

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

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

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

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

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

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

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

ACM Reference Format:

Pritam M. Gharat, Uday P. Khedker, and Alan Mycroft. 2018. Generalized Points-to Graphs: A New Abstraction of Memory in the Presence of Pointers. *ACM Trans. Program. Lang. Syst.* 1, 1 (January 2018), 74 pages. <https://doi.org/10.1145/nnnnnnnn.nnnnnnnn>

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 [4, 8, 27, 28] 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

Authors' addresses: Pritam M. Gharat, pritamg@cse.iitb.ac.in; Uday P. Khedker, uday@cse.iitb.ac.in, Indian Institute of Technology Bombay, India; Alan Mycroft, University of Cambridge, UK, Alan.Mycroft@cl.cam.ac.uk.

analysis needs to be repeated. An exhaustive analysis, on the other hand, computes all points-to information which can be queried later by a client analysis; multiple queries do not require points-to analysis to be repeated. 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 data flow 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 [36] effectively traversing the call graph top-down. In the process, it analyzes a procedure each time a new data flow value reaches it from some call. Several popular approaches fall in this category: the call-strings method [25], its value-based variants [13, 20] and the tabulation-based functional method [21, 25]. By contrast, bottom-up approaches [3, 5, 7, 11, 18, 22, 25, 30–36] 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 applies them at the call sites to compute the desired information.

1.2 Our Contributions

This paper advocates 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 GPUs (generalized points-to updates) 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 below.

- (1) Our representation of memory updates, called the *generalized points-to update* (GPU) leaves accesses of unknown pointees implicit without losing precision.
- (2) GPGs undergo aggressive optimizations that are applied repeatedly to improve the compactness of GPGs incrementally. These optimizations are similar to the optimizations performed by compilers and are governed by the following possibilities of data dependence between two memory updates (illustrated in Example 1 in Section 2.2)
 - **Case A.** The memory updates have a data dependence between them. It could be
 - **Case 1.** a read-after-write (RaW) dependence,
 - **Case 2.** a write-after-read (WaR) dependence, or
 - **Case 3.** a write-after-write (WaW) dependence.
 A read-after-read (RaR) dependence is irrelevant.
 - **Case B.** The memory updates do not have a data dependence between them.

¹We 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. Hence these terms are orthogonal. Thus, both a forward data flow analysis (e.g. available expressions analysis) and a backward data flow analysis (e.g. live variables analysis) could be implemented as a top-down or a bottom-up analysis at the interprocedural level.

- **Case C.** More information is needed to find out whether the memory updates have a data dependence between them.

These cases are exploited by the optimizations described below:

- *Strength reduction* optimization exploits case A1. It simplifies memory updates by using the information from other memory updates to eliminate data dependence between them.
- *Redundancy elimination* optimizations handle cases A2, A3, and B. They remove redundant memory updates (case A3) and minimize control flow (case B). Case A2 is an anti-dependence and is modelled by eliminating control flow and ensuring that it is not viewed as a RaW dependence (Example 6 in Section 3.1).
- *Call inlining* optimization handles case C by progressively providing more information. It inlines the summaries of the callees of a procedure. This enhances the opportunities of strength reduction and redundancy elimination and enables context-sensitive analysis.
- *Type-based non-aliasing.* We use the types specified in the program to resolve some additional instances of case C into case B.

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 it may be subjected to in search of precision. Our optimizations are effective because they eliminate data dependence wherever possible and discard irrelevant control flow without losing precision. Flow and context insensitivity discard control flow but over-approximate data dependence 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.

In order to perform these optimizations:

- We define operations of *GPU composition* (to create new GPUs by eliminating data dependence between two GPUs), and *GPU reduction* (to eliminate the data dependence of a GPU with the GPUs in a given set).
- We propose novel data flow analyses such as two variants of *reaching GPUs analysis* (to identify the effects of memory updates reaching a given statement) and *coalescing analysis* (to eliminate the redundant control flow in the GPG).
- We handle recursive calls by refining the GPGs through a fixed-point computation. Calls through function pointers are proposed to be handled through delayed inlining.

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

1.3 The Organization of the Paper

Section 2 describes the limitations of past approaches as a background to motivate our key ideas that overcome them. Section 3 introduces the concept of generalized points-to updates (GPUs) that form the basis of GPGs and provides a brief overview of GPG construction through a motivating example. Section 4 describes the strength-reduction optimization performed on GPGs by formalizing the operations such as GPU composition and GPU reduction and defining data flow equations for reaching GPUs analyses. Section 5 describes redundancy elimination optimizations performed

on GPGs. Section 6 explains the interprocedural use of GPGs by defining call inlining and shows how recursion is handled. Section 7 shows how GPGs are used for performing points-to analysis. Section 8 describes the handling of structures, unions and the heap. Section 9 describes the handling of function pointers. Section 10 presents empirical evaluation on SPEC benchmarks and Section 11 describes related work. Section 12 concludes the paper.

2 EXISTING APPROACHES AND THEIR LIMITATIONS

This section begins by reviewing some basic concepts and then describes the challenges in constructing procedure summaries for 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. Firstly we have the concrete memory view (or semantic view) corresponding to run-time operations where a memory location always points to exactly one memory location or NULL (which is a distinguished memory location). Unfortunately this is, in general, statically uncomputable. Secondly, as is traditional in program analysis, we can consider an abstract view of memory where an abstract location represents one or more concrete locations; this conflation and the uncertainty of conditional branches means that abstract memory locations can point to multiple other locations—as in the classical points-to graph. These views are not independent and abstract operations must over-approximate concrete operations to ensure soundness. Formally, let L and $P \subseteq L$ denote the sets of locations and pointers respectively. The *concrete memory* after a pointer assignment is a function $M : P \rightarrow L$. The *abstract memory* after a pointer assignment is a relation $M \subseteq P \times L$. 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 $x \in P$ contains the address of $y \in L$. Unless noted explicitly, all subsequent references to memory locations and transformers refer to the abstract view.

The (abstract) memory associated with a statement s is an over-approximation of the concrete memory associated with every occurrence of s in the same or different control flow paths.

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. It is called a *memory transformer* because it updates the memory before a call to the procedure to compute the memory after the call. Given a memory M and a memory transformer 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 memory location corresponding to the LHS of the assignment. However, in abstract memory, we may be uncertain as to which of several locations a variable (say p) points to. Hence an indirect assignment such as $*p = \&x$ does not overwrite any of its pointees, but merely *adds* x to the possible pointees. This is a *weak update*. Sometimes however, there is only one possible abstract location described by the LHS of an assignment, and in this case we may, in general, *replace* the contents of this location. This is a *strong update*. There is just one subtlety which we return to later: prior to the above assignment we may only have one assignment to p (say $p = \&a$). If this latter assignment dominates the former, then a strong update is appropriate. But if the latter assignment only appears on some control flow paths to the former, then we say that the read of p in $*p = \&x$ is *upwards exposed* (live on entry to the current procedure) and therefore may

have additional pointees unknown to the current procedure. Thus, the criterion for a strong update in an assignment is that its LHS references a single location *and* the location referenced is not upwards exposed (for more details, see Section 4.3.2). An important special case is that a direct assignment to a variable (e.g. $p = \&x$) is always a strong update.

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

2.2 Challenges in Constructing Procedure Summaries for Points-to Analysis

In the absence of 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 data flow analyses (such as live variables analysis), can be precisely represented by constant *gen* and *kill* sets or graph paths discovered using reachability [15]. In the presence of pointers, these (bit-vector) summaries can be constructed using externally supplied points-to information.

Procedure summaries for points-to analysis, however, cannot be represented in terms of constant *gen* and *kill* sets because the association between pointer variables and their pointee locations could change in the procedure and may depend on the aliases between pointer variables established in the callers of the procedure. Often, and particularly for points-to analysis, we have a situation where a procedure summary must either lose information or retain internal details which can only be resolved when its caller is known.

Example 1. Consider procedure f on the right. For many calls, $f()$ simply returns $\&a$ but until we are certain that $*p$ does not alias with x , we cannot perform this constant-propagation optimization. We say that the assignment 04 *blocks* this optimization. There are four possibilities:

- If it is known that $*p$ and x *always* alias then we can optimize f to return $\&b$.
- If it is known that $*p$ and x alias on some control flow paths containing a call to f but not on all, then the procedure returns $\&a$ in some cases and $\&b$ in other cases. While procedure f cannot be optimized to do this, a static analysis can compute such a summary.
- If it is known that they *never* alias we can optimize this code to return $\&a$.
- If nothing is known about the alias information, then to preserve precision, we must retain this blocking assignment in the procedure summary for f .

```

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

```

The first two situations correspond to case (A1) in item (2) in Section 1.2. The third and the fourth situations correspond to cases (B) and (C) respectively.

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

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

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

2.3 Limitations of Existing Procedure Summaries for Points-to Analysis

A common solution for modelling indirect accesses of unknown pointees in a memory transformer is to use *placeholders*² which 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* (MTF) approach, proposed a precise representation of a procedure summary for points-to analysis as a collection of *partial transfer functions* (PTFs) [3, 11, 32, 35].³ 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 [18, 30, 31]. 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' is constructed by the STF approach [18, 30, 31]. It can be viewed as an abstract points-to graph containing placeholders ϕ_i for modelling unknown pointees of the pointers appearing in Δ' . For example, ϕ_1 represents the pointees of y and ϕ_2 represents the pointees of pointees of y , both of which are not known in the procedure. The placeholders are pattern matched against the input memory (e.g. M_1 or M_2) to compute the corresponding output memory (M'_1 and M'_2 respectively). A crucial difference between a memory and a memory transformer is: a memory is a snapshot of points-to edges whereas a memory transformer needs to distinguish the points-to edges that are generated by it (shown by thick edges) from those that are carried forward from the input memory (shown by thin edges).

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

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

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

²Placeholders have also been known as external variables [18, 30, 31] and extended parameters [32]. They are parameters of the procedure summary and not necessarily of the procedure for which the summary is constructed.

³In level-by-level analysis [35], multiple PTFs are combined into a single function with a series of condition checks for different points-to information occurring in the calling contexts.

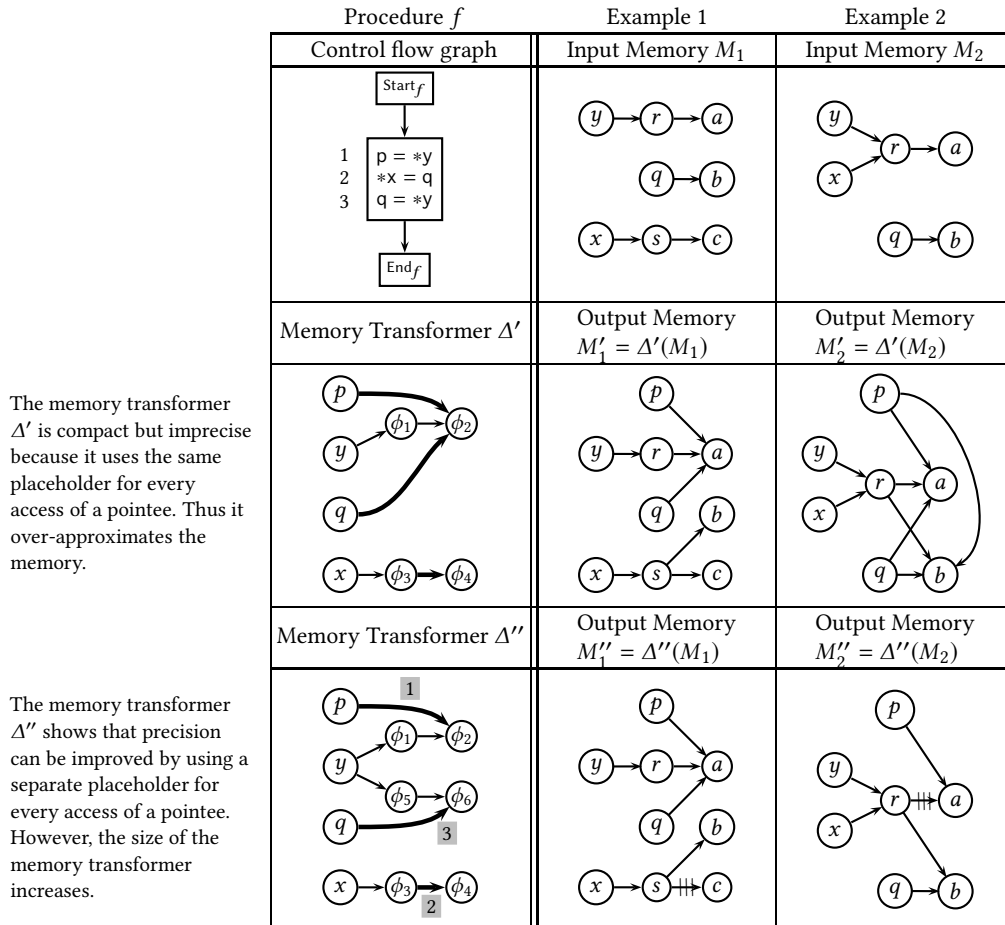


Fig. 1. An STF-style memory transformer Δ' and its associated transformations. Δ'' is its flow-sensitive version. Unknown pointees are denoted by placeholders ϕ_i . Thick edges in a memory transformer represent the points-to edges generated by it, other edges are carried forward from the input memory. Labels of the points-to edges in Δ'' correspond to the statements indicating the sequencing of edges. Edges that are killed in the memory are struck off.

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).

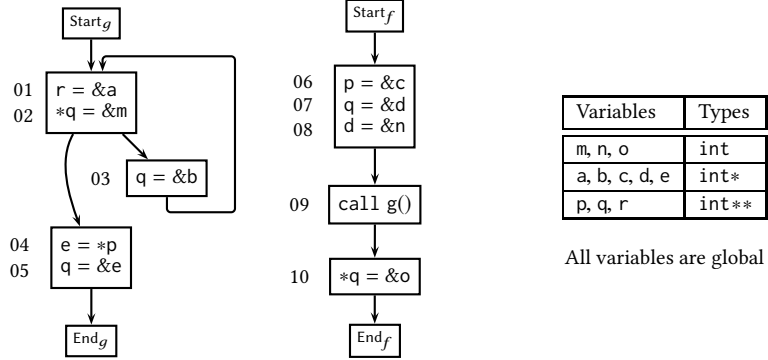


Fig. 2. A motivating example. Procedures are represented by their control flow graphs (CFGs).

The need of control flow ordering further increases the number of placeholders in the former approach. The latter approach obviates the need of ordering because the PTFs are customized for combinations of aliases.

2.4 Our Key Ideas

We propose a *generalized points-to graph* (GPG) as a representation for a memory transformer of a procedure; special cases of GPGs also represent memory as a points-to relation. 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. An interesting property of the flow relation is that it can be compressed dramatically without losing precision and can be transformed into a compact acyclic flow relation in most cases, even if the procedure it represents has loops or recursive calls.

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

3 THE GENERALIZED POINTS-TO GRAPHS AND AN OVERVIEW OF THEIR CONSTRUCTION

In this section, we define a *generalized points-to graph* (GPG) which serves as our memory transformer. It is a graph with *generalized points-to blocks* (GPBs) as nodes which contain *generalized points-to updates* (GPUs). The ideas and algorithms for defining and computing these three representations of memory transformers can be seen as a collection of abstractions, operations, data flow analyses, and optimizations. Their relationships are shown in Figure 3. A choice of key abstractions enables us to define GPU operations which are used for performing three data flow analyses. The information computed by these analyses enables optimizations over GPGs.

This section presents an overview of our approach in a limited setting of our motivating example of Figure 2. Towards the end of this section, Figure 8 fleshes out Figure 3 to list specific abstractions, operations, analyses, and optimizations.

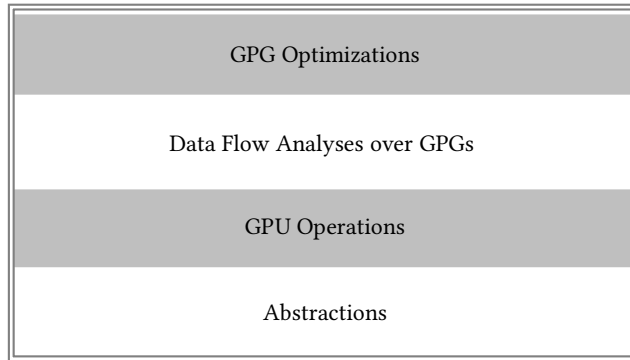


Fig. 3. Inter-relationships between ideas and algorithms for defining and computing GPUs, GPBs, and GPGs. Each layer is defined in terms of the layers below it. Figure 8 fleshes out this picture by listing specific abstractions, operations, data flow analyses, and optimizations.

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

Definition 1. *Generalized Points-to Update.*

3.1 Defining a Generalized Points-to Graph (GPG)

We model the effect of a pointer assignment on an abstract memory by defining the concept of *generalized points-to update* (GPU) in Definition 1. We use the statement label s to capture weak versus strong updates and for computing points-to information.⁴ Definition 1 gives the abstract semantics of a GPU. The concrete semantics of a GPU $x \xrightarrow[i]{j}_s y$ can be viewed as the following C-style pointer assignment with $i - 1$ dereferences of x ⁵ and j dereferences of $\&y$:

$$\underbrace{** \dots * x}_{(i-1)} = \underbrace{** \dots * \&y}_j$$

A GPU $\gamma : x \xrightarrow[i]{j}_s y$ generalizes a points-to edge⁶ 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.
- 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 the statement s in the procedure.

⁴We omit the statement labels in GPUs at some places when they are not required.

⁵Alternatively, i dereferences of $\&x$. We choose $i - 1$ dereference from x because the left-hand side cannot be $\&x$.

⁶Although a GPU can be drawn as an arrow just like a points-to edge, we avoid the term ‘edge’ for a GPU because of the risk of confusion with a ‘control flow edge’ in a GPG.

Pointer assignment	GPU	Relevant memory graph after the assignment
$s: x = \&y$	$x \xrightarrow{1 0}_s y$	$x \bullet \rightarrow \bullet \circ y$
$s: x = y$	$x \xrightarrow{1 1}_s y$	$x \bullet \rightarrow \bullet \circ \leftarrow \bullet y$
$s: x = *y$	$x \xrightarrow{1 2}_s y$	$x \bullet \rightarrow \bullet \circ \leftarrow \bullet \leftarrow \bullet y$
$s: *x = y$	$x \xrightarrow{2 1}_s y$	$x \bullet \rightarrow \bullet \rightarrow \bullet \rightarrow \bullet \circ \leftarrow \bullet y$

Fig. 4. GPUs for basic pointer assignments in C. In the memory graphs, a double circle indicates the location whose address is being assigned, a thick arrow shows the generated edges. Unnamed nodes may represent multiple pointees (implicitly representing placeholders).

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

Definition 2. *Generalized Points-to Blocks and Generalized Points-to Graphs.*

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 $r \xrightarrow{1|0}_{01} a$ 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 $q \xrightarrow{2|0}_{02} m$. 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 4 presents the GPUs for basic pointer assignments in C. (To deal with C structs and unions, GPUs are augmented to encode lists of field names—for details see Figure 18).

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

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

- The CFG could have procedure calls whereas the GPG does not.⁷ Besides, a GPG is acyclic in almost all cases, even if the procedure it represents has loops or recursive calls.

⁷In the presence of recursion and calls through function pointers (Sections 6.2 and 9), we need an intermediate form of GPG called an *incomplete* GPG containing unresolved calls that are resolved when more information becomes available.

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

A concrete semantic reading of a GPB δ is defined in terms of the semantics of executing a GPU (Definition 1). Execution of δ implies that the GPUs in δ are executed non-deterministically in any order. This gives a correct abstract reading of a GPB as a *may* property. But a stronger concrete semantic reading also holds as a *must* property: Let δ contain GPUs corresponding to some statement s . Define $X_s \subseteq \delta$ by $X_s = \{x \xrightarrow{i/j} y \in \delta\}$, $X_s \neq \emptyset$. Then, whenever statement s is reached in any execution, at least one GPU in X_s *must* be executed. This semantics corresponds to that of the points-to information generated for a statement in the classical points-to analysis. This gives GPBs their expressive power—multiple GPUs arising from a single statement, produced by GPU-reduction (see later), represent *may*-alternative updates, but one of these *must* be executed.⁸

Example 5. Consider a GPB $\{\gamma_1 : x \xrightarrow{1/0} a, \gamma_2 : x \xrightarrow{1/0} b, \gamma_3 : y \xrightarrow{1/0} c, \gamma_4 : z \xrightarrow{1/0} d, \gamma_5 : t \xrightarrow{1/0} d, \}$. After executing this GPB (abstractly or concretely) we know that the points-to sets of x is overwritten to become $\{a, b\}$ (i.e. x definitely points to one of a and b) because GPUs γ_1 and γ_2 both represent statement 11 and define a single location x . Similarly, the points-to set of y is overwritten to become $\{c\}$ because γ_3 defines a single location c in statement 12. However, this GPB causes the points-to sets of z and t to *include* $\{d\}$ (without removing the existing pointees) because γ_4 and γ_5 both represent statement 13 but define separate locations. Thus, x and y are strongly updated (their previous pointees are removed) but z and t are weakly updated (their previous pointees are augmented).

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

The *may* property of the absence of control flow between the GPUs in a GPB allows us to model a WaR dependence as illustrated in the following example:

Example 6. Consider the code snippet on the right. There is a WaR data dependence between statements 01 and 02. If the control flow is not maintained, the statements could be executed in the reverse order and y could erroneously point to a .

01	$y = x;$
02	$x = \&a;$

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

3.2 An Overview of GPG Operations

Figure 5 lists the GPG operations based on the concept of generalized points-to updates (GPUs). Each layer is defined in terms of the layers below it. For each operation, Figure 5 describes the types of its operands and result, and lists the section in which the operation is defined.

3.2.1 GPU Composition. In a compiler, the sequence $p = \&a; *p = x$ is usually simplified to $p = \&a; a = x$ to facilitate further optimizations. Similarly, the sequence $p = \&a; q = p$ is usually simplified to $p = \&a; q = \&a$. While both

⁸A subtlety is that a GPB δ may contain a spurious GPU that can never be executed because the flow functions of points-to analysis are non-distributive [15].

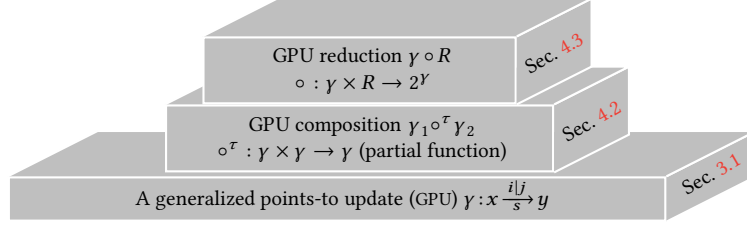


Fig. 5. A hierarchy of core operations involving GPUs. Each operation is defined in terms of the layers below it. The set of GPUs reaching a GPU γ (computed using the reaching GPUs analyses of Sections 4.4 and 4.5) is denoted by R . By abuse of notation, we use γ , δ , and R also as types to indicate the signatures of the operations. The operator “ \circ ” is overloaded and can be disambiguated using the types of the operands.

simplifications are forms of constant propagation, they play rather different roles, and in the GPG framework, are instances of (respectively) *SS* and *TS* variants of *GPU composition* (Section 4.2).

Suppose a GPU γ_1 precedes γ_2 on some control flow path. If there is a RaW dependence between γ_1 and γ_2 then, a GPU composition $\gamma_2 \circ^\tau \gamma_1$ computes a new GPU where τ is *SS* or *TS*. The resulting GPU γ_3 is a simplified version of the *consumer* GPU γ_2 obtained by using the points-to information in the *producer* GPU γ_1 such that:

- The *indlev* of γ_3 (say $i|j$) does not *exceed* that of γ_2 (say $i'|j'$), i.e. $i \leq i'$ and $j \leq j'$. The two GPUs γ_2 and γ_3 are equivalent in the context of GPU γ_1 .
- The type of GPU composition (denoted τ) is governed by the role of the common node (later called the ‘pivot’) between γ_1 and γ_2 . The forms of GPU composition important here are *TS* and *SS* compositions. In *TS* composition, the pivot is the target of GPU γ_2 and the source of γ_1 , whereas in *SS* composition, the pivot is the source of both γ_1 and γ_2 .

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

Example 7. For statement sequence $p = \&a; *p = x$, the consumer GPU $\gamma_2 : p \xrightarrow{2|1}_2 x$ (statement 2) is simplified to $\gamma_3 : a \xrightarrow{1|1}_2 x$ by replacing the source p of γ_2 using the producer GPU $\gamma_1 : p \xrightarrow{1|0}_1 a$ (statement 1). GPU γ_3 can be further simplified to one or more points-to edges (i.e. GPUs with *indlev* $1|0$) when GPUs representing the pointees of x (the target of γ_3) become available.

The above example illustrates the following:

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

3.2.2 GPU Reduction. We generalize the above operation as follows. If we have a set RGIn_s of GPUs (representing generalized-points-to knowledge from previous statements and obtained from the *reaching GPUs analyses* of Sections 4.4 and 4.5) and a single GPU $\gamma_s \in \delta_s$, representing a GPU statement s , then *GPU reduction* $\gamma_s \circ \text{RGIn}_s$ constructs a set of one or more GPUs, all of which correspond to statement s . This is considered as the information generated for

statement s and is denoted by RGen_s . It is a union of all such sets created for every GPU $\gamma_s \in \delta_s$ and is semantically equivalent to δ_s in the context of RIn_s and, as suggested above, may beneficially replace δ_s .

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

3.3 An Overview of GPG Construction

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

- (1) *creation* of the initial GPG, and *inlining* optimized GPGs of called procedures⁹ within Δ_f ,
- (2) *strength reduction* optimization to simplify the GPUs in Δ_f by performing *reaching GPUs analyses* and transforming GPBs using *GPU reduction* based on the results of these analyses,
- (3) *redundancy elimination* optimizations to improve the compactness of Δ_f .

This section illustrates GPG construction intuitively using the motivating example in Figure 2. The formal details of these steps are provided in later sections.

3.3.1 Creating a GPG and Call Inlining. In order to construct a GPG from a CFG, we first map the CFG naively into a GPG by the following transformations:

- Non-pointer assignments and condition tests are removed (treating the latter as non-deterministic control flow). GPG flow edges are induced from those of the CFG.
- Each pointer assignment labelled s is transliterated to its GPU (denoted γ_s). Figure 4 presented the GPUs for basic pointer assignments in C.
- A singleton GPB is created for every pointer assignment in the CFG.

Then procedure calls are replaced by the optimized GPGs of the callees. The resulting GPG may still contain unresolved calls in the case of recursion and function pointers (Sections 6.2 and 9).

Example 8. The initial GPG for procedure g of Figure 2 is given in Figure 6. Each assignment is replaced by its corresponding GPU. The initial GPG for procedure f is shown in Figure 7 with the call to procedure g on line 09 replaced by its optimized GPG. Examples 9 to 11 in the rest of this section explain the analyses and optimizations over Δ_f and Δ_g at an intuitive level.

3.3.2 Strength Reduction Optimization. This step simplifies GPB δ_s for each statement s by

- performing reaching GPUs analysis; this performs GPU reduction $\gamma \circ \text{RIn}_s$ for each $\gamma \in \delta_s$ which computes a set of GPUs that are equivalent to δ_s , and
- replacing δ_s by the resulting GPUs.

In some cases, the reaching GPUs analysis needs to *block* certain GPUs from participating in GPU reduction (as in Example 1 in Section 2.2) to ensure the soundness of strength reduction. When this happens, redundancy elimination

⁹ This requires a bottom-up traversal of a spanning tree of the call graph starting with its leaf nodes.

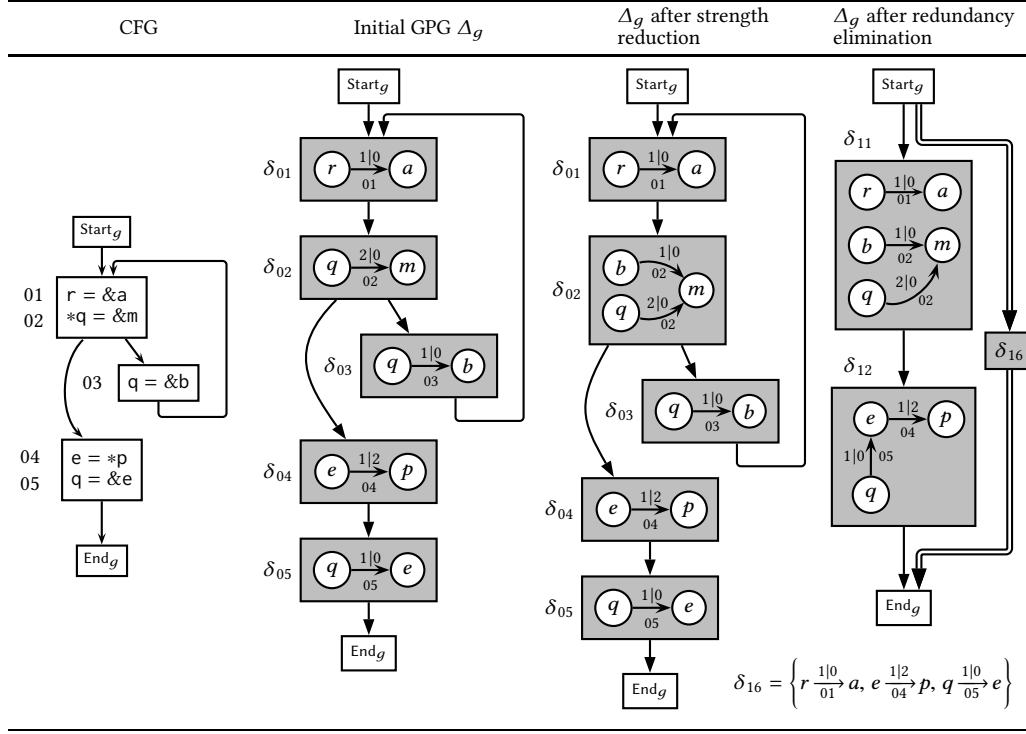


Fig. 6. Constructing the GPG for procedure g (see Figure 2). The edges with double lines are not different from the control flow edges but have been shown separately because they are introduced to represent definition-free paths for the sources of all GPUs that do not appear in GPB δ_{16} . Thus, it is a definition-free path for the sources $(b, 1)$ and $(q, 2)$ of GPUs $b \xrightarrow{1|0}_{02} m$ and $q \xrightarrow{2|0}_{02} m$.

optimizations need to know if the blocked GPUs in a GPG are useful for potential composition after the GPG is inlined in the callers. These two conflicting requirements (of ignoring some GPUs for strength reduction but remembering them for redundancy elimination) are met by performing two variants of reaching GPUs analysis: first with blocking, and then without blocking. There is no instance of blocking in our motivating example, hence we provide an overview only of reaching GPUs analysis without blocking.

Effectively, strength reduction simplifies each GPB as much as possible given the absence of knowledge of aliasing in the caller (Example 1 in Section 2.2). In the process, data dependences are eliminated to the extent possible thereby paving way for redundancy elimination (Section 3.3.3).

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

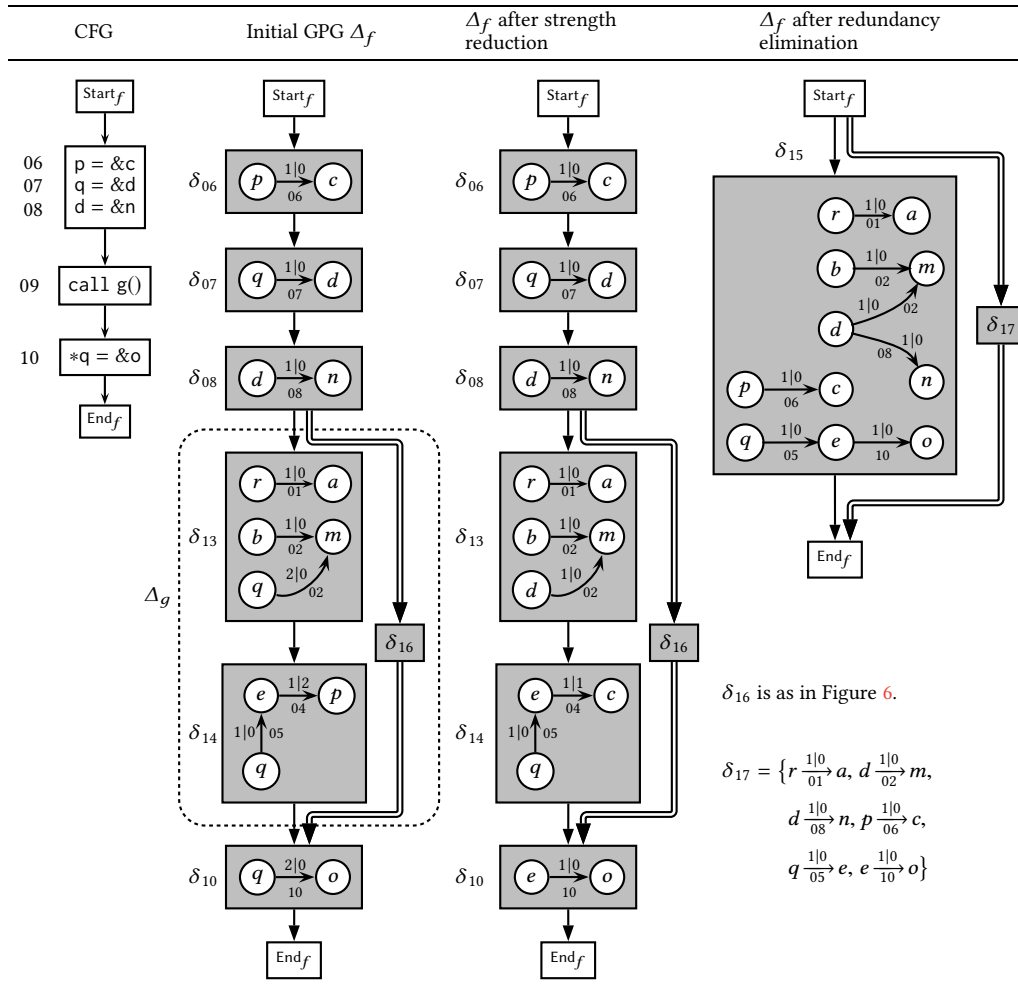


Fig. 7. Constructing the GPG for procedure f (see Figures 2 and 6). GPBs δ_{13} through δ_{14} in the GPG are the (renumbered) GPBs representing the inlined optimized GPG of procedure g . The statement labels in the GPUs of these GPBs remain unchanged. Redundancy elimination of Δ_f coalesces all of its GPBs creating a new GPB δ_{15} . GPB δ_{17} is required for modelling definition-free paths. The edges with double lines are control flow edges shown separately because they are introduced to represent definition-free paths.

follows inlining of GPGs of callee procedures, procedure calls have already been eliminated and hence this analysis is effectively intraprocedural.

There is one last bit of detail which we allude to here and explain in Section 4.3.2 where the analysis is presented formally: For the start GPB of the GPG, RGI_n is initialized to *boundary definitions*¹⁰ that help track definition-free paths to identify variables that are upwards exposed (i.e. live on entry to the procedure and therefore may have additional pointees unknown to the current procedure). This is required for making a distinction between strong and weak

¹⁰The boundary definitions represent *boundary conditions* [1].

updates (Sections 2.1.3 and 4.3.2). For the purpose of this overview, we do not show boundary definitions in our example below. They are explained in Example 16 in Section 4.3.2.

Example 9. We intuitively explain the reaching GPUs analysis for procedure g over its initial GPG (Figure 6). The final result is shown later in Figure 11. Since we ignore boundary definitions for now, the analysis begins with $\text{RGIIn}_{01} = \emptyset$. Further, since we compute the least fixed point, RGOOut values are initialized to \emptyset for all statements. The GPU corresponding to the assignment in statement 01 $\gamma_1 : r \xrightarrow{1|0} a$, forms RGOOut_{01} and RGIIn_{02} . For statement 02, $\text{RGIIn}_{02} = \{r \xrightarrow{1|0} a\}$ and $\text{RGGGen}_{02} = \{q \xrightarrow{2|0} m\}$. $\text{RGIKill}_{02} = \emptyset$ and RGOOut_{02} is computed using RGIIn_{02} which also forms RGIIn_{03} which is $\{r \xrightarrow{1|0} a, q \xrightarrow{2|0} m\}$. For statement 03, $\gamma_3 : q \xrightarrow{1|0} b$ forms RGGGen_{03} . In the second iteration of the analysis over the loop, we have $\text{RGIIn}_{01} = \text{RGOOut}_{03} = \{r \xrightarrow{1|0} a, q \xrightarrow{2|0} m, q \xrightarrow{1|0} b\}$. RGIIn_{02} is also the same set. Composing $\gamma_2 : q \xrightarrow{2|0} m$ with $q \xrightarrow{1|0} b$ in RGIIn_{02} results in the GPU $b \xrightarrow{1|0} m$. Also, the pointee information of q is available only along one path (identified with the help of boundary definitions that are not shown here) and hence the assignment causes a weak update and the GPU $q \xrightarrow{2|0} m$ is also retained. Thus, RGGGen_{02} is now updated and now contains two GPUs: $b \xrightarrow{1|0} m$ and $q \xrightarrow{2|0} m$. This process continues until the least fixed point is reached. Strength reduction optimization after reaching GPUs analysis gives the GPG shown in the third column of Figure 6 (the fourth column represents the GPG after redundancy elimination optimizations and is explained in Section 3.3.3).

3.3.3 Redundancy Elimination Optimizations. This step performs the following optimizations across GPBs to improve the compactness of a GPG.

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

Example 10. In procedure g of Figure 6, pointer q is defined in statement 03 but is redefined in statement 05 and hence the GPU $q \xrightarrow{1|0} b$ is eliminated. Hence the GPB δ_{03} becomes empty and is removed from the GPG of procedure g (Δ_g). Note that GPU $q \xrightarrow{2|0} m$ does not define q but its pointee and hence is not killed by statement 05. Thus it is not eliminated from Δ_g .

For procedure f in Figure 7, the GPU $q \xrightarrow{1|0} d$ in δ_{07} is killed by the GPU $q \xrightarrow{1|0} e$ in δ_{14} . Hence the GPU $q \xrightarrow{1|0} d$ is eliminated from the GPB δ_{07} which then becomes empty and is removed from the optimized GPG. Similarly, the GPU $e \xrightarrow{1|1} c$ in GPB δ_{14} is removed because e is redefined by the GPU $e \xrightarrow{1|0} o$ in the GPB δ_{10} (after strength reduction in Δ_f). However, GPU $d \xrightarrow{1|0} n$ in GPB δ_{08} is not removed even though δ_{13} contains a definition of d expressed by GPU $d \xrightarrow{1|0} m$. This is because δ_{13} also contains GPU $b \xrightarrow{1|0} m$ which defines b , indicating that d is not defined along all paths. Hence the previous definition of d cannot be killed—giving a weak update.

Finally, we eliminate the redundant control flow in the GPG by perform coalescing analysis (Section 5.2). It partitions the GPBs of a GPG (into *parts*) such that all GPBs in a part are coalesced (i.e., a new GPB is formed by taking a union of the GPUs of all GPBs in the part) and control flow is retained only across the new GPBs representing the parts. Given a GPB δ_s in part π_i , we can add its adjacent GPB δ_t to π_i provided the *may* property (Section 3.1) of π_i is preserved. This is possible if the GPUs in π_i and δ_t do not have a data dependence between them.

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

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

Coalescing two GPBs that do not have control flow between them may eliminate a definition-free path for the GPUs in it (see the Example 11 below). We handle this situation as follows: We create an artificial GPB by collecting all GPUs that do not have a definition-free path in the GPG. We add a path from start to end via this GPB. This introduces a definition-free path for all GPUs that do not appear in this GPB.

Example 11. For procedure g in Figure 6, the GPBs δ_1 and δ_2 can be coalesced: there is no data dependence between their GPUs because GPU $r \xrightarrow{1|0}{01} a$ 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 *`. Since types do not match, there is no data dependence.

The GPUs in δ_2 and δ_4 contain a dereference whose data dependence is unknown. We therefore use the type information. Since both q and p have the same types, there is a possibility of RaW data dependence between the GPUs $q \xrightarrow{2|0}{02} m$ and $e \xrightarrow{1|2}{04} p$ (p and q could be aliased in the caller). Thus, we do not coalesce the GPBs δ_2 and δ_4 . Also, there is no RaW dependence between the GPUs in the GPBs δ_4 and δ_5 and we coalesce them; recall that potential WaR dependence does not matter because of the *may*-property of GPBs (see Example 6).

The GPB resulting from coalescing GPBs δ_1 and δ_2 is labelled δ_{11} . Similarly, the GPB resulting from coalescing GPBs δ_4 and δ_5 is labelled δ_{12} . The loop formed by the back edge $\delta_2 \rightarrow \delta_1$ in the GPG before coalescing now reduces to a self loop over δ_{11} . Since the GPUs in a GPB do not have a dependence between them, the self loop $\delta_{11} \rightarrow \delta_{11}$ is redundant and is removed.

For procedure f in Figure 7, after performing dead GPU elimination, the remaining GPBs in the GPG of procedure f are all coalesced into a single GPB δ_{15} because there is no data dependence within the GPUs of its GPBs.

As exemplified in Example 10, the sources of the GPUs $b \xrightarrow{1|0}{02} m$ and $q \xrightarrow{2|0}{02} m$ in procedure g are not defined along all paths from Start_g to End_g leading to a weak update. This is modelled by introducing a definition-free path (shown by edges with double lines in the fourth column of Figure 6). Thus for procedure g , we have GPB δ_{16} that contains all GPUs of Δ_g that are defined along all paths to create a definition-free path for those that are not. Similarly, for procedure f , we have a definition-free path for the source of GPU $b \xrightarrow{1|0}{02} m$ (as shown in the fourth column of Figure 7). The GPB δ_{17} contains all GPUs of Δ_f except $b \xrightarrow{1|0}{02} m$. GPU $q \xrightarrow{2|0}{02} m$ which has a definition-free path in Δ_g , reduces to $d \xrightarrow{1|0}{02} m$ in Δ_f . Since d is also defined in δ_{08} , it does not have a definition-free path in Δ_f .

3.4 The Big Picture

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

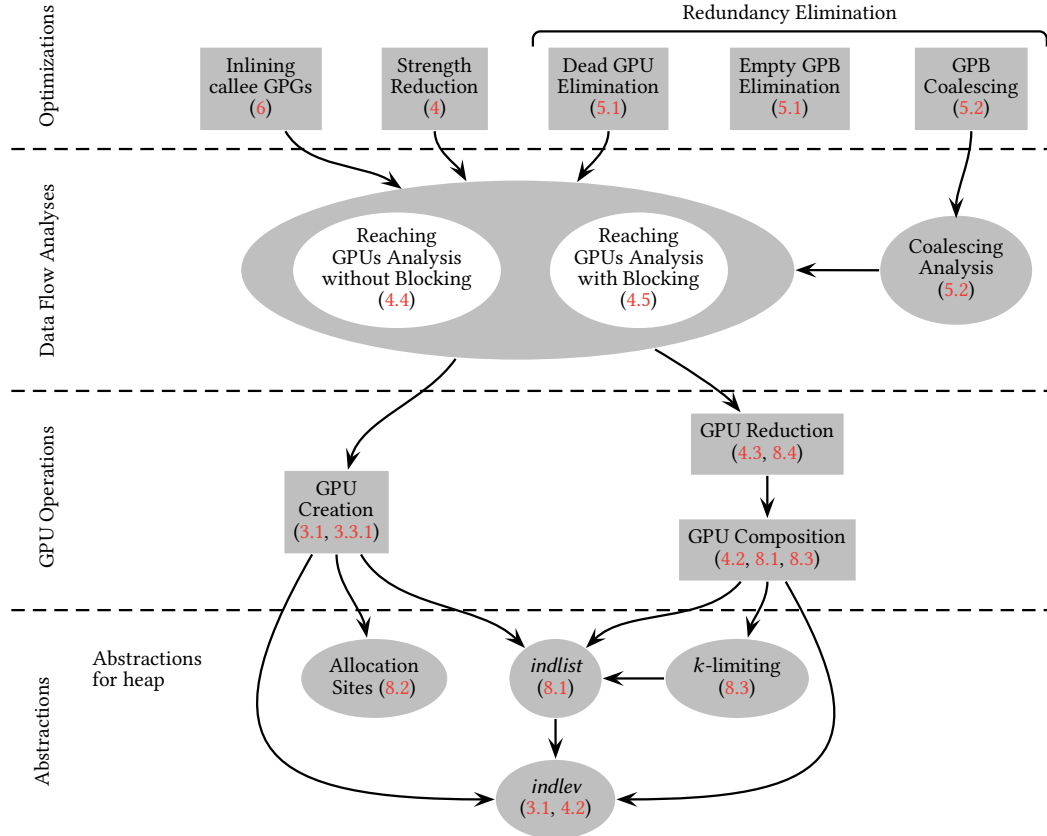


Fig. 8. The big picture of GPG construction as a fleshed out version of Figure 3. The arrows show the dependence between specific instances of optimizations, analyses, operations, and abstractions. The results of the two variants of reaching GPUs analysis are required together. The optimization of empty GPB removal does not depend on any data flow analysis. The labels in parentheses refer to relevant sections.

Figure 8 is a fleshed out version of Figure 3. It provides the big picture of GPG construction by listing specific abstractions, operations, data flow analyses, and optimizations and shows dependences between them. The optimizations use the results of data flow analyses. The two variants of reaching GPUs analysis are the key analyses; they have been clubbed together because their results are required together. They use the GPU operations which are defined in terms of key abstractions. Empty GPB removal does not require a data flow analysis.

4 STRENGTH REDUCTION OPTIMIZATION

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

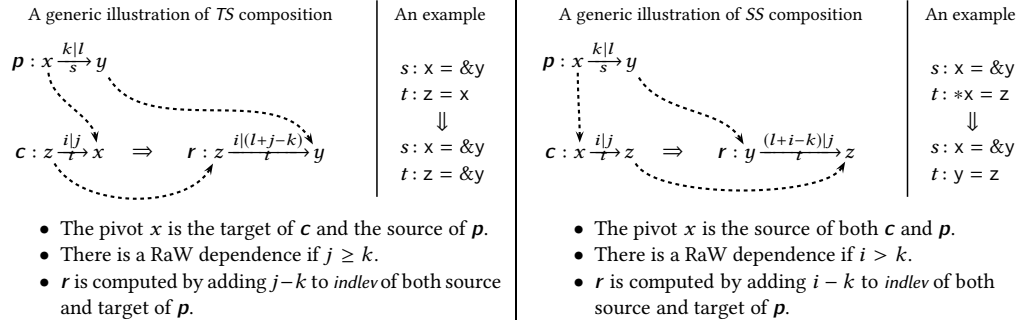


Fig. 9. Composing a consumer GPU c with a producer GPU p to compute a new GPU r which is equivalent to c in the context of p . Both *SS* and *TS* compositions exploit a RaW dependence of statement at t on the statement at s because the pointer defined in p is used to simplify a pointer used in c .

4.1 An Overview of Strength Reduction Optimization

Recall that the construction of a GPG of a procedure begins by transliterating each pointer assignment labelled s in the CFG of the procedure into a GPB δ_s containing the singleton GPU corresponding to the assignment. Then the GPUs are simplified by composing them with other GPUs. This simplification progressively converts a GPU to a classical points-to edge; as noted in Section 2.2. Some simplifications can be done immediately while others are blocked awaiting knowledge of aliasing in the callers and so are postponed. They are reconsidered in the calling context after the GPG is inlined as a procedure summary in its callers. The strength reduction optimization then replaces every GPU $\gamma \in \delta_s$ with its simplified version.

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

In some cases, the location read by c could be different from the location defined by p due to the presence of a GPU b (called a *barrier*) corresponding to an intervening assignment. The GPU p may be updated by the GPU b depending on the aliases in the calling context (Section 2.2). This could happen because the *indlev* of the source of p or b is greater than 1 indicating that the pointer being defined by this GPU is still not known. In such a situation (characterized formally in Section 4.5.1), replacing δ_s by $\text{RGG}e_n_s$ during strength reduction may be unsound. To ensure soundness, we need to *postpone* the composition $c \circ^\tau p$ explicitly by eliminating those GPUs from R which are blocked by a barrier.¹¹ We do this by performing a variant of reaching GPUs analysis called the *reaching GPUs analysis with blocking* that identifies GPUs blocked by a barrier (Section 4.5). We distinguish the two variants by using the phrase *reaching GPUs analysis without blocking* for the earlier reaching GPUs analysis. For strength reduction, it is sufficient to perform reaching GPUs

¹¹Formally the term ‘barrier’ applies to a GPU, but we abuse this and refer to its associated statement as a barrier too.

Possible SS Compositions			Possible TS Compositions		
Statement sequence	Memory graph after the stmt. sequence	GPU _s	Statement sequence	Memory graph after the stmt. sequence	GPU _s
$i < k$			$j < k$		
<div style="border: 1px solid black; padding: 2px; display: inline-block;">Ex. ss1</div> $*x = \&y$ $x = \&z$		$p: x \xrightarrow{2 0} y$ $c: x \xrightarrow{1 0} z$ <i>(invalid)</i>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">Ex. ts1</div> $*x = \&y$ $z = x$		$p: x \xrightarrow{2 0} y$ $c: z \xrightarrow{1 1} x$ <i>(invalid)</i>
$i > k$			$j > k$		
<div style="border: 1px solid black; padding: 2px; display: inline-block;">Ex. ss2</div> $x = \&y$ $*x = \&z$		$p: x \xrightarrow{1 0} y$ $c: x \xrightarrow{2 0} z$ $r: y \xrightarrow{1 0} z$	<div style="border: 1px solid black; padding: 2px; display: inline-block;">Ex. ts2</div> $x = \&y$ $z = *x$		$p: x \xrightarrow{1 0} y$ $c: z \xrightarrow{1 2} x$ $r: z \xrightarrow{1 1} y$
$i = k$			$j = k$		
<div style="border: 1px solid black; padding: 2px; display: inline-block;">Ex. ss3</div> $*x = \&y$ $*x = \&z$		$p: x \xrightarrow{2 0} y$ $c: x \xrightarrow{2 0} z$ <i>(invalid)</i>	<div style="border: 1px solid black; padding: 2px; display: inline-block;">Ex. ts3</div> $x = \&y$ $z = x$		$p: x \xrightarrow{1 0} y$ $c: z \xrightarrow{1 1} x$ $r: z \xrightarrow{1 0} y$

Fig. 10. Illustrating the *validity* of SS and TS compositions based on the *indlevs* of pivot (x in these examples) in the consumer GPU c and producer GPU p .

analysis with blocking. However, redundancy elimination optimizations need to know whether the blocked GPUs in a GPG are useful for potential composition after the GPG is inlined in the callers. These two conflicting requirements force us to perform both the variants of reaching GPUs analysis: first with blocking, and then without blocking.

Section 4.2 defines GPU composition as a family of partial operations. Section 4.3 defines GPU reduction. Section 4.4 provides data flow equations for reaching GPUs analysis without blocking while Section 4.5 provides data flow equations for reaching GPUs analysis with blocking.

4.2 GPU Composition

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

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

We name the compositions as *TS* or *SS* where the first letter indicates the role of the pivot in c and second letter indicates its role in p . If the pivot is the target of c and the source of p , the composition is called a *TS* composition. If the pivot is the source of both c and p , the composition is called an *SS* composition. We remark for completeness that there are two further GPU-composition operations which can be applied when the pivot is the target of p . These are called *ST* and *TT* compositions which are optional and we do not use them here. However, *TS* and *SS* compositions are sufficient to convert a GPU to a classical points-to edge.

Figure 9 illustrates *TS* and *SS* compositions. For *TS* composition, consider GPUs $c:z \xrightarrow{i|j} x$ and $p:x \xrightarrow{k|l} y$ with a pivot x which is the target of c and the source of p . The goal of GPU composition is to join the source z of c and the target y of p by using the pivot x as a bridge. This requires the *indlevs* 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 *indlevs* of the source and target of p to view the base GPU p in its derived form as $x \xrightarrow{j|(l+j-k)} y$. This balances the *indlevs* of x in the two GPUs allowing us to create a simplified GPU $r:z \xrightarrow{i|(l+j-k)} y$. (Given a GPU $x \xrightarrow{i|j} y$, we can create a GPU $x \xrightarrow{(i+1)|(j+1)} y$ based on the type restrictions on the *indlevs* of x and 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 $r = c \circ^{\tau} p$ is *valid* only if c reads a location defined by p and this read/write happens through the pivot of the composition.
- (b) A composition $r = c \circ^{\tau} p$ is *desirable* only if the *indlev* of r does not exceed the *indlev* of c .

Validity requires the *indlev* of the pivot in c to be greater than the *indlev* of pivot in p . For the generic *indlevs* used in Figure 9, this requirement translates to the following constraints:

$$j \geq k \quad (TS \text{ composition}) \quad (1)$$

$$i > k \quad (SS \text{ composition}) \quad (2)$$

Observe that *SS* composition condition (2) prohibits equality unlike the condition for *TS* composition (1). This is because of the fact that *SS* composition involves the source nodes of both the GPUs and when $i = k$, c overwrites the location written by p ; for a location written by p to be read by c in its source, i must be strictly greater than k .

Example 12. The following (attempted) compositions in Figure 10 are *invalid* because c does not read a location defined by p .

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

The following compositions in Figure 10 are *valid* because c reads a location defined by p .

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

The *desirability* of GPU composition characterizes progress in conversion of GPUs into classical points-to edges by ensuring that the *indlev* of the new source and the new target in r does not exceed the corresponding *indlev* in the consumer GPU c . This requires the *indlev* in the simplified GPU r and the consumer GPU c to satisfy the following constraints. In each constraint, the first term in the conjunct compares the *indlevs* of the sources of c and r while the

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

Definition 3. GPU Composition $c \circ^\tau p$

second term compares those of the targets (see Figure 9):

$$(i \leq i) \wedge (l + j - k \leq j) \quad \text{or equivalently} \quad l \leq k \quad (TS \text{ composition}) \quad (3)$$

$$(l + i - k \leq i) \wedge (j \leq j) \quad \text{or equivalently} \quad l \leq k \quad (SS \text{ composition}) \quad (4)$$

Example 13. Consider the statement sequence $x = *y; z = x$. A *TS* composition of the corresponding GPUs $p : x \xrightarrow{1|2} y$ and $c : z \xrightarrow{1|1} x$ is *valid* because $j = k = 1$ satisfying Constraint 1. However, if we perform this composition, we get $r : z \xrightarrow{1|2} y$. Intuitively, this GPU is not useful for computing a points-to edge because the *indlev* of r is “1|2” which is greater than the *indlev* of c which is “1|1”. Formally, this composition is flagged *undesirable* because $l = 2$ which is greater than $k = 1$ violating Constraint 3.

We take a conjunction of the constraints of *validity* (1 and 2) and *desirability* (3 and 4) to characterize *admissible* GPU compositions.

$$l \leq k \leq j \quad (TS \text{ composition}) \quad (5)$$

$$l \leq k < i \quad (SS \text{ composition}) \quad (6)$$

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 $r = c \circ^\tau p$ by balancing the *indlev* of the pivot in both the GPUs provided the composition (*TS* or *SS*) is *admissible*. Otherwise it fails—being a partial operation. Note that *TS* and *SS* compositions are mutually exclusive for a given pair of c and p because a variable cannot occur both in the RHS and the LHS of a pointer assignment in the case of pointers to scalars.¹²

¹² Since our language is modelled on C, GPUs for statements such as $*x = x$ or $x = *x$ are prohibited by typing rules; GPUs for statements such as $*x = *x$ are ignored as inconsequential. Further, we assume as allowed by C-standard *undefined behaviour* that the programmer has not abused type-casting to simulate such prohibited statements. Section 8 considers the richer situation with structs and unions where we can have an assignment $x \rightarrow n = x$ which might have both *TS* and *SS* compositions with a GPU p that defines x .

```

Input:  $c$            // The consumer GPU to be simplified
          $R$            // The context (set of GPUs) in which  $c$  is to be simplified
Output: Red       // The set of simplified GPUs equivalent to  $c$ 
01 GPU_reduction ( $c, R$ )
02 { Red =  $\emptyset$ 
03    $W = \{c\}$ 
04   while ( $W \neq \emptyset$ )
05   { extract  $w$  from  $W$ 
06      $composed = \text{false}$ 
07     for each  $\gamma \in R$ 
08     { if ( $r = w \circ^{ts} \gamma$ ) succeeds
09       {  $W = W \cup \{r\}$ 
10          $composed = \text{true}$ 
11       }
12     else if ( $r = w \circ^{ss} \gamma$ ) succeeds
13     {  $W = W \cup \{r\}$ 
14        $composed = \text{true}$ 
15     }
16   }
17   if ( $\neg composed$ )
18     Red = Red  $\cup \{w\}$ 
19   }
20   return Red
21 }

```

Definition 4. GPU Reduction $c \circ R$

4.3 GPU Reduction

GPU reduction $c \circ R$ uses the GPUs in R to compute a set of GPUs Red whose *indlevs* do not exceed that of c . The result of GPU reduction $c \circ R$ must ensure the semantic equivalence of Red with c in the context of R . The set R is computed using reaching GPUs analysis without blocking (Section 4.4). In some cases, we need to restrict R using the reaching GPUs analysis with blocking (Section 4.5) to ensure this semantic equivalence.

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

4.3.1 Defining GPU Reduction $c \circ R$. Definition 4 gives the algorithm for GPU reduction. The worklist W is initialized to $\{c\}$. A reduced GPU is added to W for further GPU compositions. When a GPU w cannot be reduced any further, the flag *composed* remains **false** and w is added to Red (lines 17 and 18 of Definition 4). This algorithm assumes that

the graph induced by the GPUs in R is acyclic. This holds for scalar pointers. However, in the presence of structures the graph may contain cycles via fields of structures; Section 8.4 extends the algorithm to handle cycles.

Example 14. Consider the statements on the right. For $c : x \xrightarrow{1|2} y$, $R = \{y \xrightarrow{1|0} a, a \xrightarrow{1|0} b\}$. The reduction $c \circ R$ involves two consecutive TS compositions. The first composition involves $y \xrightarrow{1|0} a$ as p , resulting in $r = x \xrightarrow{1|1} a$ which is added to the worklist. In the second iteration of the **while** loop on line 04 of Definition 4, the reduced GPU $x \xrightarrow{1|1} a$ in the previous iteration now becomes the consumer GPU. It is composed with $a \xrightarrow{1|0} b$ which results in a reduced GPU $x \xrightarrow{1|0} b$. This GPU is added to the worklist. However, since it cannot be reduced further as it is already in the classical points-to form, the loop terminates. The flag *composed* remains **false** for the final GPU $x \xrightarrow{1|0} b$ because no further composition is possible and $\text{Red} = \{x \xrightarrow{1|0} b\}$.

$y \xrightarrow{1 0} a$ as p ,	21 $y = \&a;$
resulting in $r = x \xrightarrow{1 1} a$	22 $a = \&b;$
which is added to the worklist.	23 $x = *y;$

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

- A GPU w extracted from the worklist will never be added to it again. If there is no reduction, then w is added to Red directly. This is ensured by setting the flag *composed* appropriately.
- Reduction of *indlev* of source and target of a GPU w is performed independently, hence there is no oscillation across iterations of fixed-point computation.
- The process terminates only when the GPUs in Red are either in their simplified form or no more GPUs are available in R for further GPU compositions.
- The order in which a GPU γ is selected from R for composition with w does not matter because of the following properties of R that are established by the reaching GPUs analysis with and without blocking (Sections 4.4 and 4.5).

Consider two GPUs γ_1 and γ_2 in R . Then γ_1 and γ_2 cannot compose with each other: If the composition $\gamma_2 \circ \gamma_1$ were possible, it would have been performed during the reaching GPUs analysis (Section 4.4) and γ_2 would not exist in R because it would be replaced by the result of the composition. Similarly if the composition $\gamma_1 \circ \gamma_2$ were possible, γ_1 would not exist in R . Hence we examine the possible reasons of existence of both γ_1 and γ_2 in R and explain why the order of performing the compositions $w \circ \gamma_1$ and $w \circ \gamma_2$ does not matter.

- (a) There is no data dependence between γ_1 and γ_2 because there is no pivot between them or one does not follow the other on any control flow path. Hence a composition between them is ruled out. In this case, the order between $w \circ \gamma_1$ and $w \circ \gamma_2$ is irrelevant because of the absence of data dependence between γ_1 and γ_2 .
- (b) There is data dependence between γ_1 and γ_2 potentially enabling a composition. Without any loss of generality, consider the composition $\gamma_2 \circ \gamma_1$. Then there are two possibilities that may have prohibited the composition:
 - (i) $\gamma_2 \circ \gamma_1$ is *inadmissible* because it is *undesirable*. Then, $w \circ \gamma_1$ also is *undesirable* because the *desirability* constraint is based solely on the *indlev* of γ_1 (Constraints 3 and 4). Thus w may compose only with γ_2 and the issue of an order between $w \circ \gamma_1$ and $w \circ \gamma_2$ does not arise.
 - (ii) $\gamma_2 \circ \gamma_1$ is *admissible* but has been postponed because of a barrier (introduced in Section 2.2 and explained later in Section 4.5) between γ_2 and γ_1 . In this case, the barrier also prohibits a composition of w with γ_1 and it can compose only with γ_2 . Thus the issue of an order between $w \circ \gamma_1$ and $w \circ \gamma_2$ does not arise.

4.3.2 Modelling Caller-Defined Pointer Variables. In abstract memory, we may be uncertain as to which of several locations a variable points to. Hence, for an indirect assignment ($*p = \&x$ say), GPU reduction returns a set of GPUs which define multiple pointers (or different pointees of the same pointer) leading to a weak update. In this case we do not overwrite any of its pointees, but merely add $\&x$ to the possible values they can contain. Sometimes however, we may discover that p has a single pointee within the procedure and conclude that there is only one possible abstract location defined by the assignment. In this case we may, in general, *replace* the contents of this location. This is a strong update. However, this is necessary but not sufficient for a strong update because the pointer may not be defined along all paths—there may be a path along which the pointer (or some pointee of the pointer) may not be defined within the procedure but may be 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. Also, the effect of definition-free paths has to be taken into account during strength reduction optimization: if γ_1 is simplified to γ_2 , γ_2 can replace γ_1 provided there is no definition-free path reaching γ_1 ; otherwise γ_1 should also be included with γ_2 to allow the composition of γ_1 with the GPUs in a caller.

Example 15. Figure 6 shows the set of GPUs corresponding to statement 02 (δ_{02} in the GPG after strength reduction) of procedure g of Figure 2. There is a definition-free path for q meaning that δ_{11} in the optimized Δ_g must include GPU $q \xrightarrow{2|0} m$ along with its reduced GPU $b \xrightarrow{1|0} m$.

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

The boundary definitions are introduced for global variables and formal parameters because they could be read in a procedure before being defined. They are symbolic in that they are not introduced in the GPG of a procedure but are included in RGIn of the Start GPB during reaching GPUs analysis. They are of the form $x \xrightarrow{\ell|\ell} x'$ where x' is a symbolic representation of the initial value of x at the start of the procedure and ℓ ranges from 1 to the maximum depth of the indirection level which depends on the type of x , and 00 is the label of the Start GPB. For type (int **), ℓ ranges from 1 to 2. Variable version x' is called the *upwards exposed* [15] version of x . This is similar to Hoare-logic style specifications in which postconditions use (immutable) *auxiliary variables* x' to be able to talk about the original value of variable x (which may have since changed). Our upwards-exposed versions serve a similar purpose, so that logically on entry to each procedure the statement $x = x'$ provides a definition of x .

A reduced GPU $x \xrightarrow{i|j} y$ along any path kills the boundary definition $x \xrightarrow{i|i} x'$ on that path indicating that $(i - 1)^{th}$ pointees of x are redefined. Including boundary definitions at the start ensures that if a boundary definition $x \xrightarrow{i|i} x'$ reaches a program point s , there is a definition-free path from Start to s ; its absence at s guarantees that the source of $x \xrightarrow{i|i} x'$ has been defined along all paths reaching s . This leads to a simple necessary and sufficient condition for strong updates: All GPUs corresponding to a statement s must define the same location.

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

Example 16. Consider reaching GPUs analysis for the GPB corresponding to statement 02 in the initial GPG of procedure g (δ_{02} in Figure 6). We include the boundary definitions for each global variable and the parameters of a procedure as RGIn of the Start GPB of the GPG of procedure g . Although Figure 6 does not show boundary definitions for simplicity, they are shown in Figure 11 for variable q (boundary definitions of other variables are not

required for strong updates in this example). These boundary definitions capture the effect of definition-free paths to distinguish between weak and strong updates.

The GPU $\gamma_2 : q \xrightarrow{2|0}{02} m$ is composed with GPUs from RGIIn_{02} which contains a GPU $q \xrightarrow{1|0}{03} b$ indicating that pointer b is being defined by statement 02. However, this is not the case of strong update as b is not the only pointer that is being defined by the assignment. There is a definition-free path along which pointee of q is not available indicating that q may have a definition the callers of procedure g which is also required in statement 02 of g but is currently unavailable. The presence of boundary definition $q \xrightarrow{1|1}{00} q'$ in RGIIn_{02} indicates the presence of a definition-free path and the composition of this GPU results in a reduced GPU $q' \xrightarrow{2|0}{02} m$ which is also a part of δ_{02} . The GPU $q' \xrightarrow{2|0}{02} m$ has been represented by the GPU $q \xrightarrow{2|0}{02} m$ in Figure 6 because it ignores boundary definitions.

At the call site in procedure f , after the composition of GPU $q \xrightarrow{1|0}{07} d$ and $q \xrightarrow{2|0}{02} m$ (the upwards-exposed version q' is replaced by q during call inlining; for more details see Section 6), the set of reduced GPUs corresponding to statement 02 in procedure f (GPB δ_{13}) contains two GPUs $b \xrightarrow{1|0}{02} m$ and $d \xrightarrow{1|0}{02} m$ (Figure 7). Since, the assignment defines two pointers d and b , no GPU is removed and hence the GPU $d \xrightarrow{1|0}{08} n$ in GPB δ_{08} is retained owing to a weak update.

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

4.4 Reaching GPUs Analysis without Blocking

In this section, we present the data flow equations for computing RGIIn and RGOOut for every GPB δ in the GPG of a procedure. These equations ignore the effect of barriers; Section 4.5 incorporates the effects of barriers and performs reaching GPUs analysis with blocking to compute $\overline{\text{RGIIn}}$ and $\overline{\text{RGOOut}}$ for every GPB δ .

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

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

GPU reduction allows us to model Kill (i.e., GPU removal from RGIIn) in the case of strong update as follows: The reduced GPUs should define the same pointer (or the same pointee of a given pointer) along every control flow path reaching the statement represented by γ . This is captured by the requirement $|\text{Def}(X, \gamma)| = 1$ in the definition of $\text{Kill}(X, R)$ in Definition 5 where $\text{Def}(X, \gamma)$ extracts the source nodes and their indirection levels of the GPUs (i.e. pair (x, i) for GPU $x \xrightarrow{i|j}{s} y$) in X that are constructed for the same statement s . The GPUs that are killed are determined by the GPUs in RGGGen_s and not those in δ_s .

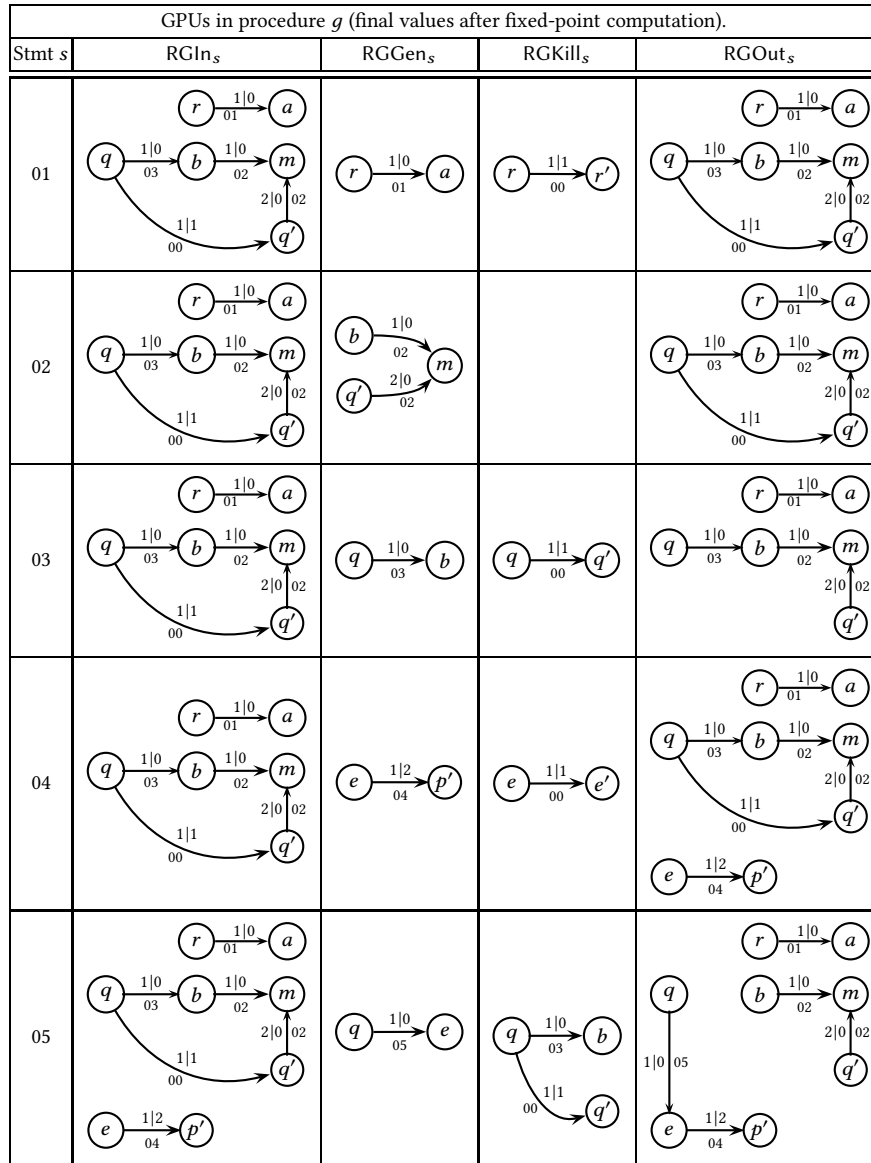


Fig. 11. The data flow information computed by reaching GPUs analysis for procedure g of the motivating example given in Figure 2. In RGI n and RGI Out , we show only one boundary definition $q \xrightarrow{1|1}_{00} q'$ because other boundary definitions do not participate in GPU reduction for this example. However, the boundary definitions that are removed are shown in RGI $Kill$.

Example 17. Figure 11 gives the final result of reaching GPUs analysis for procedure g of our motivating example. We have shown the boundary GPU $q \xrightarrow{1|1}_{00} q'$ for q . Other boundary GPUs are not required for strong updates in this example and have been omitted. This result has been used to construct GPG Δ_g shown in Figure 6. For procedure

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

Definition 5. Data flow equations for Reaching GPUs Analysis without Blocking

f , we do not show the complete result of the analysis but make some observations. The GPU $q \xrightarrow{2|0}{10} o$ is composed with the GPU $e \xrightarrow{1|0}{05} e$ to create a reduced GPU $e \xrightarrow{1|0}{10} o$. Since, only a single pointer (in this case e) is being defined by the assignment, this is a case of strong update and hence kills $e \xrightarrow{1|1}{04} c$. The GPU to be killed is identified by $\text{Match}(e \xrightarrow{1|0}{10} o, \text{RGIn}_{10})$ 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 $e \xrightarrow{1|0}{10} o$) and not the consumer GPU (in this case $q \xrightarrow{2|0}{10} o$).

4.5 Reaching GPUs Analysis with Blocking

Given a GPB δ_s , strength reduction seeks to replace a consumer GPU $c \in \delta_s$ with the GPUs obtained by reducing c . During GPU reduction, it is possible that c has an *admissible* composition with some producer GPU p , but the location read by c could be different from the location defined by p due to the presence of a barrier GPU b (Sections 2.2 and 4.1). The barrier may change the pointer chain established by p thereby altering the data dependence between p and c . In this case, c should not be composed with p and should be left unsimplified. If $c \circ^\tau p$ is performed, then RGGen_s will not contain c . Hence, when strength reduction optimization replaces δ_s by RGGen_s , c will be replaced by the result of composition, possibly leading to unsoundness.

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

4.5.1 The Need of Blocking. The location read by a GPU c could be different from the location defined by p because of a combined effect of the GPUs in a calling context and the GPUs corresponding to the intervening assignments on a control flow path from p to c which may update the GPU p . We characterize these situations by building on Section 2.2 and defining the notion of a *barrier GPU* which *blocks* certain GPUs so that GPU compositions leading to potentially unsound strength reduction optimization are *postponed*. After inlining the GPG in a caller, more information may

<pre> int a, b, *p, *q, **x; 01 void h() 02 { p = &a; /* GPU p */ 03 *x = &b; /* GPU b */ 04 q = p; /* GPU c */ 05 } </pre> <p>If x points-to p then q points-to b else q points-to a.</p> <p>(a) Composition across an indirect GPU b</p>	<pre> int a, b, *p, *q, **x; 01 void h() 02 { *x = &a; /* GPU p */ 03 p = &b; /* GPU b */ 04 q = *x; /* GPU c */ 05 } </pre> <p>If x points-to p then q points-to b else q points-to a.</p> <p>(b) Composition with an indirect GPU across the GPU b</p>
--	--

Fig. 12. Risk of unsoundness in GPU reduction caused by a barrier GPU.

become available. Thus, it may resolve any uncertain data dependence between c and p —so a composition which was earlier postponed may now safely be performed. This is explained in the rest of the section.

We define a barrier as follows. Let an *indirect* GPU refer to a GPU whose *indlev* of the source is greater than 1 (i.e., the pointer being defined by the GPU is not known). Then, a GPU b corresponding to an assignment between c and p on some control flow path is a barrier if:

- b is an indirect GPU. This is a composition across an indirect GPU b (Figure 12(a)).
- p is an indirect GPU (b need not be an indirect GPU). This is a composition with an indirect GPU across the GPU b (Figure 12(b)).

We illustrate these situations in the following example.

Example 18. Consider the procedure in Figure 12(a). The composition between the GPUs for statements 02 and 04 is *admissible*. However, statement 03 may cause a side-effect by indirectly defining p (if x points to p in the calling context). Thus, q in statement 04 would point to b if x points to p ; otherwise it would point to a . If we replace the GPU $q \xrightarrow{1|1}_{04} p$ by $q \xrightarrow{1|0}_{04} a$ (which is the result of composing $q \xrightarrow{1|1}_{04} p$ with $p \xrightarrow{1|0}_{02} a$), then we would miss the GPU $q \xrightarrow{1|0}_{04} b$ if x points to p in the calling context—leading to unsoundness. Since the calling context is not available during GPG construction, we postpone this composition to eliminate the possibility of unsoundness. This is done by blocking the GPU $p \xrightarrow{1|0}_{02} a$ by an indirect GPU $x \xrightarrow{2|0}_{03} b$ which acts as a barrier. This corresponds to the first case described above.

For the second case, consider statement 02 of the procedure in Figure 12(b) which may indirectly define p (if x points to p). Statement 03 directly defines p . Thus, q in statement 04 would point to b if x points to p ; otherwise it would point to a . We postpone the composition $c : q \xrightarrow{1|2}_{04} x$ with $p : x \xrightarrow{2|0}_{02} a$ by blocking the GPU p where the GPU $p \xrightarrow{1|0}_{03} b$ acts as a barrier.

A barrier GPU is likely to have a WaW or WaR dependence with some preceding GPUs which cannot be ascertained without the alias information in the calling context. In the absence of alias information from the calling context, we use the type information to identify some such GPUs as non-blocking. The barrier blocks such GPUs, so that the compositions of c with them are postponed (Section 2.2). Consider a GPU p originally blocked by a barrier b where p or b is an indirect GPU. 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 (p or b) is reduced to 1 thereby identifying the pointer being defined by the GPU. In this case, b ceases being a barrier and so no longer blocks p leading to the following two situations:
 - (a) b redefines the pointer defined by p , killing p thereby obviating the composition $c \circ^\tau p$.
 - (b) b does not redefine the pointer defined by p thereby allowing the composition $c \circ^\tau p$.
- (2) The *indlev* of the source of the indirect GPU (p or b) remains greater than 1. In this case, b continues to block p awaiting further inlining.

In case 1(a), an eager reduction of c without blocking p would cause c to be replaced by the result of composition $c \circ^\tau p$, thereby causing unsoundness. Reaching GPUs analysis with blocking helps to postpone the composition until all information becomes available. 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.

Example 19. Case 1(a) above could arise if x points to p in the calling context of the procedure in Figure 12(a). As a result, GPU $p \xrightarrow{1|0}{02} a$ is killed by the barrier GPU $p \xrightarrow{1|0}{03} b$ (which is the simplified version of the barrier GPU $x \xrightarrow{2|0}{03} b$) and hence the composition is prohibited and q points to b for statement 04. Case 1(b) could arise if x points to any location other than p in the calling context. In this case, the composition between $q \xrightarrow{1|1}{04} p$ and $p \xrightarrow{1|0}{02} a$ is sound and q points to a for statement 04. Case 2 could arise if pointee of x is not available even in the calling context. In this case, the barrier GPU $x \xrightarrow{2|0}{03} b$ continues to block $p \xrightarrow{1|0}{02} a$.

Example 20. To see how reaching GPUs analysis with blocking helps, consider the example in Figure 12(b). The set of GPUs reaching the statement 04 is $\overline{\text{RGIn}}_{04} = \{x \xrightarrow{2|0}{02} a, p \xrightarrow{1|0}{03} b\}$. The GPU $x \xrightarrow{2|0}{02} a$ is blocked by the barrier GPU $p \xrightarrow{1|0}{03} b$ and hence $\overline{\text{RGIn}}_{04} = \{p \xrightarrow{1|0}{03} b\}$. Thus, GPU reduction for $w: q \xrightarrow{1|2}{04} x$ (in the context of $\overline{\text{RGIn}}_{04}$) computes Red as $\{w\}$ with the flag *composed* set to **false** because w cannot be reduced further within the GPG of the procedure. However, w is still not a points-to edge and can be simplified further after the GPG is inlined in its callers. Hence we postpone the composition of w with $p: x \xrightarrow{2|0}{02} a$ until p is simplified.

4.5.2 Data Flow Equations for Computing $\overline{\text{RGIn}}$ and $\overline{\text{RGOu}}$. A barrier may not necessarily block all preceding GPUs. We use the type information to identify absence of data dependence between a barrier and the GPUs reaching it. This allows us to minimize blocking by identifying GPUs that need not be blocked. A barrier $b \in \overline{\text{RGGen}}_s$ may block a producer GPU $p \in \overline{\text{RGIn}}_s$ if it writes into a location read by or written by p . Thus, they could share a WaW or a WaR data dependence. Recall that a barrier GPU b is either an indirect GPU or a GPU that follows an indirect GPU (Section 4.5.1). Thus the following GPUs should be blocked:

- If $\overline{\text{RGGen}}_s$ contains an indirect GPU b , then all GPUs reaching δ_s that share a data dependence with b should be blocked regardless of the nature of other GPUs (if any) in $\overline{\text{RGGen}}_s$.
- If $\overline{\text{RGGen}}_s$ does not contain an indirect GPU and is not \emptyset , then all indirect GPUs reaching δ_s that share a data dependence with a GPU in $\overline{\text{RGGen}}_s$ should be blocked.

We define a predicate $\overline{\text{DDep}}(B, I)$ to check the presence of data dependence between the set of GPUs B and I (Definition 6). When the types of $b \in B$ and $p \in I$ match¹³, we assume the possibility of data dependence and b blocks p .

¹³Although C11 standard allows type casting for pointers, there is no guarantee of the expected behaviour if there is alignment mismatch. For example, the runtime behaviour of assigning 'int*' to 'float*' depends on the compiler and the architecture. However, assigning 'void*' to 'int*' does

$\overline{\text{RGIn}}_s := \begin{cases} \{x \xrightarrow{s} x' \mid x \in P, 0 < \ell \leq \kappa\} & s = \text{Start}, \kappa \text{ is the largest } \textit{indlev} \\ \bigcup_{p \in \textit{pred}(s)} \overline{\text{RGOu}}_p & \text{otherwise} \end{cases}$
$\overline{\text{RGOu}}_s := \left(\overline{\text{RGIn}}_s - \left(\overline{\text{RGMK}}_s \cup \text{Blocked}(\overline{\text{RGIn}}_s, \overline{\text{RGGen}}_s) \right) \right) \cup \overline{\text{RGGen}}_s$
$\text{Blocked}(I, G) := \begin{cases} \emptyset & G = \emptyset \\ \{\gamma \mid \gamma \in I, \overline{\text{DDep}}(\text{IndGPUUs}(G), \{\gamma\})\} & \text{IndGPUUs}(G) > 1 \\ \{\gamma \mid \gamma \in \text{IndGPUUs}(I), \overline{\text{DDep}}(G, \{\gamma\})\} & \text{otherwise} \end{cases}$
$\text{IndGPUUs}(X) := \{x \xrightarrow{s} y \mid x \xrightarrow{s} y \in X, i > 1\}$
$\overline{\text{RGGen}}_s := \text{Gen}(\delta_s, \overline{\text{RGIn}}_s)$
$\overline{\text{RGMK}}_s := \text{Kill}(\overline{\text{RGGen}}_s, \overline{\text{RGIn}}_s)$
$\overline{\text{DDep}}(B, I) \Leftrightarrow \text{TDef}(B) \cap (\text{TDef}(I) \cup \text{TRef}(I)) \neq \emptyset$
$\text{TDef}(X) := \{\text{typeof}(x, i) \mid x \xrightarrow{s} y \in X\}$
$\text{TRef}(X) := \{\text{typeof}(x, k) \mid 1 \leq k < i, x \xrightarrow{s} y \in X\} \cup \{\text{typeof}(y, k) \mid 1 \leq k < j, x \xrightarrow{s} y \in X\}$
<p>Note: The definitions of Gen and Kill are same as in Definition 5</p>

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

$\text{TDef}(B)$ 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 a WaW and WaR dependence. The type of the i^{th} pointee of x is given by $\text{typeof}(x, i)$ defined as illustrated below.

Example 21. If the declaration of a pointer x is ‘int **x’, then $\text{typeof}(x, 1)$ is ‘int **’ and $\text{typeof}(x, 2)$ is ‘int *’. Note that $\text{typeof}(x, 0)$ is not a pointer and $\text{typeof}(x, 3)$ is undefined because x cannot be dereferenced thrice.

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

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

not result in misalignment. In our implementation, we trust the types recorded in the GIMPLE IR used by gcc and assume that there is no undefined behaviour of the program.

Example 22. For the procedure in Figure 12(b), $\overline{\text{RGI}}_{n_02} = \emptyset$ and $\overline{\text{RGen}}_{n_02}$ is $\{x \xrightarrow{2|0} a\}$. Although $\overline{\text{RGen}}_{n_02}$ contains an indirect GPU, since no GPUs reach 02 (because it is the first statement), $\overline{\text{RGO}}_{n_02}$ is $\{x \xrightarrow{2|0} a\}$ indicating that no GPUs are blocked.

For statement 03, $\overline{\text{RGI}}_{n_03} = \{x \xrightarrow{2|0} a\}$ and $\overline{\text{RGen}}_{n_03} = \{p \xrightarrow{1|0} b\}$. $\overline{\text{RGen}}_{n_03}$ is non-empty and does not contain an indirect GPU and thus $\overline{\text{RGO}}_{n_03} = \{p \xrightarrow{1|0} b\}$ according to the third case in the Blocked equation in Definition 6 indicating that the GPU $x \xrightarrow{2|0} a$ is blocked and should not be used for composition by the later GPUs. The indirect GPU in $\overline{\text{RGI}}_{n_03}$ is excluded from $\overline{\text{RGO}}_{n_03}$. Note that the indirect GPU $x \xrightarrow{2|0} a$ is blocked by the GPU $p \xrightarrow{1|0} b$ because $\text{typeof}(x, 2)$ matches with $\text{typeof}(p, 1)$ indicating a possibility of WaW dependence.

For statement 04, $\overline{\text{RGI}}_{n_04} = \{p \xrightarrow{1|0} b\}$ and $\overline{\text{RGen}}_{n_04}$ is $\{q \xrightarrow{1|2} x\}$. For this statement, the composition $(q \xrightarrow{1|2} x \circ^{\text{ts}} x \xrightarrow{2|0} a)$ is postponed because the GPU $x \xrightarrow{2|0} a$ is blocked. In this case, $\overline{\text{RGen}}_{n_04}$ does not contain an indirect GPU and $\overline{\text{RGO}}_{n_04} = \{p \xrightarrow{1|0} b, q \xrightarrow{1|2} x\}$.

Similarly in Figure 12(a), the GPU $p \xrightarrow{1|0} a$ is blocked by the barrier GPU $x \xrightarrow{2|0} b$ because $\text{typeof}(p, 1)$ matches with $\text{typeof}(x, 2)$. Hence, the composition $(q \xrightarrow{1|1} p \circ^{\text{ts}} p \xrightarrow{1|0} a)$ is postponed.

In the GPG of procedure g (of our motivating example) shown in Figure 6, the GPUs $r \xrightarrow{1|0} a$ and $q \xrightarrow{1|0} b$ are not blocked by the GPU $q \xrightarrow{2|0} m$ because they have different types. However, the GPU $e \xrightarrow{1|2} p$ blocks the indirect GPU $q \xrightarrow{2|0} m$ because there is a possible WaW data dependence (e and q could be aliased in the callers of g).

5 REDUNDANCY ELIMINATION OPTIMIZATIONS

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

- *Dead GPU and empty GPB elimination.*
- *Coalescing of GPBs.*

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

5.1 Dead GPU and Empty GPB Elimination

We perform dead GPU elimination to remove a redundant GPU $\gamma \in \delta_s$ that is killed along every control flow path from s to the End GPB of the procedure. However, the following two kinds of GPUs should not be removed even if they are killed in reaching GPUs analyses: (a) GPUs that are blocked, or (b) GPUs that are producer GPUs for *undesirable* compositions that have been postponed (Section 4.2.2). For the former, we check that a GPU considered for dead GPU elimination does not belong to $\overline{\text{RGO}}_{\text{End}}$ (the result of reaching GPUs analysis without blocking); for the latter we check that the GPU is not a producer GPU for a postponed composition. We record such GPUs in the set `Queued`

computed for every GPG. It is computed during GPU reduction.¹⁴ Thus, we perform dead GPU elimination and remove a GPU $\gamma \in \delta_s$ if $\gamma \notin (\text{RGO}_{\text{End}} \cup \text{Queued})$.

Example 23. In procedure g of Figure 6, pointer q is defined in statement 03 but is redefined in statement 05 and hence the GPU $q \xrightarrow{1|0}{03} b$ is killed and does not reach the End GPB. Since no composition with the GPU $q \xrightarrow{1|0}{03} b$ is postponed, it does not belong to set Queued either. Hence the GPU $q \xrightarrow{1|0}{03} b$ is eliminated from the GPB δ_{03} as an instance of dead GPU elimination.

Similarly, the GPUs $q \xrightarrow{1|0}{07} d$ (in δ_{07}) and $e \xrightarrow{1|1}{04} c$ (in δ_{14}) in the GPG of procedure f (Figure 7) are eliminated from their corresponding GPBs.

Example 24. For the procedure in Figure 12(a), the GPU $p \xrightarrow{1|0}{02} a$ is not killed but is blocked by the barrier $x \xrightarrow{2|0}{03} b$; hence it is present in RGO_{05} but not in $\overline{\text{RGO}}_{05}$ (05 is the End GPB). This GPU may be required when the barrier $x \xrightarrow{2|0}{03} b$ is reduced after call inlining (and ceases to block $p \xrightarrow{1|0}{02} a$). Thus, it is not removed by dead GPU elimination.

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

Example 25. In the GPG of procedure g of Figure 6, the GPB δ_{03} becomes empty after dead GPU elimination. Hence, δ_{03} can be removed by connecting its predecessors to successors. This transforms the back edge $\delta_{03} \rightarrow \delta_{01}$ to $\delta_{02} \rightarrow \delta_{01}$. Similarly, the GPB δ_{07} is deleted from the GPG of procedure f in Figure 7.

5.2 Minimizing the Control Flow by Coalescing GPBs

Strength reduction eliminates data dependence between GPUs rendering the control flow redundant. Eliminating redundant control flow is important to make a GPG as compact as possible—in the absence of control flow minimization, the size of the GPG of a procedure tends to increase exponentially because of transitive inlinings of calls in the procedure. This effect is aggravated by the fact that many procedures are called multiple times in the same procedure. Besides, recursion causes multiple inlinings of the GPGs of procedures in the cycle of recursion (Section 6.2).

5.2.1 Coalescing GPBs by Partitioning a GPG. We eliminate redundant control flow by coalescing adjacent GPBs. This amounts to partitioning the set of GPBs in a GPG such that each part contains the GPBs whose GPUs do not have a data dependence between them and hence can be seen essentially as executed non-deterministically in any order in accordance with abstract semantics of a GPB as a *may* property (Section 3.1).

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

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

¹⁴The revised definition is available at <https://www.cse.iitb.ac.in/~uday/soft-copies/gpg-pta-paper-appendix.pdf>.

- The possibility of data dependence between GPBs δ_m and δ_n matters only if there is control flow between them. Otherwise, they are executed in different execution instances of the program and there is no data dependence between them even if the variables or abstract locations accessed by them are same. Hence the successors of a GPB can be coalesced with each other in the same part provided there is no control flow between them.
- As a design choice, a successor (predecessor) of a GPB is included in the part containing the GPB *iff* all successors (predecessors) of the GPB are included in the part: Consider GPBs δ_l , δ_m and δ_n such that $\text{succ}(l) = \{m, n\}$ and neither m is a successor of n nor vice-versa. Let $\delta_l \in \pi_i$. Since there is no control flow between δ_m and δ_n , including only one of them in π_i will create a spurious control flow between them. This ordering could introduce a spurious data dependence between their GPUs which may cause imprecision (through a RaW dependence that may create spurious GPUs).
- Coalescing may eliminate a definition-free path for the source of a GPU. This may convert the GPU from *may-def* (i.e., source is defined along some path) to *must-def* (i.e., source is defined along all paths) in the GPG. Consider GPBs δ_l , δ_m , δ_n , and δ_o such that $\text{succ}(l) = \{m, n\}$ and $\text{pred}(o) = \{m, n\}$. Let $\pi_i = \{\delta_l, \delta_m, \delta_n\}$ and $\pi_j = \{\delta_o\}$. The source of some GPU $\gamma_m \in \delta_m$ may have a definition-free path $\delta_l \rightarrow \delta_n \rightarrow \delta_o$. After coalescing, this definition-free path ceases to exist because of the control flow edge $\pi_i \rightarrow \pi_j$. This may lead to strong updates instead of weak updates thereby leading to unsoundness. Hence, we add a separate definition-free path for such GPUs.

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

- Start GPB and End GPB form singleton parts and no other GPB is included in these parts. This is required for modelling definition-free paths from Start to End to distinguish between strong and weak updates by a callee GPG in a caller GPG.
- The process of identifying the partition begins with Start GPB. Thus Start forms $\pi_1 \in \Pi$. As a consequence, a part $\pi_i \in \Pi$ grows only in the “forward” direction including only successor GPBs. It never grows in the “backward” direction by considering predecessors.
- Consider δ_n and δ_s , $s \in \text{succ}(n)$ such that $\delta_n \rightarrow \delta_s$ is a back edge. Then δ_n and δ_s belong to the same partition π_i *iff* all GPBs in the loop formed by the back edge (i.e. all GPBs that appear on all paths from δ_s to δ_n) belong to π_i .

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

5.2.2 The Role of Data Dependence in Blocking and Coalescing. The main differences between the use of data dependence for blocking (Definition 6 in Section 4.5) and for coalescing are:

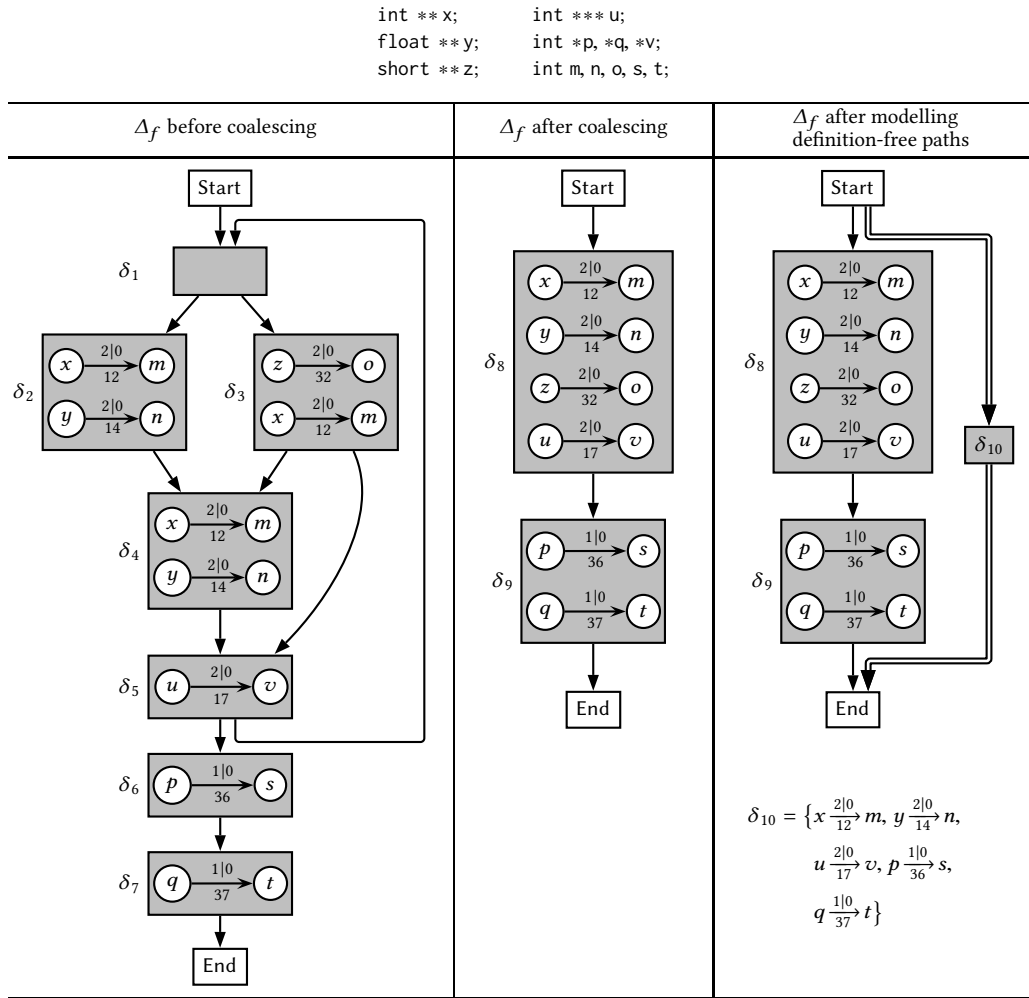


Fig. 13. An example demonstrating the effect of coalescing. The loop formed by the back edge $\delta_5 \rightarrow \delta_1$ reduces to a self loop over GPB δ_8 after coalescing. Since self loops are redundant, they are eliminated. Control flow edges with double lines represent definition-free paths.

- *The motivation behind using data dependence.* When analyzing for blocking, we identify the possibility of a barrier updating a location accessed by a previous GPU. In coalescing we wish to establish that no control flow needs to be maintained between two GPUs.
- *The way data dependence is used.* For blocking, we use the *possible presence* of data dependence between a barrier and reaching GPUs to block some of the reaching GPUs. For coalescing, we use the *guaranteed absence* of data dependence between the GPUs of a GPB and those reaching it from within a part to coalesce the GPB with the part.
- *Relevant data dependences.* Coalescing removes control flow between two GPUs enabling their non-deterministic execution with respect to each other which is oblivious to any data dependence between the GPUs. Hence, a

$\text{CIn}_n := \begin{cases} \text{false} & n \text{ is Start} \\ \bigwedge_{p \in \text{pred}(n)} \text{coalesce}(p, n) & \text{otherwise} \end{cases}$
$\text{COut}_n := \begin{cases} \text{false} & n \text{ is End} \\ \bigwedge_{s \in \text{succ}(n)} \text{CIn}_s & \text{otherwise} \end{cases}$
$\text{coalesce}(p, n) \Leftrightarrow \text{COut}_p \wedge (\text{GOut}_p = \emptyset \vee \text{gpuFlow}(p, n) \neq \emptyset)$
$\text{GIn}_n := \begin{cases} \emptyset & n \text{ is Start} \\ \bigcup_{p \in \text{pred}(n)} \text{gpuFlow}(p, n) & \text{otherwise} \end{cases}$
$\text{GOut}_n := \begin{cases} \text{GIn}_n \cup \delta_n & \text{CIn}_n = \text{true} \\ \delta_n & \text{otherwise} \end{cases}$
$\text{gpuFlow}(p, n) := \begin{cases} \emptyset & \neg \text{CIn}_n \wedge \text{DDep}(\text{GOut}_p, \delta_n) \\ \text{GOut}_p & \text{otherwise} \end{cases}$
$\text{DDep}(X, Y) \Leftrightarrow (\text{deref}(X) \vee \text{deref}(Y)) \wedge (\text{TDef}(Y) \cup \text{TRef}(Y)) \cap \text{TDef}(X - Y) \neq \emptyset$
$\text{deref}(X) \Leftrightarrow \exists x \xrightarrow{ij} y \in X \text{ s.t. } (i > 1) \vee (j > 1)$

Definition 7. Data flow equations for Coalescing Analysis.

RaW and WaW dependences need to be preserved by prohibiting coalescing. However, a WaR dependence is not affected by coalescing. On the other hand, blocking by a barrier does not involve RaW dependence (see the motivation above) and needs to handle only WaW and WaR dependences.

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

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

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

Compare the predicates $\overline{\text{DDep}}$ (Definition 6) for blocking and DDep (Definition 7) for coalescing to see the above differences. For establishing the absence of dependence, we match the types of $\gamma_1 \in X$ with the types of $\gamma_2 \in Y$. This is meaningful only when $\gamma_1 \neq \gamma_2$. The term $X - Y$ in the definition of predicate DDep ensures this.

5.2.3 *Partitioning Analysis.* We define two interdependent data flow analyses that inductively

- construct part π_i using data flow variables $\text{CIn}_n/\text{COut}_n$, and
- compute the GPUs accumulated in $G(\pi_i, n)$ in data flow variables $\text{GIn}_n/\text{GOut}_n$.

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

Name for GPUs. Statement ids do not matter											
γ_1	$x \xrightarrow{2 0}_{12} m$	γ_2	$y \xrightarrow{2 0}_{14} n$	γ_3	$z \xrightarrow{2 0}_{32} o$	γ_4	$u \xrightarrow{2 0}_{17} v$	γ_5	$p \xrightarrow{1 0}_{36} s$	γ_6	$q \xrightarrow{1 0}_{37} t$
GPB n	TDef (n)	TRef (n)	GIn $_n$	GOut $_n$	CIn $_n$	COut $_n$					
δ_1	\emptyset	\emptyset	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	F	T					
δ_2	$\{\text{int}^*, \text{float}^*\}$	\emptyset	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	T	T					
δ_3	$\{\text{short}^*, \text{int}^*\}$	\emptyset	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	T	T					
δ_4	$\{\text{int}^*, \text{float}^*\}$	\emptyset	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	T	T					
δ_5	$\{\text{int}^{**}\}$	\emptyset	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	$\{\gamma_1, \gamma_2, \gamma_3, \gamma_4\}$	T	F					
δ_6	$\{\text{int}^*\}$	\emptyset	\emptyset	$\{\gamma_5\}$	F	T					
δ_7	$\{\text{int}^*\}$	\emptyset	$\{\gamma_5\}$	$\{\gamma_5, \gamma_6\}$	T	F					

Fig. 14. The data flow information computed by coalescing analysis for example in Figure 13. The CIn and COut values indicate that GPBs $\delta_1, \delta_2, \delta_3, \delta_4, \delta_5$ can be coalesced. Similarly, GPBs δ_6 and δ_7 can be coalesced. GPBs δ_5 and δ_6 must remain in different coalesced groups.

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

The data flow equations to compute CIn $_n$ /COut $_n$ are given in Definition 7. The initialization is *true* for all GPBs. Predicate *coalesce*(p, n) uses *gpuFlow*(p, n) to check if GPUs $G(\pi_i, p)$ are allowed to flow from p to n —if yes, then p and n belong to the same part. If GOut $_p$ is \emptyset , they belong to the same part regardless of *gpuFlow*(p, n). The presence of COut $_p$ in the equation of *coalesce* (Definition 7) ensures that GPB δ_p is considered for coalescing with δ_n only if δ_p has not been found to be a “boundary” in coalescing because it cannot coalesce with some successor.

Another striking difference between the equations for CIn/COut in Definition 7 and the usual data flow equations is that the data flow variables CIn $_n$ and COut $_n$ for GPB n are independent of each other—CIn $_n$ depends only on the COut of its predecessors and COut $_n$ depends only on the CIn of its successors. Intuitively, this form of data flow equations attempts to *melt* the boundaries of GPB n to explore fusing it with its successors and predecessors.

- When CIn $_n$ is true, it melts the boundary at the top of the GPB and glues it with all its predecessors that are already in the part. Thus, a part grows in a forward direction.
- When COut $_n$ is true, it melts the boundary at the bottom of the GPB and includes all its successors in the part thereby growing a part in the forward direction.

The incremental expansion of a part in a forward direction influences the flow of GPUs accumulated in a part leading to a forward data flow analysis for computing $G(\pi_i, n)$ using data flow variables GIn $_n$ /GOut $_n$. The data flow equations to compute them are given in Definition 7. Function *gpuFlow*(p, n) in the equation for GIn computes the set of GPUs $G(\pi_i, p)$ that flow from p to n . It establishes the absence of data dependences using predicate DDep defined in Section (5.2.2). If no data dependence exists, the GPUs accumulated in GOut $_p$ are propagated to n . The presence of \neg CIn $_p$ in equation for *gpuFlow* ensures that GPUs in GOut $_p$ are propagated to δ_n only if δ_n has not been found to be a “boundary” in coalescing because it cannot coalesce with some predecessor.

Example 26. Figure 14 gives the data flow information for the example of Figure 13. GPBs δ_1 and δ_2 can be coalesced because COut_1 is *true* and GOut_1 is \emptyset . Thus, $\text{DDep}(1, 2)$ returns *false* indicating that types do not match and hence there is no possibility of a data dependence between the GPUs of δ_1 and δ_2 . Similarly, GPBs δ_1 and δ_3 can be coalesced. Thus COut_1 , CIn_2 , and CIn_3 are *true*. We check the data dependence between the GPUs of GPBs δ_2 and δ_4 using the type information. However, $\text{DDep}(2, 4)$ returns *false* because the term $(\text{GOut}_2 - \delta_4)$ is \emptyset . Thus, GPBs δ_2 and δ_4 belong to the same part and can be coalesced. For GPBs δ_3 and δ_4 , the possibility of data dependence is resolved based on the type information. The term $(\text{GOut}_3 - \delta_4)$ returns $z \xrightarrow{\frac{2|0}{32}} o$ whose $\text{typeof}(z, 1)$ 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 $\text{DDep}(\delta_4, \delta_5)$ returns *false* indicating that there is no type matching and hence no possibility of data dependence, thereby allowing the coalescing of the two GPBs. The $\text{DDep}(\delta_5, \delta_6)$ returns *true* (type of source of the GPU $x \xrightarrow{\frac{2|0}{12}} m \in \text{GOut}_5$ matches the source of the GPU $p \xrightarrow{\frac{1|0}{36}} s \in \delta_6$) indicating a possibility of data dependence in the caller through aliasing and hence the two GPBs cannot be coalesced. Thus, the first part in the partition contains only GPBs $\delta_1, \delta_2, \delta_3, \delta_4$, and δ_5 . GPB δ_6 now marks the first GPB of the new part. GPBs δ_6 and δ_7 can be coalesced as there is no data dependence between their GPUs. The loop $\delta_5 \rightarrow \delta_1$ before coalescing now reduces to self loop over GPB δ_8 after coalescing. The self loop is redundant and hence eliminated. GPBs δ_5 and δ_1 can be coalesced because all the GPBs of the loop belong to the same part.

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

Example 27. In the example of Figure 6, GPBs δ_1 and δ_2 can be coalesced because $\text{DDep}(\delta_1, \delta_2)$ returns *false* indicating that there is no type matching and hence no possible data dependence between their GPUs. Thus, COut_1 and CIn_2 are set to *true*. The loop formed by the back edge $\delta_2 \rightarrow \delta_1$ reduces to a self loop over GPB δ_{11} after coalescing. The self loop is redundant and hence it is eliminated. For GPBs δ_2 and δ_4 , $\text{DDep}(\delta_2, \delta_4)$ returns *true* because $\text{typeof}(q, 2)$ (for the GPU $q \xrightarrow{\frac{2|0}{02}} m$ in δ_{02}) matches $\text{typeof}(p, 2)$ (for the GPU $e \xrightarrow{\frac{1|2}{04}} p$ in δ_{04}) which is `int *`. This indicates the possibility of a data dependence between the GPUs of GPBs δ_2 and δ_4 (q and p could be aliased in the caller) and hence these GPBs cannot be coalesced. Thus, COut_2 and CIn_4 are set to *false*. For GPBs δ_4 and δ_5 , $\text{DDep}(\delta_4, \delta_5)$ returns *false* because there is no possible data dependence. Hence COut_4 and CIn_5 are set to *true* and the two GPBs can be coalesced.

Recall that our coalescing heuristics requires us to prohibit

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

The data flow equations for Coalescing (CIn/COut in Definition 7) do not have any provision of these requirements; they are enforced separately during the actual transformation.

5.2.4 Preserving Definition-Free Paths. Consider a GPU γ that reaches the exit of a GPG along some path but not all. It means that there is some path in the GPG along which the source of γ is not defined (i.e., the source of γ is *may*-defined in the GPG). According to our heuristics of coalescing, a GPB is coalesced either with all its successors

or with none. Hence, after coalescing with all successors, a definition-free path may get subsumed and γ may reach the exit of a GPG along all paths indicating that the source of γ is now *must*-defined. This would lead to a strong update instead of a weak update thereby introducing unsoundness. Hence, we need to add an explicit definition-free path for such GPUs. The GPUs with definition-free paths are identified by the corresponding boundary definitions. A definition-free path for the source of GPU $x \xrightarrow{i|j}{s} y$ exists in a GPG only if the boundary definition $x \xrightarrow{i|i}{00} x'$ reaches the exit of the GPG.

Example 28. In the example of Figure 13, the definition-free path is shown by edges with double lines in the GPG obtained after coalescing. The GPU $z \xrightarrow{2|0}{32} o$ does not reach the exit along the path $\delta_1 \rightarrow \delta_2 \rightarrow \delta_4 \rightarrow \delta_5 \rightarrow \delta_6 \rightarrow \delta_7$ which forms the definition-free path. We add a definition-free path between Start and End GPBs of a GPG with a GPB that contains all GPUs that do not have any definition-free path. Thus, we have a GPB δ_{10} which contains all GPUs except $z \xrightarrow{2|0}{32} o$.

Example 29. In Figures 6 and 7, definition-free paths are shown by edges with double lines in the GPGs of procedures f and g obtained after coalescing. For procedure g , the GPUs $b \xrightarrow{1|0}{02} m$ and $q \xrightarrow{2|0}{02} m$ undergo a weak update and hence do not kill their corresponding boundary definitions. This indicates that the source of these GPUs are *may*-defined and hence a definition-free path is required for these GPUs. Thus, we add a definition-free path between Start and End GPBs of Δ_g with GPB δ_{16} which contains the set of GPUs $\{r \xrightarrow{1|0}{01} a, e \xrightarrow{1|2}{04} p, q \xrightarrow{1|0}{05} e\}$.

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

6 CALL INLINING

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

The GPG of a procedure can be constructed completely only when (a) all callees are known, and (b) their GPGs have been constructed completely. The first condition is violated by a call through function pointer and the second condition is violated by a recursive call. We classify procedure calls into the following three categories and explain the handling of the first two in this section. The third category is handled in Section 9 because it requires the concepts introduced in Section 7.

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

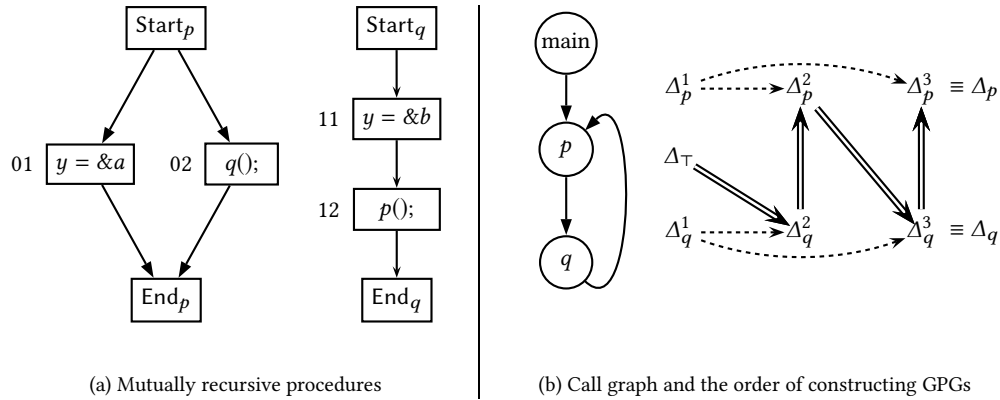


Fig. 15. Constructing GPGs for recursive procedures by successive refinements.

6.1 Callee is Known and the Call is Non-Recursive

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

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

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

- GPUs that correspond to the actual-to-formal-parameter mapping.
- A GPU that maps the return variable of the callee to the receiver variable of the call in the caller (or zero GPUs for a void function).

Some GPUs in the GPG of the callee may have upwards-exposed versions of variables. For example, if the callee reads a global variable x defined in the caller, it would have a GPU referring to the initial value x' (see Section 4.3.2). Hence when a GPG is inlined in a caller procedure, we substitute the callee's upwards-exposed variable x' occurring in a callee's GPU by the original variable x when the GPU is included in the caller's GPG. Note that x may be a global variable or a formal parameter.

Inlining of procedure calls with the callee's optimized GPG allows reaching GPU analyses to remain intraprocedural analyses. However, recursive and indirect calls need to be handled specially. These cases are discussed in Section 6.2 immediately below and Section 9.

6.2 Callee is Known and the Call is Recursive

Consider Figure 15 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

<pre> 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: Δ_p^{i+1} // Optimized $(i + 1)^{th}$ GPG for procedure p 01 Refine_GPG ($p, \Delta_p^1, \Delta_p^i$) 02 { 03 $R_{prev} = \text{RGOuT}_{\text{End}}(\Delta_p^i)$ 04 $\overline{R}_{prev} = \overline{\text{RGOuT}}_{\text{End}}(\Delta_p^i)$ 05 Compute Δ_p^{i+1} by inlining recursive calls in Δ_p^1 with their latest GPGs 06 Perform both variants of reaching GPUs analysis over Δ_p^{i+1} 07 $R_{curr} = \text{RGOuT}_{\text{End}}(\Delta_p^{i+1})$ 08 $\overline{R}_{curr} = \overline{\text{RGOuT}}_{\text{End}}(\Delta_p^{i+1})$ 09 if $((R_{curr} \neq R_{prev}) \vee (\overline{R}_{curr} \neq \overline{R}_{prev}))$ 10 Push callers of p on the worklist 11 Perform strength reduction and redundancy elimination optimizations over Δ_p^{i+1} 12 return Δ_p^{i+1} 13 }</pre>

Definition 8. Computing GPGs for Recursive Procedures by Successive Refinement

incomplete. We handle this mutual dependency by successive refinement of incomplete GPGs of p and q through a fixed-point computation.

A set of recursive procedures is represented by a strongly connected component in a call graph which is formed by a collection of back edges that represent recursive calls. Since we traverse a call graph bottom up, the construction of GPGs for a set of recursive procedures begins with the procedures that are the sources of back edges. The GPGs of some callees of these procedures (i.e. the callees that are targets of back edges in the call graph) have not been constructed yet. We handle such situations by using a special GPG Δ_{\top} that represents the effect of a call when the callee's GPG is not available. The GPG Δ_{\top} is the \top element of the lattice of all possible procedure summaries. It kills all GPUs and generates none (thereby, when applied, computes the \top value— \emptyset —of the lattice for *may* points-to analysis) [15]. Semantically, Δ_{\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 Δ_{\top} as their initial GPGs, until no further inlining is required. This is achieved as follows: Since data flow analysis over incomplete GPGs under-approximates the effect of some calls through Δ_{\top} , the data flow values so computed need to be refined further. This is achieved by inlining the calls by including incomplete GPGs of the callees to compute a new GPG over which the data flow analysis is repeated. Let Δ_p^1 denote the GPG of procedure p in which all the calls to the procedures that are not part of the strongly connected component are inlined by their respective optimized GPGs. Note that the GPGs of these procedures have already been constructed because of the

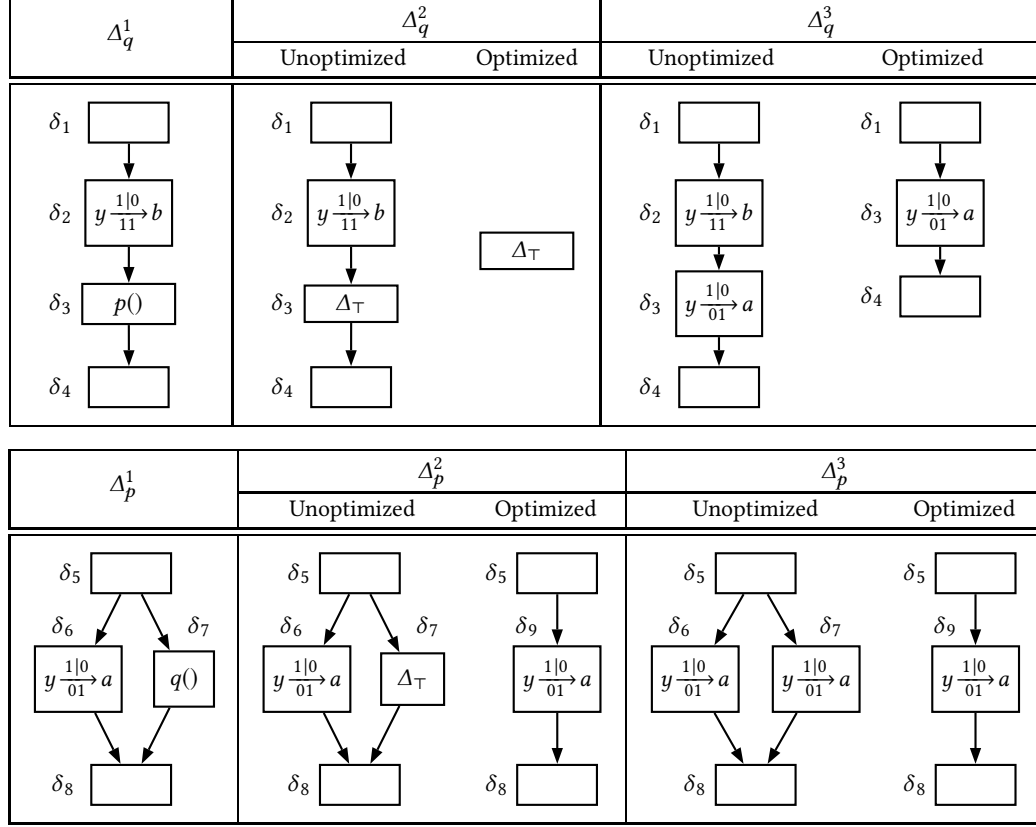


Fig. 16. Series of GPGs of procedures p and q of Figure 15. They are computed in the order shown in Figure 15(b). See Example 30 for explanation.

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 Δ_\top when no GPG of the callee has been constructed, or
- by an incomplete GPG of a callee in which some calls are under-approximated using Δ_\top .

Thus we compute a series of GPGs Δ_p^i , $i > 1$ for every procedure p in a strongly connected component until the termination of fixed-point computation. For this purpose, we initialize a worklist with all procedures in a strongly connected component. This worklist is ordered by the postorder relation between the procedures in the call graph. A procedure is added to the worklist based on the following criterion; the process terminates when the worklist becomes empty. Once Δ_p^i is constructed, we decide to construct Δ_q^j for a caller q of p if the data flow values of the End GPB of Δ_p^i differ from those of the End GPB of Δ_p^{i-1} . This is because the overall effect of a procedure on its callers is reflected by the values reaching its End GPB (because of forward flow of information in points-to analysis). If the data values of the End GPBs of Δ_p^{i-1} and Δ_p^i are same, then they would have identical effect on their callers. Thus, the GPGs are semantically identical as procedure summaries even if they differ structurally. This step is described in Definition 8.

The convergence of this fixed-point computation differs subtly from the usual fixed-point computation in the following manner: in each step of computation, the GPGs continue to change. And yet, we stop the fixed-point computation when the *data flow* values of the End GPB converge across the changing GPGs, not when the resultant GPGs converge.

Example 30. In the example of Figure 15, the sole strongly connected component contains procedures p and q . Since procedure q is the source of the back edge in the call graph, the GPG of procedure q is constructed first. There are no calls in procedure q to procedures outside the strongly connected component. Thus, Δ_q^1 contains a single call to procedure p whose GPG is not constructed yet and hence the construction of Δ_q^2 requires inlining of Δ_\top . Since Δ_\top represents a procedure call which never returns, the GPB End_q becomes unreachable from the rest of the GPBs in Δ_q^2 . The optimized Δ_q^2 is Δ_\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 Δ_\top , is inlined during construction of Δ_p^2 . The optimized version of Δ_p^2 is shown in Figure 16. Then, Δ_p^2 is used to construct Δ_q^3 . Reaching GPUs analyses with and without blocking are performed on Δ_q^2 and Δ_q^3 . The data flow values for Δ_q^2 are $Rprev = \bar{Rprev} = \emptyset$ whereas the data flow values for Δ_q^3 are $Rcurr = \bar{Rcurr} = \{y \xrightarrow{1|0} a\}$. Since the data flow values have changed, caller of q i.e., p is pushed on the worklist and Δ_p^3 is constructed by inlining Δ_q^3 . The data flow values computed for Δ_p^2 and Δ_p^3 are identical $Rprev = \bar{Rprev} = Rcurr = \bar{Rcurr} = \{y \xrightarrow{1|0} a\}$ and hence caller of p i.e., procedure q is not added to the worklist. The worklist becomes empty and hence the process terminates. Note that the data flow values of Δ_q^2 and Δ_q^3 differ and yet we do not construct the GPG Δ_q^4 . This is because Δ_q^4 constructed by inlining Δ_p^3 will have the same effect as that of Δ_q^3 constructed by inlining Δ_p^2 since the impact of Δ_p^2 and Δ_p^3 is identical.

The process of fixed-point computation is guaranteed to terminate because of the finiteness of the set of GPUs $Rprev$, \bar{Rprev} , $Rcurr$, \bar{Rcurr} : For two variables x and y , the number of GPUs $x \xrightarrow{i|j} y$ depends on the number of possible *indlevs* ($i|j$) and the number of statements. Since the number of statements is finite, we need to examine the number of *indlevs*. For pointers to scalars, the number of *indlevs* between any two variables is bounded because of type restrictions. For pointers to structures (Section 8), *indlevs* are replaced by indirection lists (*indlists*). Sections 8.2 and 8.3 summarize *indlists* restricting them to a finite number. Hence the number of GPUs is also finite.

7 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 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 u with *indlev* 1 and the target is the pointee which is being read. Thus a condition ‘if ($x == *y$)’ where both x and y are pointers, is modelled as a GPB $\left\{ u \xrightarrow{1|1} x, u \xrightarrow{1|2} y \right\}$ whereas an integer assignment ‘ $*x = 5;$ ’ is modelled as a GPB $\left\{ u \xrightarrow{1|2} x \right\}$.

Example 31. Consider the code snippet on the right. There is a non-pointer assignment in

which the pointee of x (which is the location a) is being defined. A client analysis would like to know the pointees of x for statement 02. We model this use of pointee of x as a GPU $u \frac{1|2}{02} \rightarrow x$.

01	$x = \&a;$
02	$*x = 5;$

This GPU can be composed with $x \frac{1|0}{01} \rightarrow a$ to get a reduced GPU $u \frac{1|1}{02} \rightarrow a$ 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.

With the provision of a GPU for a use statement, the process of computing points-to information can be seen as a two step process:

- creating def-use or use-def chains for pointers to view producer GPUs as definitions of pointers and consumer GPUs as the use of pointers, and
- performing strength reduction of the consumer GPUs using the information from the producer GPUs to reduce the *indlevs* of the consumer GPUs.

Since our first phase does this for constructing procedure summaries, it is sufficient to compute points-to information in the first phase.

This process is easy to visualize if the definitions and uses are in the same procedure. Consider a producer GPU p and a consumer GPU c that are not in the same procedure. We can facilitate strength reduction involving them by

- (a) propagating p to the procedure containing c ,
- (b) propagating c to the procedure containing p , or
- (c) propagating both p and c to a common procedure.

The propagation of information in cases (a) and (b) is similar to that in a top-down analysis; case (a) corresponds to a forward analysis and case (b) corresponds to a backward analysis. However, case (c) is only possible in bottom-up analysis.

A typical second phase of a bottom-up approach involves propagation of information similar to cases (a) and (b). This is illustrated in Example 32. We use propagation similar to case (c) which is subsumed in the first phase of a bottom-up approach rendering the second phase redundant. It is illustrated in Example 33.

Example 32. Consider procedures f, g, h and s defined in Figure 17. We can facilitate strength reduction in the following ways for cases (a) and (b):

- *Propagating p to the procedure containing c .* A top-down forward analysis would propagate the GPU $x \frac{1|0}{1} \rightarrow a$ from procedure f to procedure g .
- *Propagating c to the procedure containing p .* A top-down backward analysis in the spirit of liveness could propagate the GPU $y \frac{1|1}{4} \rightarrow x$ from procedure g to procedure f .

We handle case (c) by interleaved call inlining and strength reduction. Call inlining enhances the opportunities for strength reduction by providing more information from the callers. The interleaving of strength reduction and call inlining gradually converts a GPU $x \frac{i|j}{s} \rightarrow y$ to a set of points-to edges $\{a \frac{1|0}{s} \rightarrow b \mid a \text{ is } i^{\text{th}} \text{ pointee of } x, b \text{ is } j^{\text{th}} \text{ pointee of } y\}$. This is achieved by propagating the *use* of a pointer¹⁵ and its *definitions* to a common context. This may require propagating:

¹⁵This use could be in a pointer assignment or a use statement.

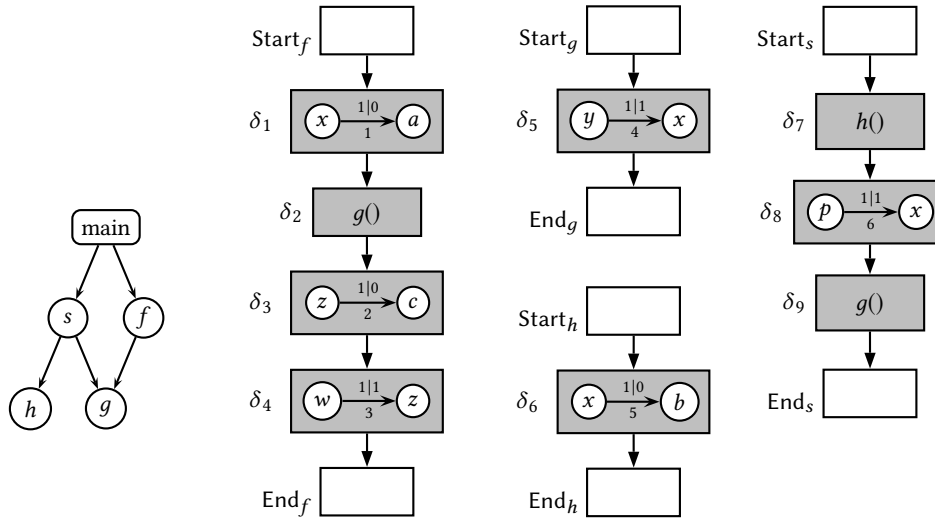


Fig. 17. Computing points-to information using GPGs. The first column gives the call graph while the other columns give GPGs before call inlining. The GPG of procedure *main* has been omitted.

- (1) a consumer GPU (i.e. a use of a pointer variable) to a caller,
- (2) a producer GPU (i.e. a definition of a pointer variable) to a caller,
- (3) both consumer and producer GPUs involving a pointer variable to a caller, and
- (4) neither (if they are same in the procedure).

Since statement numbers are unique across all procedures and are not renamed on inlining, the points-to edges computed across different contexts for a given statement represent the flow- and context-sensitive points-to information for the statement.

Example 33. The four variants of hoisting p and c to a common procedure in the first phase of a bottom-up method are illustrated below. Effectively, they make the second phase redundant.

- (c.1) When Δ_g is inlined in f , $c: y \xrightarrow{1|1}_4 x$ from procedure g is hoisted to procedure f that contains GPU $p: x \xrightarrow{1|0}_1 a$ thereby propagating the use of pointer x in procedure g to caller f . Strength reduction reduces c to $y \xrightarrow{1|0}_4 a$.
- (c.2) When Δ_h is inlined in s , $p: x \xrightarrow{1|0}_5 b$ from procedure h is hoisted to procedure s that contains $c: p \xrightarrow{1|1}_6 x$ thereby propagating the definition of x in procedure h to the caller s . Strength reduction reduces c to $p \xrightarrow{1|0}_6 b$.
- (c.3) When Δ_g and Δ_h are inlined in s , $c: y \xrightarrow{1|1}_4 x$ in procedure g and $p: x \xrightarrow{1|0}_5 b$ in procedure h are both hoisted to procedure s thereby propagating both the use and definition of x in procedure s . Strength reduction reduces c to $y \xrightarrow{1|0}_4 b$.
- (c.4) Both the definition and use of pointer z are available in procedure f with $c: w \xrightarrow{1|1}_3 z$ and $p: z \xrightarrow{1|0}_2 c$. Strength reduction reduces c to $w \xrightarrow{1|0}_3 c$.

Thus, y points-to a along the call from procedure f and it points-to b along the call from procedure s . Thus, the points-to information $\{y \xrightarrow{1|0} a, y \xrightarrow{1|0} b\}$ represents flow- and context-sensitive information for statement 4.

8 HANDLING HEAP FOR POINTS-TO ANALYSIS USING GPGS

So far we have created the concept of GPGs for pointers to scalars allocated on the stack or in the static area. This section extends the concepts to data *structures* containing named fields created using C style **struct** or **union** and possibly allocated on the heap (as well as on the stack or in static memory). For clarity, in this section, we show only the set of GPUs reaching a given statement and do not show the complete GPG of a procedure.

Extending GPGs to handle structures and heap-allocated data requires the following changes:

- The concept of *indlevs* is generalized to indirection lists (*indlists*) to handle structures and heap accesses field sensitively.
- Heap locations are abstracted using allocation sites. In this abstraction, all locations allocated at a particular allocation site are treated alike. This approximation allows us to handle the unbounded nature of heap as if it were bounded [12]. Hence only weak updates can be performed on heap locations.¹⁶
- When the GPG of a procedure is being constructed, the allocation sites may appear in a caller procedure and hence may not be known. We deal with this by an additional summarization based on k -limiting to bound the accesses in a loop. Both these summarization techniques are required to create a decidable version of our method of constructing procedure summaries in the form of GPGs. The resulting points-to analysis is a precise flow-sensitive, field-sensitive, and context-sensitive analysis (relative to these two summarization techniques).¹⁷
- Introduction of *indlists* and k -limiting summarization requires extending the concept of GPU composition to handle them.
- The allocation-site-based abstraction and k -limiting summarization may create cycles in GPUs; a simple extension to GPU reduction handles them naturally.

The optimizations performed on GPGs and the required analyses remain the same. Hence, the discussion in these sections is driven mainly by examples that illustrate how the theory developed earlier is adapted to handle structures (typically, but not necessarily, heap-allocated).

8.1 Extending GPU Composition to Indirection Lists

The *indlev* “ $i|j$ ” of a GPU $x \xrightarrow{i|j} y$ represents i dereferences of x and j dereferences of y using the dereference operator $*$. We can also view the *indlev* “ $i|j$ ” as lists (also referred to as indirection list or *indlist*) containing i and j occurrences of $*$. This representation naturally allows field-sensitive handling of structures by using indirection lists containing field dereferences. Consider the statements $x = *y$ and $x = y \rightarrow n$ involving pointer dereferences. Since $x = y \rightarrow n$ is equivalent to $x = (*y).n$, we can represent the two statements by GPUs as shown below:

¹⁶We also perform weak updates for address-escaped variables (Section 10.1) because they share many similarities with heap locations. Like heap locations, address-escaped variables could outlive the lifetime of the procedures that create them. They potentially represent multiple concrete locations because of multiple calls to the procedure. Further, this number could be unbounded in case of recursive calls.

¹⁷In a top-down analysis, k -limiting is not required because allocation sites are propagated from callers to callees. While the use of k -limiting in a bottom-up approach seems like an additional restriction, unless the locations involved in a pointer chain are allocated by $m > k$ distinct allocation sites, there is no loss of precision compared to a top-down approach.

Pointer assignment	GPU	Remark
$x = \text{malloc}(\dots)$	$x \xrightarrow{[*][i]} h_i$	The allocation site name is i
$x = \text{NULL}$	$x \xrightarrow{[*][i]} \text{NULL}$	NULL is distinguished location
$x = y.n$	$x \xrightarrow{[*][n]} y$	
$x.n = y$	$x \xrightarrow{[n][*]} y$	
$x = y \rightarrow n$	$x \xrightarrow{[*][*,n]} y$	
$x \rightarrow n = y$	$x \xrightarrow{[*],n][[*]} y$	

Fig. 18. GPUs with indirection lists (*indlist*) for basic pointer assignments in C for structures.

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

We achieve field sensitivity by enumerating field names. Having a field-insensitive representation which does not distinguish between different fields, makes no difference for a statement $x = *y$, but loses precision for a statement $x = y \rightarrow n$. Figure 18 illustrates the GPUs corresponding to the basic pointer assignments involving structures.

The dereference in the pointer expression $y \rightarrow n$ is represented by an *indlist* written as $[*], n$ associated with pointer variable y . It means that, first the address in y is read and then the address in field n is read. On the other hand, the access $y.n$ as shown in the third row of Figure 18 can be mapped to location by adding the offset of field n to the virtual address of y at compile time. Hence, it can be treated as a separate variable which is represented by a node $y.n$ with an *indlist* $[*]$. We can also represent $y.n$ with a node y and an *indlist* $[n]$. For our implementation, we chose the former representation. However, the latter representation is more convenient for explaining the GPU compositions and hence we use it in the rest of the paper. For structures, we ensure field sensitivity by maintaining *indlist* in terms of field names. We choose to handle unions field-insensitively to capture aliasing between its fields.

Recall that a GPU composition $c \circ^\tau p$ involves balancing the *indlev* of the pivot in c and p (Section 4.2). With *indlist* replacing *indlev*, the operations remain similar in spirit, although now they become operations on lists rather than operations on numbers. To motivate the operations on *indlists*, let us recall the operations on *indlevs*: GPU composition $c \circ^\tau p$ requires balancing *indlevs* of the pivot which involves computing the difference between the *indlev* of the pivot in c and p . This difference is then added to the *indlev* of the non-pivot node in p . Recall that a GPU composition is *valid* (Section 4.2.2) only when the *indlev* of the pivot in c is greater than or equal to the *indlev* of the pivot in p . For convenience, we illustrate it again in the following example.

Example 34. Consider $p: y \xrightarrow{1|0} x$ and $c: w \xrightarrow{1|2} y$ where y is the pivot. Then a *TS* composition $c \circ^{\text{ts}} p$ is *valid* because *indlev* of y in c (which is 2) is greater than *indlev* of y in p (which is 1). The difference $(2 - 1)$ is added to the *indlev* of x (which then becomes 1) resulting in a reduced GPU $r: w \xrightarrow{1|(2-1+0)} x$, i.e. $r: w \xrightarrow{1|1} x$.

We define similar operations for *indlists*. A GPU composition is *valid* if the *indlist* of the pivot in GPU p is a prefix of the *indlist* of the pivot in GPU c . For example, the *indlist* “ $[*]$ ” is a prefix of the *indlist* “ $[*, n]$ ”. The addition (+) of

$$\begin{array}{l}
(z \xrightarrow[t]{il_1|il_2} x) \circ^{ts} (v \xrightarrow[u]{il_3|il_4} y) := \begin{cases} z \xrightarrow[t]{il_1|il_5} y & (v = x) \wedge (il_2 = il_3@il_6) \wedge (il_5 = il_4@il_6) \\ \text{fail} & \text{otherwise} \end{cases} \\
(x \xrightarrow[t]{il_1|il_2} z) \circ^{ss} (v \xrightarrow[u]{il_3|il_4} y) := \begin{cases} y \xrightarrow[t]{il_5|il_2} z & (v = x) \wedge (il_1 = il_3@il_6) \wedge (il_5 = il_4@il_6) \\ & \wedge il_6 \neq [] \\ \text{fail} & \text{otherwise} \end{cases}
\end{array}$$

Definition 9. GPU Composition $c \circ^{\tau} p$ using *indlists*

the difference $(-)$ in the *indlevs* of the pivot to the *indlev* of one of the other two nodes is replaced by the list-append operation denoted $@$.

Similarly computing the difference $(-)$ in the *indlev* of the pivot is replaced by the ‘list-difference’ or ‘list-remainder’ operation, $\text{Remainder} : \text{indlist} \times \text{indlist} \rightarrow \text{indlist}$; this takes two *indlists* as its arguments where the first is a prefix of the second and returns the suffix of the second *indlist* that remains after removing the first *indlist* from it. Given $il_2 = il_1 @ il_3$, $\text{Remainder}(il_1, il_2) = il_3$. When $il_1 = il_2$, the remainder il_3 is an empty *indlist* (denoted $[]$). A GPU composition is *valid* only when il_1 is a prefix of il_2 ; $\text{Remainder}(il_1, il_2)$ is computed only for *valid* GPU compositions. This is again a natural generalization of the integer *indlev* formulation earlier.

Example 35. Consider the statement sequence $y = x; w = y \rightarrow n$; In order to compose the corresponding GPUs $p : y \xrightarrow{[*][*]} x$ and $c : w \xrightarrow{[*][*,n]} y$ we find the list remainder of the *indlists* of y in the two GPUs. This operation ($\text{Remainder}([*], [*, n])$) returns $[n]$ which is appended to the *indlist* of node x (which is $[*]$) resulting in a new *indlist* $[*] @ [n] = [*, n]$ and thus, we get a reduced GPU $w \xrightarrow{[*][*,n]} x$ representing $w = x \rightarrow n$.

The formal definition of GPU composition using *indlists* is similar to that using *indlevs* (Definition 3) and is given in Definition 9. Note that for *TS* and *SS* compositions in the equations, the pivot is x . Besides, for *SS* composition, the condition $il_6 \neq []$ (generalizing the strict inequality ‘ $<$ ’ in Definition 3) ensures that the consumer GPU does not redefine the location defined by the producer GPU. Unlike the case of pointers to scalars, *TS* and *SS* compositions are not mutually exclusive for pointers to structures. For example, an assignment $x \rightarrow n = x$ could have both *TS* and *SS* compositions with a GPU p defining x . The two compositions are independent because *SS* composition resolves the source of a GPU whereas *TS* composition resolves the target of the GPU. Hence, they can be performed in any order.

A GPU composition is *desirable* if the *indlev* of r does not exceed that of c . Similarly, in the case of *indlists*, a GPU composition is *desirable* if *indlists* of r (say $il_1|il_2$) does not exceed that of c (say $il'_1|il'_2$), i.e. $|il_1| \leq |il'_1| \wedge |il_2| \leq |il'_2|$ where $|il|$ denotes the length of *indlist* il . Note that, for *desirability*, we only need a smaller length and not a prefix relation between *indlists*. In fact, the *indlist* in r is always a suffix of the *indlist* in c as illustrated by the following example.

Example 36. Consider the code snippet on right. The effect of statement 22 in the context of statement 21 can be seen as an assignment $z = y.n$. The composition of GPUs $c : z \xrightarrow{[*][*,n]} x$ and $p : x \xrightarrow{[*][*]} y$ results in the GPU $r : z \xrightarrow{[*][n]} y$. The *indlist* of the target (y) of r is not a prefix of that of target (x) of c but is a suffix.

21 : $x = \&y;$
22 : $z = x \rightarrow n;$


```

struct node * x;

01 struct node {
02     struct node * n;
03     int d;
04 };
05 void g() {
06     struct node * y;
07     while (...) {
08         print x → d;
09         x = x → n;
10     }
11 }

12 void f() {
13     struct node * y;
14     y = malloc(...);
15     x = y;
16     while (...) {
17         y → n = malloc(...);
18         y = y → n;
19     }
20     g();
21 }

```

(a) A program for creating a linked list and traversing it. We have omitted the null assignment for the last node of the list and the associated GPUs

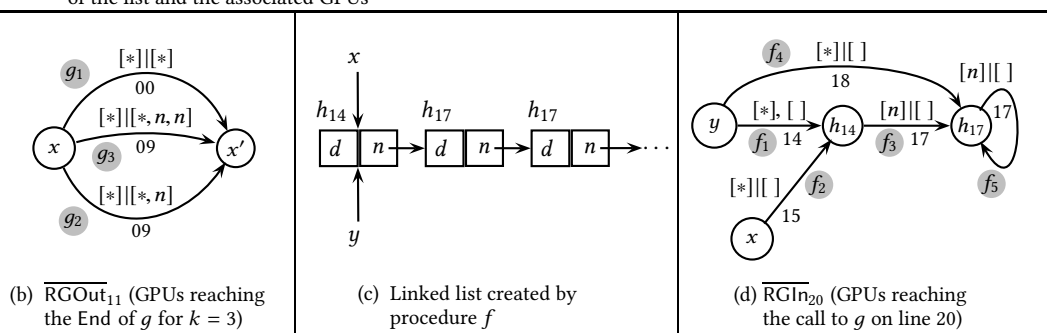


Fig. 19. An example demonstrating the need of k -limiting summarization technique in addition to allocation-site-based abstraction for the heap. h_{14} and h_{17} are the heap nodes allocated on lines 14 and 17 respectively.

8.2 Summarization Using Allocation Sites

Under the allocation-site-based abstraction for the heap, the objects created by an allocation statement are collectively named by the allocation site and undergo weak update. Thus, a statement $x = \text{malloc}(\dots)$ is represented by a GPU $x \xrightarrow{\frac{[*][[]]}{i}} h_i$ where h_i is the heap location created at the allocation site i . The example below illustrates how this bounds an unbounded heap in a GPG. For convenience, we identify GPUs using procedure names.

Example 37. For procedure f shown in Figure 19 we create heap objects h_{14} and h_{17} allocated at line numbers 14 and 17. The GPU set $\overline{\text{RGIn}}_{20}$ in procedure f represents a linked list with x as its head pointer (Figure 19(d)) and h_{14} as its first node. The remaining nodes in the list are represented by the heap location h_{17} and are summarized by a self-loop over the node. This set of GPUs is computed as follows: The GPU $f_1 : y \xrightarrow{\frac{[*][[]]}{14}} h_{14}$ is created for allocation-site 14. The GPU $x \xrightarrow{\frac{[*][[*]]}{15}} y$ composes with f_1 (under TS composition) to create a new GPU $f_2 : x \xrightarrow{\frac{[*][[]]}{15}} h_{14}$. When statement 17 is processed for the first time, GPU $y \xrightarrow{\frac{[*][n][[]]}{17}} h_{17}$ composes with f_1 (under SS composition) to create a GPU $f_3 : h_{14} \xrightarrow{\frac{[n][[]]}{17}} h_{17}$. When statement 18 is processed for the first time, the GPU $y \xrightarrow{\frac{[*][[*][n]]}{18}} y$ composes with f_1

(under *TS* composition) to create a GPU $y \xrightarrow{[*][n]}_{18} h_{14}$ which is further composed with f_3 (under *TS* composition) to create a GPU $f_4 : y \xrightarrow{[*][n]}_{18} h_{17}$. GPU f_4 kills GPU f_1 because y is redefined by statement 18. This completes the first iteration of the loop and the set of GPUs $\overline{\text{RGO}}_{19}$ is $\{f_2, f_3, f_4\}$ representing the following information:

- f_2 indicates that x points to the head of the linked list.
- f_3 indicates that the field n of heap location h_{14} points to heap location h_{17} .
- f_4 indicates that y points to heap location h_{17} .

In the second iteration of the reaching GPUs analysis over the loop, $\overline{\text{RGO}}_{15}$ and $\overline{\text{RGO}}_{19}$ are merged to compute $\overline{\text{RGN}}_{16}$ as $\{f_1, f_2, f_3, f_4\}$. When statement 17 is processed for the second time, the GPU $y \xrightarrow{[*][n]}_{17} h_{17}$ composes with

- f_1 (under *SS* composition) to create f_3 , and with
- f_4 (under *SS* composition) to create $f_5 : h_{17} \xrightarrow{[n][l]}_{17} h_{17}$.

When statement 18 is processed for the second time, f_4 is recreated killing f_1 . This completes the second iteration of the loop and the set of GPUs $\overline{\text{RGN}}_{20}$ is $\{f_1, f_2, f_3, f_4, f_5\}$. The new GPU f_5 implies that the field n of heap location h_{17} holds the address of heap location h_{17} . The self loop represents an unbounded list $(h_{17} \xrightarrow{n} h_{17} \xrightarrow{n} h_{17} \xrightarrow{n} h_{17} \dots)$ under the allocation-site-based abstraction. The third iteration of reaching GPUs analysis over the loop does not add any new information and reaching GPUs analysis reaches a fixed point.

The following example discusses the absence of blocking in the procedures in Figure 19.

Example 38. The GPUs in $\overline{\text{RGN}}_{14}$ reach statement 17 unblocked because there is no barrier. Since the pointee of y is available, the set $\overline{\text{RGG}}_{14}$ does not contain any indirect GPUs and hence do not contribute to the blocking of any GPUs. If the allocation site at statement 14 was not available, then the GPU for statement 17 would not have been reduced and hence the set $\overline{\text{RGG}}_{17}$ would contain an indirect GPU $y \xrightarrow{[*][n]}_{17} h_{17}$. This GPU would block all GPUs in $\overline{\text{RGN}}_{18}$ and in turn would be blocked by the GPUs in $\overline{\text{RGG}}_{18}$ so that it cannot be used for reduction of any successive GPUs.

8.3 Summarization Using k -Limiting

This section shows why allocation-site-based abstraction is not sufficient for a bottom-up points-to analysis although it serves the purpose well in a top-down analysis.

8.3.1 The Need for k -Limiting. In some cases, the allocation site may not be available during the construction of the GPG of a procedure. For our example in Figure 19, when the GPG is constructed for procedure g , we do not know the allocation site because the accesses to heap in procedure g refer to the data-structure created in procedure f . Thus allocation-site-based abstraction is not applicable for procedure g and the indirection lists grow without bound.

In a top-down analysis, k -limiting is not required because allocation sites are propagated from callers to callees.

Example 39. When the GPG for procedure g in Figure 19 is constructed, we have a boundary definition $g_1 : x \xrightarrow{[*][*]}_{00} x'$ at the start of the procedure. In the first iteration of the analysis over the loop, the GPU $x \xrightarrow{[*][*,n]}_{09} x$ composes with g_1 (under *TS* composition) creating a reduced GPU $g_2 : x \xrightarrow{[*][*,n]}_{09} x'$. The GPU g_2 kills GPU g_1 because x is redefined by statement at 09. However, the merge at the top of the loop reintroduces it. In the second iteration, the GPU $x \xrightarrow{[*][*,n]}_{09} x$ composes with g_1 to recreate g_2 , and with g_2 to create $g_3 : x \xrightarrow{[*][*,n,n]}_{09} x'$. In the third iteration, we get

an additional GPU $g_4 : x \xrightarrow{\frac{[*][*,n,n,n]}{09}} x'$ apart from g_2 and g_3 . This continues and the indirection lists of the GPUs between x and x' grow without bound leading to non-termination.

There are two ways of handling traversals of data structures created in some other procedure.

- As the above example illustrates, we perform compositions involving upwards exposed variables inspite of these compositions being *valid* but *undesirable*.
- Alternatively, we can postpone these compositions (as suggested before) until call inlining enables their reduction.

We use the first approach and bound the length of indirection lists using k -limiting. This limits the participation of the GPUs in the fixed-point computation for the procedures containing them. The second approach requires the GPUs to participate in the fixed-point computations for the callers as well. This could cause inefficiency.

While the use of k -limiting in a bottom-up approach seems like an additional restriction, unless the locations involved in a pointer chain are allocated by $m > k$ distinct allocation sites, there is no loss of precision compared to a top-down approach.

8.3.2 Incorporating k -Limiting. We limit the length of *indlists* to k such that the *indlist* is exact up to $k - 1$ dereferences and approximate for k or more dereferences in terms of an unbounded number of dereferences. Besides, the dereferences are field-insensitive beyond k . This summarization is implemented by redefining the list concatenation operator $@$ such that for $il_1 @ il_2$, the result is a k -limited prefix of the concatenation of il_1 and il_2 .

Example 40. The set of GPUs $\overline{\text{RGO}}_{11}$ reaching the End of procedure g of Figure 19, for $k = 3$ is given in the Figure 19(b). A GPU between x and x' has an *indlist* $[*, n]$ of length 2 and all *indlists* of length ≥ 3 are approximated by $[*, n, n]$.

GPU $g_1 : x \xrightarrow{\frac{[*][*]}{00}} x'$ in the GPG for procedure g represents the effect of **while** loop not executed even once. GPU $g_2 : x \xrightarrow{\frac{[*][*,n]}{09}} x'$ represents the effect of the first iteration of the **while** loop. The GPU $g_3 : x \xrightarrow{\frac{[*][*,n,n]}{09}} x'$ represents the combined effect of the second and all subsequent iterations of the **while** loop. The GPG of procedure g (Δ_g) contains a single GPB which in turn contains a set of GPUs $\{g_2, g_3\}$.

Note that an explicit summarization is required only for heap locations and address-escaped stack locations in recursive procedures because the *indlists* can grow without bound only in these cases (see Footnote 16).

The GPU composition defined in Section 8.1 (Definition 9) is extended to handle k -limited *indlists* in the following manner: The removal of a prefix from a k -limited *indlist* in the Remainder operation is over-approximated by suffixing special field-insensitive dereferences denoted by “ \dagger ” where \dagger represents any field. For an operation $\text{Remainder}(il_1, il_2)$, il_1 must be a prefix of il_2 as explained in Section 8.1. Let $il_2 = il_1 @ il_3$ for $\text{Remainder}(il_1, il_2)$. We define a summarized list-remainder operation $\text{sRemainder} : \text{indlist} \times \text{indlist} \rightarrow 2^{\text{indlist}}$ which takes two *indlists* as its arguments and computes a set of *indlists* as shown below:

$$\text{sRemainder}(il_1, il_2) = \begin{cases} \{il_3 \mid il_2 = il_1 @ il_3\} & |il_2| < k \\ \{il_3 @ \sigma \mid il_2 = il_1 @ il_3, \sigma \text{ is a sequence of } \dagger, 0 \leq |\sigma| \leq |il_1|\} & \text{otherwise} \end{cases}$$

Observe that sRemainder is a generalization of Remainder defined in Section 8.1 because it computes a set of *indlists* when its second argument is a k -limited *indlist*; for non k -limited *indlist*, sRemainder returns a singleton set. The longest *indlist* in the set computed by sRemainder represents a summary whereas the other *indlists* are exact in length

```

Input:  $c$            // The consumer GPU to be simplified
          $R$            // The context (set of GPUs) in which  $c$  is to be simplified
         Used        // The set of GPUs used for GPU reduction for a GPU
Output: Red       // The set of simplified GPUs equivalent to  $c$ 
01 GPU_reduction ( $c, R, Used$ )
02 { Red =  $\emptyset$ 
03    $composed = false$ 
04   for each  $\gamma \in (R - Used)$ 
05   { for each  $r \in (c \circ^{ts} \gamma)$ 
06     { Red = Red  $\cup$  GPU_reduction ( $r, R, Used \cup \{\gamma\}$ )
07      $composed = true$ 
08     }
09   for each  $r \in (c \circ^{ss} \gamma)$ 
10     { Red = Red  $\cup$  GPU_reduction ( $r, R, Used \cup \{\gamma\}$ )
11      $composed = true$ 
12     }
13   }
14   if ( $\neg composed$ )
15     Red = Red  $\cup$   $\{c\}$ 
16   return Red
17 }

```

Definition 10. GPU Reduction $c \circ R$ for Handling Heap

but approximate in terms of fields because of field insensitivity introduced by \dagger .¹⁸ This is illustrated in the example below.

Example 41. For $k = 3$, some examples of the sets of *indlists* computed by the sRemainder operation are shown below:

$$\begin{aligned} \text{sRemainder}([*], [*], n, n) &= \{[n, n], [n, n, \dagger]\} \\ \text{sRemainder}([*, n], [*], n, n) &= \{[n], [n, \dagger], [n, \dagger, \dagger]\} \\ \text{sRemainder}([*, n, n], [*], n, n) &= \{[], [\dagger], [\dagger, \dagger], [\dagger, \dagger, \dagger]\} \end{aligned}$$

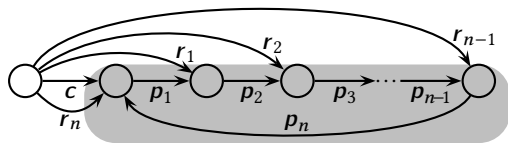
For the last case, the sRemainder operation can be viewed as an operation that creates an intermediate set $S = \{[*], n, n, [*, n, n, \dagger], [*, n, n, \dagger, \dagger], [*, n, n, \dagger, \dagger, \dagger]\}$ obtained by adding upto 3 occurrences of \dagger (because $k = 3$). The sRemainder operation can then be viewed as a collection of Remainder($[*, n, n], \sigma$) for each σ in this set:

$$\text{sRemainder}([*, n, n], [*], n, n) = \{\text{Remainder}([*, n, n], \sigma) \mid \sigma \in S\}$$

The first two cases in this example can also be explained in a similar manner.

GPU composition using *indlevs* (Section 4.2.2) or using *indlists* (Section 8.1) is a partial operation defined to compute a single GPU as its result when it succeeds. Since we do not have a representation for an “invalid” GPU, we model failure

¹⁸This is somewhat similar to materialization [23] which extracts copies out of summary representation of an object to create some exact objects.



- The shaded part shows the GPUs in \overline{RGIn} .
- Let $r_0 = c$. Then $r_i = r_{i-1} \circ^\tau p_i$, $i > 0$.
- For simplicity, the directions chosen in the GPUs illustrate only *TS* compositions.

Fig. 20. Series of compositions and its consequence when the graph induced by the GPUs in \overline{RGIn} (shown by the shaded part) has a cycle. The compositions may happen more than the required number of times, resulting in a points-to edge.

by defining GPU composition as a partial function for GPUs containing *indlevs* or non-*k*-limited *indlists*. However, when *indlists* are summarized using *k*-limiting, *sRemainder* naturally computes a set of *indlists* (unlike *Remainder* which computes a single *indlist*). This allows us to define GPU composition as a total function, since we can express the previous partiality simply by returning an empty set.

8.4 Extending GPU Reduction to Handle Cycles in GPUs

In the presence of a heap, the graph induced by the set of GPUs reaching a GPB can contain cycles of the following two kinds:

- Cycles arising out of creation of a recursive data structure in a procedure under allocation-site-based abstraction. This manifests itself in the form of a cycle involving heap nodes h_i as illustrated in Example 37 in Section 8.2. These cycles are closed form representations of acyclic unbounded paths in the memory.
- Cycles arising out of cyclic data structures. These cycles represent cycles in the memory.

Both these cases of cycles are handled by GPU composition using *sRemainder* operation over indirection lists. Definition 10 extends the algorithm for GPU reduction to use the new definition of GPU composition which computes a set of GPUs instead of a single GPU.

For GPU reduction $c \circ R$, an *admissible* composition $r_1 = c \circ^\tau p_1$ (where $p_1 \in \overline{RGIn}$) may lead to another composition $r_2 = r_1 \circ^\tau p_2$ (where $p_2 \in \overline{RGIn}$). This in turn may lead to another composition thereby creating a chain of compositions. If the graph induced by the reaching GPUs (i.e. GPUs in \overline{RGIn}) has a cycle (as illustrated in Example 37 in Section 8.2), some p_m must be adjacent to p_1 with the length of the cycle being $m + 1$ as illustrated in Figure 20. The lengths of *indlists* in r_i would be smaller than (or equal to) those in r_{i-1} because of *admissibility*. If the length of an *indlist* in c exceeds m , the series of compositions would resume with p_1 after the composition with p_m . In other words, after computing r_{m-1} using the composition $r_{m-2} \circ p_m$, the next GPU r_m would be computed using the composition $r_{m-1} \circ p_1$ and the process will continue until some r_j , $j \geq m$ is a points-to edge.¹⁹ Thus, we will have more compositions than required and the result of GPU reduction may not represent the updates of locations that are updated by the original GPU c . In order to prohibit this, we allow a GPU p to be used only once in a chain of compositions.

Hence, the new definition of GPU reduction (Definition 10) uses an additional argument, *Used*, which maintains a set of GPUs that have been used in a chain of GPU compositions. For the top level non-recursive call to *GPU_reduction*, *Used* = \emptyset . In the case of pointers to scalars, a graph induced by a set of GPUs cannot have a cycle, hence a GPU p cannot be used multiple times in a series of GPU compositions. Therefore, we did not need set *Used* for defining GPU reduction in the case of pointers to scalars (Definition 4).

¹⁹Note that this happens for reducing a single GPU c in the context of \overline{RGIn} and does not require a cycle in the GPG.

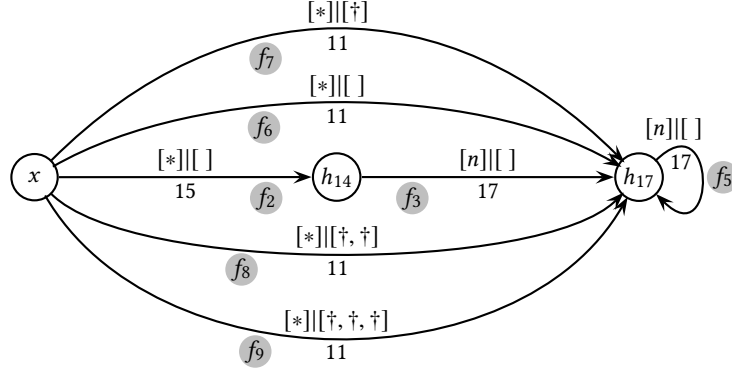


Fig. 21. The set of GPUs $\overline{\text{RGOut}}_{20}$ after the call to procedure g in procedure f of Figure 19. Local variable y has been eliminated.

Example 42. This example illustrates GPU reduction with 3-limited *indlists* using GPU g_3 of Δ_g shown in Figure 19(b). At the call site 20 in procedure f of Figure 19(a), the upwards-exposed variable x' in Δ_g is substituted by x in Δ_f (see Section 6). All GPU compositions for this examples are *TS* compositions. The GPUs in $\overline{\text{RGIn}}_{20}$ (Figure 19(d)) are used for composition. The set $\overline{\text{RGOut}}_{20}$ is same as $\overline{\text{RGOut}}_{21}$ shown in Figure 21 except that $\overline{\text{RGOut}}_{20}$ also contains the GPUs involving y which is a local variable of f and is not in the scope of the caller procedures.

The GPU composition $g_2 \circ f_2$ for $f_2: x \xrightarrow{15} h_{14}$ and $g_2: x \xrightarrow{11} x$ (with x substituting for x') creates a reduced GPU $x \xrightarrow{11} h_{14}$ which is further composed with $f_3: h_{14} \xrightarrow{17} h_{17}$ to create a reduced GPU $f_6: x \xrightarrow{11} h_{17}$ (Figure 21).

Now GPU g_3 must be composed with f_2 , f_3 and f_5 . The composition $g_3 \circ f_2$ for $g_3: x \xrightarrow{11} x$ creates two GPUs $x \xrightarrow{11} h_{14}$ and $x \xrightarrow{11} h_{14}$. The newly created GPU $x \xrightarrow{11} h_{14}$ is further composed with f_3 to create GPU $x \xrightarrow{11} h_{17}$ which is further composed with f_5 to recreate GPU $f_6: x \xrightarrow{11} h_{17}$. The GPU composition between the other newly created GPU $x \xrightarrow{11} h_{14}$ and f_3 creates GPUs $x \xrightarrow{11} h_{17}$ and $x \xrightarrow{11} h_{17}$. The GPU $x \xrightarrow{11} h_{17}$ further composes with f_5 creating a GPU $f_7: x \xrightarrow{11} h_{17}$ while the composition between GPUs $x \xrightarrow{11} h_{17}$ and f_5 creates two reduced GPUs $f_8: x \xrightarrow{11} h_{17}$ and $f_9: x \xrightarrow{11} h_{17}$.

Note that GPU f_5 is used only once in a series of compositions (Example 43 explains this).

The final reduced GPUs f_6 , f_7 , f_8 , and, f_9 are members of the set $\overline{\text{RGOut}}_{21}$ containing the GPUs reaching the End of procedure f (as shown in Figure 21). These reduced GPUs represent the following information:

- f_6 implies that x now points-to heap location h_{17} .
- f_7 imply that x points-to heap locations that are one dereference away from h_{17} .
- f_8 imply that x points-to heap locations that are two dereferences away from h_{17} .
- f_9 imply that x points-to heap locations that are beyond two dereferences from h_{17} .

Thus, x points to every node in the linked list.

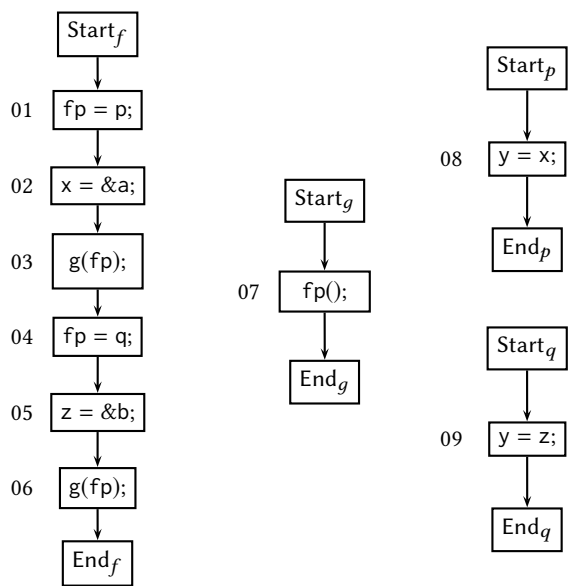


Fig. 22. An example demonstrating the handling of function pointers.

Example 43. To see why GPU reduction in Definition 10 excludes a GPU used for composition once, observe that GPUs f_7 , f_8 and f_9 can be further composed with GPU f_5 . The composition of f_7 with f_5 creates GPU f_6 . Similarly, repetitive compositions of f_8 with f_5 also creates GPU f_6 . This indicates that x points to only h_{17} and misses out the fact that x points to every location in the linked list which is represented by h_{17} and is represented by GPUs f_7 , f_8 and f_9 .

A cycle in a graph induced by a set of GPUs could also occur because of a cyclic data structure.

Example 44. Let an assignment $y \rightarrow n = x$ be inserted in procedure f after line 19 in Figure 19. This creates a circular linked list instead of a simple linked list. This will cause inclusion of the GPU $h_{17} \xrightarrow{[n][l]} h_{14}$ in Figure 19(d), thereby creating a cycle between the nodes h_{14} and h_{17} .

9 HANDLING CALLS THROUGH FUNCTION POINTERS

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

Our goal is to convert a call through a function pointer into a direct call for every pointee of the function pointer. Interleaving of strength reduction and call inlining reduces the GPU $u \xrightarrow{1|1}_s fp$ and provides the pointees of fp . This is identical to computing points-to information (Section 7). Until the pointees become available, the GPU $u \xrightarrow{1|1}_s fp$ acts as a barrier. Once the pointees become available, the indirect call converts to a set of direct calls and are handled as explained in Section 6.

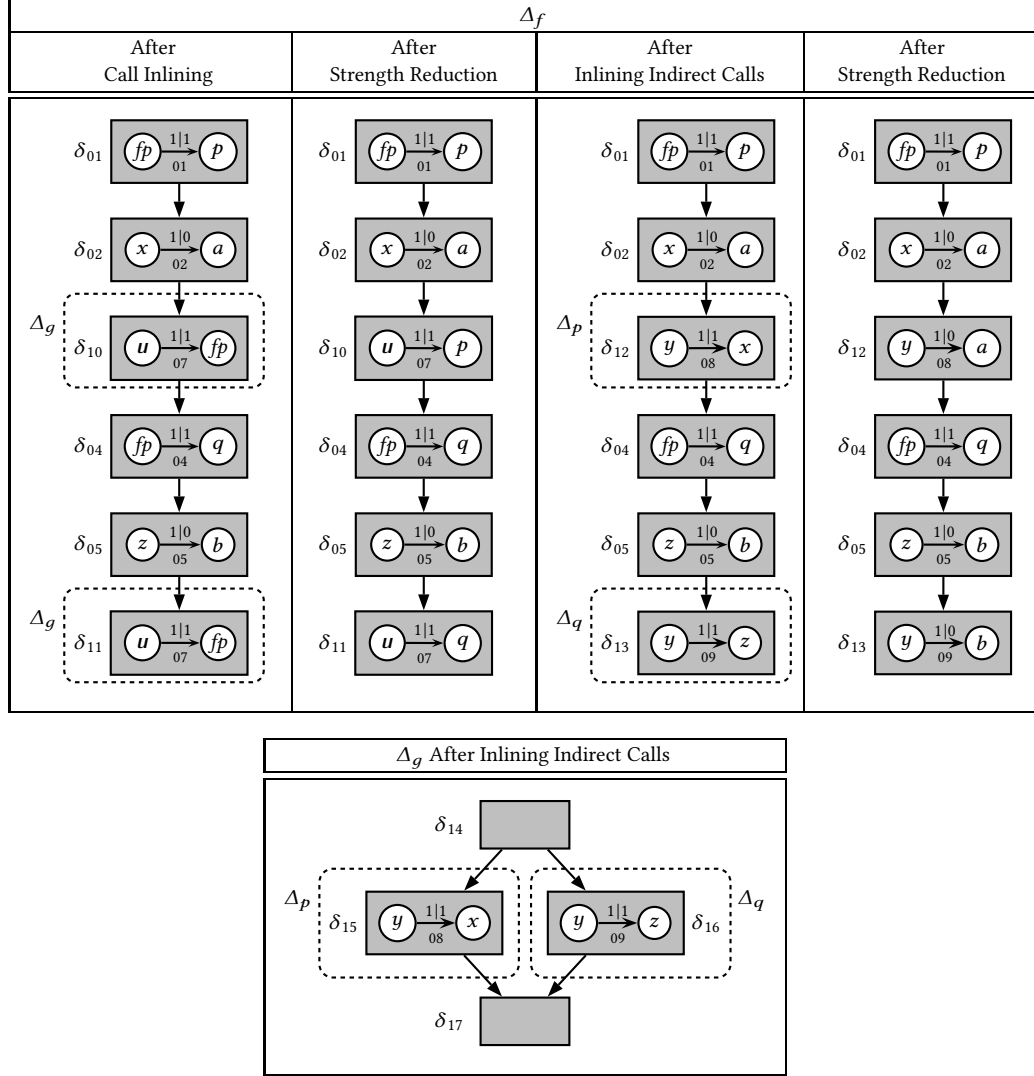


Fig. 23. Handling function pointers for the example in Figure 22. First, the direct calls are inlined leading to the discovery of pointees of the function pointer fp causing further inlining and strength reduction. See Example 45 for explanation.

Example 45. Figure 22 provides an example of procedures containing calls through function pointers. Figure 23 provides the GPGs of the procedures before and after resolving all calls through function pointers. Procedure g has an indirect call through function pointer fp in statement 07 and is modelled by a GPB containing a single GPU $u \xrightarrow{1|1}_{07} fp$ where u models a use (Section 7). This GPG is inlined in procedure f in statement 03 as δ_{10} and in statement 06 as δ_{11} .

Since we have $fp \xrightarrow{1|1}_{01} p \in \overline{\text{RGI}}n_{10}$, the GPU in δ_{10} reduces to $u \xrightarrow{1|1}_{07} p$ indicating that the callee of this indirect call is p . Similarly, the callee for the indirect call in δ_{11} is q . Hence we inline Δ_p in δ_{10} which then becomes δ_{12} . Similarly,

Program	kLoC	# of pointer stmts	# of call sites	# of procs.	Proc. count for different buckets of # of calls				# of procs. requiring different no. of PTFs based on the no. of aliasing patterns					
					2-5	5-10	10-20	20+	2-5	6-10	11-15	15+	2-5	15+
	A	B	C	D	E				F				G	
lbn	0.9	370	30	19	5	0	0	0	8	0	0	0	13	0
mcf	1.6	480	29	23	11	0	0	0	0	0	0	0	4	0
libquantum	2.6	340	277	80	24	11	4	3	7	3	1	0	14	4
bzip2	5.7	1650	288	89	35	7	2	1	22	0	0	0	28	2
milc	9.5	2540	782	190	60	15	9	1	37	8	0	1	35	25
sjeng	10.5	700	726	133	46	20	5	6	14	3	1	3	10	14
hmmmer	20.6	6790	1328	275	93	33	22	11	62	5	3	4	88	32
h264ref	36.1	17770	2393	566	171	60	22	16	85	17	5	3	102	46
gobmk	158.0	212830	9379	2699	317	110	99	134	206	30	9	10	210	121

Table 1. Benchmark characteristics relevant to our analysis.

Δ_q is inlined in δ_{11} which then becomes δ_{13} . This information is reflected in g by recording p and q as the pointees of fp in statement 07. The indirect call in g is converted to two direct calls leading to the inlining of Δ_p and Δ_q in Δ_g .

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

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 between memory updates when the data dependence between them is not known. Thus, the size of a summary can be potentially large. This effect is exacerbated by the transitive inlining of the summaries of the callee procedures which can increase the size of a summary exponentially thereby hampering the scalability of analysis.

Section 10.1 describes our implementation, Section 10.2 describes the metrics that we have used for our measurements, Section 10.3 describes our empirical observations, and Section 10.4 analyzes our observations and describes the lessons learnt.

10.1 Implementation and Experiments

We have implemented GPG-based points-to analysis in GCC 4.7.2 using the LTO framework and have carried out measurements on SPEC CPU2006 benchmarks on a machine with 16 GB RAM with eight 64-bit Intel i7-4770 CPUs running at 3.40GHz.

Our method eliminates non-address-taken local variables using the def-use chains explicated by the SSA-form. Although we construct GPUs involving such variables, they are used for computing the points-to information within the

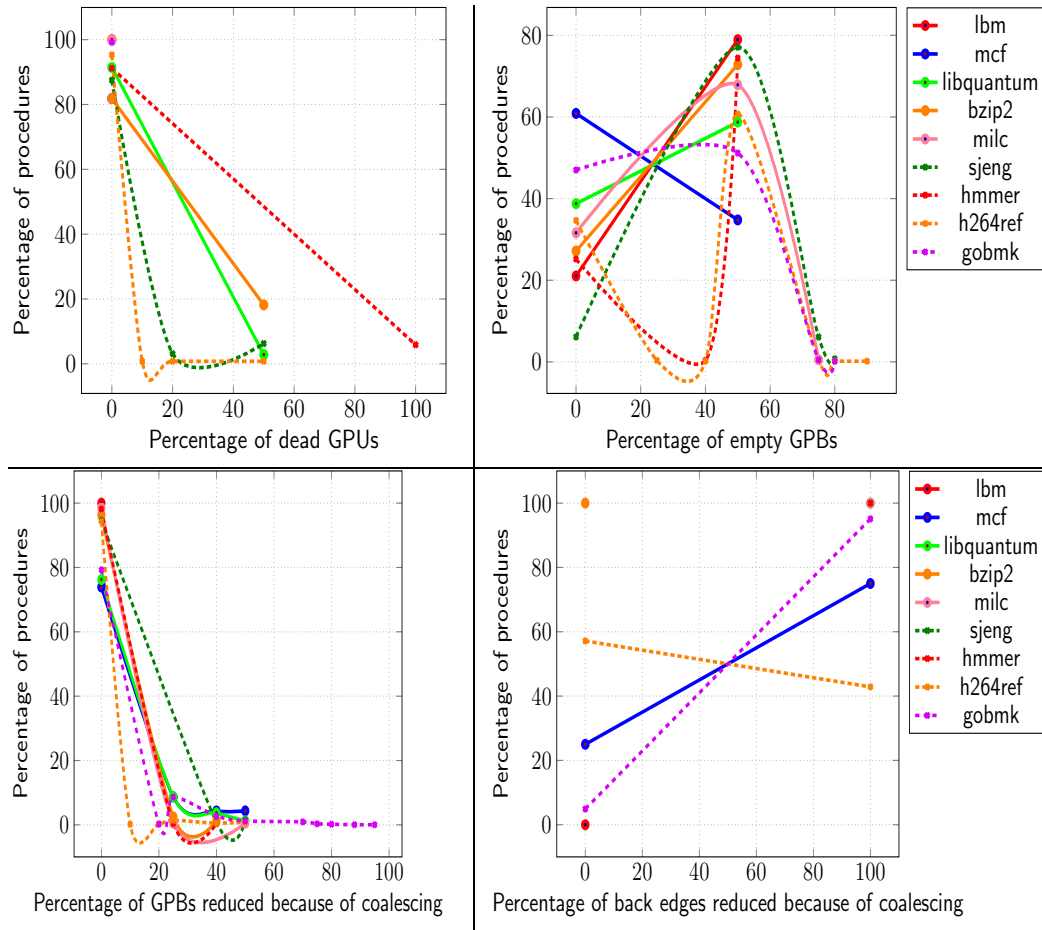


Fig. 24. Effectiveness of redundancy elimination optimizations. Benchmarks libquantum, milc, sjeng, and hmmer have all procedures whose all back edges are eliminated because of coalescing shown by the same point (100, 100) in the fourth plot. Hence they are not visible separately.

procedure and do not appear in the GPG of the procedure. If a GPU defining a global variable or a parameter reads a non-address-taken local variable, we identify the corresponding producer GPUs by traversing the def-use chains transitively. This eliminates the need for filtering out the local variables from the GPGs for inlining them in the callers. As a consequence, a GPG of a procedure consists of GPUs that involve global variables²⁰, parameters of the procedure, and the return variable which is visible in the scope of its callers. Since non-address-taken local variables have SSA versions, storing the GPUs that define them flow-insensitively results in no loss of precision.

All address-taken local variables in a procedure are treated as global variables because they can escape the scope of the procedure. However, these variables are not strongly updated because they could represent multiple locations.

We approximate the heap memory by maintaining k -limited indirection lists of field dereferences for $k = 3$ (see Section 8). An array is treated as a single variable in the following sense: accessing a particular element is seen as

²⁰ From now on we regard static, heap-summary nodes, and address-taken local variables as ‘global variables’.

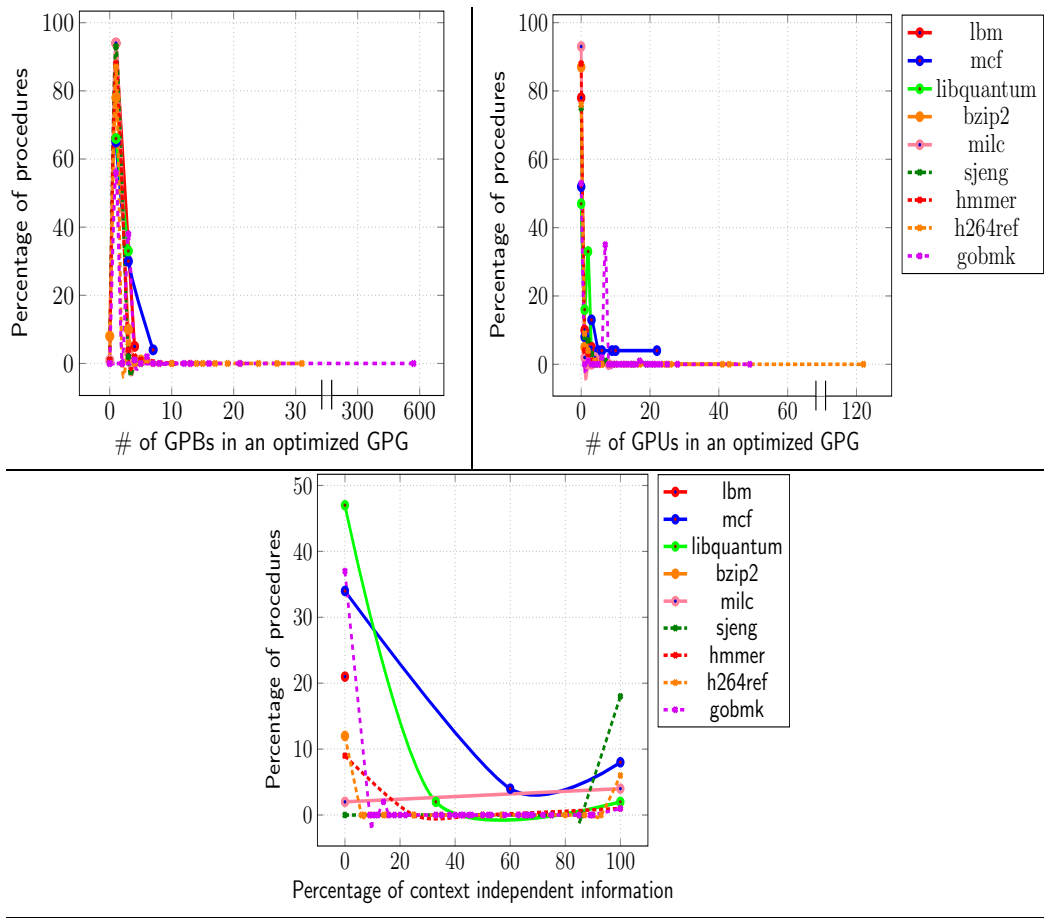


Fig. 25. Goodness measure of procedure summaries. A break in X-axis shown by two parallel lines is a discontinuity necessitated by wide variation in the number of GPUs and GPBs across benchmarks.

accessing every possible element and updates are treated as weak updates. This applies to both when arrays of pointers are manipulated, as well as when arrays are accessed through pointers. Since there is no kill owing to weak update, arrays are maintained flow-insensitively by our analysis.

For pointer arithmetic involving a pointer to an array, we approximate the pointer being defined to point to every element of the array. For pointer arithmetic involving other pointers, we approximate the pointer being defined to point to every possible location. Our current implementation handles only locally defined function pointers (Section 9) but can be easily extended to handle function pointers defined in the calling contexts too.

We have also implemented flow-insensitive points-to analysis by collecting the GPUs in a *GPG store* which differs from a GPB in that GPUs within a store can compose with each other whereas those in GPB cannot. This allowed us to implement the following variants:

- Flow- and context-insensitive (FICI) points-to analysis. For each benchmark program, we collected all GPUs across all procedures in a common store and performed all possible reductions. The resulting GPUs were classical points-to edges representing the flow- and context-insensitive points-to information.
- Flow-insensitive and context-sensitive (FICS) points-to analysis. For each procedure of a benchmark program, all GPUs within the procedure were collected in a store for the procedure and all possible reductions were performed. The resulting store was used as a summary in the callers of the procedure giving context-sensitivity. In the process the GPUs are reduced to classical points-to edges using the information from the calling context. This represents the flow-insensitive and context-sensitive points-to information for the procedure.

The third variant i.e., flow-sensitive and context-insensitive (FSCI) points-to analysis can be modelled by constructing a supergraph by joining the control flow graphs of all procedures such that calls and returns are replaced by gotos. This amounts to a top-down approach (or a bottom-up approach with a single summary for the entire program instead of separate summaries for each procedure). For practical programs, this initial GPG is too large for our analysis to scale. Our analysis achieves scalability by keeping the GPGs as small as possible at each stage. Therefore, we did not implement this variant of points-to analysis. Note that the FICI variant is also not a bottom-up approach because a separate summary is not constructed for every procedure. However, it was easy to implement because of a single store.

10.2 Measurements

We have measured the following for each benchmark program. The number of procedures varies significantly across the benchmark programs. Besides, the number of GPUs and GPBs varies across GPGs. Hence we have plotted such data in terms of percentages.²¹

- 1) Characteristics of benchmark programs (Table 1).
- 2) Effectiveness of redundancy elimination optimizations (Figure 24):
 - a) The number of dead GPUs for each procedure.
 - b) The number of empty GPBs for each procedure created by strength reduction, call inlining and dead GPU elimination.
 - c) A reduction in the number of GPBs due to coalescing.
 - d) A reduction in the number of back edges due to coalescing.
- 3) The goodness metric of the optimized procedure summaries (Figure 25):
 - a) Number of GPBs in the optimized GPGs.
 - b) Number of GPUs in the optimized GPGs.
 - c) Number of GPUs that are dependent on locally defined pointers alone.
- 4) The number of GPBs in a GPG (Figure 26):
 - a) After call inlining, relative to the number of basic blocks in the CFG.
 - b) After all optimizations, relative to the number of basic blocks in the CFG.
 - c) After all optimizations, relative to the number of GPBs after call inlining.
- 5) The number of GPUs in a GPG (Figure 27):
 - a) After call inlining, relative to the number of pointer assignments in the CFG.
 - b) After all optimizations, relative to the number of pointer assignments in the CFG.
 - c) After all optimizations, relative to the number of GPUs in the GPG after call inlining.

²¹The actual procedure counts are available at <https://www.cse.iitb.ac.in/~uday/soft-copies/gpg-pta-paper-appendix.pdf>.

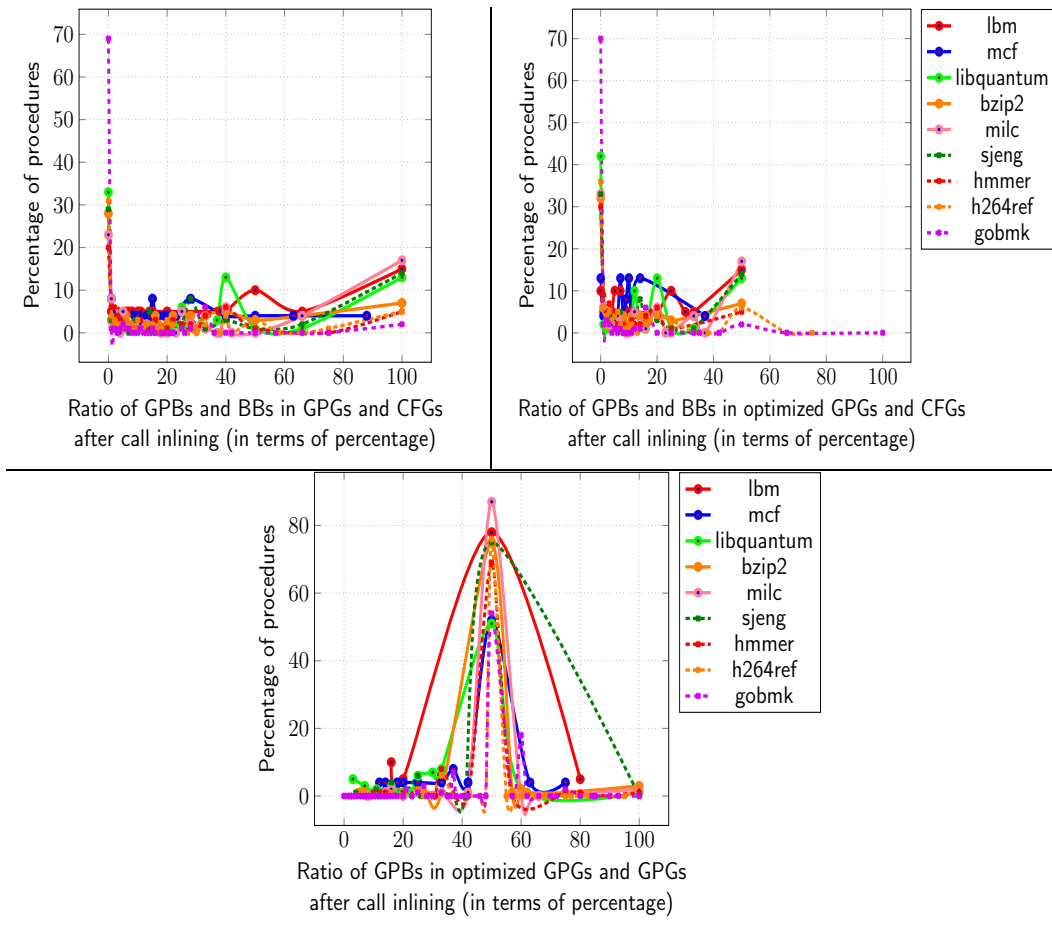


Fig. 26. Size of GPGs relative to the size of corresponding procedures in terms of GPBs and basic blocks.

- 6) The number of control flow edges in a GPG (Figure 28):
 - a) After call inlining, relative to the number of edges in the CFG.
 - b) After all optimizations, relative to the number of edges in the CFG.
 - c) After all optimizations, relative to the number of edges in the GPG after call inlining.
- 7) Miscellaneous data about GPGs (Table 2).
- 8) Time measurements (Figure 29):
 - a) FSCS (with and without blocking), FICI, and FICS variants of points-to analyses (second plot).
 - b) Time for different optimizations without blocking (third plot).
 - c) Time for different optimizations with blocking (fourth plot).
- 9) Average points-to pairs per procedure in FSCS, FICI, and FICS variants of points-to analyses. This data is plotted in the first plot of Figure 29.

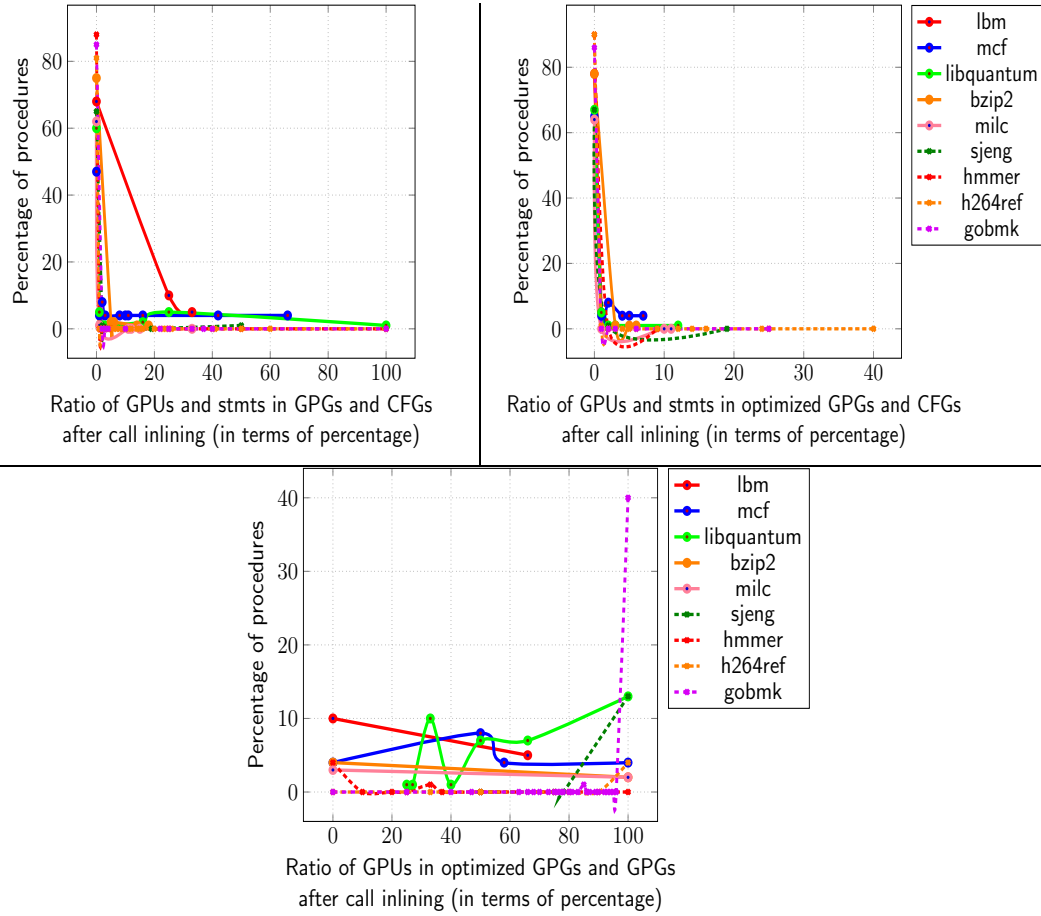


Fig. 27. Size of GPGs relative to the size of procedures in terms of GPUs and pointer assignments.

10.3 Observations

We describe our observations about the sizes of GPGs, GPG optimizations, and performance of the analysis. Observations related to the time measurements are presented in the end. Section 10.4 discusses these observations by analyzing them.

10.3.1 Effectiveness of Redundancy Elimination Optimizations. We observe that:

- The percentage of dead GPUs is very small and the dead GPU elimination optimization is the least effective of all the optimizations. Also, this optimization requires very little time compared to other optimizations (see Figure 29). Hence, disabling the optimization will neither improve the efficiency or scalability of the analysis nor will it affect the compactness of the GPGs.
- 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%-5% or close to 50% of GPBs are empty.

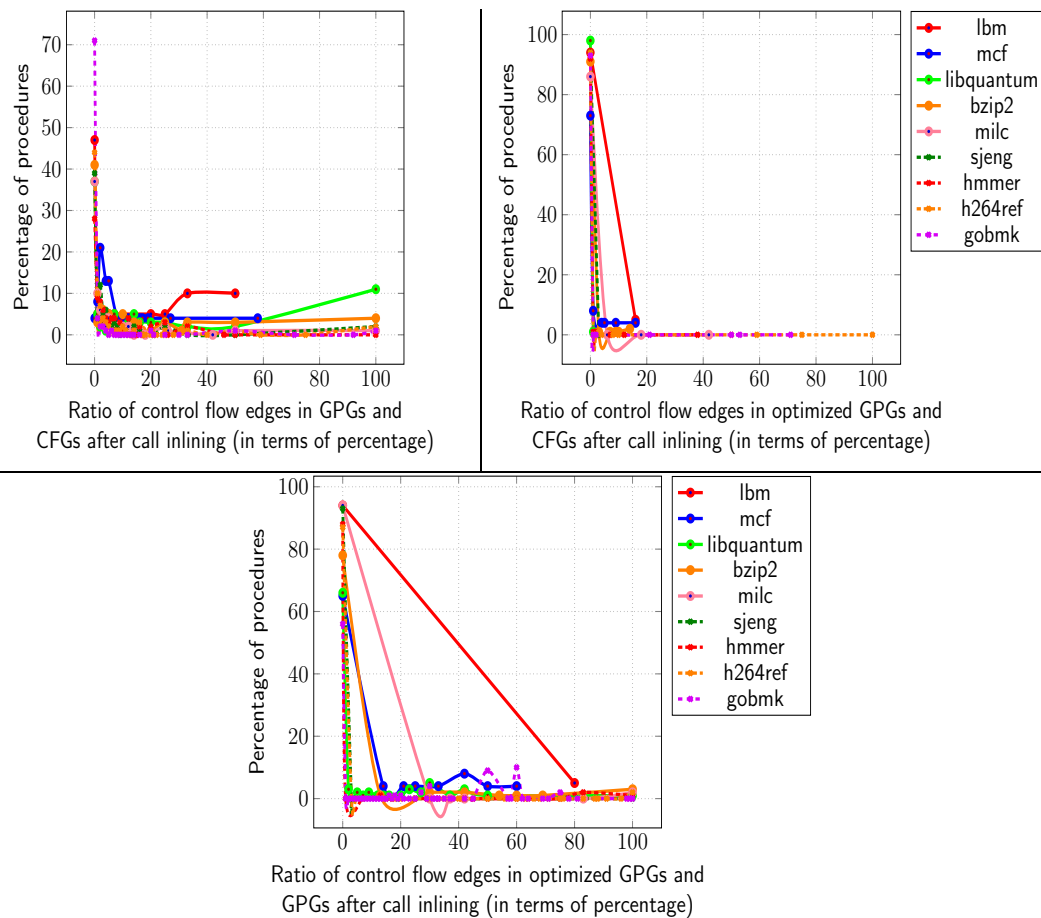


Fig. 28. Size of GPGs relative to the size of corresponding procedures in terms of control flow edges.

- (c) The last optimization among the redundancy elimination optimizations, coalesces the adjacent GPBs that do not require control flow between them. In our experience, many benchmarks had some very large GPGs in the presence of recursion. GPGs for recursive procedures are constructed by repeated inlinings of recursive calls. Coalescing was most effective for such procedures. Once these GPGs were optimized, the GPGs of the caller procedures did not have much scope for coalescing. In other words, coalescing did not cause uniform reduction across all GPGs but helped in the most critical GPGs. Hence we observe a reduction of 20% to 50% of GPBs for some but not majority of procedures.

Even if coalescing did not reduce the number of GPBs uniformly, it eliminated almost all back edges as shown in fourth plot in Figure 24. 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.

10.3.2 Goodness of Procedure Summaries. This data is presented in Tables 1, 2, and Figure 25. We use the following goodness metrics on procedure summaries:

Program	# of Proc. which have 0 GPUs	# of Proc. which have Δ_T as GPG	# of Proc. in which back edges are present in a CFG	# of Proc. in which back edges are present in a GPG	Exported Definitions	Imported Uses	# Queued GPUs	# Soundness Alerts
lbm	15	0	10	0	1.68	16.63	0	0
mcf	12	0	20	1	12.30	29.26	117	0
libquantum	38	0	36	0	1.54	1.89	0	0
bzip2	78	8	43	1	1.21	17.37	0	0
milc	184	3	94	0	0.70	6.14	0	0
sjeng	101	2	65	0	0.81	1.77	0	0
hmmmer	242	5	153	0	2.26	13.02	19	0
h264ref	434	3	308	5	1.60	26.75	13	0
gobmk	1436	2	464	8	0.39	1.36	6	0

Table 2. Miscellaneous data about the GPGs.

- (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 use multiple times. From column *E* in Table 1, it is clear that most procedures are called from many call sites. This indicates a high reusability of procedure summaries.
- (b) Compactness of a procedure summary. For scalability of a bottom-up approach, a procedure summary should be as compact as possible. Figure 25 and Table 2 show that the procedure summaries are indeed small in terms of number of GPBs and GPUs. GPGs for a large number of procedures have 0 GPUs because they do not manipulate global pointers (and thereby represent the identity flow function). Further, the majority of GPGs have 1 to 3 GPBs. Note that this is an absolute size of GPGs. Observations about the relative size of GPGs with respect to their CFGs are presented in Section 10.3.3 below.
- (c) Percentage of context-independent information. A procedure summary is very useful if it contains high percentage of context-independent information. We observe that the number of procedures with a high amount of context-independent information is larger in the larger benchmarks. Thus, a bottom-up approach is particularly useful for large programs.

10.3.3 Relative Size of GPGs with respect to the Size of Corresponding Procedures. For an exhaustive study, we compare three representations of a procedure with each other: (I) the CFG of a procedure, (II) the initial GPG obtained after call inlining, and (III) the final optimized GPG. Since GPGs have callee GPGs inlined within them, for a fair comparison, the CFG size must be counted by accumulating the sizes of the CFGs of the callee procedures. This is easy for non-recursive procedures. For recursive procedures, we accumulate the size of a CFG as many times as the number of inlinings of the corresponding GPG (Section 6.2). Further, the number of statements in a CFG is measured only in terms of the pointer assignments.

- (a) The first plot in these figures gives the size of the initial GPG (i.e. II) relative to that of the corresponding CFG (i.e. I). It is easy to that the reduction is immense: a large number of initial GPGs are in the range 0%-20% of the corresponding CFGs.
- (b) The second plot in these figures gives the size of the optimized GPG (i.e. III) relative to that of the corresponding CFG (i.e. I). The number of procedures in the range of 0%-20% is larger here than in the first plot indicating more reduction because of optimizations.

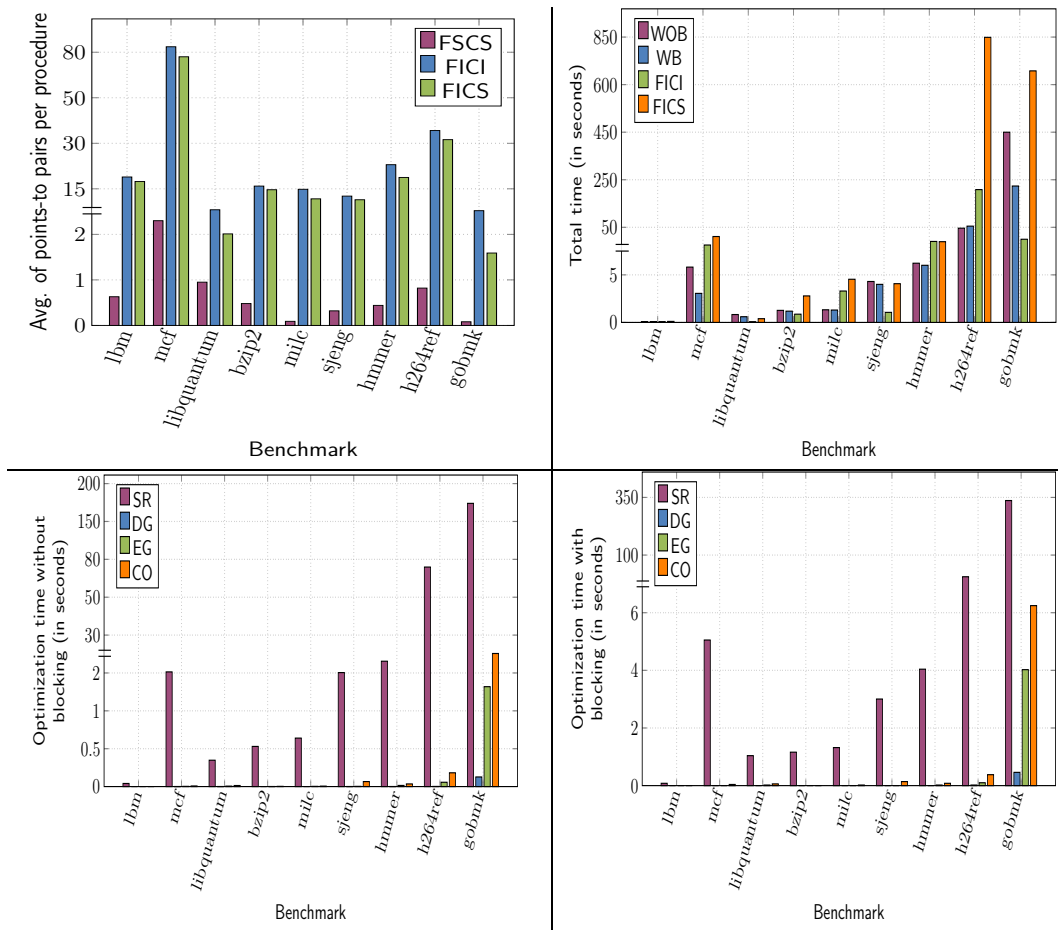


Fig. 29. Final points-to information measurements (first plot) and time measurements (the remaining three plots). FSCS (flow- and context-sensitive), FICI (flow- and context-insensitive), FICS (flow-insensitive and context-sensitive), WOB (our analysis without blocking), WB (our analysis with blocking), SR (strength reduction optimization), DG (dead GPU elimination), EG (empty GPB elimination), CO (coalescing). The time taken by dead GPU elimination, empty GPB elimination, and coalescing is negligible for small benchmarks and hence the corresponding bars are not visible.

- (c) The third plot in these figures gives the size of the optimized GPG (i.e. III) relative to that of the initial GPG (i.e. I). Here the distribution of procedures is different for GPBs, GPUs, and control flow edges. In the case of GPBs, the reduction factor is 50%. For GPUs, the reduction varies widely. The largest reduction is found for control flow: a large number of procedures fall in the range 0%-20%. The number of procedures in this range is larger than in the case of GPBs or GPUs indicating that the control flow is optimized the most.
- (d) As a special case of control flow reduction, we have measured the effect of our optimizations on back edges. This is because the presence of back edges increases the number of iterations required for fixed-point computation in an analysis. If a procedure summary needs to encode control flow, it is desirable to eliminate back edges to the extent possible. The data in Table 2 shows that most of the GPGs are acyclic in spite of the fact that the number of procedures with back edges in CFG is large.

Program	# of Proc.	# of Stmts.	FSCS			FICI	FICS
			FS	FI	FS+FI		
			Avg (per stmt)	Avg (per proc)	Avg (per proc)	Avg (per proc)	Avg (per proc)
lbn	19	367	1.99	0.79	0.63	19.26	17.11
mcf	23	484	4.12	9.30	2.30	82.13	77.39
libquantum	80	342	0.58	0.57	0.95	3.46	2.01
bzip2	89	1645	2.18	0.65	0.48	14.72	12.96
milc	196	2504	1.18	3.10	0.09	13.21	8.71
sjeng	133	684	1.44	1.83	0.32	10.04	8.17
hammer	275	6719	1.28	1.14	0.44	25.12	19.01
h264ref	566	17253	2.35	12.02	0.82	35.04	30.75
gobmk	2699	10557	0.74	6.36	0.08	2.95	1.59

Table 3. Final points-to information. FSCS (flow- and context-sensitive), FICI (flow- and context-insensitive), FICS (flow-insensitive and context-sensitive).

10.3.4 Final Points-to Information. We compared the amount of points-to information computed by our approach with flow- and context-insensitive (FICI) and flow-insensitive and context-sensitive (FICS) methods (first plot of Figure 29 and Table 3). For this purpose, we computed number of points-to pairs per procedure in all the three approaches by dividing the total number of unique points-to pairs across all procedures by the total number of procedures. Predictably, this number is smallest for our analysis (FSCS) and largest for FICI method.

10.3.5 Time measurements. We have measured the overall time as well as the time taken by each of the optimizations (Figure 29). We have also measured the time taken by the FICI and FICS variants of points-to analysis. Our observations are:

- (a) Our analysis takes less than 8 minutes on gobmk.445 which is a large benchmark with 158 kLoC. Our current implementation does not scale beyond that.
- (b) Strength reduction is the most expensive optimization followed by coalescing which is the most expensive among the redundancy elimination optimizations.
- (c) We introduced reaching GPUs analysis with blocking to ensure soundness of strength reduction so that a barrier GPU does not cause a side-effect invalidating strength reduction. However, our intuition was that very few of us write programs where a pointer is manipulated in such a manner. Hence we identified possible soundness alerts. The soundness alerts arise when a GPU whose composition was postponed, is updated by a GPU within the same GPG after inlining in a caller GPG. This is identified by checking if a GPU in the set `Queued` of a GPG is killed by the GPU of the same GPG when it is inlined in a caller.

We also measured the number of GPUs that were queued (i.e. not used as producer GPUs). Our measurements show that the number of GPUs in the `Queued` set is relatively small (see Table 2). We did not find a single instance of a soundness alert that was valid; we did find a very small number of false positives that were manually examined and rejected.

- (d) FICI variant is consistently faster than the FICS variant, and faster than FSCS in most programs. Further, FSCS is faster than FICS in most cases.

10.4 Discussion: Lessons From Our Empirical Measurements

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

- (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 learnt these lessons the hard way in the situations described in the rest of this section.

10.4.1 Handling Recursion. In our first attempt of handling recursion, we converted indirect recursion to self recursion, and repeatedly inlined the recursive calls to optimize them. This failed because in some cases, the size of GPG after inlining calls became too big and our analyses and optimizations did not scale. Hence, instead of first creating a naively large GPG and then optimizing it to bring down the size, we decided to keep the GPGs small at every stage by successive refinements of mutually recursive GPGs starting from Δ_{\top} .

10.4.2 Handling Large Size of Context-Dependent Information. Some GPGs had a large amount of context-dependent information (i.e. GPUs with upwards-exposed versions of variables) and the GPGs could not be optimized much. This caused the size of the caller GPGs to grow significantly, threatening the scalability of our analysis. Hence we devised a heuristic threshold beyond which the procedure summary will be inlined as a symbolic Δ_{\top} GPG with an additional feature that it carries with it in a single GPB, all context-dependent GPUs (i.e., the GPUs that have upwards-exposed versions of variables after optimizations). This keeps the size of the caller GPG small and at the same time, allows reduction of the context-dependent GPUs. 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.

Note that, in our implementation, we discovered very few cases (and only in large benchmarks) where the threshold actually exceeded.²² 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.

10.4.3 Handling Function Pointers. Function pointers used in a procedure but defined in its callers is another case where we had to inline unoptimized GPGs in the callers because the GPGs of the procedure's callees were not known and hence their flow function was Δ_{\top} . This hampered scalability. Since our primary goal was to evaluate the effectiveness of our optimizations, our current implementation handles only locally defined function pointers (Section 9) Our implementation can be easily extended to handle function pointers defined in the calling contexts. We can handle such function pointers by using a symbolic Δ_{\top} GPG and introducing a small touch of top-down analysis as was done above when handling a large number of context-dependent GPUs. We leave this as future work.

²²We 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.

10.4.4 Handling Arrays and SSA Form. Pointers to arrays were weakly updated, hence we realized early on that maintaining this information flow sensitively prohibited scalability. This was particularly true for large arrays with static initializations. Similarly, GPUs involving SSA versions of variables were not required to be maintained flow sensitively. This allowed us to reduce the propagation of data across control flow without any loss in precision.

10.4.5 Making Coalescing More Effective. Unlike dead GPU elimination, coalescing proved to be a very significant optimization for boosting the scalability of the analysis. The points-to analysis failed to scale in the absence of this optimization. However, this optimization was effective (i.e. coalesced many GPBs) only when we brought in the concept of types. In cases where the data dependence between the GPUs was unknown because of the dependency on the context information, we used type-based non-aliasing to enable coalescing.

10.4.6 Estimating the Number of Context-Dependent Summaries. Constructing context-dependent procedure summaries (i.e. partial transfer functions) using the aliases or points-to information from calling contexts obviates the need of control flow. Since control flow is the real bottleneck as per our findings, we computed the number of aliases after computing the final points-to information to estimate the number of context-dependent summaries that may be required for real program. This number (column *F* in Table 1) is large suggesting that it is undesirable to construct multiple PTFs for a procedure using the aliases from the calling contexts.

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 [9, 10, 12, 16, 26, 29]. 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.

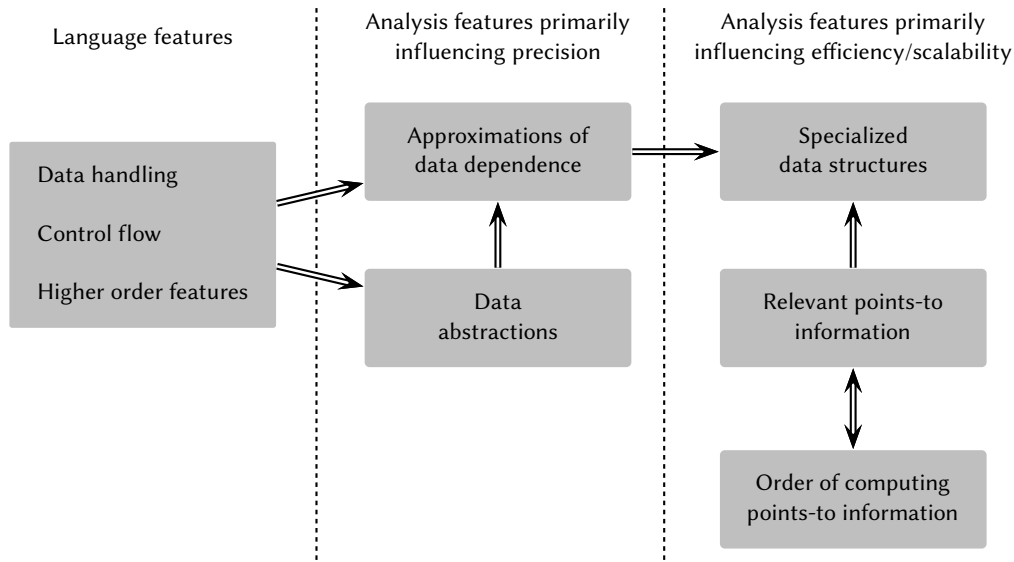
11.1 Factors Influencing the Precision, Efficiency, and Scalability of Points-to Analysis

Figure 30 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 over-approximate the control flow: flow-insensitivity effectively creates a complete graph out of a control flow graph whereas context-insensitivity treats call and returns as simple goto statements as far as the control transfer between procedures is concerned.

Observe that control flow in imperative languages is a proxy for implicit data dependence. As a consequence, an over-approximation of control flow amounts to over-approximation of data dependence. In other words, control flow over-approximation may introduce spurious data dependences between pointer assignments that may have not existed if the analysis respected the control flow. This causes imprecision.



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, <i>eval</i> 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. 30. 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.

Note that SSA form also discards control flow but it avoids over-approximation in data dependences by creating use-def chains in the form of SSA edges.

- *Data abstractions.* An abstract location usually represents a set of concrete locations. An over-approximation of this set of locations leads to spurious data dependences 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 which is then used to compute further points-to information. For example, staged points-to analyses begin with conservative points-to information which is then made more precise. Similarly, some methods begin by computing points-to information for top-level pointers whose indirections are then eliminated. This uncovers a different set of pointers as top-level pointers whose points-to information is then computed.
- *Order of computing points-to information.* Most methods order computations based on relevant points-to information which may also be defined in terms of a chosen order of traversal over the call graph (eg. top-down or bottom-up).
- *Specialized data structures.* A method may use specialized data structures for encoding information efficiently (e.g. BDDs or GPUs and GPGs) or may use them for modelling relevant points-to information (e.g. use of placeholders to model accesses of unknown pointees in a bottom-up method).

11.1.3 *Interaction between the Features.* In this section we explain the interaction between the features indicated by the arrows in Figure 30.

- *Data abstraction influences approximations of data dependence.* An abstract location may be over-approximated to represent a larger set of concrete locations in many situations such as in field-insensitivity, type-based abstraction, allocation site-based abstraction. This over-approximation 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 [35], 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 [14] only the live pointers are relevant. Thus, points-to information is computed only when the liveness of pointers is generated. Thus, the generation of liveness information influences the relevant points-to information to be computed.

11.1.4 *Our Work in the Context of 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 top-level local variables. Unlike the approaches that over-approximate control flow indiscriminately, we discard control flow as much as possible but only when there is a guarantee that it does not over-approximate data dependence.

Our analysis is field-sensitive. It over-approximates arrays by treating all its elements alike. We use allocation-site-based abstraction for representing heap locations and use k -limiting for summarizing the unbounded accesses of heap where allocation sites are not known.

Like every bottom-up approach, points-to information is computed when all the information is available in the context. Our analysis computes points-to information for all pointers.

11.2 Approaches of Constructing Procedure Summaries

We restrict our description of related work to bottom-up approaches. We begin with the two broad categories of approaches introduced in Section 2.3.

11.2.1 MTF Approach. In this approach [11, 32, 35, 36], control flow is not required to be recorded between memory updates. This is because the data dependency between memory updates (even the ones which 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.

Level-by-level analysis [35] 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 extended SSA form. When used in conjunction with conditions based on points-to information from calling contexts, the chains become context sensitive.

The scalability of these approaches depends on the number of aliases/points-to pairs in the calling contexts, which could be large. Thus, this approach may not be useful for constructing summaries for library functions which have to be analyzed without the benefit of different calling contexts. Saturn [7] creates sound summaries but they may not be precise across applications because of their dependence on context information.

Relevant context inference [3] constructs a procedure summary by inferring the relevant potential aliasing between unknown pointees that are accessed in the procedure. Although, it does not use the information from the context, it has multiple versions of the summary depending on the alias and the type context. This analysis could be inefficient if the inferred possibilities of aliases and types do not actually occur in the program. It also over-approximates the alias and the type context as an optimization thereby being only partially context-sensitive.

11.2.2 STF Approach. This approach does not make any assumptions about the calling contexts [17, 18, 24, 30, 31] but constructs large procedure summaries causing inefficiency in fixed-point computation at the intraprocedural level. It introduces separate placeholders for every distinct access of a pointee (Section 2.3). Also, the data dependence is not known in the case of indirect accesses of unknown pointees and hence control flow is required for constructing the summary for a flow-sensitive points-to analysis. However, these methods do not record control flow between memory updates in the summaries so constructed. Thus, in order to ensure soundness, the procedure summaries do not assume any ordering between the memory updates and are effectively applied flow-insensitively even though they are constructed flow-sensitively. This introduces imprecision by prohibiting killing of points-to information. However, it may not have much adverse impact on programs written in Java because all local variables in Java have SSA versions, thanks to the absence of indirect assignments to variables (there is no addressof operator). Besides, there are few static variables in Java programs and absence of kill for them may not matter much; the points-to relations of heap locations are not killed in any case.

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

11.2.3 The Hybrid Approach. Hybrid approaches use customized summaries and combine the top-down and bottom-up analyses to construct summaries [36]. This choice is controlled by the number of times a procedure is called. If this number exceeds a fixed threshold, a summary is constructed using the information of the calling contexts that have been recorded for that procedure. A new calling context may lead to generating a new precondition and hence a new summary. If the threshold is set to zero, then a summary is constructed for every procedure and hence we have a pure bottom-up approach. If the threshold is set to a very large number, then we have a pure top-down approach and no procedure summary is constructed.

Additionally, we can set a threshold on the size of procedure summary or the percentage of context-dependent information in the summary or a combination of these choices. In our implementation, we have used the percentage of context-dependent information as a threshold—when a procedure has a significant amount of context-dependent information, it is better to introduce a small touch of top-down analysis (Section 10.4.2). If this threshold is set to 0%, our method becomes purely bottom-up approach; if it is set to 100%, our method becomes a top-down approach.

12 CONCLUSIONS 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.

The first issue has been handled by modelling accesses of unknown pointees using placeholders. However, it may require a large number of placeholders. The second issue has been handled by constructing multiple versions of a procedure summary for different aliases in the calling contexts. The third issue can only be handled by inlining the summaries of the callees. However, it can increase the size of a summary exponentially thereby hampering the scalability of analysis.

We have handled the first issue by proposing the concept of generalized points-to updates (GPUs) which track indirection levels. Simple arithmetic on indirection levels allows composition of GPUs to create new GPUs with smaller indirection levels; this reduces them progressively to classical points-to edges.

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

In order to achieve the above, we have devised novel data flow analyses such as reaching GPUs analysis (with and without blocking) and coalescing analysis which is a bidirectional analysis. Interleaved call inlining and strength reduction of GPGs facilitated a novel optimization that computes flow- and context-sensitive points-to information in the first phase of a bottom-up approach. This obviates the need for the second phase.

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. Two important takeaways from our empirical evaluation are:

- (a) Flow- and context-sensitive points-to information is small and sparse.
- (b) The real killer of scalability in program analysis is not the amount of data 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. Besides, it would be interesting to explore the possibility of restricting GPG construction to live pointer variables [14] 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 modelling 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. This direction can also be explored in future.

ACKNOWLEDGMENTS

Pritam Gharat is partially supported by a TCS Research Fellowship.

REFERENCES

- [1] Alfred V. Aho, Monica S. Lam, Ravi Sethi, and Jeffrey D. Ullman. 2006. *Compilers: Principles, Techniques, and Tools (2Nd Edition)*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA.
- [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, USA, 1–3. <https://doi.org/10.1145/503272.503274>
- [3] 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, USA, 133–146. <https://doi.org/10.1145/292540.292554>
- [4] 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, USA. <https://doi.org/10.1145/1375581.1375615>
- [5] Yu Feng, Xinyu Wang, Isil Dillig, and Thomas Dillig. 2015. Bottom-Up Context-Sensitive Pointer Analysis for Java. In *Programming Languages and Systems - 13th Asian Symposium, APLAS 2015, Pohang, South Korea, November 30 - December 2, 2015, Proceedings*. https://doi.org/10.1007/978-3-319-26529-2_25
- [6] 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)*. Springer-Verlag, Berlin, Heidelberg.
- [7] 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, USA. <https://doi.org/10.1145/1181775.1181785>
- [8] 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, USA. <https://doi.org/10.1145/378795.378802>
- [9] Michael Hind and Anthony Pioli. 1998. Assessing the Effects of Flow-Sensitivity on Pointer Alias Analyses. In *Static Analysis, 5th International Symposium, SAS '98, Pisa, Italy, September 14-16, 1998, Proceedings*. 57–81. https://doi.org/10.1007/3-540-49727-7_4
- [10] 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, USA, 113–123. <https://doi.org/10.1145/347324.348916>
- [11] 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, USA, 249–259. <https://doi.org/10.1145/1375581.1375613>
- [12] Vini Kanvar and Uday P. Khedker. 2016. Heap Abstractions for Static Analysis. *ACM Comput. Surv.* 49, 2, Article 29 (June 2016), 47 pages. <https://doi.org/10.1145/2931098>
- [13] 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 17th international conference on Compiler construction (CC'08/ETAPS'08)*.
- [14] 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)*. Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-33125-1_19
- [15] U. P. Khedker, A. Sanyal, and B. Sathe. 2009. *Data Flow Analysis: Theory and Practice*. Taylor & Francis (CRC Press, Inc.), Boca Raton, FL, USA.

- [16] Ondrej Lhotak, Yannis Smaragdakis, and Manu Sridharan. 2013. Pointer Analysis (Dagstuhl Seminar 13162). *Dagstuhl Reports* 3, 4 (2013), 91–113. <https://doi.org/10.4230/DagRep.3.4.91>
- [17] 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, USA. <https://doi.org/10.1145/2464157.2466483>
- [18] 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)*. Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-33125-1_25
- [19] Erik M. Nystrom, Hong-Seok Kim, and Wen-mei W. Hwu. 2004. Bottom-Up and Top-Down Context-Sensitive Summary-Based Pointer Analysis. In *Static Analysis, 11th International Symposium, SAS 2004, Verona, Italy, August 26-28, 2004, Proceedings*. https://doi.org/10.1007/978-3-540-27864-1_14
- [20] 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, USA. <https://doi.org/10.1145/2487568.2487569>
- [21] 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, USA. <https://doi.org/10.1145/199448.199462>
- [22] 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, The Netherlands. <http://dl.acm.org/citation.cfm?id=243753.243762>
- [23] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. 1998. Solving Shape-analysis Problems in Languages with Destructive Updating. *ACM Trans. Program. Lang. Syst.* 20, 1 (Jan. 1998), 1–50. <https://doi.org/10.1145/271510.271517>
- [24] Lei Shang, Xinwei Xie, and Jingling Xue. 2012. On-demand Dynamic Summary-based Points-to Analysis. In *Proceedings of the Tenth International Symposium on Code Generation and Optimization (CGO '12)*. ACM, New York, NY, USA. <https://doi.org/10.1145/2259016.2259050>
- [25] A. Sharir M., Pnueli. 1981. Two approaches to interprocedural data flow analysis. *S.S., Jones, N.D. (eds.) Program Flow Analysis: Theory and Applications, (ch. 7)* (1981).
- [26] Yannis Smaragdakis and George Balatsouras. 2015. Pointer Analysis. *Foundations and Trends in Programming Languages* 2, 1 (2015), 1–69. <https://doi.org/10.1561/2500000014>
- [27] Johannes Späth, Lisa Nguyen, Karim Ali, and Eric Bodden. 2016. Boomerang: Demand-Driven Flow- and Context-Sensitive Pointer Analysis for Java. In *European Conference on Object-Oriented Programming (ECOOP)*.
- [28] Manu Sridharan, Denis Gopan, Lexin Shan, and Rastislav Bodik. 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, USA. <https://doi.org/10.1145/1094811.1094817>
- [29] Stefan Staiger-Stöhr. 2013. Practical Integrated Analysis of Pointers, Dataflow and Control Flow. *ACM Trans. Program. Lang. Syst.* 35, 1 (2013), 5:1–5:48. <https://doi.org/10.1145/2450136.2450140>
- [30] 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)*. Springer-Verlag, Berlin, Heidelberg. https://doi.org/10.1007/978-3-540-30579-8_14
- [31] 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, USA. <https://doi.org/10.1145/320384.320400>
- [32] 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>
- [33] 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, USA. <https://doi.org/10.1145/2259051.2259053>
- [34] 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, USA. <https://doi.org/10.1145/1328438.1328467>
- [35] 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, USA, 218–229. <https://doi.org/10.1145/1772954.1772985>
- [36] 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, USA. <https://doi.org/10.1145/2594291.2594328>