of constructing GPGs

Fig. 14. Constructing GPGs for recursive procedures by successive refinements.

7.2 Callee Is Known and the Call Is Recursive

Consider Figure 14, in which procedure P calls procedure Q and Q calls P. The GPG of Q depends on that of P and vice versa, leading to *incomplete* GPGs: the GPGs of the callees of some calls either have not been constructed or are incomplete. We handle this mutual dependency by successive refinement of incomplete GPGs of P and Q, which involves inlining GPGs of the callee procedures, followed by GPG optimizations, repeatedly until a fixed point is reached. The rest of the section explains how refinement is performed and how a fixed point is defined and detected.

A set of recursive procedures is represented by a strongly connected component in a call graph. We construct GPGs for a set of recursive procedures by visiting the procedures in a post order obtained through a topological sort of the call graph. Because of recursion, the GPGs of some callees of the leaf are not available in the beginning. 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. The GPG $\Delta_\top$ is the $\top$ element of the lattice of all possible procedure summaries. It kills all GPUs and generates none (thereby, when applied, it computes the $\top$ value—$\emptyset$—of the lattice for *may*-points-to analysis) [22]. This is consistent with using $\top$ value as the initialization for computing the maximum fixed-point solution in iterative dataflow analysis. Semantically, $\Delta_\top$ 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. Let Δ_P^1 denote the GPG of procedure P in which all of 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 of refinement, the recursive calls in Δ_P^1 are inlined either by

- $\Delta_\top$ when no GPG of the callee has been constructed or
- an incomplete GPG of a callee in which some calls are underapproximated using $\Delta_\top$.

Thus, we compute a series of GPGs Δ_P^i, $i > 1$ for every procedure P in a strongly connected component in the call graph until the termination of fixed-point computation. Once Δ_P^i is constructed, we decide to construct Δ_Q^j for a caller Q of P if the dataflow 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. Thus, the convergence of this fixed-point computation differs subtly from the usual fixed-point computation in that it does not depend on matching the values at corresponding nodes (or the structure of the GPGs) across successive iterations. Instead, it depends on matching the *dataflow* values of the End GPB. This process is described in Definition 10.

Example 35. In the example of Figure 14, the sole strongly connected component contains procedures P and Q. The GPG of procedure Q is constructed first and 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 that 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 15. 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 dataflow values for Δ_Q^2 are $Rprev = \overline{R}prev = \emptyset$, whereas the dataflow values for Δ_Q^3 are $Rcurr = \overline{R}curr = \{y \xrightarrow[01]{1|0} a\}$. Since the dataflow values have changed, the caller of Q (i.e., P) is pushed on the worklist and Δ_P^3 is constructed by inlining Δ_Q^3. The dataflow 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 the 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 dataflow 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.

We give an informal argument for termination of GPG construction in the presence of recursion. A formal and complete proof can be found in Gharat [10]. We first describe a property that holds for intraprocedural dataflow analysis over CFGs and then extend it to GPGs.

Consider a CFG C_Q representing procedure Q such that the flow functions associated with the nodes in C_Q are monotonic and compute values in a finite lattice L. Let the dataflow value associated with the entry of Start_Q and exit of End_Q be denoted by *In* and *Out*, respectively. We denote their relationship by writing $Out = C_Q(In)$. Consider an arbitrary node n in C_Q whose flow function is f_n. Let n be replaced by n' with flow function f_n' such that $f_n' \sqsubseteq f_n$ (i.e., $\forall x \in L, f_n'(x) \sqsubseteq f_n(x)$), giving us the CFG C_Q'. Let $Out' = C_Q'(In)$. We claim that $Out' \sqsubseteq Out$. This follows from the fact that the control flow in C_Q and C_Q' is the same, all flow functions are same except f_n has been replaced by $f_n' \sqsubseteq f_n$, and the same *In* value is supplied to both C_Q and C_Q'.

The preceding situation models call inlining in GPGs. From Section 3.1.3, the set of GPUs is finite, and from Gharat [10], they form a lattice with $\subseteq$ as the partial order. The flow function for a call GPB is initially assumed to be $f_\top$, and then the GPB is replaced by the GPG of the callee. The control

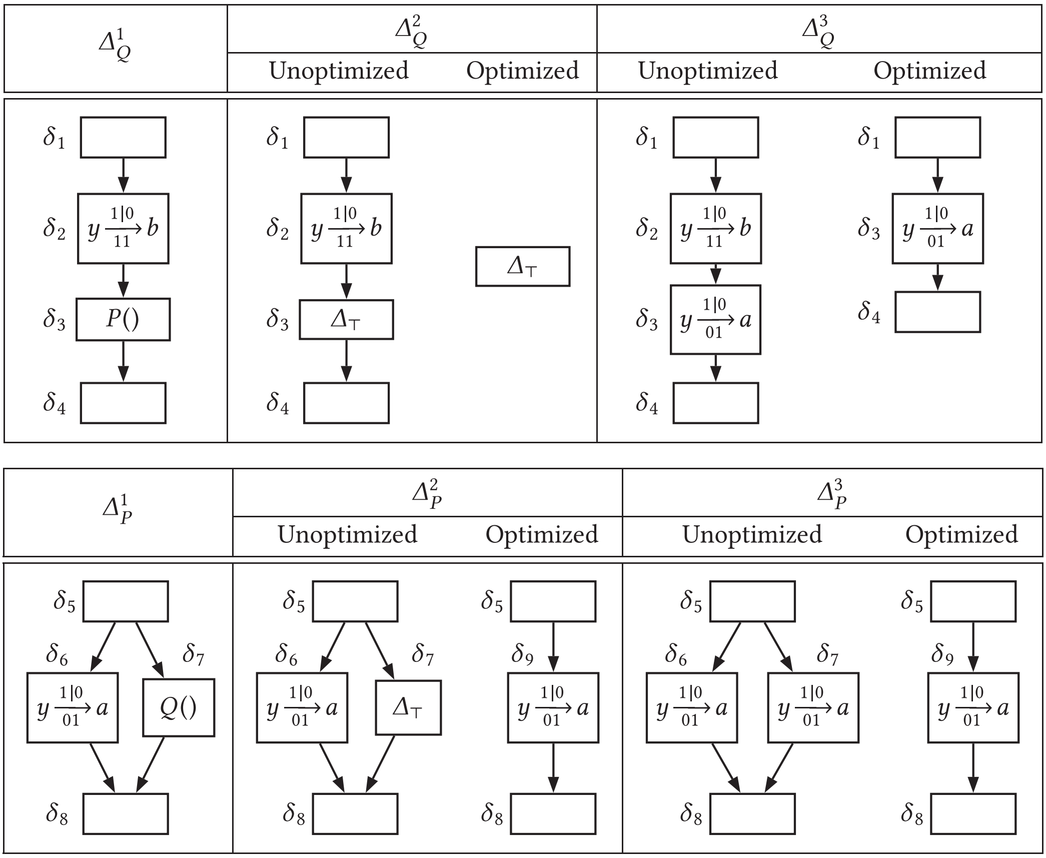


Fig. 15. Series of GPGs of procedures P and Q of Figure 14. They are computed in the order shown in Figure 14(b). See Example 35 for an explanation.

flow surrounding this call remains same. Let the effect of the callee GPG be described by a flow function f. Clearly, $f \sqsubseteq f_\top$ because $f_\top$ computes $\top$ value. The process of successive refinements for handling recursion replaces call GPBs by the GPGs of the callees repeatedly. Consider a sequence of refinement, $\Delta^1_Q, \Delta^2_Q, \ldots \Delta^i_Q$. It can be proved by induction on the length of the sequence that the GPUs reaching the End GPB of the successive GPBs follow a descending chain because the boundary definitions at the Start GPB of every Δ^i_Q are identical. Since the set of all possible GPUs is finite, this descending chain must contain two successive elements that are identical. Thus, there must exist Δ^k_Q and Δ^{k+1}_Q such that the GPUs reaching their End GPB are identical.

7.3 Handling Calls Through Function Pointers

We model a call through function pointer (say *fp*) at call site s as a use statement with a GPU $\boldsymbol{u} \xrightarrow[s]{1|1} fp$ (Section 8). Interleaving of strength reduction and call inlining reduces the GPU $\boldsymbol{u} \xrightarrow[s]{1|1} fp$ and provides the pointees of *fp*. This is identical to computing points-to information (Section 8). Until the pointees become available, the GPU $\boldsymbol{u} \xrightarrow[s]{1|1} fp$ acts as a barrier. Once the pointees become available, the indirect call converts to a set of direct calls (see Appendix C for an illustrative example). A naive approach to function pointer resolution would inline an indirect callee first into its immediate callers. This may require as many rounds of GPG construction as the maximum number of indirect calls in any call chain. Instead, we allow inlining directly in a transitive callee when a pointee of the function pointer of an indirect call becomes available. Hence, we can resolve all indirect calls in a call chain in a single round beginning with the indirect call closest to *main*. This is explained in Appendix C.

8 COMPUTING POINTS-TO INFORMATION USING GPGS

The second phase of a bottom-up approach, which uses procedure summaries created in the first phase, is redundant in our method. This is because our first phase computes the points-to information as a side effect of the construction of GPGs. Since statement labels in GPUs are unique across all procedures and are not renamed on inlining, the points-to edges computed across different contexts for a given statement can be back-annotated to the statements giving the flow- and context-sensitive points-to information for the statement.

Since we also need points-to information for statements that read pointers but do not define them, we model them as *use* statements. 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 modeled as a GPB $\{\boldsymbol{u} \xrightarrow[s]{1|1} x, \boldsymbol{u} \xrightarrow[s]{1|2} y\}$, whereas an integer assignment "`*x = 5;`" is modeled as a GPB $\{\boldsymbol{u} \xrightarrow[s]{1|2} x\}$.

Example 36. Consider the assignment sequence $01\colon x = \&a;\ 02\colon *x = 5;$. 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.

When a use involves multiple pointers such as "`if (x == *y)`," the corresponding GPB contains multiple GPUs. 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 that have been replaced by their pointees in the simplified GPUs.

With the provision of a GPU for a use statement, the process of computing points-to information can be seen simply as a process of simplifying consumer GPUs (including those with a use node $\boldsymbol{u}$. 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 $\{a \xrightarrow[s]{1|0} b \mid a \text{ is } i^{th} \text{ pointee of } x, b \text{ is } j^{th} \text{ pointee of } y\}$. This is achieved by propagating the use of a pointer (in a pointer assignment or a use statement) and its definitions to a common context. This may require propagating

(a) a consumer GPU $\boldsymbol{c}$ (i.e., a use of a pointer variable) to a caller,
(b) a producer GPU $\boldsymbol{p}$ (i.e., a definition of a pointer variable) to a caller,
(c) both consumer $\boldsymbol{c}$ and producer $\boldsymbol{p}$ to a common (transitive) caller, and
(d) neither (if they are same in the procedure).

Example 37. eg.phase1.pta The four variants of hoisting $\boldsymbol{p}$ and $\boldsymbol{c}$ to a common procedure in the first phase of a bottom-up method are illustrated in the following with the help of Figure 16; effectively, they make the second phase redundant:

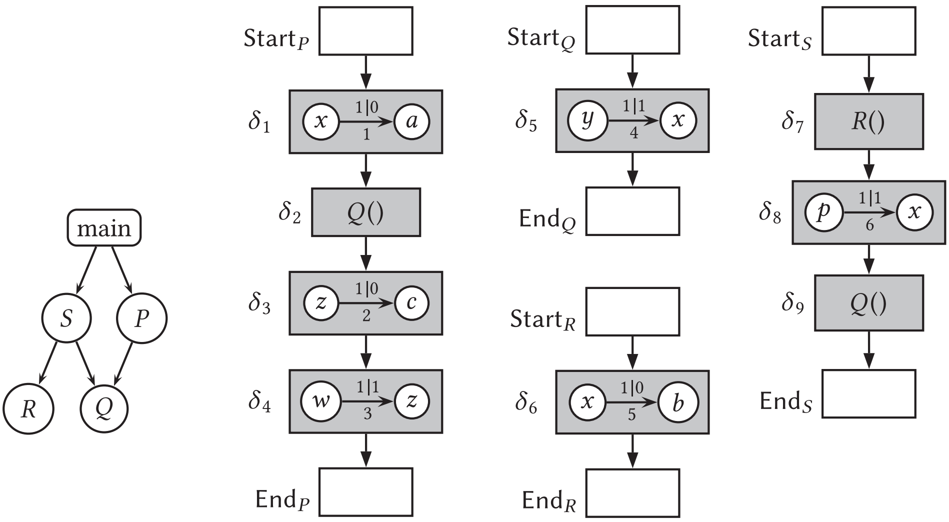


Fig. 16. Computing points-to information using GPGs. The first column gives the call graph, whereas the other columns give GPGs before call inlining. The GPG of procedure *main* has been omitted.

(a) When Δ_Q is inlined in P, $\boldsymbol{c} : y \xrightarrow[4]{1|1} x$ from procedure Q is hoisted to procedure P that contains GPU $\boldsymbol{p} : x \xrightarrow[1]{1|0} a$, thereby propagating the use of pointer x in procedure Q to caller P. Strength reduction reduces $\boldsymbol{c}$ to $y \xrightarrow[4]{1|0} a$.

(b) When Δ_R is inlined in S, $\boldsymbol{p} : x \xrightarrow[5]{1|0} b$ from procedure R is hoisted to procedure S that contains $\boldsymbol{c} : p \xrightarrow[6]{1|1} x$, thereby propagating the definition of x in procedure R to the caller S. Strength reduction reduces $\boldsymbol{c}$ to $p \xrightarrow[6]{1|0} b$.

(c) When Δ_Q and Δ_R are inlined in S, $\boldsymbol{c} : y \xrightarrow[4]{1|1} x$ in procedure Q and $\boldsymbol{p} : x \xrightarrow[5]{1|0} b$ in procedure R are both hoisted to procedure S, thereby propagating both the use and definition of x in procedure S. Strength reduction reduces $\boldsymbol{c}$ to $y \xrightarrow[4]{1|0} b$.

(d) Both the definition and use of pointer z are available in procedure P with $\boldsymbol{c} : w \xrightarrow[3]{1|1} z$ 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 P, and it points-to b along the call from procedure S. Thus, the points-to information $\{y \overset{\rightarrow}{1|0} a, y \overset{\rightarrow}{1|0} b\}$ represents flow- and context-sensitive information for statement 4.

9 SOUNDNESS AND PRECISION

In this section, we prove the soundness and precision of GPG-based points-to analysis by comparing it with a classical top-down flow- and context-sensitive points-to analysis. We first describe our

$$\mathsf{PTin}_n := \begin{cases} \Big\{ (Y,Y) \mid \exists X.\ (X,Y) \in \mathsf{PTin}_m,\, m \in \mathsf{CallSitesOf}(Q) \Big\} & n \text{ is } \mathsf{Start}_Q \\ \qquad \cup \ \Big\{ (L_P \times \{\mathsf{NULL}\},\ L_P \times \{\mathsf{NULL}\}) \mid Q \text{ is } \mathit{main} \Big\} & \text{for some } Q \\ \Big\{ (X,Y) \mid Y = \displaystyle\bigcup_{m \in \mathit{pred}(n)} \{ Z \mid (X,Z) \in \mathsf{PTout}_m \} \Big\} & \text{otherwise} \end{cases}$$

$$\mathsf{PTout}_n := \begin{cases} \Big\{ (X,Y) \mid \exists\, Z.\ (X,Z) \in \mathsf{PTin}_n \wedge (Z,Y) \in \mathsf{PTout}_{\mathsf{End}_Q} \Big\} & n \text{ calls } Q \\ \Big\{ (X,\ Y - \mathsf{Kill}_n(Y) \cup \mathsf{Gen}_n(Y)) \mid (X,Y) \in \mathsf{PTin}_n \Big\} & \text{otherwise} \end{cases}$$

$$\mathsf{Kill}_n(X) := \mathsf{Overwritten\_Ptrs}_n(X) \ \times \ (L \cup \{\mathsf{NULL}\})$$

$$\mathsf{Gen}_n(X) := \mathsf{Updated\_Ptrs}_n(X) \times \mathsf{RHS\_Pointees}_n(X)$$

In the following definitions, variables $w, x, y \in L_P$ (i.e., they are pointers) and $z \in L \cup \{\mathsf{NULL}\}$ (i.e., it need not be a pointer).

$$\mathsf{Overwritten\_Ptrs}_n(X) := \begin{cases} \{ w \mid x \rightarrow w \in X, \forall z.\ x \rightarrow z \in X \Rightarrow z = w \} & n \text{ is } *x = y \\ \{x\} & n \text{ is } x = \ldots \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{Updated\_Ptrs}_n(X) := \begin{cases} \{ w \mid x \rightarrow w \in X \} & n \text{ is } *x = y \\ \{x\} & n \text{ is } x = \ldots \\ \emptyset & \text{otherwise} \end{cases}$$

$$\mathsf{RHS\_Pointees}_n(X) := \begin{cases} \{z\} & n \text{ is } x = \&z \\ \{ z \mid y \rightarrow z \in X \} & n \text{ is } x = y \text{ or } *x = y \\ \{ z \mid \exists w.\ y \rightarrow w \in X \wedge w \rightarrow z \in X \} & n \text{ is } x = *y \\ \emptyset & \text{otherwise} \end{cases}$$

Definition 11. Classical top-down interprocedural flow- and context-sensitive points-to analysis. Points-to sets are subsets of $L_P \times (L \cup \{\mathsf{NULL}\})$ whose elements (x, y) are written $x \rightarrow y$ as is conventional.

assumptions, review the classical points-to analysis, and then provide the main proof obligation. This is followed by a series of lemmas proving the soundness of our analyses and operations.

9.1 Assumptions

We do a whole-program analysis and assume that the entire source is available for analysis. Practically, there are very few library functions that influence the points-to relations of pointers to scalars in a C program. Library functions manipulating pointers into the heap can be manually represented by a GPB representing a sound overapproximation of their summaries.

For simplicity of reasoning, our proof does not talk about heap pointers. Our analysis computes a sound overapproximation of classical points-to analysis for heap pointers because (a) we use a simple allocation-site-based abstractions in which heap locations are not cloned context sensitively, and (b) we use k-limiting for heap pointers that are live on entry.

In the proof, we often talk about reaching GPUs analysis without making a distinction between reaching GPUs analysis with and without blocking. Blocking is discussed only in the proof of Lemma 9.11 because it is required to ensure soundness of GPU reduction.

Finally, our proofs use a simplistic model of programs where all variables are global and there is no parameter or return value mapping when making a call or when returning from a call. Including local variables, function parameters, and these mapping functions in the reasoning is a matter of detail and is not required for the spirit of the arguments made in the proof.

9.2 Classical Top-Down Flow- and Context-Sensitive Points-to Analysis

This section describes the top-down interprocedural flow- and context-sensitive points-to analysis. In keeping with the requirements of our proof of soundness (assumptions in Section 9.1), our formulation is restricted to global (non-structure) variables and direct procedure calls. Our formulation can be easily extended to support local variables, parameter mappings, return-value mappings, and structures; calls through function pointers can be handled using a standard approach of augmenting the call graph on the fly.

Our formulation is based on the classical Sharir-Pnueli tabulation method [34]. This method maintains pairs (X, Y) of input-output dataflow values (hence the name *tabulation method*) for every procedure Q where X reaches Start_Q and Y is the corresponding value reaching End_Q. The input value X forms a context for context-sensitive analysis of Q. Every time a call to Q is seen with the dataflow value X reaching the call, the dataflow value Y is used as the effect of the call. In other words, procedure Q is reanalyzed only when a dataflow $X' \neq X$ reaches a call to Q; the corresponding value Y' reaching End_Q is then memoized as the pair (X', Y').[11] However, since the tabulation method is algorithmic, we use the ideas from value-contexts-based method [29] for a declarative description using dataflow equations.

The value-contexts-based method is subtly different the tabulation method in the following way. For each procedure Q, this method creates a mapping represented as a set of pairs (X, Y), with X being a possible points-to graph reaching Start_Q and and Y being its associated points-to graph reaching End_Q. During intraprocedural analysis, X is held constant and represents the calling context, and Y, at each program point n within Q, represents the points-to graph reaching n. This association of dataflow values at a program point with its context enables a declarative description of the method. Definition 11 provides the dataflow equations for *may*-points-to analysis using dataflow variables $\mathsf{PTin}_n/\mathsf{PTout}_n$ for node n. The following two situations in the dataflow equations require special handling for maintaining context sensitivity:

- Context X is generated at Start_Q in the second case of the equation for PTin_n. Thus, the data flow value at Start_Q is a set of pairs (Y, Y).
- The context-sensitive data value after a call to some procedure Q is computed by the first case in the equation for PTout_n. This is achieved by extracting the dataflow value (from $\mathsf{PTout}_{\mathsf{End}_Q}$) that corresponds to the context in which the call to Q was made.

For other statements, the generated points-to information is the cross product of the pointers being defined by the statement ($\mathsf{Updated\_Ptrs}_n$) and the locations whose addresses are read by the pointers on the RHS ($\mathsf{RHS\_Pointees}_n$). The points-to information is killed by a statement when a strong update is possible (Section 2.1.3), which is the case for every direct assignment because the pointer in the LHS is overwritten ($\mathsf{Overwritten\_Ptrs}_n$). For an indirect assignment, when the pointer appearing on the LHS has exactly one pointee and the pointer is not live on entry to *main*, the pointer is overwritten and the earlier pointees are removed. This is possible only when there is no definition-free path for the pointer from Start_{main} to statement n. We eliminate such definition-free paths by making every pointer point to NULL at the Start_{main} for its outermost call (the first case in PTin_n equation); this is consistent with C semantics for global variables. This initialization may be adapted suitably to handle other static initializations in the program.

[11] Since all input-output values are memoized, the method requires the lattice of dataflow values to be finite. In our case, X and Y are points-to graphs involving global variables and their lattice is finite.

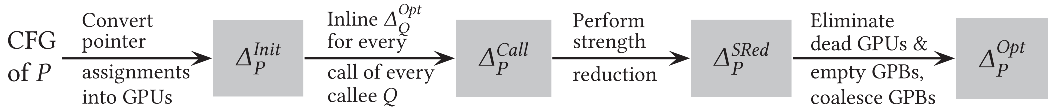

CFG of P
Convert pointer assignments into GPUs
Δ_P^{Init}
Inline Δ_Q^{Opt} for every call of every callee Q
Δ_P^{Call}
Perform strength reduction
Δ_P^{SRed}
Eliminate dead GPUs & empty GPBs, coalesce GPBs
Δ_P^{Opt}

Fig. 17. Different representations of the GPG of procedure P.

9.3 Notations Used in the Proof

We need to name different versions of a GPG as it undergoes optimizations, analyses on these different versions, and GPBs of a callee inlined in a caller's GPG.

9.3.1 Naming the GPG Versions and Analyses. Recall that GPG construction creates a series of GPGs by progressively transforming them. For the purpose of proof, it is convenient to use notation that distinguishes between them. We use the notation in Figure 17 for different versions of a GPG. These different versions can have different possibilities of analyses that we show are equivalent using the following notation:

- TPT. Top-down flow- and context-sensitive classical points-to analysis (Section 9.2).
- TRG. Top-down flow- and context-sensitive reaching GPUs analysis.
- IRG. Intraprocedural reaching GPUs analysis.

Note that our implementation does not perform TPT or TRG.

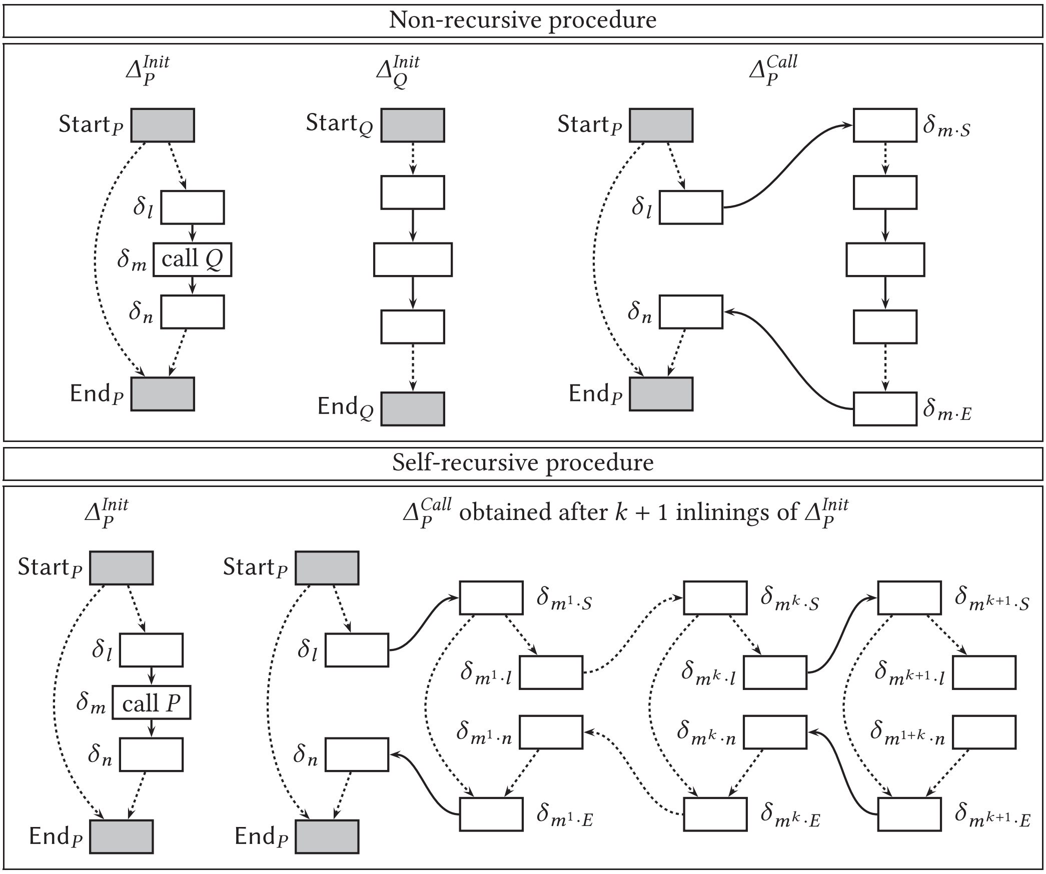
9.3.2 Naming the GPBs After Call Inlining. Let procedure P call procedure Q. Then, as illustrated in Figure 18, Δ_P^{Call} contains the Δ_Q^{Opt} as a subgraph that is obtained by expanding GPB δ_{m} containing the call to Q, by connecting the predecessors of δ_m to the Start GPB of Δ_Q^{Opt} and the End GPB of Δ_Q^{Opt}, and to the successors of δ_m.

Consider node n in procedure Q. After Q is inlined in its caller, say P, the label of the inlined instance of the node is a sequence $m \cdot n$, where m is the label of the node in P that contains the call to Q. When P is inlined in a caller R, the label of the further inlined instance of the node becomes $l \cdot m \cdot n$, where node l in R calls P. Thus, the nodes labels are sequences of the labels of the call nodes with the last element in the sequence identifying the actual node in the inlined callers. Letters S and E are used for distinguishing the inlined Start and End nodes.

We handle recursion by repeated inlining of recursive procedures. This process constructs for series of GPGs Δ_P^i, $i > 1$ for every procedure P in a cycle of recursion. GPG Δ_P^{i+1} is constructed by inlining GPGs Δ_Q^1 in Δ_P^i, for all callees Q of P. As explained in Section 7.2, this sequence is bounded by some k when Δ_P^k is equivalent to Δ_P^{k+1} in terms of the GPUs reaching the End nodes are identical. This converts a recursive GPG into a non-recursive one.

For the purpose of reasoning in the proofs, we assume without any loss of generality that indirect recursion has been converted into self-recursion [18]. The resulting inlining has been illustrated in Figure 18. Note that the successors and predecessors of the call node after $k + 1$ inlinings are disconnected (e.g., there is no control flow from $\delta_{m^{k+1} \cdot l}$ to $\delta_{m^{k+1} \cdot n}$ in Figure 18). For self-recursive procedures, we use the notation $\delta_{m^k \cdot n}$ to denote the sequence $m \cdot m \ldots m \cdot S$ of k occurrences of m followed by n, where n could be letter S or E apart from the usual node labels.

9.3.3 Naming the Dataflow Variables in Different Contexts of a Recursive Procedure. The top-down context-sensitive reaching GPUs analysis over Δ_P^{Init} computes the values of dataflow variables $\mathsf{RGIn}_n/\mathsf{RGOut}_n$ for different contexts reaching node n for different recursion depth. We distinguish between these different values of the same dataflow variable by writing $\mathsf{RGIn}_n^i/\mathsf{RGOut}_n^i$, where i denotes the depth of the recursive call. Note that there is some k for which $\mathsf{RGIn}_n^k =$



When P is non-recursive, label $\delta_{m\cdot S}$ in Δ_P^{Call} denotes the Start of the callee procedure for the call in node l in P. When P is self recursive, label $\delta_{m^k\cdot S}$ of a node in Δ_P^{Call} of self-recursive procedure P indicates a sequence $m\cdot m\ldots m\cdot S$ of k occurrences of m followed by S indicating the Start node of the procedures reached after k inlinings of the call to P in node m.

Fig. 18. Constructing a GPG by call inlining.

RGIn_n^{k+1} and $\text{RGOut}_n^k = \text{RGOut}_n^{k+1}$. It follows from the fact that the flow functions are monotonic and the lattice of GPUs is finite because of only a finite number of combinations of *indlev*s are possible due to the type restrictions in C (as explained in footnote 7). A formal proof of the convergence of GPG construction for recursive calls can be found in Gharat [10].

9.4 The Overall Proof

We use the classical points-to analysis defined in Section 9.2 as the gold standard and show that the GPG-based points-to analysis computes identical information (except when k-limiting is used for bounding *indlist*s for live-on-entry pointers to heap). Thus, our analysis is both sound and precise (for k-limited heap pointers, the precision can be controlled by choosing a suitable value of k).

Theorem 9.1. *Given the complete source of a C program, the GPG-based points-to analysis of the program computes the same points-to information that would be computed by a top-down fully flow- and context-sensitive classical points-to analysis of the program.*

Equivalence of analyses and associated lemmas. X denotes P or its transitive callees.

$$\begin{array}{ll}
\mathsf{TPT}(\Delta_P^{Init}, X) & \\
\Updownarrow & \text{(Claim A, Lemmas 9.2, 9.3)} \\
\mathsf{TRG}(\Delta_P^{Init}, X) & \\
\Updownarrow & \text{(Claim B, Lemmas 9.4, 9.5, 9.6, and 9.7)} \\
\mathsf{IRG}(\Delta_P^{Call}, X) & \\
\Updownarrow & \text{(Claim C, Lemma 9.8)} \\
\mathsf{IRG}(\Delta_P^{SRed}, X) & \\
\Updownarrow & \text{(Claim D, Lemma 9.9)} \\
\mathsf{IRG}(\Delta_P^{Opt}, X) &
\end{array}$$

Soundness and precision of analyses and operations

Soundness and precision of GPU composition	Lemma 9.10
Soundness and precision of GPU reduction	Lemma 9.11
Soundness and precision of classical flow-sensitive points-to analysis	Assumed
Soundness of reaching GPUs analyses	Lemmas 9.12, 9.13
Precision of reaching GPUs analyses	Lemma 9.14

Equivalence $A(\Delta, X) \Leftrightarrow A'(\Delta', X)$ indicates that for every statement in procedure X, the information computed by analysis A over Δ is identical to that computed by analysis A' over Δ'.

Fig. 19. The overall organization of the soundness proof listing the lemmas that show equivalence of different analyses over different representations.

Proof. Figure 19 illustrates the proof outline listing the lemmas that prove some key results. Since Δ_P^{Init} is constructed by simple transliteration (Section 3.3.1), we assume that it is a sound representation of the CFG of procedure P with respect to classical flow- and context-sensitive points-to analysis. Then, using the points-to relations created by static initializations as memory M (or set of GPUs) for boundary information for *main*,

$$\begin{aligned}
\mathsf{TPT}(\Delta_{main}^{Init}, main) &\Leftrightarrow \mathsf{TRG}(\Delta_{main}^{Init}, main) && \text{(from Lemma 9.2)} \\
&\Leftrightarrow \mathsf{IRG}(\Delta_{main}^{Call}, main) && \text{(from Lemma 9.5)} \\
&\Leftrightarrow \mathsf{IRG}(\Delta_{main}^{SRed}, main) && \text{(from Lemma 9.8)} \\
&\Leftrightarrow \mathsf{IRG}(\Delta_{main}^{Opt}, main) && \text{(from Lemma 9.9).}
\end{aligned}$$

For all transitive callees Q of *main*,

$$\begin{aligned}
\mathsf{TPT}(\Delta_{main}^{Init}, Q) &\Leftrightarrow \mathsf{TRG}(\Delta_{main}^{Init}, Q) && \text{(from Lemma 9.3)} \\
&\Leftrightarrow \mathsf{IRG}(\Delta_{main}^{Call}, Q) && \text{(from Lemma 9.7)} \\
&\Leftrightarrow \mathsf{IRG}(\Delta_{main}^{SRed}, Q) && \text{(from Lemma 9.8)} \\
&\Leftrightarrow \mathsf{IRG}(\Delta_{main}^{Opt}, Q) && \text{(from Lemma 9.9).}
\end{aligned}$$

Hence the theorem. □

9.5 Equivalence of Analyses over Different Representations of a GPG

Recall that a top-down analysis uses the dataflow information reaching the call sites to compute the context-sensitive dataflow information within the callees. Thus, the information reaching the call sites is boundary information for an analysis of the callee procedures.

Lemma 9.2 (Claim A for Procedure P). *Consider points-to information represented by memory M that defines all pointers that are live on entry in procedure P. Then, with M as boundary information,* $\mathsf{TPT}(\Delta_P^{Init}, P) \Leftrightarrow \mathsf{TRG}(\Delta_P^{Init}, P)$.

Proof. Since all pointers are defined before their use, GPU reduction computes GPUs of the form $x \xrightarrow[s]{1|0} y$. Thus, there are no potential dependences and hence no blocking. In such a situation, the dataflow equations in Definition 5 reduce to those of the classical flow-sensitive points-to analysis. Assuming that the two analyses maintain context sensitivity using the same mechanism, they would compute the same points-to information at the corresponding program points. □

Lemma 9.3 (Claim A for Callees of Procedure P). *Let procedure Q be a transitive callee of procedure P. Consider points-to information represented by memory M that defines all pointers that are live on entry in procedure P. Then, with M as boundary information,* $\mathsf{TPT}(\Delta_P^{Init}, Q) \Leftrightarrow \mathsf{TRG}(\Delta_P^{Init}, Q)$.

Proof. Similar to that of Lemma 9.2. □

Lemma 9.4 argues about the GPUs that reach the GPBs for the statements in P (and not the statements of the inlined callees), whereas Lemma 9.6 argues about the GPUs that reach the GPBs for the statements belonging to the (transitive) callees inlined in Δ_P^{Call}.

Lemma 9.4 (Claim B for Procedure P in Non-Recursive Case). *Consider a non-recursive procedure P such that all of its transitive callees are also non-recursive. For a given boundary information (possibly containing points-to information and boundary definitions),* $\mathsf{TRG}(\Delta_P^{Init}, P) \Leftrightarrow \mathsf{IRG}(\Delta_P^{Call}, P)$.

Proof. We prove the lemma by inducting on two levels. At the outer level, we use structural induction on the call structure rooted at P. To prove the inductive step of the outer induction, we use an inner induction on the iteration number in the Gauss-Seidel method of fixed-point computation (the dataflow values in iteration $i + 1$ are computed only from those computed in iteration i):

- *Basis of structural induction*: The base case is when P does not contain any call. Since Δ_P^{Init} and Δ_P^{Call} are identical in the absence of a call within P, the lemma trivially holds.
- *Inductive step of structural induction*: The inductive step requires us to prove that the lemma holds for P when it contains calls. For inductive hypothesis, we assume that the lemma holds for the callees in P. For every GPB δ_m, we need to prove the following equivalences:

  The IN equivalence: If δ_{m} contains a call, then RGIn_m in Δ_P^{Init} is identical in $\mathsf{RGIn}_{m \cdot S}$ in Δ_P^{Call}; otherwise, RGIn_m in Δ_P^{Init} is identical in RGIn_m in Δ_P^{Call}.

  The OUT equivalence: If δ_m contains a call, then RGOut_m in Δ_P^{Init} is identical in $\mathsf{RGOut}_{m \cdot E}$ in Δ_P^{Call}; otherwise, RGOut_m in Δ_P^{Init} is identical in RGOut_m in Δ_P^{Call}.

  We prove these equivalences on the number of iteration i:

  — *Basis of induction on the number of iterations:* The basis is the first iterations (i.e., $i = 1$). By initialization, RGIn is $\emptyset$ for each node in both Δ_P^{Init} and Δ_P^{Call} (except for Start_P for which it contains boundary definitions). Thus, IN equivalence holds for each m belonging to P. For the OUT equivalence, there are two cases:

  (a) δ_m *does not contain a call*: These GPBs are identical in Δ_P^{Init} and Δ_P^{Call}. For each such GPB δ_m, RGOut_m trivially contains all GPUs in δ_{m} because RGIn_m is $\emptyset$ and there is no GPU reduction. Thus, the OUT equivalence also holds for such GPBs.

  (b) δ_m *contains a call*: By the hypothesis of structural induction, the lemma holds for the callees. Since $\mathsf{TRG}(\Delta_Q^{Init}, Q) \Leftrightarrow \mathsf{IRG}(\Delta_Q^{Call}, Q)$, for every value of boundary information, it also holds for $\emptyset$ as boundary information. Hence, it holds for the End

GPB of Q. In Δ_P^{Init}, it becomes the value RGOut_m, whereas in Δ_P^{Call}, it becomes the value $\mathsf{RGOut}_{m \cdot E}$. Hence, the OUT equivalence also holds for such GPBs.

- *Inductive step for number of iterations*: For the hypothesis for the inner induction on the number of iterations, assume that the lemma holds for iteration i. Since RGIn_m for each m in iteration $i + 1$ is computed from RGOut of the predecessors nodes and these values have been computed in an iteration $i' \le i$, the IN equivalence holds for iteration $i + 1$. Since the RGIn_m values are same for each δ_m in both Δ_P^{Init} and Δ_P^{Call}, the RGOut_m must also be same proving the OUT equivalence and the inductive step.

This completes the proof of the lemma. □

Lemma 9.5 (Claim B for Procedure P in Recursive Case). *The claim of Lemma 9.4 also holds for recursive procedures.*

Proof. We prove the lemma by showing that for all $0 < i \le k$:

The IN equivalence: RGIn_m^i computed for the GPB δ_m in Δ_P^{Init} is identical to $\mathsf{RGIn}_{m^i \cdot S}$ for the GPB $\delta_{m^i \cdot S}$ in Δ_P^{Call}, and

The OUT equivalence: RGOut_m^i computed for the GPB δ_m in Δ_P^{Init} is identical to $\mathsf{RGOut}_{m^i \cdot E}$ for the GPB $\delta_{m^i \cdot E}$ in Δ_P^{Call}.

If the IN equivalence holds, then the OUT equivalence holds because by Lemma 9.9, the inlined version Δ_P^{Opt} is same as Δ_P^{SRed}, which is same as Δ_P^{Call} by Lemma 9.8. Thus, our proof obligation reduces to showing the IN equivalence, which is easy to argue using induction on recursion depth i. The base case $i = 1$ represents the first (i.e., "outermost") recursive call. Since no recursive call has been encountered so far, it is easy to see that $\mathsf{RGIn}_m^1 = \mathsf{RGIn}_{1 \cdot S}$. Assuming that it holds for recursion depth i, it also holds for recursion depth $i + 1$ because as explained earlier, the effect of Δ_P^{Opt} is same as Δ_P^{Call} by Lemmas 9.9 and 9.8. For $i = k$, since a fixed point has been reached in both Δ_P^{Init} and Δ_P^{Call}, the absence of recursive call $k + 1$ in Δ_P^{Call} does not matter. □

In the following lemma, we need to consider the contexts of the calls within procedure P. For our reasoning, the way a context is defined does not matter, and we generically denote a context as σ. We assume that σ denotes the full context without any approximation.

Lemma 9.6 (Claim B for Callees of Procedure P in Non-Recursive Case). *Consider a non-recursive procedure P such that all of its transitive callees are also non-recursive. Assume that P calls procedure Q possibly transitively. For any boundary information (possibly containing points-to information and boundary definitions) for P, $\mathsf{TRG}(\Delta_P^{Init}, Q) \Leftrightarrow \mathsf{IRG}(\Delta_P^{Call}, Q)$.*

Proof. There could be multiple paths in the call graph from P to Q. We assume without any loss of generality that these calls of Q have different contexts and boundary information reaching them. Assume that there are i calls to Q with contexts σ_i, $i > 0$. Let the corresponding boundary information (the sets of GPUs reaching the calls) be R_i, $i > 0$. Then, $\mathsf{TRG}(\Delta_P^{Init}, Q)$ analysis would analyze Q separately for these contexts with the corresponding boundary information. Observe that Δ_P^{Call} contains i separate instances of Δ_Q^{Call}, which are analyzed independently by IRG over Δ_P^{Call}. The dataflow information for the statements of Q is a union of the dataflow information reaching these statements in different contexts. Thus, it is sufficient to argue about a particular call to Q from within P independently of other calls from within P.

Consider a particular to call to Q with context σ. We prove the lemma on the length j of the call chain from P to Q for this context. The base case is $j = 1$, representing the situation when

Q is a direct callee of P. Let this call be in the GPB δ_m. Then, from Lemma 9.4, RGIn_m is same as $\mathsf{RGIn}_{m\cdot S}$. TRG analysis of Δ_P^{Init} will visit Δ_Q^{Init} for the context σ with the boundary information RGIn_m. However, IRG analysis of Δ_P^{Call} will analyze the GPG subgraph between $\delta_{m\cdot S}$ to $\delta_{m\cdot E}$ with the dataflow reaching $\mathsf{RGIn}_{m\cdot S}$. Since the information reaching the Start GPB of Q is same in both the cases, the dataflow values for statements in Q for analysis within this context would be identical in both of the analyses proving the base case.

For the inductive step, we assume that the lemma holds for length j of the call chain. To prove the inductive step for length $j + 1$, let the procedure that calls Q be Q'. Then, the length of the call chain from P to Q is m and the lemma holds for Q' by the inductive hypothesis. We can argue about the call to Q from within Q' in a manner similar to the base case described earlier. This proves the inductive step. □

Lemma 9.7 (Claim B for Callees of Procedure P in Recursive Case). *The claim of Lemma 9.6 also holds for recursive procedures.*

Proof. The proof is essentially along the lines of Lemma 9.5 because all we need to argue is that RGIn_m^i computed for the GPB δ_m in Δ_P^{Init} is identical to $\mathsf{RGIn}_{m^i\cdot S}$ for the GPB $\delta_{m^i\cdot S}$ in Δ_P^{Call}. □

Lemma 9.8 (Claim C). *Let Q denote procedure P or its transitive callees. Then, for a given boundary information (possibly containing points-to information and boundary definitions) for procedure P,*

$$\mathsf{IRG}(\Delta_P^{\mathsf{Call}}, Q) \Leftrightarrow \mathsf{IRG}(\Delta_P^{\mathsf{SRed}}, Q).$$

Proof. It is easy to show that the intraprocedural reaching GPUs analysis over Δ_P^{Call} and Δ_P^{SRed} compute the same set of GPUs reaching the corresponding GPBs because the changes made by strength reduction are local in nature—there is no change in the control flow, only the GPUs in GPBs are replaced by equivalent GPUs. Thus, it is sufficient to argue that the effect of the GPUs in a GPB is preserved by strength reduction.

Although the GPBs are not renumbered by strength reduction, it is useful to distinguish between the GPBs before and after strength reduction for reasoning: let the GPB obtained after strength reduction of δ_m be denoted by δ'_m. Then, $\delta'_m = \bigcup_{\gamma\in\delta_m} \gamma \circ \mathsf{RGIn}_m$. Since reaching GPUs analysis is sound (Lemmas 9.12 and 9.13), all relevant producer GPUs reach each δ_m. Hence, from Lemma 9.11, γ is equivalent to $\gamma \circ \mathsf{RGIn}_m$. Thus, it follows that δ'_m is equivalent to δ_m, proving the lemma. □

Lemma 9.9 (Claim D). *Let Q denote procedure P or its transitive callees. Then, for a given boundary information (possibly containing points-to information and boundary definitions) for procedure P,*

$$\forall Q,\ \mathsf{IRG}(\Delta_P^{SRed}, Q) \Leftrightarrow \mathsf{IRG}(\Delta_P^{Opt}, Q).$$

Proof. GPGs Δ_P^{SRed} and Δ_P^{Opt} do not contain any call, and hence the following reasoning holds for all corresponding statements between them regardless of whether they belong to P or a transitive callee of P. We prove the equivalence Δ_P^{SRed} and Δ_P^{Opt} in three steps for the three optimizations:

(1) *Dead GPU elimination*: A GPU whose source is redefined along every path is a dead GPU. Since both the variants of reaching GPUs analysis are sound (Lemmas 9.12 and 9.13), it follows that if $\gamma \notin (\mathsf{RGOut}_{\mathsf{End}} \cup \overline{\mathsf{RGOut}}_{\mathsf{End}} \cup \mathsf{Queued})$ (Section 5), it really has no use within Δ_P^{SRed} or in the GPGs of the callers of P. Hence, removing γ does not change anything in Δ_P^{SRed}.

(2) *Empty GPB elimination*: Since empty GPBs do not influence the reaching GPUs analysis in any way, removing them by connecting their predecessors to their successors does

not change anything (because this transformation preserves the paths through the empty GPBs).

(3) *Coalescing*: This transformation does not add, remove, or simplify any GPU. It only rearranges the GPUs by merging GPBs wherever possible without creating new dependences and without missing any existing dependences. To prove that the GPGs before and after coalescing are identical in terms of their effect on the callers and in terms of the points-to information computed within them, we need to show that the two soundness conditions and the three precision conditions of Section 6.3 are ensured by coalescing:

- *Ensuring soundness*: As described in Section 6.4.1, soundness condition (S1) is ensured by considering only adjacent nodes whose GPUs do not have any dependence between them (steps 1 and 2 in Section 6.4.1), whereas condition (S2) is ensured by computing may-definition sets associated with coalesced GPBs to maintain definition-free paths (step 3 in Section 6.4.1).
- *Ensuring precision*: Precision condition (P1) is ensured by considering only those nodes whose GPUs do not have any dependence between them (step 2 in Section 6.4.1), whereas conditions (P2) and (P3) are satisfied by ensuring that no spurious control flow paths are created: only adjacent nodes (step 1 in Section 6.4.1) for coalescing and coherence ensure that there are no "cross connections" between exits and entries of adjacent parts with multiple entries or exits (Section 6.4.2).

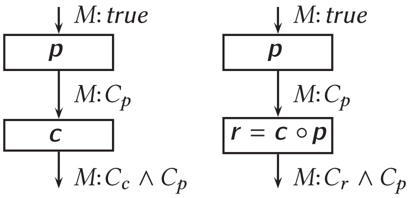
This proves the lemma. □

9.6 Soundness and Precision of Analyses and Operations

Recall that the abstract memory computed by a points-to analysis is a relation $M \subseteq L_P \times L$, where L denotes the set of locations and $L_P \subseteq L$ denotes the set of pointers. Given M, the direct pointees of a set of pointers $X \subseteq L_P$ are computed by the relation application $M\ X = \{y \mid (x, y) \in M, x \in X\}$. Let M^i denote a composition of degree i. 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. By construction, $M^0\{x\} = \{x\}$. Abstract execution of GPU $x \xrightarrow{i|j} y$ in memory M imposes the constraint $M^i\{x\} \supseteq M^j\{y\}$ on M with weak updates; with strong updates, the constraint is stronger: $M^i\{x\} = M^j\{y\}$. Observe that $M^i\{x\} \supseteq M^j\{y\} \Rightarrow M^{i+k}\{x\} \supseteq M^{j+k}\{y\}, k \geq 0$; this also holds for equality (i.e., "=" instead of "$\supseteq$").

Lemma 9.10 (Soundness and Precision of GPU Composition). *Consider GPU composition $\boldsymbol{r} = \boldsymbol{c} \circ^{\tau} \boldsymbol{p}$. Let the source of $\boldsymbol{p}$ be (x, k). Then, if no other GPU with the source $(x, k'), k' \leq k$, reaches $\boldsymbol{c}$, then the abstract executions of $\boldsymbol{r}$ and $\boldsymbol{c}$ are identical in the memory obtained after the abstract execution of $\boldsymbol{p}$.*

Proof. Consider the picture on the right. The memory before the execution of $\boldsymbol{p}$ is M with no constraint, whereas the memory obtained after the execution of $\boldsymbol{p}$ is M with the constraint C_p. The memory obtained after the execution of $\boldsymbol{c}$ and $\boldsymbol{r}$ is M with the constraints $C_c \wedge C_p$ and $C_r \wedge C_p$, respectively. Then, the lemma can be proved by showing that $C_c \wedge C_p$ and $C_r \wedge C_p$ are identical. We first consider *TS* composition.



Initially, assume that $\boldsymbol{c}$ causes a weak update; this assumption can be relaxed later. Let $\boldsymbol{p}$ and $\boldsymbol{c}$ be the GPUs illustrated in the first column of Figure 7. Since no other GPU with the source (x, k'), $k' \leq k$, reaches $\boldsymbol{c}$, the constraint C_p is $M^k\{x\} = M^l\{y\}$. The constraints C_c is $M^i\{z\} \supseteq M^j\{x\}$ and

C_r is $M^i\{z\} \supseteq M^{(l+j-k)}\{y\}$.

$$\begin{aligned}
C_c \wedge C_p &= M^i\{z\} \supseteq M^j\{x\} \wedge M^k\{x\} = M^l\{y\} \\
&\Rightarrow M^i\{z\} \supseteq M^j\{x\} \wedge M^{k+(j-k)}\{x\} = M^{l+(j-k)}\{y\} \wedge M^k\{x\} = M^l\{y\} \quad \text{(adding } j-k \text{ to } C_p\text{)} \\
&\Rightarrow \underbrace{M^i\{z\} \supseteq M^j\{x\} \wedge M^j\{x\} = M^{l+(j-k)}\{y\}} \wedge M^k\{x\} = M^l\{y\} \\
&\Rightarrow M^i\{z\} \supseteq M^{l+j-k}\{y\} \wedge M^k\{x\} \supseteq M^l\{y\} \qquad \text{(combining the first two terms)} \\
&\Rightarrow C_r \wedge C_p
\end{aligned}$$

We also need to prove the implication in the reverse direction to show the equivalence.

$$\begin{aligned}
C_r \wedge C_p &= M^i\{z\} \supseteq M^{l+j-k}\{y\} \wedge M^k\{x\} = M^l\{y\} \\
&= M^i\{z\} \supseteq M^{l+j-k}\{y\} \wedge M^l\{y\} = M^k\{x\} \\
&\Rightarrow M^i\{z\} \supseteq M^{l+j-k}\{y\} \wedge M^{l+(j-k)}\{y\} = M^{k+(j-k)}\{x\} \wedge M^k\{x\} = M^l\{y\} \text{ (adding } j-k \text{ to } C_p\text{)} \\
&\Rightarrow \underbrace{M^i\{z\} \supseteq M^{l+j-k}\{y\} \wedge M^{l+j-k}\{y\} = M^j\{x\}} \wedge M^k\{x\} = M^l\{y\} \\
&\Rightarrow M^i\{z\} \supseteq M^j\{x\} \wedge M^k\{x\} = M^l\{y\} \qquad \text{(combining the first two terms)} \\
&\Rightarrow C_c \wedge C_p
\end{aligned}$$

This proves the lemma for *TS* composition when $\boldsymbol{c}$ perform a weak update. When $\boldsymbol{c}$ performs a strong update, the superset relation "$\supseteq$" is replaced by equality relation "=," and it is easy to see that the two-way implication still holds. Similar arguments can be made for *SS* composition. □

Lemma 9.11 (Soundness and Precision of GPU Reduction). *Consider the set* Red $= \boldsymbol{c} \circ \mathcal{R}$. *Let M be the memory obtained after executing the GPUs in $\mathcal{R}$. Then, the execution of the GPUs in* Red *in M is identical to the execution of the GPU $\boldsymbol{c}$ in M.*

Proof. Recall that Red is the fixed point of function GPU_reduction(Red, $\mathcal{R}$) (Definition 4) with the initial value Red $= \{\boldsymbol{c}\}$. As explained in Section 4.3, this computation is monotonic and is guaranteed to converge. Hence, this lemma can be proved by induction on step i in the fixed-point iteration that computes Red^i. The base case is $i = 0$, which follows trivially because Red$^0 = \{\boldsymbol{c}\}$.

For the inductive hypothesis, assume that the lemma holds for Red^i. For the inductive step, we observe that Red^{i+1} is computed by reducing the GPUs in Red^i by composing them with those in $\mathcal{R}$. Consider the composition of a GPU $\gamma_1 \in$ Red^i with a GPU $\boldsymbol{p} \in \mathcal{R}$ such that $\gamma_2 = \gamma_1 \circ \boldsymbol{p}$; then, $\gamma_2 \in$ Red^{i+1}. Let the source of γ_1 be (x, i). Then, from Lemma 9.10, γ_2 can replace γ_1 if $\mathcal{R}$ does not contain any GPU γ_1' with a source (x, i'), where $i' \leq i$. Thus, there are two cases to consider:

- There is no GPU γ_1' in $\mathcal{R}$ with a source (x, i'), $i' \leq i$. Then, Red$^{i+1} = \big(\text{Red}^i - \{\gamma_1\}\big) \cup \{\gamma_2\}$. Since the execution of the GPUs in Red^i is identical to that of $\boldsymbol{c}$ by the inductive hypothesis, the execution of the GPUs in Red^{i+1} is identical to that of $\boldsymbol{c}$.
- There is a GPU γ_1' in $\mathcal{R}$ with a source (x, i'), $i' \leq i$. Then, γ_1 will be composed with γ_1' also giving some simplified GPU γ_2'. Assume that γ_1' is the only such GPU in $\mathcal{R}$, then Red^{i+1} is computed as Red$^{i+1} = \big(\text{Red}^i - \{\gamma_1\}\big) \cup \{\gamma_2, \gamma_2'\}$. Since the execution of the GPUs in Red^i is identical to that of $\boldsymbol{c}$ by the inductive hypothesis, the execution of the GPUs in Red^{i+1} is identical to that of $\boldsymbol{c}$.

  Replacement of γ_1 by the simplified GPUs is sound only when it is composed with all possible GPUs with which it has a RaW dependence. This requires us to argue the following:

  - The source (x, i') used for reducing γ_1 for computing γ_2 is defined along every path reaching the node. This follows from the property of completeness of $\mathcal{R}$ (Section 4.3), which is trivially ensured by (a) reaching GPUs analysis without blocking (Section 4.5) because of the presence of boundary definitions at the Start, and by (b) reaching GPUs analysis with blocking (Section 4.6) because of the presence of boundary definitions at the Start, and their reintroduction when some GPUs are blocked.

- Reaching GPUs analyses are sound and precise—that is, no GPU on which γ_1 may have a RaW dependence is missed from $\mathcal{R}$, nor does $\mathcal{R}$ contain a spurious GPU. This follows from Lemmas 9.12 and 9.14.

Thus, the execution of the GPUs in Red^{i+1} is identical to that of $\boldsymbol{c}$, thereby proving the inductive step.

Hence the lemma. □

Lemma 9.12 shows the soundness of reaching GPUs analysis without blocking. The soundness of reaching GPUs analysis with blocking is shown in Lemma 9.13. Lemma 9.14 shows the precision of reaching GPUs analyses by arguing that every GPU that reaches a GPB is either generated by a GPB or is a boundary definition.

Lemma 9.12 (Soundness of Reaching GPUs Analysis Without Blocking). *Consider a GPU $\gamma : x \xrightarrow[s]{i|j} y$ obtained after the strength reduction of the GPUs in δ_l using the simplified GPUs reaching δ_l. Assume that there is a control flow path from δ_l to δ_m along which the source (x, i) is not strongly updated. Then, γ reaches δ_m.*

Proof. We prove the lemma by induction on the number of nodes k between δ_l and δ_m. The basis is $k = 0$ when δ_m is a successor of δ_l. Since γ has been obtained after strength reduction, of the GPUs in δ_l, $\gamma \in \text{RGOut}_l$. Since RGIn_m is a union of RGOut of all predecessors (Definition 5), it follows that $\gamma \in \text{RGIn}_m$.

For the inductive hypothesis, assume that the lemma holds when there are k nodes between δ_l and δ_m. To prove that it holds for $k + 1$ nodes between them, let the k^{th} node be δ_n. Then, $\gamma \in \text{RGIn}_n$ by the inductive hypothesis. Since δ_n does not strongly update the source (x, i), it means that $\gamma \notin \text{RGKill}_n$ and thus $\gamma \in \text{RGOut}_n$. Since δ_m is a successor of δ_n, it follows that $\gamma \in \text{RGIn}_m$, proving the inductive step, and hence the lemma. □

Lemma 9.13 (Soundness of Reaching GPUs Analysis with Blocking). *Let GPU $\gamma : x \xrightarrow[s]{i|j} y$ be obtained after the strength reduction of the GPUs in δ_l using the simplified GPUs reaching δ_l. Assume that there is a control flow path from δ_l to δ_m along which the source (x, i) is neither strongly updated nor blocked. Then, γ reaches δ_m.*

Proof. The proof is similar to that of Lemma 9.12, but now we additionally reason about Blocked (I, G) (Definition 7). □

Lemma 9.14 (Precision of Reaching GPUs Analysis). *If a GPU γ reaches a GPB δ_l, then there must be a GPB δ_m such that there is a control flow path from δ_m to δ_l that does not kill or block γ and either δ_m is* Start *and γ is a boundary definition, or γ is generated in δ_m due to GPU reduction.*

Proof. Without any loss of generality, we generically use RGIn/RGOut to represent both variants of reaching GPUs analysis. We prove the lemma by induction on the number of iterations in the Gauss-Seidel method of fixed-point computation (the dataflow values in iteration $i + 1$ are computed only from those computed in iteration i). The basis is $i = 1$ when RGIn is $\emptyset$ by initialization for each node (except for Start, for which it contains the boundary definitions). Hence, the lemma holds vacuously.

For the inductive hypothesis, assume that the lemma holds for iteration i. Consider a GPU γ in RGIn_l in iteration $(i + 1)$. Then, γ must be in RGOut_m for some predecessor δ_m of δ_l in some iteration $i' < i$. If it is generated by GPU reduction, then the lemma is proved. If not, then it must be the case that it reached RGIn_m in iteration $i' < i$ and was neither killed nor blocked in δ_m. By the inductive hypothesis, it is either a boundary definition reaching from Start or there exists some GPB δ_n that generated it after the reduction. Hence, it follows in iteration $i + 1$ that γ is either a boundary definition reaching from Start or there exists some GPB δ_n that generated it after the reduction. This proves the lemma. □

10 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. Thus, the size of a summary can potentially be large. Further, the transitive inlining of the summaries of the callee procedures can increase the size of a summary exponentially, thereby hampering the scalability of analysis.

Section 10.1 describes our implementation, Section 10.2 describes our measurements that include comparisons with client analyses, and Section 10.3 analyzes our observations and describes the lessons learned.

10.1 Implementation and Experiments

We 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 of RAM with eight 64-bit Intel i7-7700 CPUs running at 4.20 GHz. The implementation can be downloaded from https://github.com/PritamMG/GPG-based-Points-to-Analysis.

10.1.1 Modeling Language Features. Our method eliminates all non-address-taken local variables[12] using def-use chains explicated by the SSA-form; this generalizes the technique in Section 3.3.1 that removes compiler-added temporaries. 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 *only* of GPUs that involve global variables,[13] parameters of the procedure, and the return variable that is visible in the scope of its callers. 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 heap memory using context-insensitive allocation-site-based abstraction and by maintaining k-limited indirection lists of field dereferences for $k = 3$ (see Appendix B) for heap locations that are live on entry to a procedure. An array is treated index insensitively. 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.

[12] An address-taken variable is a global or stack-allocated variable to which the C *address-of* operator, "&," is applied.
[13] From now on, we also regard heap-summary nodes and address-taken local variables as "variables."

10.1.2 Variants of Points-to Analysis Implemented. For comparing the precision of our analysis, we implemented the following variants. For convenience, we implemented them using GPUs and not by using special data structures for efficient flow-insensitive analysis:

- *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 between the GPUs in the store. 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 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—that is, flow-sensitive and context-insensitive (FSCI) points-to analysis—can be modeled by constructing a supergraph by joining the CFGs 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.

10.1.3 Client Analyses Implemented. We implemented mod-ref analysis and call-graph construction to measure the effectiveness of our points-to analysis. The mod-ref analysis computes interprocedural reference and modification side effects for each variable caused by a procedure on the callers of the procedure. Call graph represents caller-callee relationships between the procedures in a program. Standard compilers like GCC and LLVM construct call graphs for only direct calls and do not resolve the calls through function pointers. We constructed the call graph that includes the effect of indirect calls using the points-to information computed for function pointers.

10.1.4 Comparison with Other Points-to Analysis. We also computed corresponding data for client analyses using *static value flow analysis* (SVF) [41].[14] SVF is used for comparison because its implementation is readily available. SVF is a static analysis framework implemented in LLVM that allows value-flow construction and context-insensitive pointer analysis to be performed in an iterative manner (sparse analysis performed in stages, from cheap overapproximate analysis to precise expensive analysis). It uses the points-to information from Andersen's analysis and constructs an interprocedural memory SSA (Static Single Assignment) form where def-use chains of both top-level (i.e., non-address-taken) and address-taken variables are included. The scalability and precision of the analysis is controlled by designing memory SSA that allows users to partition memory into a set of regions.

10.2 Measurements

This section describes the evaluations made on SPEC CPU 2006 benchmarks. The characteristics of benchmark programs in terms of number of procedures, number of pointer assignments, and the number of call sites is given in Table 1.

[14] Downloaded commit 03c6eb0 of SVF for LLVM 9.0 from https://github.com/SVF-tools/SVF on October 23, 2019.

Table 1. Benchmark Characteristics Relevant to Our Analysis

Program	kLoC	No. of Statements Involving Pointers	No. of Call Sites	No. of Procedures	Proc. Count for Different Buckets of No. of Calls			
					2–5	6–10	11–20	21+
	A	*B*	*C*	*D*	*E*			
lbm	0.9	370	30	19	5	0	0	0
mcf	1.6	480	29	23	11	0	0	0
libquantum	2.6	340	277	80	24	11	4	3
bzip2	5.7	1,650	288	89	35	7	2	1
milc	9.5	2,540	782	190	60	15	9	1
sjeng	10.5	700	726	133	46	20	5	6
hmmer	20.6	6,790	1,328	275	93	33	22	11
h264ref	36.1	17,770	2,393	566	171	60	22	16
gobmk	158.0	212,830	9,379	2,699	317	110	99	134

Column E omits procedures with a single call.

(We measured the data for the following two categories of evaluations:

- Comparing the precision of GPG-based FSCS points-to analysis and SVF-based points-to analysis (Section 10.2.1):
  - —effect on mod-ref analysis and call graph construction, and
  - —number of points-to pairs per procedure.
- Measuring the effectiveness of GPG-based FSCS points-to analysis (Section 10.2.2):
  - —effectiveness of control flow optimizations,
  - —quality of procedure summaries in terms of reusability, compactness (both absolute and relative), and
  - —precision gain over FICS and FICI points-to analyses (in terms of number of points-to pairs per procedure). For our FSCS analysis, we compute this number by adding all points-to pairs computed as described in Section 8 across all procedures in a benchmark and then divide it by the number of procedures.

10.2.1 Comparison of GPG-Based and SVF Analyses. We compared the data for mod-ref analysis and call graph for GPG-based points-to analysis with that of SVF. However, the comparison is not straightforward, because the two implementations use two differently engineered intermediate representations of programs. The underlying compiler frameworks (GCC and LLVM) use different strategies for function inlining and function cloning (for creating specialized versions of the same functions), leading to a different number of procedures in the call graph for the same benchmark program. Although we suppressed the two optimizations in GCC with the appropriate flags, GCC continues to perform function inlining and cloning at a smaller scale, indicating that we do not have a direct control over the IR. We therefore make the comparison only on the common part of the benchmark programs:

- *Call graph construction*: Table 2 provides the call graph nodes (number of functions in a program) and the call graph edges (representing the caller-callee relationships between functions). The table also provides the number of monomorphic calls (single callee at a call site) and polymorphic calls (multiple callees at a call site) for indirect calls. An approach A_1 is said to be more precise than approach A_2 if the number of call graph edges computed by A_1 is smaller than that computed by A_2 (given that the number of nodes in the call graph are

Table 2. Call Graph Statistics for the Common Part of the Program as Discovered by GCC (for GPG-Based Analysis) and LLVM (for SVF-Based Analysis)

Program	No. of Call Graph Nodes		No. of Call Graph Edges		No. of Monomorphic Calls		No. of Polymorphic Calls	
	GPG	SVF	GPG	SVF	GPG	SVF	GPG	SVF
lbm	19	19	20	20	0	0	0	0
mcf	23	23	26	26	0	0	0	0
libquantum	80	80	156	156	0	0	0	0
bzip2	88	88	140	144	22	20	3	5
milc	174	174	434	434	0	0	0	0
sjeng	121	121	367	367	0	0	1	1
hmmer	263	263	709	723	7	0	2	9
h264ref	560	560	1,231	1,521	9	1	343	351
gobmk	2,679	2,679	8,889	8,889	0	0	44	44

Table 3. Mod/Ref Statistics

Program	No. of Calls with Mod			No. of Calls with Ref			No. of Mods Across All Calls			No. of Refs Across All Calls		
	GPG	SVF	RR	GPG	SVF	RR	GPG	SVF	RR	GPG	SVF	RR
lbm	2	6	0.33	7	7	1.00	3	6	0.50	12	16	0.75
mcf	8	16	0.50	13	21	0.62	9	58	0.16	21	108	0.19
libquantum	13	60	0.22	22	64	0.34	14	208	0.07	41	224	0.18
bzip2	16	30	0.53	61	181	0.34	30	150	0.20	36	128	0.28
milc	31	63	0.49	18	94	0.19	36	228	0.16	30	650	0.05
sjeng	32	42	0.76	39	70	0.56	93	291	0.32	128	455	0.28
hmmer	32	152	0.21	126	207	0.61	173	896	0.19	1091	1384	0.79
h264ref	183	204	0.90	178	402	0.44	2607	2722	0.96	4232	7342	0.58
gobmk	105	622	0.17	261	681	0.38	10194	13426	0.76	5225	27842	0.19
Geometric mean			0.39			0.41			0.27			0.27

RR denotes the reduction ratio of GPG-based analysis over SVF-based analysis computed by dividing the counts for the GPG-based method by the counts for the SVF-based method. A value smaller than 1.00 indicates that GPG-based analysis is more precise than SVF-based analysis, and the smaller the value, the higher the precision.

same). In addition, A_1 is more precise than A_2 if it discovers more monomorphic calls than A_2.

Table 2 shows that lbm, mcf, libquantum, and milc have no function pointers (indicated by zero counts for polymorphic and monomorphic calls for function pointers). Since we compared only common parts in the IRs of both GCC and LLVM, these benchmarks have identical call graphs. In addition, there is no difference in the precision of call graphs for sjeng (one polymorphic indirect call) and gobmk (44 polymorphic indirect calls). However, the data shows that our approach finds a larger number of monomorphic calls in hmmer and h264ref than the SVF method. This reduces the number of edges in the call graph.

- *Mod-ref analysis*: Table 3 provides the number of calls to procedures in which a pointer variable was either modified or referenced. It also gives the number of pointer variables (globals, parameters, and heap locations) that are modified and referenced across all calls. An approach A_1 is said to be more precise than approach say A_2, if the numbers computed

Table 4. Final Points-to Information: Average Points-to Pairs per Procedure

Program	No. of Proc.	No. of Stmts.	FSCS	FICI	FICS	SVF
lbm	19	367	0.05	3.26	2.11	0.21
mcf	23	484	0.63	8.13	7.39	0.92
libquantum	80	396	0.12	3.99	2.42	0.28
bzip2	89	1,645	0.18	4.72	2.94	2.44
milc	190	2,467	0.29	3.43	2.87	1.05
sjeng	133	684	0.42	1.12	1.9	0.39
hmmer	275	6,717	0.07	5.10	1.52	3.44
h264ref	566	17,253	0.49	5.02	3.08	0.41
gobmk	2,699	10,557	0.24	2.95	1.39	7.58
Geometric mean			0.21	3.74	2.51	0.94

FSCS, FICI, FICS, and SVF analysis.

in this table by A_1 are smaller than the numbers computed by A_2. Table 3 shows that our approach is more precise than the SVF method across all benchmarks. The geometric mean of the reduction ratio of GPG over SVF in the number of calls is 0.39 with a geometric standard deviation of 1.81. The same numbers for ref are 0.41 and 1.60, respectively. The geometric mean of the reduction ratio of the number of mods across all calls is 0.27 with a geometric standard deviation of 2.34. The same numbers for ref are 0.27 and 2.43, respectively.

- *Average points-to information*: A smaller value of average points-to pairs per procedure indicates higher precision. Table 4 shows that in general, the average points-to pairs per procedure for GPG-based points-to analysis is substantially smaller than that of SVF. More specifically, the geometric mean of average points-to pairs per procedure in GPG-based FSCS points-to analysis is 0.21 with a geometric standard deviation of 2.40. The number of average points-to pairs is larger for mcf, sjeng, and h264ref because they contain a large number of heap pointers and we used simple context-insensitive allocation-site-based heap abstraction. For SVF-based points-to analysis, the geometric mean of average points-to pairs per procedure is much larger at 0.94 with a still larger geometric standard deviation of 3.43. This is expected because SVF is context insensitive. However, the numbers are smaller for SVF for sjeng and h264ref benchmarks perhaps because they have a better handling of heap. For FICS, average points-to information is much larger with a geometric mean of 2.51. As expected, it is maximum for FICI with a geometric mean of 3.74.

10.2.2 Data for Points-to Analysis Using GPGs. We describe our observations about the sizes of GPGs, GPG optimizations, and performance of the analysis. Data related to the time measurements are presented in Section 10.2.3. Section 10.3 discusses these observations by analyzing them. Our observations include the following:

- *Effectiveness of control flow minimization*: The effectiveness of control flow minimization optimizations is presented in Table 5. The data is represented in terms of percentage of dead GPUs, percentage of empty GPBs, percentage of GPBs reduced because of coalescing, and percentage of back edges removed because of coalescing.

  We compute the percentage of dead GPUs 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). We then create five buckets that associate the number of procedures having percentage of dead GPUs within a given range. Similarly, we create buckets for the percentage of empty GPBs, percentage of GPBs reduced because of coalescing, and percentage of back edges removed because of coalescing as shown in Table 5. We observe the following:

(a) The percentage of dead GPUs is very small, and the dead GPU elimination optimization is the least effective of all optimizations. For most procedures, less than 20% of the GPUs are eliminated as dead. The geometric mean of the percentage of procedures for this bucket is 97.01% with a geometric standard deviation of 1.04 across different benchmarks. The absolute numbers for this optimizations are very small because the number of candidate procedures for this optimization is small—as shown in Table 6, a large number of GPGs have zero GPUs and a small number GPGs are $\Delta_\top$ because they are disconnected in that the corresponding CFGs have an exit node with no successors.

(b) The transformations performed by call inlining, strength reduction, and dead GPU elimination create empty GPBs which are removed by empty GPB elimination. For most procedures, 0% to 5% or close to 50% of GPBs are empty. More specifically, the geometric mean of the percentage of procedures for empty GPB elimination in the bucket of 41% to 60% is 65.7% with a geometric standard deviation of 1.30. For the 0% to 20% bucket, the same numbers are 25.09 and 1.86, respectively.

(c) Coalescing was most effective for recursive procedures whose GPGs are constructed by repeated inlinings of recursive calls. 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 in the most critical GPGs. Hence, we observe a reduction of 20% to 50% of GPBs for some but not the majority of procedures. More specifically, the geometric mean of the percentage of procedures that undergo a reduction of less than 20% is 91.6% with a geometric standard deviation of 1.09. Although this number may look small, it should be noted that without coalescing, the GPGs became too large and our implementation failed to scale. In other words, the transitive effect of coalescing is significant and is presented in the discussion on relative sizes of GPGs before and after optimizations for measuring the quality of procedure summaries. In any case, coalescing eliminated almost all back edges as shown in the table. 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.

- *Quality of procedure summaries*: This data is presented in Tables 1, 5, and 6. We use the following quality metrics on procedure summaries:

  (a) *Reusability*: The number of calls to a procedure is a measure for the reusability of its summary. The construction of a procedure summary is meaningful only if it is used multiple times. From column E in Table 1, it is clear that most procedures are called from many call sites. We counted only the procedures that were called multiple times, ignoring the procedures that have only one call.

  (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.

  Tables 7 and 6 show that the procedure summaries are indeed small in terms of the number of GPBs and GPUs. GPGs for a large number of procedures have zero GPUs because they do not manipulate global pointers (and thereby represent the identity flow function). More specifically, the geometric mean of the percentage of procedures with

Table 5. Effectiveness of Control Flow Minimization

Program	Count of Procedures in Each Bucket of Percentages																		
	Dead GPUs (%)					Empty GPBs Eliminated (%)					GPBs Reduced Because of Coalescing (%)					Back Edges Reduced Because of Coalescing (%)			
	0–20	21–40	41–60	61–80	81–100	0–20	21–40	41–60	61–80	81–100	0–20	21–40	41–60	61–80	81–100	0–20	21–40	40–80	81–100
lbm	4	0	0	0	0	4	0	15	0	0	19	0	0	0	0	0	0	0	0
mcf	10	1	0	0	0	14	0	8	0	1	17	2	3	1	0	1	0	0	3
libquantum	10	0	0	0	0	10	0	68	2	0	77	2	1	0	0	0	0	0	1
bzip2	11	0	0	0	0	23	0	58	0	0	75	4	2	0	0	0	0	0	1
milc	12	0	0	0	0	60	0	126	1	0	184	1	2	0	0	0	0	0	1
sjeng	29	1	2	0	0	10	0	99	18	4	124	1	3	2	1	0	0	0	9
hmmer	32	0	1	0	0	100	0	170	0	0	234	11	15	8	2	3	0	0	0
h264ref	123	2	1	1	1	207	2	331	18	5	523	12	17	9	2	4	0	1	4
gobmk	549	2	0	0	0	701	0	1,952	40	4	2,478	72	46	67	34	33	2	6	63
Geometric mean	97.01%					25.09%		65.7%			91.6%								

The geometric mean has been shown for the percentage of procedures in the buckets with the largest numbers. The percentages for dead GPU elimination is computed against a much smaller number of procedures (obtained by omitting the procedures that have zero GPUs).

Table 6. Miscellaneous Data

Program	No. of TotalProc.	No. of Stmts.	No. of Proc. with Zero GPUs	No. of Proc. Whose GPG is $\Delta_\top$	No. of Proc. Containing Back Edgesin the CFG	No. of Proc. Containing Back Edgesin the GPG	No of Queued GPUs Computed When GPU Compositions Were Postponed
lbm	19	367	15	0	10	0	0
mcf	23	484	12	0	20	1	115
libquantum	80	396	70	0	36	0	0
bzip2	89	1,645	70	8	43	0	0
milc	190	2,467	175	3	92	0	0
sjeng	133	684	99	2	65	0	0
hmmer	275	6,717	237	5	153	0	9
h264ref	566	17,253	435	3	308	8	15
gobmk	2,699	10,557	2,146	2	464	45	3

Table 7. Measurement of the Quality of Procedure Summaries

Program	Procedure Count for Different Buckets of No. of GPBs						Procedure Count for Different Buckets of No. of GPUs							
	0	1–3	4–10	11–25	26–35	36+	0	1–3	4–6	7–10	11–30	31–50	51–70	71+
lbm	0	18	1	0	0	0	15	4	0	0	0	0	0	0
mcf	0	22	1	0	0	0	12	6	2	2	0	0	0	1
libquantum	0	80	0	0	0	0	70	8	2	0	0	0	0	0
bzip2	8	79	2	0	0	0	70	11	5	3	0	0	0	0
milc	3	186	1	0	0	0	175	7	6	2	0	0	0	0
sjeng	2	130	1	0	0	0	99	26	3	3	2	0	0	0
hmmer	5	253	13	3	1	0	237	29	4	5	0	0	0	0
h264ref	3	544	15	4	0	0	435	81	20	8	17	3	1	1
gobmk	2	2,514	150	9	0	24	2,146	75	16	361	63	37	1	0
Geo. Mean		95.06%					77.63%	18.08%						

The geometric mean of percentages of procedures with 1 to 3 GPBs is 95.06%, whereas that of procedures with 0 GPUs and 1 to 10 GPUs is 77.63% and 18.08%, respectively. Some procedures have zero GPBs because they have an exit node with no successors.

zero GPUs across all benchmarks is 77.63% with a geometric standard deviation of 1.18. The geometric mean of the percentages of procedures with 1 to 10 GPUs is 18.08% with a geometric standard deviation of 1.61. Further, the majority of GPGs have 1 to 3 GPBs; the geometric mean of the percentages of such procedures is 95.06%.

Note that this is an absolute size of GPGs. Since the relative sizes were measured on several parameters, the associated observations are presented separately in the following.

- *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: (1) the CFG of a procedure (with a cumulative effect of call inlining), (2) the initial GPG obtained after call inlining, and (3) 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 7.2). The number of statements in a CFG is measured only in terms of the pointer assignments.

  The data is represented in terms of ratio $u = (x/y) \times 100$ (rounded to the nearest integer), where x and y represent the following:

  (a) x is the number of GPBs/GPUs/control flow edges in a GPG obtained after call inlining, and y is the number of basic blocks/pointer statements/control flow edges in the CFG after call inlining.

  (b) x is the number of GPBs/GPUs/control flow edges in a GPG after all optimizations, and y is the number of basic blocks/pointer statements/control flow edges in the CFG.

  (c) x is the number of GPBs/GPUs/control flow edges in a GPG after all optimizations, and y is the number of GPBs/GPUs/control flow edges in a GPG obtained after call inlining.

  We then create five buckets that associate the number of procedures having the computed ratio within a given range. This data is presented in Table 8 (in terms of GPBs and basic blocks), Table 9 (in terms of GPUs and pointer assignments), and Table 10 (in terms of control flow edges) and is described next:

Table 8. Relative Size of GPGs with Respect to Corresponding Procedures in Terms of GPBs and Basic Blocks

A:	Procedure count for different buckets of ratio of GPBs/BBs in CFG and GPG after inlining														
B:	Procedure count for different buckets of ratio of GPBs/BBs in CFG and optimized GPG														
C:	Procedure count for different buckets of ratio of GPBs in GPG before and after optimizations														
Program	A					B					C				
	0–20	21–40	41–60	61–80	81+	0–20	21–40	41–60	61–80	81+	0–20	21–40	41–60	61–80	81+
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	14	1	0
libquantum	42	14	12	1	11	56	13	11	0	0	26	17	36	0	1
bzip2	53	16	10	4	6	71	12	6	0	0	13	4	70	1	1
milc	115	20	14	7	34	134	22	34	0	0	10	6	169	1	4
sjeng	87	17	7	3	19	105	9	19	0	0	19	13	99	1	1
hmmer	205	34	18	1	17	239	19	16	0	1	62	32	164	8	9
h264ref	401	71	49	10	35	476	51	38	1	0	46	79	412	17	12
gobmk	2,336	275	24	6	58	2,610	29	56	1	3	210	163	2,038	235	53
Geo. mean	63.3%					79.13%							69.31%		

The geometric mean has been shown for the percentages of procedures in buckets with the largest numbers.

Table 9. Relative Size of GPGs with Respect to Corresponding Procedures in Terms of GPUs and Pointer Assignments

A:	Procedure count for different buckets of ratio of GPUs/stmts in a CFG and GPG after inlining														
B:	Procedure count for different buckets of ratio of GPUs/stmts in a CFG and an optimized GPG														
C:	Procedure count for different buckets of ratio of GPBs in a GPG before and after optimizations														
Program	A					B					C				
	0–20	21–40	41–60	61–80	81+	0–20	21–40	41–60	61–80	81+	0–20	21–40	41–60	61–80	81+
lbm	16	3	0	0	0	19	0	0	0	0	18	0	0	1	0
mcf	21	0	0	1	1	23	0	0	0	0	17	0	3	0	3
libquantum	75	4	0	0	1	80	0	0	0	0	47	1	1	0	31
bzip2	89	0	0	0	0	89	0	0	0	0	85	0	0	0	4
milc	189	1	0	0	0	190	0	0	0	0	185	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	6	1	0	2
h264ref	540	12	10	1	3	563	2	1	0	0	505	3	1	1	56
gobmk	2,688	4	2	0	5	2,697	1	1	0	0	2,189	0	4	7	499
Geo. mean	95.59%					99.93%					84.14%				

The geometric mean has been shown for the percentages of procedures in buckets with the largest numbers.

(a) Column A 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 initial GPGs are in the range of 0% to 20% of the corresponding CFGs. The geometric mean of the percentage of procedures in this bucket for relative size in terms of GPUs and pointer assignments across all benchmarks is 95.59% with a geometric standard deviation of 1.09 (Table 9). The same number for relative size in terms of GPBs and basic blocks is 63.3% with a geometric standard deviation of 1.2 (Table 8), and those for relative size in terms of control flow edges is 86.13% with a geometric standard deviation of 1.12 (Table 10).

Table 10. Relative Size of GPGs with Respect to Corresponding Procedures in Terms of Control Flow Edges

A:	Procedure count for different buckets of ratio of control flow edges in a CFG and GPG after inlining														
B:	Procedure count for different buckets of ratio of control flow edges in a CFG and an optimized GPG														
C:	Procedure count for different buckets of ratio of GPBs in a GPG before and after optimizations														
Program	A					B					C				
	0–20	21–40	41–60	61–80	81+	0–20	21–40	41–60	61–80	81+	0–20	21–40	41–60	61–80	81+
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	4	2	1	0
libquantum	61	8	2	0	9	80	0	0	0	0	78	1	0	0	1
bzip2	72	9	2	2	4	89	0	0	0	0	79	7	1	1	1
milc	180	3	5	0	2	189	0	1	0	0	182	1	3	0	4
sjeng	124	5	1	0	3	133	0	0	0	0	130	2	0	0	1
hmmer	246	24	3	0	2	274	1	0	0	0	252	8	1	5	9
h264ref	509	26	13	1	17	562	0	2	1	1	516	15	14	11	10
gobmk	2,572	72	31	1	23	2,693	1	2	1	2	2,336	43	92	162	66
Geo. mean	86.13%					99.8%					89.97%				

The geometric mean has been shown for the percentages of procedures in buckets with the largest numbers.

(b) Column B 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% to 20% is larger in this column than in column A, indicating more reduction because of optimizations. The geometric mean of the percentage of procedures in this bucket for relative size in terms of GPUs and pointer assignments across all benchmarks is a whopping 99.93% with a geometric standard deviation of 1.0 (Table 9). The same number for relative size in terms of GPBs and basic blocks is 79.13% with a geometric standard deviation of 1.18 (Table 9), and those for relative size in terms of control flow edges is a whopping 99.8% with a geometric standard deviation of 1.0 (Table 10). We believe that this is the key to the scalability gain of GPG-based points-to analysis over top-down context-sensitive points-to analysis.

(c) Column C 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 maximum reduction is found for control flow—a large number of procedures fall in the range 0%-20% and the number is larger than in this range for GPBs or GPUs indicating that the control flow is optimized the most. The geometric mean of percentage of procedures in this bucket for relative size in terms of GPUs and pointer assignments across all benchmarks is 84.14% with a geometric standard deviation of 1.18 (Table 9). The same number for relative size in terms of GPBs and basic blocks is 69.31% with a geometric standard deviation of 1.23 (Table 8), and those for relative size in terms of control flow edges is 89.97% with a geometric standard deviation of 1.11 (Table 10).

We also measured the effect of control flow minimization on the number of back edges that get removed because fixed-point computation requires a larger number of iterations in the presence of back edges. The data in Table 6 shows that most of the GPGs are acyclic despite the fact that the number of procedures with back edges in the CFG is large.

Table 11. Time Measurements

Program	Time (in Seconds)	
	FSCS (with Blocking)	SVF
lbm	0.070	0.01
mcf	8.690	0.062
libquantum	1.514	0.031
bzip2	1.066	0.534
milc	1.133	0.236
sjeng	3.702	0.131
hmmer	4.961	2.032
h264ref	73.779	1.852
gobmk	938.949	17.959

- *Precision gain of GPG-based FSCS over FICS, and FICI points-to analyses*:

  We compared the points-to information computed by our approach with FICI and FICS methods. For this purpose, we computed number of points-to pairs per procedure in all 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 the FICI method. The summary statistics for this were presented in Section 10.2.1.

10.2.3 Time Measurements. We measured the overall time (Table 11). We also measured the time taken by the SVF points-to analysis. We observe that our analysis takes less than 16 minutes on gobmk.445, which is a large benchmark with 158 kLoC. Our current implementation does not scale beyond that. SVF is faster than all variants of points-to analysis that we implemented. This is expected because SVF is context insensitive.

10.3 Discussion: Lessons from Our Empirical Measurements

Our experiments and empirical data leads us to some important learnings as described next:

(1) 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.
(2) For scalability, the bottom-up summaries must be kept as small as possible at each stage.
(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 learned these lessons the hard way in the situations described in the rest of this section.

10.3.1 Handling a 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 simultaneously 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 (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.

10.3.2 Handling Arrays and SSA Form. Pointers to arrays were weakly updated, and 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.

10.3.3 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.

11 RELATED WORK: THE BIG PICTURE

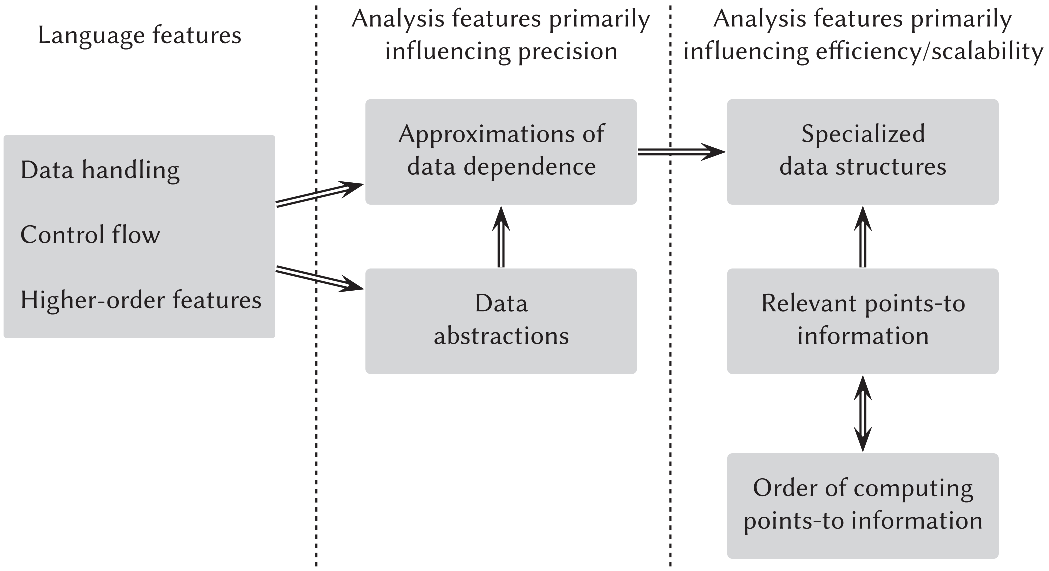
Many investigations reported in the literature have described the popular points-to analysis methods and have presented a comparative study of the methods with respect to scalability and precision [14, 15, 17, 24, 35, 38]. Instead of discussing these methods, we devise a metric of features that influence 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.

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

Figure 20 presents our metric. At the top level, we have language features and analysis features. 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 categorization of language features is obvious. Here we describe our categorization of analysis features.

11.1.1 Features Influencing Precision. Two important sources of imprecision in an analysis are approximation of data dependence and abstraction of data.

- *Approximations of data dependence*: The approaches that compromise on control flow by using flow insensitivity or context insensitivity overapproximate the control flow: flow


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

Fig. 20. 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.

insensitivity admits arbitrary orderings of statements, whereas context insensitivity treats call and returns as simple goto statements admitting interprocedurally invalid paths.
Observe that honoring control flow in imperative languages preserves data dependence, and its overapproximation causes overapproximation of data dependence. This may introduce spurious data dependences, causing imprecision.
Note that SSA form also discards control flow, but it avoids overapproximation in data dependences by creating use-def chains between renamed variables.

- *Data abstractions*: An abstract location usually represents a set of concrete locations. An overapproximation of this set of locations leads to spurious data dependences, causing imprecision in points-to analysis.

11.1.2 Features Influencing Efficiency and Scalability. Different methods use different techniques to achieve scalability. We characterize them based on the following three criteria:

- *Relevant points-to information.* Many methods choose to compute a specific kind of points-to information that is then used to compute further points-to information. For example, staged points-to analyses begin with conservative points-to information that is then made more precise. Similarly, some methods begin by computing points-to information for top-level (i.e., non-address-taken) pointers whose indirections are then eliminated. This uncovers a different set of pointers as top-level pointers whose points-to information is then computed.
- *Order of computing points-to information*: Most methods order computations based on relevant points-to information that may also be defined in terms of a chosen order of traversal over the call graph (e.g., top-down or bottom-up).
- *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 modeling relevant points-to information (e.g., use of placeholders to model accesses of unknown pointees in a bottom-up method).

11.1.3 Interaction Between the Features. In this section, we explain the interaction between the features indicated by the arrows in Figure 20:

- *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, and allocation-site-based abstraction. This overapproximation creates spurious data dependence between the concrete locations represented by the 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 [46], 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 LFCPA [21], only the live pointers are relevant. Thus, points-to information is computed only when the liveness of pointers is generated. The order of computing liveness influences the relevant points-to information to be computed.

11.1.4 Our Work in the Context of the Big Picture of Points-to Analysis. GPG-based points-to analysis preserves data dependence by being flow- and context sensitive. It is path insensitive and uses SSA form for non-address-taken local variables. Unlike the approaches that overapproximate control flow indiscriminately, we discard control flow as much as possible, but only when there is a guarantee that it does not overapproximate data dependence.

Our analysis is field sensitive. It overapproximates arrays by treating all of its elements alike. We use context-insensitive 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 of the information is available in the context. Our analysis computes points-to information for all pointers.

11.2 Approaches of Constructing Procedure Summaries

There is a large body of work on flow-insensitive or context-insensitive points-to (or alias) analysis. In addition, the literature is abound with investigations exploring analysis of Java programs. Finally, a large number of investigations focus on demand-driven methods. We restrict ourselves to exhaustive flow- and context-sensitive points-to analysis of primarily C programs and mention Java-related works that are directly related to our ideas.

Most of the top-down approaches to flow- and context-sensitive pointer analysis of C programs have not scaled [8, 21, 31] with the largest program successfully analyzed by them consisting of 35 kLoC [21]. It is no surprise, then, that the literature of flow- and context-sensitive points-to analyses is dominated by bottom-up approaches. Our work also belongs to this category, and hence we focus on them in this section by classifying them into the MTF or STF approach (Section 2.3).

11.2.1 MTF Approach for Bottom-Up Summaries. In this approach [16, 43, 46, 47], control flow is not required to be recorded between memory updates. This is because the data dependency between memory updates (even the ones that access unknown pointers) is known by using either the alias information or the points-to information from the calling context. These approaches construct symbolic procedure summaries. This involves computing preconditions and corresponding postconditions (in terms of aliases or points-to information). A calling context is matched against a precondition, and the corresponding postcondition gives the result.

Two approaches that stand out among these from the viewpoint of scalability are bootstrapping [16] and level-by-level analysis [46]. The bootstrapping approach partitions the pointers using flow-insensitive analyses such that each subset is an equivalence class with respect to alias information and then analyses are performed in a cascaded fashion in a series $A_0, A_1, \ldots, A_k$, where analysis A_i uses the points-to information computed by the analysis A_{i-1}. In addition, analysis of different equivalence classes at any level can be performed in parallel. This process involves constructing MTF procedure summaries using a top-down traversal of call graph to compute alias information using FSCI analysis. This may cause some imprecision in the computed summaries. However, it is not clear if the precision loss is significant, as there are no formal guarantees of precision, nor does the work provide empirical evaluation of precision—the focus being solely on scalability. The analysis is reported to scale to 128 kLoC.

Level-by-level analysis [46] constructs a procedure summary with multiple interprocedural conditions. It matches the calling context with these conditions and chooses the appropriate summary for the 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 an extended SSA form. When used in conjunction with conditions based on points-to information from calling contexts, the chains become context sensitive. This method is claimed to scale to 238 kLoC; however, similar to the bootstrapping method, there are no formal guarantees or empirical evaluation of precision.[15]

[15] In addition, this method has been followed by the SVF method that is flow sensitive but context insensitive [41], which has further been followed by a flow- and context-sensitive method that is demand driven [40].

Since these approaches depend on the number of aliases/points-to pairs in the calling contexts, the procedure summaries are not context independent. Thus, this approach may not be useful for constructing summaries for library functions that have to be analyzed without the benefit of different calling contexts. Saturn [12] creates sound summaries, but they may not be precise across applications because of their dependence on context information.

Relevant context inference [5] 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 overapproximates the alias and the type context as an optimization, thereby being only partially context sensitive.

11.2.2 STF Approach for Bottom-Up Summaries. This approach does not make any assumptions about the calling contexts [4, 6, 23, 25–27, 33, 39, 42, 48] and uses multiple placeholders for distinct accesses of the pointees of the same pointer (Section 2.3). This tends to increase the size of the resulting procedure summaries. This problem is mitigated by choosing carefully where the placeholders are required [39, 42], by employing optimizations that merge placeholders [26], by maintaining restricted control flow [4], by overapproximating the control flow through flow insensitivity [23], or a combination of them [27]. In some cases, the overapproximation is only in the application of procedure summaries even though they are constructed flow sensitively [27]. Many of these approaches scale to millions of lines of code.

Although the attempts to minimize the placeholders prohibits killing of points-to information of pointer variables in C/C++ programs, it does not have much adverse impact on Java programs. This is because all local variables in Java have SSA versions, thanks to the absence of indirect assignments to variables (there is no addressof operator). In addition, there are few static variables in Java programs, and absence of kill for them may not matter much.

Lattner et al. [23] 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 Hwu [6] 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. The abstraction of access paths is similar to the indirection lists (*indlist*s) used by our approach. The approach uses allocation-site-based abstraction and cycles in the access paths to bound the length of access paths. This approach is flow insensitive and hence does not maintain any control flow between these access paths.

Access fetch graphs [4] is another representation for procedure summaries for points-to analysis. This approach presents two versions of a summary: a flow-insensitive version and a flow-aware version that is a flow-insensitive version augmented by encoding control flow using a total order. The latter is sound and more precise than the flow-insensitive version but less precise than a flow-sensitive version.

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.

11.2.3 The Hybrid Approach. Hybrid approaches use customized summaries and combine the top-down and bottom-up analyses to construct summaries [47]. 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 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 10.3.1). If this threshold is set to 0%, our method becomes a purely bottom-up approach; if it is set to 100%, our method becomes a top-down approach.

12 CONCLUSION AND FUTURE WORK

Constructing compact procedure summaries for flow- and context-sensitive points-to analysis seems hard because it

(a) needs to model the accesses of pointees defined in callers without examining their code,
(b) needs to preserve data dependence between memory updates, and
(c) needs to incorporate the effect of the summaries of the callee procedures transitively.

These issues have been handled in the past as follows. The first issue has been handled by modeling 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 handled the first issue by proposing the concept of GPUs that 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.

To handle the second issue, we maintain control flow within a GPG and perform optimizations of strength reduction and control flow minimization. Together, these optimizations reduce the indirection levels of GPUs, eliminate data dependences between GPUs, and significantly reduce control flow. These optimizations also mitigate the impact of the third issue.

We achieved the preceding by devising novel dataflow analyses such as two variants of reaching GPUs analysis and coalescing 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 usual second phase.

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. Our work differs from most other investigations exploring scalable exhaustive flow- and context-sensitive points-to analysis of C in the following ways:

- To achieve scalability and precision simultaneously, most approaches start with a scalable method and try to increase its precision. Our work starts with a precise method and optimizes it for scalability without compromising soundness or precision.
- This reversal of priorities in our approach has a significant benefit that it facilitates formal guarantees of soundness and precision. In addition, we have provided extensive empirical evidence of scalability and precision; most other scalable methods focus primarily on scalability and do not provide formal guarantees or empirical evidence of precision.

Two important takeaways from our empirical evaluation are the following:

(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 but the amount of control flow that it may be subjected to in search of precision. Our analysis scales because it minimizes the control flow significantly.

Our empirical measurements show that most of the GPGs are acyclic even if they represent procedures that have loops or are recursive.

As a possible direction of future work, 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 addition, it would be interesting to explore the possibility of restricting GPG construction to live pointer variables [21] for scalability. It would also be useful to extend the scope of the implementation to C++ and Java programs.

The concept of GPG provides a useful abstraction of memory and memory transformers involving pointers by directly modeling load, store, and copy of memory addresses. Any client program analysis that uses these operations may be able to use GPGs by combining them with the original abstractions of the analysis. In particular, we expect to integrate this method into an in-house bounded model checking infrastructure being developed at IIT Bombay.

APPENDIX

The appendix is available at https://github.com/PritamMG/GPG-based-Points-to-Analysis.

ACKNOWLEDGMENTS

We would like to thank the anonymous referees for their incisive comments on the earlier draft of the article. In particular, their insistence on soundness proof forced us to investigate deeper. In the process, many concepts became more rigorous, and we could formally show that our method is equivalent to a top-down flow- and context-sensitive classical points-to analysis, proving both soundness and precision.

We would like to thank Akshat Garg and N. Venkatesh for empirical measurements of points-to information computed by SVF analysis as also the data for the client analyses.

REFERENCES

[1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools* (2nd ed.). Addison Wesley Longman, Boston, MA.

[2] Thomas Ball and Sriram K. Rajamani. 2002. 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)*. ACM, New York, NY, 1–3. DOI: https://doi.org/10.1145/503272.503274

[3] A. J. Bernstein. 1996. Analysis of programs for parallel processing. *IEEE Trans. Elec. Comp.* EC-15, 5 (1996), 746–757. https://ci.nii.ac.jp/naid/10009998541/en/.

[4] Marcio Buss, Daniel Brand, Vugranam Sreedhar, and Stephen A. Edwards. 2010. A novel analysis space for pointer analysis and its application for bug finding. *Sci. Comput. Program.* 75, 11 (Nov. 2010), 921–942. DOI: https://doi.org/10.1016/j.scico.2009.08.002

[5] Ramkrishna Chatterjee, Barbara G. Ryder, and William A. Landi. 1999. Relevant context inference. In *Proceedings of the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'99)*. ACM, New York, NY, 133–146. DOI: https://doi.org/10.1145/292540.292554

[6] Ben-Chung Cheng and Wen-Mei W. Hwu. 2000. 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)*. ACM, New York, NY, 57–69. DOI: https://doi.org/10.1145/349299.349311

[7] Isil Dillig, Thomas Dillig, and Alex Aiken. 2008. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 29th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'08)*. ACM, New York, NY. DOI: https://doi.org/10.1145/1375581.1375615

[8] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN 1994 Conference on Programming Language Design and Implementation (PLDI'94)*. ACM, New York, NY, 242–256. DOI: https://doi.org/10.1145/178243.178264

[9] Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. 2015. Bottom-up context-sensitive pointer analysis for Java. In *Proceedings of the 13th Asian Symposium on Programming Languages and Systems (APLAS'15)*. DOI: https://doi.org/10.1007/978-3-319-26529-2_25

[10] Pritam M. Gharat. 2018. *Generalized Points-to Graph: A New Abstraction of Memory in Presence of Pointers*. Ph.D. Dissertation. Indian Institute of Technology Bombay, Mumbai, India.

[11] Pritam M. Gharat, Uday P. Khedker, and Alan Mycroft. 2016. Flow- and context-sensitive points-to analysis using generalized points-to graphs. In *Proceedings of the 23rd Static Analysis Symposium (SAS'16)*.

[12] Brian Hackett and Alex Aiken. 2006. 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)*. ACM, New York, NY. DOI: https://doi.org/10.1145/1181775.1181785

[13] Nevin Heintze and Olivier Tardieu. 2001. Demand-driven pointer analysis. In *Proceedings of the ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation (PLDI'01)*. ACM, New York, NY. DOI: https://doi.org/10.1145/378795.378802

[14] Michael Hind and Anthony Pioli. 1998. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Proceedings of the 5th International Symposium on Static Analysis (SAS'98)*. 57–81. DOI: https://doi.org/10.1007/3-540-49727-7_4

[15] Michael Hind and Anthony Pioli. 2000. Which pointer analysis should I use? In *Proceedings of the 2000 ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'00)*. ACM, New York, NY, 113–123. DOI: https://doi.org/10.1145/347324.348916

[16] Vineet Kahlon. 2008. 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)*. ACM, New York, NY, 249–259. DOI: https://doi.org/10.1145/1375581.1375613

[17] Vini Kanvar and Uday P. Khedker. 2016. Heap abstractions for static analysis. *ACM Comput. Surv.* 49, 2 (June 2016), Article 29, 47 pages. DOI: https://doi.org/10.1145/2931098

[18] Owen Kaser, C. R. Ramakrishnan, and Shaunak Pawagi. 1993. On the conversion of indirect to direct recursion. *ACM Lett. Program. Lang. Syst.* 2, 1-4 (March 1993), 151–164. DOI: https://doi.org/10.1145/176454.176510

[19] Ken Kennedy and John R. Allen. 2002. *Optimizing Compilers for Modern Architectures: A Dependence-Based Approach*. Morgan Kaufmann, San Francisco, CA.

[20] Uday P. Khedker and Bageshri Karkare. 2008. 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 and the 17th International Conference on Compiler Construction (CC'08/ETAPS'08)*.

[21] Uday P. Khedker, Alan Mycroft, and Prashant Singh Rawat. 2012. Liveness-based pointer analysis. In *Proceedings of the 19th International Static Analysis Symposium (SAS'12)*. DOI: https://doi.org/10.1007/978-3-642-33125-1_19

[22] U. P. Khedker, A. Sanyal, and B. Sathe. 2009. *Data Flow Analysis: Theory and Practice*. Taylor & Francis (CRC Press), Boca Raton, FL.

[23] Chris Lattner, Andrew Lenharth, and Vikram Adve. 2007. Making context-sensitive points-to analysis with heap cloning practical for the real world. In *Proceedings of the 28th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*. ACM, New York, NY, 278–289. DOI: https://doi.org/10.1145/1250734.1250766

[24] Ondrej Lhotak, Yannis Smaragdakis, and Manu Sridharan. 2013. Pointer analysis (Dagstuhl seminar 13162). *Dagstuhl Reports* 3, 4 (2013), 91–113. DOI: https://doi.org/10.4230/DagRep.3.4.91

[25] Lian Li, Cristina Cifuentes, and Nathan Keynes. 2013. Precise and scalable context-sensitive pointer analysis via value flow graph. In *Proceedings of the 2013 International Symposium on Memory Management (ISMM'13)*. ACM, New York, NY. DOI: https://doi.org/10.1145/2464157.2466483

[26] Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani. 2012. Modular heap analysis for higher-order programs. In *Proceedings of the 19th International Conference on Static Analysis (SAS'12)*. DOI: https://doi.org/10.1007/978-3-642-33125-1_25

[27] Ravichandhran Madhavan, G. Ramalingam, and Kapil Vaswani. 2015. A framework for efficient modular heap analysis. *Found. Trends Program. Lang.* 1, 4 (Jan. 2015), 269–381. DOI: https://doi.org/10.1561/2500000020

[28] Steven S. Muchnick. 1997. *Advanced Compiler Design and Implementation*. Morgan Kaufmann, San Francisco, CA.

[29] Rohan Padhye and Uday P. Khedker. 2013. 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)*. ACM, New York, NY. DOI: https://doi.org/10.1145/2487568.2487569

[30] Thomas Reps, Susan Horwitz, and Mooly Sagiv. 1995. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'95)*. ACM, New York, NY. DOI: https://doi.org/10.1145/199448.199462

[31] Barbara G. Ryder, William A. Landi, Philip A. Stocks, Sean Zhang, and Rita Altucher. 2001. A schema for interprocedural modification side-effect analysis with pointer aliasing. *ACM Trans. Program. Lang. Syst.* 23, 2 (March 2001), 105–186. DOI: https://doi.org/10.1145/383043.381532

[32] Mooly Sagiv, Thomas Reps, and Susan Horwitz. 1996. 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).* Elsevier Science Publishers B. V., Amsterdam, The Netherlands. http://dl.acm.org/citation.cfm?id=243753.243762

[33] Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand dynamic summary-based points-to analysis. In *Proceedings of the 10th International Symposium on Code Generation and Optimization (CGO'12).* ACM, New York, NY. DOI: https://doi.org/10.1145/2259016.2259050

[34] M. Sharir and A. Pneuli. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones (Eds.). Prentice Hall, 189–234.

[35] Yannis Smaragdakis and George Balatsouras. 2015. Pointer analysis. *Foundations and Trends® in Programming Languages* 2, 1 (2015), 1–69. DOI: https://doi.org/10.1561/2500000014

[36] Johannes Späth, Lisa Nguyen, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-driven flow- and context-sensitive pointer analysis for Java. In *Proceedings of the European Conference on Object-Oriented Programming (ECOOP'16).*

[37] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodík. 2005. 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).* ACM, New York, NY. DOI: https://doi.org/10.1145/1094811.1094817

[38] Stefan Staiger-Stöhr. 2013. Practical integrated analysis of pointers, dataflow and control flow. *ACM Trans. Program. Lang. Syst.* 35, 1 (2013), Article 5, 48 pages. DOI: https://doi.org/10.1145/2450136.2450140

[39] Alexandru Sălcianu and Martin Rinard. 2005. Purity and side effect analysis for Java programs. In *Proceedings of the 6th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI'05).* DOI: https://doi.org/10.1007/978-3-540-30579-8_14

[40] Yulei Sui and Jingling Xue. 2016. On-demand strong update analysis via value-flow refinement. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE'16).* ACM, New York, NY, 460–473. DOI: https://doi.org/10.1145/2950290.2950296

[41] Yulei Sui and Jingling Xue. 2016. SVF: Interprocedural static value-flow analysis in LLVM. In *Proceedings of the 25th International Conference on Compiler Construction (CC'16).* ACM, New York, NY, 265–266. DOI: https://doi.org/10.1145/2892208.2892235

[42] John Whaley and Martin Rinard. 1999. 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).* ACM, New York, NY. DOI: https://doi.org/10.1145/320384.320400

[43] R. P. Wilson and M. S. Lam. 1995. Efficient context-sensitive pointer analysis for C programs. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'95).* citeseer.ist.psu.edu/wilson95efficient.html

[44] Dacong Yan, Guoqing Xu, and Atanas Rountev. 2012. 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).* ACM, New York, NY. DOI: https://doi.org/10.1145/2259051.2259053

[45] Greta Yorsh, Eran Yahav, and Satish Chandra. 2008. Generating precise and concise procedure summaries. In *Proceedings of the 35th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages (POPL'08).* ACM, New York, NY. DOI: https://doi.org/10.1145/1328438.1328467

[46] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. 2010. 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).* ACM, New York, NY, 218–229. DOI: https://doi.org/10.1145/1772954.1772985

[47] Xin Zhang, Ravi Mangal, Mayur Naik, and Hongseok Yang. 2014. Hybrid top-down and bottom-up interprocedural analysis. In *Proceedings of the 35th ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14).* ACM, New York, NY. DOI: https://doi.org/10.1145/2594291.2594328

[48] Jianwen Zhu and Silvian Calman. 2005. Context sensitive symbolic pointer analysis. *IEEE Trans. Comput. Aided Des. Integr. Circ. Syst.* 24 (May 2005), 516–531. DOI: https://doi.org/10.1109/TCAD.2005.844092

Received January 2018; revised October 2019; accepted January 2020