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

For this purpose, we propose a novel abstraction called the Generalized Points-to Graph (GPG) which views points-to relations as memory updates and generalizes them using the counts of indirection levels leaving the unknown pointees implicit. This allows us to construct GPGs as compact representations of bottom-up procedure summaries in terms of memory updates and control flow between them. Their compactness is ensured by strength reduction (which reduces the indirection levels), control flow minimization (which removes control flow edges while preserving soundness and precision), and call inlining (which enhances the opportunities of these optimizations).

The effectiveness of GPGs lies in the fact that they discard as much control flow as possible without losing precision. This is the reason 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 to represent and transform memory in the presence of pointers. Future investigations can try to combine it with other abstractions for static analyses that can benefit from points-to information.
intensive analyses such as model checking are noted as being ineffective on programs containing pointers, partly because of imprecision of points-to analysis [2].

1.1 The Context of this Work

We focus on exhaustive as against demand-driven [7, 13, 37, 38] points-to analysis. A demand-driven points-to analysis computes points-to information that is relevant to a query raised by a client analysis; for a different query, the points-to analysis needs to be repeated. An exhaustive analysis, 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 [48] 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 [35], its value-based variants [20, 30] and the tabulation-based functional method [31, 35]. By contrast, bottom-up approaches [5, 9, 12, 16, 26, 33, 35, 40, 43–48] 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. A flow- and context-sensitive interprocedural analysis using procedure summaries is performed in two phases: the first phase constructs the procedure summaries and the second phase uses them to represent the effect of the calls at the call sites.

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

1.2 Our Contributions

Most approaches to precise points-to analysis begin with a scalable but imprecise method and then seek to increase its precision. We take the opposite approach in that we begin with a precise method and increase its scalability. We create naive, possibly non-scalable, procedure summaries and then use novel optimizations to compact them while retaining their soundness and precision. More specifically, we advocate a new form of bottom-up procedure summaries, called the generalized points-to graphs (GPGs) for flow- and context-sensitive points-to analysis. GPGs represent memory transformers (summarizing the effect of a procedure) and contain GPUs (generalized points-to

---

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. At the interprocedural level, a forward data flow analysis (e.g. available expressions analysis) could be top-down or a bottom-up and so can be a backward data flow analysis (e.g. live variables analysis).
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), denoted \( \gamma \), leaves accesses of unknown pointees implicit without losing precision.

2. GPGs undergo aggressive optimizations that are applied repeatedly to improve the compactness of GPGs incrementally. They are governed by the following possibilities of data dependence of a GPU \( \gamma_2 \) on another GPU \( \gamma_1 \) (illustrated in Section 2.2).

   - **Case A.** The dependence of \( \gamma_2 \) on \( \gamma_1 \) can be determined. Then, there are two possibilities:
     - (i) GPU \( \gamma_2 \) follows \( \gamma_1 \) on some control flow path and has the following kind of dependence on \( \gamma_1 \): (a) a read-after-write (RaW) dependence, (b) a write-after-read (WaR) dependence, or (c) a write-after-write (WaW) dependence. A read-after-read (RaR) dependence is irrelevant.
     - (ii) GPU \( \gamma_2 \) does not have a dependence on \( \gamma_1 \).

   - **Case B.** More information is needed to determine whether or not \( \gamma_2 \) has a dependence on \( \gamma_1 \). Then, \( \gamma_2 \) has a potential dependence on \( \gamma_1 \). We use source-language type information ubiquitously to rule out potential dependence following C-style rules on indirect accesses via type-casted pointers. This resolves some instances of case B into case A.ii.

   These cases are exploited by three classes of optimizations as described below.

   - **Elimination of data dependence.** These optimizations attempt to eliminate data dependences between GPUs so that the control flow can be minimized. The strength reduction optimization exploits the RaW dependence (case A.i.a) of GPU \( \gamma_2 \) on GPU \( \gamma_1 \). It simplifies GPU \( \gamma_2 \) by reducing the pointer indirection levels in it by using the pointer information from \( \gamma_1 \) to eliminate the data dependence between them. The dead GPU elimination optimization exploits the WaW dependence (case A.i.b) between GPU \( \gamma_1 \) and \( \gamma_2 \). If the locations written by \( \gamma_1 \) are rewritten by \( \gamma_2 \) along every path reaching the end of the procedure, \( \gamma_1 \) does not have any effect on the callers. Deleting it eliminates the WaW dependence between \( \gamma_1 \) and \( \gamma_2 \).

   - **Control flow minimization.** These optimizations exploit the WaR dependence (case A.i.c) and the absence of data dependence (case A.i.i). They discard control flow selectively by converting some sequentially ordered GPUs into parallel GPUs when there is WaR dependence or no dependence between them—since all reads precede any write in parallel assignments, they preserve WaR dependences inherently. When there are RaW or WaW dependences between GPUs, we preserve control flow between them.

   - **Call inlining.** This optimization handles case B by progressively providing more information. It inlines the summaries of the callees of a procedure enhancing the opportunities of strength reduction and control flow optimization and enabling context-sensitive analyses. Recursive calls are handled by refining the GPGs through a fixed-point computation. Calls through function pointers are handled through delayed inlining.

Our measurements suggest that the real killer of scalability in program analysis is not the amount of data but the amount of control flow that the data propagation may be subjected to in search of precision. Our optimizations are effective because they eliminate data dependence wherever possible and discard irrelevant control flow. This aids the scalability of points-to analysis without violating soundness or causing imprecision.

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

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

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

The core ideas of GPGs have been presented before [11]. This 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. 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. Section 5 explains dead GPU elimination. Section 6 describes control flow minimization optimizations performed on GPGs. Section 7 explains the interprocedural use of GPGs by defining call inlining and shows how recursion is handled. Section 8 shows how GPGs are used for performing points-to analysis. Section 9 proves soundness and precision of our method by showing its equivalence with a top-down flow- and context-sensitive classical points-to analysis. Section 10 presents empirical evaluation on SPEC benchmarks and Section 11 describes related work. Section 12 concludes the paper.

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

### 2 EXISTING APPROACHES AND THEIR LIMITATIONS

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

#### 2.1 Basic Concepts

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

**Abstract and Concrete Memory.** There are two views of memory and operations on it. Firstly we have the concrete memory view corresponding to run-time operations where a memory location representing a pointer always points to exactly one memory location or NULL (which is a distinguished memory location). Unfortunately this is, in general, statically uncomputable. 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 \( L_p \subseteq L \) denote the sets of locations\(^2\) and pointers respectively. The **concrete memory** after a pointer assignment is a function \( M : L_p \rightarrow L \). The **abstract memory** after a pointer assignment is a relation \( M \subseteq L_p \times L \). In either case, we view \( M \) as a graph with \( L \) as the set of nodes. An edge \( x \rightarrow y \) in \( M \) is a **points-to edge** indicating that \( x \in L_p \) contains the address of \( y \in L \). The abstract memory associated with a statement is an over-approximation of the concrete memory associated with every occurrence of

\(^2\)https://github.com/PritamMG/GPG-based-Points-to-Analysis.

\(^3\)Here we talk about non-heap locations. Heap locations are handled as explained in the electronic appendix.
the statement in the same or different control flow paths. Unless noted explicitly, all subsequent references to memory and its transformations refer to the abstract view.

2.1.2 Memory Transformer. A procedure summary for points-to analysis should represent memory updates in terms of copying locations, loading from locations, or storing to locations. We call it a memory transformer because it computes the memory after a call to a procedure based on the memory before the call. Given a memory $M$ and a memory transformer $\Delta$, the updated memory $M'$ is computed by $M' = \Delta(M)$ as illustrated in Example 2 (Section 2.3).

2.1.3 Strong and Weak Updates. In concrete memory, every assignment overwrites the contents of the (single) memory location corresponding to the LHS of the assignment. However, in abstract memory, we may be uncertain as to which of several locations a variable (say $p$) points to. Hence an indirect assignment such as $*p = &x$ does not overwrite any of these locations but merely adds $x$ to their possible pointees. This is a weak update. Sometimes however, there is only one possible abstract location described by the LHS of an assignment, and in this case we may, in general, replace the contents of this location. This is a strong update. There is just one subtlety 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 (i.e., live on entry to the current procedure) and therefore may have additional pointees unknown to the current procedure. Thus, the criterion for a strong update in an assignment is that its LHS references a single location and the location referenced is not upwards exposed (for more details, see Section 4.4). A direct assignment to a variable (e.g. $p = &x$) is special case of a strong update.

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

2.2 Challenges in Constructing Procedure Summaries for Points-to Analysis

In the absence of indirect assignments involving pointers, data dependence between memory updates within a procedure can be inferred by using variable names without requiring any information from the callers. In such a situation, procedure summaries for some analyses, including various bit-vector data flow analyses (such as live variables analysis), can be precisely represented by constant gen and kill sets [1, 22] or graph paths discovered using reachability [31].

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. For many calls, procedure $Q()$ on the right simply returns &a but until we are certain that $*p$ does not alias with $x$, we cannot perform this constant-propagation optimization; assignment 04 blocks it. If it is known that $*p$ and $x$ always alias then we can optimize $Q$ to return &b (WaW dependence of statement 04 on statement 03 and RaW dependence of statement 05 on statement 04). If it is known that they never alias we can optimize this code to return &a (no dependence between statements 04 and 03 but RaW dependence of statement 05 on statement 03). If nothing is known about the alias information, then we must retain assignment 04 in the procedure.
Memory transformer $\Delta'$ is compact but imprecise because it uses the same placeholder for every access of a pointee. Thus it over-approximates the memory.

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

![Procedure Q](image1)

![Example 1](image2)

![Example 2](image3)

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

summary for $Q$ (potential dependence of statement 04 on statement 03 and of statement 05 on statements 04 and 03). The key idea is that information from the calling context(s) can determine whether a potentially blocking assignment really blocks an optimization or not.

The 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, a flow-sensitive memory transformer for points-to analysis requires a compact representation for memory updates that captures the minimal control flow between them succinctly.
2.3 Limitations of Existing Procedure Summaries for Points-to Analysis

A common solution for modelling indirect accesses of unknown pointees in a memory transformer is to use placeholders (also known as external variables [26, 40, 43] and extended parameters [44]). They are pattern-matched against the input memory to compute the output memory. Here we describe two broad approaches that use placeholders.

The first approach, which we call a multiple transfer functions (MTFs) approach, proposed a precise representation of a procedure summary for points-to analysis as a collection of (conditional) partial transfer functions (PTFs) [5, 16, 44, 47]. Each PTF corresponds to a combination of aliases that might occur in the callers of a procedure. Our work is inspired by the second approach, which we call a single transfer function (STF) approach [4, 6, 23, 26, 27, 40, 43]. 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 (A’ and A’’) for it and the associated input and output memories. The effect of A’ is explained in Example 2 and that of A’’, in Example 3.

Example 2. Transformer A’ in Figure 1 is constructed by the STF approach. It is an abstract points-to graph containing placeholders \( \phi_1 \) for modelling unknown pointees. For example, \( \phi_1 \) represents the pointees of \( y \) and \( \phi_2 \) represents the pointees of pointees of \( y \). Note that a memory is a snapshot of points-to edges whereas a memory transformer needs to distinguish the points-to edges that are generated by it (shown by thick edges) from those that are carried forward from the input memory (shown by thin edges).

The two accesses of \( y \) in statements 1 and 3 may or may not refer to the same location because of a possible side-effect of the intervening assignment in statement 2. If \( x \) and \( y \) are aliased in the input memory (e.g. in \( M_2 \)), statement 2 redefines the pointee of \( y \) and hence \( p \) and \( q \) will not be aliased in the output memory. However, A’ uses the same placeholder for all accesses of a pointee. Further, A’ also suppresses strong updates because the control flow between memory updates is not recorded. Hence, points-to edge \( s \rightarrow c \) in \( M_1' \) is not deleted. Similarly, points-to edge \( r \rightarrow a \) in \( M_2' \) is not deleted and \( q \) spuriously points to \( a \). Additionally, \( p \) spuriously points-to \( b \). Hence, \( p \) and \( q \) appear to be aliased in the output memory \( M_2' \).

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

Example 3. In Figure 1, memory transformer A’’ differs from A’ in two ways. First, it uses a separate placeholder for every access of a pointee to avoid an over-approximation of memory (e.g. placeholders \( \phi_1 \) and \( \phi_2 \) to represent \( *y \) in statement 1, and \( \phi_3 \) and \( \phi_6 \) 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 \( M_2'' \)). Second, the points-to edges generated by the memory transformer are ordered based on the control flow of a procedure, thereby adding some form of flow-sensitivity which A’ lacks. To see the role of control flow, observe that if the points-to edge corresponding to statement 2 is considered first, then \( p \) and \( q \) will always be aliased because the possible side-effect of statement 2 will be ignored.

The output memories \( M_1'' \) and \( M_2'' \) computed using A’’ are more precise than the corresponding output memories \( M_1' \) and \( M_2' \) computed using A’.

Observe that, although A’’ is more precise than A’, it uses a larger number of placeholders and also requires control flow information. This affects the scalability of points-to analysis.
A fundamental problem with placeholders is that they use a low-level representation of memory expressed in terms of classical points-to edges. Hence a placeholder-based approach is forced to explicate unknown pointees by naming them, resulting in either a large number of placeholders (in the STF approach) or multiple PTFs (in the MTF approach). The need of control flow ordering further increases the number of placeholders in the former approach.

2.4 Our Key Ideas

We propose a 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. Interestingly, it can be compressed dramatically without losing precision and can be optimized into a compact acyclic flow relation in most cases, even if the procedure it represents has loops or recursive calls.

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

3 THE 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). We provide an overview of our the ideas and algorithms in a limited setting of our motivating example of Figure 2. Towards the end of this section, Figure 6 summarizes them as a collection of abstractions, operations, data flow analyses, and optimizations.

3.1 Defining a Generalized Points-to Graph (GPG)

We model the effect of a pointer assignment on an abstract memory by defining the concept of generalized points-to update (GPU) in Definition 1 which gives the abstract semantics of a GPU. The concrete semantics of a GPU \( x \leftarrow y \) can be viewed as the following C-style pointer assignment with \( i - 1 \) dereferences of \( x \) (or \( i \) dereferences of &\( x \)) and \( j \) dereferences of &\( y \).

\[
s : \ast \ldots \ast x = \ast \ldots \ast &y \\
\text{((i - 1))}
\]

Given variables $x$ and $y$ and $i > 0$, $j \geq 0$, a generalized points-to update (GPU) $x \xrightarrow{ij} y$ 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 $ij$ represents indirection levels and is called the indlev of the GPU ($i$ is the indlev of $x$, and $j$ is the indlev of $y$). The pair $(x, i)$ is called the source and the pair $(y, j)$ is called the target of the GPU. The letter $\gamma$ is used to denote a GPU unless named otherwise.

**Definition 1. Generalized Points-to Update.**

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

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

Statement labels, $s$, in GPUs are unique across procedures to distinguish between the statements of different procedures after call inlining. They facilitate distinguishing between strong and weak updates by identifying may-defined pointers (Section 3.3.1). Further, since GPUs are simplified in the calling contexts, statement labels allow back-annotation of points-to information within the procedure that they belong to. For simplicity, we omit the statement labels from GPUs when they are not required.

A GPU $\gamma : x \xrightarrow{ij} y$ generalizes a points-to edge\(^5\) from $x$ to $y$ with the following properties:

- The direction indicates that the source $x$ with indlev $i$ identifies the locations being defined and the target $y$ with indlev $j$ identifies the locations whose addresses are read. We often refer to $(x, i)$ as the source of $\gamma$ and $(y, j)$ as its target.
- The GPU $\gamma$ abstracts away $i - 1 + j$ placeholders.
- The GPU $\gamma$ represents may information because different locations may be reached from $x$ and $y$ along different control flow paths reaching statement $s$ in the procedure.

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

**Example 4.** The pointer assignment in statement 01 in Figure 2 is represented by a GPU $r \xrightarrow{10} a$ where the indication 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 $q \xrightarrow{20} m$. 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.

---

\(^4\) Section 3.3.1 explains how this transformation is effectively reversed when transliterating intermediate code instructions for the ’Initial GPG’.

\(^5\) Although a GPU is a generalization of a points-to edge, we reserve the term edge for a ’control flow edge’ in a GPG.
A generalized points-to block (GPB), denoted $\delta$, is a set of GPUs abstracting memory updates. A generalized points-to graph (GPG) of a procedure, denoted $\Delta$, is a graph $(N, E)$ whose edges in $E$ abstract the control flow of the procedure. Nodes $n$ in $N$ are labelled both with GPBs, $\delta_n$, and with may-definition sets, $\mu_n$, containing GPU sources that are may defined. The latter set is empty initially, but is populated during GPG optimizations.

Finally, we use standard terminology from CFGs for GPGs: distinguished Start and End nodes, the notions of control flow path, dominance, the function $\text{pred}$ giving the control flow predecessors (and its transitively closed version $\text{pred}^+$) and similarly $\text{succ}$ and $\text{succ}^+$ for successors. By common abuse of notation, we often conflate nodes and their GPB labelings.

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

Figure 3 presents the GPUs for basic pointer assignments in C and for the general GPU $x \xrightarrow{l_1\to} y$. (To deal with C structs and unions, GPUs are extended to encode lists of field names—for details see Figure B.1 in Appendix B).

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

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

- The CFG could have procedure calls but an optimized GPG does not. We observe that an optimized GPG is acyclic in almost all cases, even if a procedure has loops or recursive calls.
- The GPBs 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.

3.1.1 Abstract Semantics of GPBs. Abstract semantics of GPBs is a generalization of the semantics of pointer assignment in two ways. The first generalization is from a pointer assignment to a GPU and the second generalization is from a single statement to multiple statements.
The semantics of a GPU (Definition 1) forms the basis of the semantics of a GPB. However, since a GPB has no control flow ordering on its GPUs and may contain multiple (simplified forms of) GPUs for a single source-language statement, or GPUs for multiple statements, we need to specify the combined effect of these multiple GPUs. In particular, differing concrete runs may execute a only a subset of the GPUs in some order. Let \( \delta \) be a GPB, \( \mu \) be its associated may-definition set and let \( S \) be the set of source-language labels \( s \) occurring as labels of GPUs in \( \delta \). Now write \( \delta|_s \) for \( \{x \xrightarrow{\gamma} y \in \delta\} \). The abstract execution of \( \delta \) is characterized by the following two features:

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

In the simplest case, when the GPUs in \( \delta|_s \) define multiple sources, all sources are included in \( \mu \)—since each concrete execution of statement \( s \) defines only one source, there is a concrete run for every source that does not define the source. In other cases, the source of a GPU \( y \in \delta|_s \) may be included in \( \mu \) if there is concrete run of \( \delta \) that does not execute statement \( s \).

**Example 5.** Consider a GPB \( \delta = \{y_1:x \xrightarrow{10/11} a, y_2:x \xrightarrow{10/11} b, y_3:y \xrightarrow{10/12} c, y_4:z \xrightarrow{10/13} d, y_5:t \xrightarrow{10/13} d, \} \) and its associated may-definition set \( \mu = \{(z,1),(t,1)\} \) because \( y_4 \) and \( y_5 \) correspond to a single statement (statement 13) but define multiple sources. Note that \( y_1 \) and \( y_2 \) also correspond to a single statement (statement 11) but they define a single source (\( x,1 \)). Then, after executing \( \delta \) abstractly we know that the points-to set of \( x \) is overwritten to become \( \{a,b\} \) (i.e. \( x \) definitely points to one of \( a \) and \( b \)). Similarly, the points-to set of \( y \) is overwritten to become \( \{c\} \) because \( y_5 \) defines a single location \( c \) in statement 12. However, \( \delta \) causes the points-to sets of \( z \) and \( t \) to include \( \{d\} \) (without removing the existing pointees) because their sources are members of \( \mu \). Thus, \( x \) and \( y \) are strongly updated (their previous pointees are removed) but \( z \) and \( t \) are weakly updated (their previous pointees are augmented).

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

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

- **WaW**: \( (w = x \land k = i) \)
- **WaR**: \( (w = x \land k < i) \lor (w = y \land k \leq j) \)
- **RaW**: \( (w = x \land k > i) \lor (z = x \land l \geq i) \)

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

We call these dependences as *definite* dependences. They correspond to case A.ii in Section 1.2. Also, if \( \gamma_2 \) post-dominates \( \gamma_1 \) (i.e., follows \( \gamma_1 \) on every control flow path) we call the dependence *strict*. As illustrated in Example 1, \( \gamma_1 \) and \( \gamma_2 \) can have a dependence even when they do not have a common variable. Such a dependence is called a *potential* dependence (case B in Section 1.2).

Two GPUs on a control flow path cannot be placed within a single GPB if there is a definite or potential RaW or WaW dependence between them. However it is safe to include them in the case of WaR dependence because of the ‘all reads precede all writes’ semantics of GPBs.
Example 6. Consider the code snippet on the right. There is a WaR data dependence between statements 01 and 02. If the control flow was simply ignored, the statements could be executed in the reverse order causing $y$ to erroneously point to $a$. We construct a GPB $\{y \xrightarrow{\gamma 11} x, x \xrightarrow{\gamma 10} a\}$ for the code snippet. Since all reads precede any write, the execution of this GPB in the context of the memory represented by GPU $x \xrightarrow{\gamma 12} b$, computes the points-to information $\{y \rightarrow b, x \rightarrow a\}$ and excludes $y \rightarrow a$ thereby preserving the WaR dependence.

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

3.2 An Overview of GPG Operations

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

3.2.1 GPU Composition. In a compiler, the sequence $p = &a; *p = x$ is usually simplified to $p = &a; a = x$ to facilitate further optimizations. Similarly, the sequence $p = &a; q = p$ is usually simplified to $p = &a; q = &a$. GPU composition facilitates similar simplifications: Suppose a GPU $\gamma_1$ precedes $\gamma_2$ on some control flow path. If $\gamma_2$ has a RaW dependence on $\gamma_1$, then, $\gamma_2$ is a consumer of the pointer information represented by the producer $\gamma_1$. In such a situation a GPU composition $\gamma_3 = \gamma_2 \circ \gamma_1$ computes a new GPU $\gamma_3$ such that the indlev of $\gamma_3$ (say $i$|$j$) does not exceed that of $\gamma_2$ (say $i'$|$j'$), i.e. $i \leq i'$ and $j \leq j'$. The two GPUs $\gamma_2$ and $\gamma_3$ are equivalent in the context of GPU $\gamma_1$ and, while we might prefer $\gamma_3$, we cannot delete $\gamma_2$ until we have considered all control flow paths (see Section 3.2.2 below). GPU composition is a partial function—either succeeding with a simplified GPU or signaling failure. A comparison of indlevs allows us to determine whether a composition is possible; if so, simple arithmetic on indlevs computes the indlev of the resulting GPU.

Example 7. For statement sequence $p = &a; *p = x$, the consumer GPU $\gamma_2 : p \xrightarrow{\gamma 21} x$ (statement 2) is simplified to $\gamma_3 : a \xrightarrow{\gamma 31} x$ by replacing the source $p$ of $\gamma_2$ using the producer GPU $\gamma_1 : p \xrightarrow{\gamma 10} a$ (statement 1). GPU $\gamma_3$ 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 $\gamma_3$) become available.

The above example illustrates that multiple GPU compositions may be required to reduce the indlev of a GPU to convert it to an equivalent GPU with indlev 1|0 (a classical points-to edge).

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

GPU reduction $\gamma \circ \mathcal{R}$ eliminates the RaW data dependence of $\gamma$ on the GPUs in $\mathcal{R}$, wherever possible, thereby eliminating the need for control flow between $\gamma$ and the GPUs in $\mathcal{R}$.
3.3 An Overview of GPG Construction
The GPG of procedure $R$ (denoted $\Delta_R$) is constructed by traversing a spanning tree of the call graph starting with its leaf nodes. It involves the following steps:

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

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

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 by treating the former as empty statements and the latter as non-deterministic control flow.
- Each pointer assignment, labelled $s$, in the CFG is transliterated to a GPU $x \rightarrow y$, following Figure 3. If earlier compiler stages have broken compound C assignments such as $**p=***q$ into a sequence of simpler SSA-form intermediate-language assignments using temporaries as in Equation (1), then compound statements are reconstructed by following def-use chains to eliminate such temporaries.
- A GPG node $n$ is created for each such assignment with its GPB, $\delta_n$, being the singleton set containing this GPU and with its associated may-definition set, $\mu_n$, being empty.
- The control flow between GPBs is induced from their order within a basic block in the CFG and from the control flow edges of the CFG.
- The procedure calls are replaced by the optimized GPGs of the callees. Every time we inline a GPG, we must take a fresh copy of its nodes, here achieved by simple renumbering. Note that the statement labels $s$ appearing within GPUs are not renumbered.

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

Examples 9 to 12 explain the analyses and optimizations over $\Delta_Q$ and $\Delta_R$ (GPGs for procedures $Q$ and $R$) at an intuitive level.

3.3.2 Strength Reduction Optimization. This step simplifies GPB $\delta_n$ for each node $n$ by

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

Effectively, strength reduction simplifies each GPB as much as possible without needing the knowledge of aliasing in the caller. In the process, data dependences are eliminated to the extent possible, facilitating dead GPU elimination and control flow minimization. Note that strength reduction does not create new GPBs; it only creates new (equivalent) GPUs within the same GPB. The statement labels in GPUs remain unchanged because the simplified GPUs of a statement continue to represent the same statement.

In order to reduce the **indlews** 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 computes sets of GPUs. It identifies the simplified GPUs for a GPB in the context of the GPUs reaching the GPB. By construction, all resulting GPUs are equivalent to the original GPUs of the GPB and have indirection levels that do not exceed that of the original GPUs. This process requires a fixed-point computation in the presence of loops. Since this step follows inlining of GPGs of callee procedures, procedure calls have already been eliminated and the analysis is intraprocedural.6

Two issues in reaching GPUs analysis that are not illustrated in this section are:

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

**Example 9.** We intuitively explain the reaching GPUs analysis for procedure \( Q \) over its initial GPG (Figure 4). The final result is shown later in Figure 8. GPU \( r \xrightarrow{10} a \) representing statement 01 reaches \( \delta_{02} \) in the first iteration. However, it does not simplify GPU \( q \xrightarrow{20} m \) in \( \delta_{02} \). The GPUs \( \{r \xrightarrow{10} a, q \xrightarrow{20} m\} \) reach the GPB \( \delta_{03} \). GPU \( q \xrightarrow{10} b \) cannot be simplified any further. In the second

---

6In the presence of recursion and function pointers, the effect of calls gets progressively refined through repeated analyses of a procedure and its callees but the analysis still remains intraprocedural (see Section 7.2 and Appendix C).
### Generalized Points-to Graphs: A Precise and Scalable Abstraction for Points-to Analysis

<table>
<thead>
<tr>
<th>CFG</th>
<th>Initial GPG $\Delta_R$</th>
<th>$\Delta_R$ after strength reduction</th>
<th>$\Delta_R$ after control flow elimination</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="CFG Diagram" /></td>
<td><img src="image" alt="Initial GPG Diagram" /></td>
<td><img src="image" alt="Strength Reduction" /></td>
<td><img src="image" alt="Control Flow Elimination" /></td>
</tr>
</tbody>
</table>

#### 3.3.3 Dead GPU Elimination

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

**Example 10.** In procedure $Q$ of Figure 4, pointer $q$ is defined in $\delta_{03}$ but is redefined in $\delta_{05}$ and hence GPU $q \xrightarrow{\delta_{03}} b$ is eliminated. Hence GPB $\delta_{03}$ becomes empty and is removed from $\Delta_Q$.
Since GPU $q_{02}$ does not define $q$ but its pointee, it is not killed by $\delta_{05}$ and is not eliminated from $\Delta_Q$.

For procedure $R$ in Figure 5, GPU $q_{07}$ is killed by GPU $q_{05}$ in $\delta_{14}$. Hence GPU $q_{07}$ is eliminated from GPB $\delta_{07}$. Similarly, GPU $e_{04}$ in GPB $\delta_{14}$ is removed because $e$ is redefined by GPU $e_{04}$ in GPB $\delta_{10}$ (after strength reduction in $\Delta_R$). However, GPU $d_{08}$ in GPB $\delta_{08}$ is not removed even though $\delta_{13}$ contains a definition of $d$ expressed GPU $d_{02}$ in $\Delta_R$. This is because $\delta_{13}$ also contains GPU $b_{02}$ which defines $b$. Since statement $02$ defines two sources, both of them are may-defined in $\delta_{13}$ (i.e., are included in $\mu_{13}$). Thus the previous definition of $d$ cannot be killed—giving a weak update.

**3.3.4 Control Flow Minimization.** This step improves the compactness of a GPG by eliminating empty GPBs from a GPG and then minimizing control flow by coalescing adjacent GPBs into a single GPB wherever there is no RaW or WaW dependences between them.
Example 11. After eliminating GPU \( q \xrightarrow{10}{07} d \) from the GPG of procedure \( R \) in Figure 5 (because it is dead), GPB \( \delta_{07} \) becomes empty and is removed from the optimized GPG.

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

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

Example 12. For procedure \( Q \) in Figure 4, the GPBs \( \delta_1 \) and \( \delta_2 \) can be coalesced: there is no data dependence between their GPUs because GPU \( r \xrightarrow{10}{01} a \) in \( \delta_1 \) defines \( r \) whose type is ‘\( \text{int} \star \star \)’ whereas the GPUs in \( \delta_2 \) read the address of \( m \), pointer \( b \), and pointee of \( q \). The type of latter two is ‘\( \text{int} \star \)’. Thus, a potential dependence between the GPUs in \( \delta_1 \) and \( \delta_2 \) is ruled out using types. However, GPUs \( q \xrightarrow{20}{02} m \) in \( \delta_2 \) and \( e \xrightarrow{12}{04} p \) in \( \delta_4 \) have a potential RaW dependence (\( p \) and \( q \) could be aliased in the caller) which is not ruled out by type information. Thus, we do not coalesce GPBs \( \delta_2 \) and \( \delta_4 \). Since there is no RaW dependence between the GPUs in the GPBs \( \delta_4 \) and \( \delta_5 \) we coalesce them (potential WaW dependence does not matter because all reads precede any write).

The GPB resulting from coalescing GPBs \( \delta_1 \) and \( \delta_2 \) is labelled \( \delta_{11} \). Similarly, \( \delta_{12} \) is the result of coalescing GPBs \( \delta_4 \) and \( \delta_5 \). The loop formed by the back edge \( \delta_2 \rightarrow \delta_1 \) in the GPG before coalescing now becomes a self loop over \( \delta_{11} \). Since, by definition, the GPUs in a GPB can never have a dependence between each other, the self loop \( \delta_{11} \rightarrow \delta_{11} \) is redundant and is hence removed.

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

As shown in Example 10, the GPUs \( b \xrightarrow{10}{02} m \) and \( q \xrightarrow{20}{02} m \) in procedure \( Q \) cause inclusion of the sources \((b, 1)\) and \((q, 2)\) in \( \mu_{02} \) leading further to their inclusion in \( \mu_{11} \) for the coalesced GPB \( \delta_{11} \). Similarly, for procedure \( R \), \((b, 1)\) is may-defined in GPB \( \delta_{15} \) but not \((d, 1)\) because the latter is defined along all paths through procedure \( R \) but not the former as shown Figure 5.

3.4 The Big Picture

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

4 STRENGTH REDUCTION OPTIMIZATION

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

Strength reduction optimization uses the knowledge of a producer GPU $p$, to simplify a consumer GPU $c$ (on a control flow path from $p$) through an operation called GPU composition denoted $c \circ p$ (Section 4.2). A consumer GPU may require multiple GPU compositions to reduce it to an equivalent GPU with $\text{indlev}\ 1\{0\}$ (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 GPBs. It is obtained by performing a data flow analysis called the *reaching GPUs analysis* (Sections 4.5, and 4.6) which computes the sets $\text{RGIn}_n, \text{RGOut}_n, \text{RGGen}_n, \text{and RGKill}_n$ for every GPB $\delta_n$. These data flow variables represent the GPUs reaching the entry of GPB $\delta_n$, its exit, the GPUs obtained through GPU reduction, and the GPUs whose propagation is killed by $\delta_n$, respectively. The set $\text{RGGen}_n$ is semantically equivalent to $\delta_n$ in the context of $\text{RGIn}_n$ and may beneficially replace $\delta_n$.

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

---

#### A Syntactic View of GPU Composition

- **A generic illustration of TS composition**
  
  $p : x \xrightarrow{k|l} y$
  
  $c : z \xrightarrow{i|j} x \Rightarrow r : z \xrightarrow{l|(l+k)} y$

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

- **A generic illustration of SS composition**
  
  $p : x \xrightarrow{k|l} y$
  
  $c : x \xrightarrow{i|j} z \Rightarrow r : y \xrightarrow{(l+k)|j} z$

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

#### A Semantic View of GPU Composition

- **TS composition in the memory**
  
  $p: x \xrightarrow{k|l} y$
  
  $c: z \xrightarrow{i|j} x$
  
  $r: z \xrightarrow{i|(l+k)} y$

  where, $l \leq k \leq j$

  The effect of TS composition in the memory

- **SS composition in the memory**
  
  $p: x \xrightarrow{k|l} y$
  
  $c: x \xrightarrow{i|j} z$
  
  $r: y \xrightarrow{(l+k)|j} z$

  where, $l \leq k < i$

  The effect of SS composition in the memory

---

Fig. 7. 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 labelled $t$ on the statement labelled $s$ because the pointer defined in $p$ is used to simplify a pointer used in $c$. 

---

illustrates these compositions. For a composition, consider $c: z \xrightarrow{[l]} x$ and $p: x \xrightarrow{[k]} y$ with pivot $x$ which is the target of $c$ and the source of $p$. The goal of the 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 $x \xrightarrow{[l+j-k]} y$. This balances the indlevs of $x$ in the two GPUs allowing us to create a simplified GPU $r: z \xrightarrow{[l+j-k]} y$.

### 4.2.2 Defining GPU Composition

Before we define the GPU composition formally, we need to establish the properties of validity and desirability that allow us to characterize meaningful GPU compositions. We say that a GPU composition is admissible if and only if it is valid and desirable.

(a) A composition $r = c \circ T p$ is valid only if $c$ has a RaW dependence on $p$ through the pivot of the composition.

(b) A composition $r = c \circ T p$ 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 7, this requirement translates to the following constraints:

$$j \geq k \quad \text{(TS composition)}$$
$$i > k \quad \text{(SS composition)}$$

---

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

---

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

The desirability of GPU composition characterizes progress in conversion of GPUs into classical points-to edges by ensuring that the $indlev$ of the new source and the new target in $r$ does not exceed the corresponding $indlev$ in the consumer GPU $c$. This requires the $indlev$ in the simplified GPU $r$ and the consumer GPU $c$ to satisfy the following constraints. In each constraint, the first term in the conjunct compares the $indlev$s of the sources of $c$ and $r$ while the second term compares those of the targets (see Figure 7):

\[
\begin{align*}
(i \leq i) \land (l + j - k \leq j) & \quad \text{or equivalently } l \leq k \quad (TS \text{ composition}) \quad (4) \\
(l + i - k \leq i) \land (j \leq j) & \quad \text{or equivalently } l \leq k \quad (SS \text{ composition}) \quad (5)
\end{align*}
\]

**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 2. 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 4.

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

\[
\begin{align*}
l \leq k & \leq j \quad (TS \text{ composition}) \quad (6) \\
l \leq k < i & \quad (SS \text{ composition}) \quad (7)
\end{align*}
\]

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 $r = c \circ_r p$ by balancing the $indlev$ of the pivot in both the GPUs provided the composition ($TS$ or $SS$) is admissible. Otherwise it fails—being a partial operation.

### 4.3 GPU Reduction

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

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

Output: Red’ // The set of simplified GPUs equivalent to c

01 GPU_reduction (Red, \( \mathcal{R} \)) // One step in fixed point computation
02 { Red’ = \( \emptyset \)
03 for each \( y_1 \in \text{Red} \)
04 { temp = \( \emptyset \) // The set of simplified GPUs equivalent to \( y_1 \)
05 for each \( y_2 \in \mathcal{R} \)
06 { if \( (r = y_1 \circ_{TS} y_2) \) succeeds then temp = temp \cup \{ r \}
07 else if \( (r = y_1 \circ_{SS} y_2) \) succeeds then temp = temp \cup \{ r \}
08 }
09 if temp \neq \emptyset then Red’ = Red’ \cup temp
10 else Red’ = Red’ \cup \{ y_1 \}
11 }
12 return Red’
13 }

Definition 4. GPU Reduction \( c \circ \mathcal{R} \)

Example 14. Consider \( c \circ x \xrightarrow{1/2} y \) with \( \mathcal{R} = \{ y \xrightarrow{10/21} a, a \xrightarrow{10/22} b \} \). The reduction \( c \circ \mathcal{R} \) involves two consecutive TS compositions. The first step with \( y \xrightarrow{10/21} a \) as pivot, computes \( \text{Red}’ = \{ x \xrightarrow{11/23} a \} \). Then, the reduced GPU \( x \xrightarrow{11/23} a \) becomes the consumer GPU and is composed with \( a \xrightarrow{10/22} b \) from \( \mathcal{R} \) which results in \( \text{Red}’ = \{ x \xrightarrow{10/23} b \} \). It cannot be reduced further as it is already in the classical points-to form and the computation has reached the fixed point.

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

- **Reduced form.** GPUs in \( \mathcal{R} \) must be in their reduced form: Consider the GPU \( x \xrightarrow{i/j} y \). Then,
  - either \( i = 1 \) or \( x \) is live on entry (represented by a special variable \( x’ \)), and
  - either \( j = 0 \) or \( y \) is live on entry (represented by a special variable \( y’ \)).

The special variables are explained in Section 4.4.

- **Acyclicity.** The graph induced by the GPUs in \( \mathcal{R} \) should be acyclic: \( \mathcal{R} \) cannot have a subset like \( \{ x \xrightarrow{11/1} y, y \xrightarrow{11/1} x \} \). Taking the reduced form serves to replace this subset with \( \{ x \xrightarrow{11/1} y’, y \xrightarrow{11/1} x’ \} \) thereby ruling out the cycles.

- **Completeness.** If \( \mathcal{R} \) contains a GPU \( x \xrightarrow{i/j} y \), then the source \( (x, i) \) must be defined along all paths reaching the GPB that is undergoing reduction: if \( x \) is the pivot of composition with \( c \) but \( (x, i) \) is not defined along some path, it means that the simplification of \( c \) is not complete and \( \text{Red} \) cannot replace \( c \). This is ensured by the introduction of boundary definitions in Section 4.4. Example 21 in Section 4.6 illustrates why completeness of \( \mathcal{R} \) is required.

---

\(^8\)In the presence of structures, cycles may occur via fields of structures; Appendix B.4 shows how they are handled.
• **Absence of Data Dependence.** GPUs in $\mathcal{R}$ should not have a data dependence between them. This is preserved by reaching GPUs analyses (Sections 4.5, and 4.6): If GPU $\gamma_2 \in \mathcal{R}$ had a RaW dependence on GPU $\gamma_1 \in \mathcal{R}$, then $\gamma_1$ would have been simplified during reaching GPUs analysis; if $\gamma_2$ had a WaW dependence on $\gamma_1$ along a control flow path, $\gamma_2$ would be killed during reaching GPUs analysis along the path. Finally, if $\gamma_2$ had a potential dependence on $\gamma_1$, $\gamma_2$ would have been blocked and not included in $\mathcal{R}$.

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

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

• The indlevs in the GPUs in Red in step $i + 1$ are smaller than the indlevs in the GPUs in step $i$ and the number of GPUs is finite (Section 3.1.3). Since there are no cycles in $\mathcal{R}$, once a GPU $\gamma$ is simplified, further simplifications cannot recreate $\gamma$ again; hence there is no oscillation across the iterations of fixed-point computation, ensuring the termination of GPU reduction.

• The order in which GPU $\gamma_2$ is selected from $\mathcal{R}$ for composition with $\gamma_1$ does not matter because the GPUs in $\mathcal{R}$ do not have a data dependence between them.

### 4.4 Boundary Definitions

Recall that for an indirect assignment ($*p = &x$ say) as a consumer, GPU reduction typically returns a set of GPUs which define multiple abstract locations leading to a weak update. Sometimes however, we may discover that $p$ has a single pointee within the procedure and the assignment defines only one abstract location. In this case we may, in general, perform a strong update. However, this condition, while necessary, is not sufficient for a strong update because the source of $p$ may not be defined along all paths—there may be a path along which the source of $p$ is not defined within the procedure (i.e., is live on entry to the procedure) and is defined in a caller. In the presence of such a definition-free path in a procedure, even if we find a single pointee of $p$ in the procedure, we cannot guarantee that a single abstract location is being defined. This makes it difficult to distinguish between strong and weak updates.

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

• no node $n_i$, $1 \leq i \leq k$, kills (Definition 5) or blocks (Definition 7) GPUs with source $(x, i)$,

• node $n_1$ is either Start or has a predecessor that kills or blocks $(x, i)$, and

• node $n_k$ is either End or has a successor that kills or blocks $(x, i)$.

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

• in RGIn of Start for global variables and formal parameters, and

• in RGOut of the nodes that block some GPUs for the sources of the blocked GPUs.

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

The boundary definitions are of the form $x \downarrow_{\downarrow} x'$ where $x'$ is a symbolic representation of the initial value of $x$ at the start of the procedure and $i$ ranges from 1 to the maximum depth of the indirection level which depends on the type of $x$ (e.g., for type (int **), $i$ ranges from 1 to 2). Variable $x'$ is called the **upwards-exposed** [22] version of $x$. This is similar to Hoare-logic style specifications in which postconditions use (immutable) auxiliary variables $x'$ to denote the original value of variable $x$ (which may have since changed). Our upwards-exposed versions serve a similar purpose; logically on entry to each procedure the statement $x = x'$ provides a definition of $x$. The rationale behind the label 0 in the boundary definitions is explained after the following example.
Variables can appear in simplified GPUs. The labels of producer GPUs are irrelevant because they only provide information that is used for simplifying consumer GPUs labelled \( s \) into one or more GPUs all labelled \( s \). The boundary definitions participate in GPU reduction algorithm (without requiring any change in GPU composition) like any other producer GPU. After GPU reduction, upwards-exposed versions of variables can appear in simplified GPUs.

**4.5 Reaching GPUs Analysis without Blocking**

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

---

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

- GPUs \( p \xrightarrow{1|0} q \) and \( p \xrightarrow{1|1} p' \) reach \( \delta_n \). Then \( \delta_n \) will be replaced by \( \text{Red} \) containing both \( q \xrightarrow{1|0} a \) and \( p' \xrightarrow{2|0} a \). Thus sources (\( q, 1 \)) and (\( p', 2 \)) are included in \( \mu_n \) causing a weak update (because both of them are defined by the same source).
- Only \( p \xrightarrow{1|0} q \) reaches \( \delta_n \). Then, \( \text{Red} \) contains only \( q \xrightarrow{1|0} a \) causing a strong update because statement \( s \) defines a single source (\( q, 1 \)).
- Only \( p \xrightarrow{1|1} p' \) reaches \( \delta_n \). Then, \( \text{Red} \) contains only \( p' \xrightarrow{2|0} a \). Since \( p' \) could have multiple pointees in the callers, we perform a weak update.

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

---

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

\[
\begin{align*}
\text{RGIn}_n & := \begin{cases} 
\{ x \xrightarrow{ij} x' \mid x \in L_p, 0 < i \leq \kappa \} & n = \text{Start}, \kappa \text{ is the largest indlev} \\
\bigcup_{m \in \text{pred}(n)} \text{RGOut}_m & \quad \text{otherwise}
\end{cases} \\
\text{RGOut}_n & := (\text{RGIn}_n - \text{RGKill}_n) \cup \text{RGGen}_n \\
\text{RGGen}_n & := \text{Gen}(\delta_n, \text{RGIn}_n) \\
\text{RGKill}_n & := \text{Kill}(\text{RGGen}_n, \text{RGIn}_n) \\
\text{Gen}(X, \mathcal{R}) & := \bigcup_{\gamma \in X} \gamma \circ \mathcal{R} \\
\text{Kill}(X, \mathcal{R}) & := \bigcup_{y \in \mathcal{R}} \text{Match}(y, \mathcal{R}) \\
\text{Match}(x \xrightarrow{ij} y, \mathcal{R}) & := \{ y \in \mathcal{R} \mid y = w \xrightarrow{k|l} z, x = w, i = k, -\text{Upex}(x), (x, i) \notin \mu_n \} \\
\text{Def}(X, w \xrightarrow{k|l} z) & := \{ (x, i) \mid x \xrightarrow{ij} y \in \mathcal{R} \} \\
\text{Upex}(x) & := \begin{cases} \text{true} & x \text{ is an upwards-exposed version of some variable} \\
\text{false} & \text{otherwise}
\end{cases}
\end{align*}
\]
The reaching GPUs analysis is an intra-procedural forward data flow analysis in the spirit of the classical reaching definitions analysis. Its data flow equations are presented in Definition 5. They compute set RGln_n of GPUs reaching a given GPB δ_n by combining the GPUs in RGOut_m of the predecessor GPBs δ_m. RGln_n is then used to reduce the GPUs in δ_n to compute RGGen_n which is semantically equivalent to δ_n (except for the effect of blocking) but with the additional property that the indlevs of GPUs in RGGen_n do not exceed those of the corresponding GPUs in δ_n.

RGKill_n contains the GPUs that are to be excluded from RGOut_n because of a strong update. A GPU in γ′ from RGln_n is included in RGKill_n if its source (x, i) matches the source of a reduced GPU γ ∈ δ|_s ⊆ RGGen_n corresponding to some statement s (identified by Match(γ, RGln_n)) provided:

1. All GPUs in δ|_s define the same source (x, i). Condition |Def(X, γ)| = 1 for Kill(X, R) in Definition 5 ensure this where Def(X, γ) extracts the sources of GPUs of the same statement.
2. Variable x in (x, i) is not an upwards-exposed version z′ because then i > 1 and z′ could point to multiple pointees in the caller (note that an upwards-exposed version can appear in the source of a reduced GPU only if the GPU represents an indirect assignment).
3. Source (x, i) is not in µ_n.

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

Example 16. Figure 8 gives the final result of reaching GPUs analysis for procedure Q of our motivating example. We have shown the boundary GPU q 11 by 10 o 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 ΔQ shown in Figure 4. For procedure R, we do not show the complete result of the analysis but make some observations. The GPU q 10 by 10 e is composed with the GPU q 10 by 10 o to create a reduced GPU e 10 by 10 o. Since, only a single pointer e is being defined by the assignment and source (e, 1) is not may-defined (i.e. not in µ_0), this is a strong update and hence kills e 10 by 04 c.

The GPU to be killed is identified by Match(e 10 by 10 o, RGln_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 10 by 10 o) and not the consumer GPU (in this case q 10 by 10 o).

4.6 Reaching GPUs Analysis with Blocking

This section extends the reaching GPUs analysis to incorporate the effect of blocking by defining a data flow analysis that computes RGln and RGOut for the purpose.

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

Since this potential dependence between p and b cannot be resolved without the alias information in the calling context, we block such producer GPUs so that such GPU compositions leading to potentially unsound strength reduction optimization are postponed. Wherever possible, we use the type information to rule out some GPUs as barriers. After inlining the GPG in a caller, more information may become available. Thus, it may resolve potential data dependence of a barrier.
We show only one boundary definition \( q_1 \rightarrow 00 \) in \( \text{RGIn} \) and \( \text{RGOut} \) because other boundary definitions do not participate in GPU reduction for this example.

However, the boundary definitions that are removed are shown in \( \text{RGKill} \).

Fig. 8. The data flow information computed by reaching GPUs analysis for procedure \( Q \) of Figure 2.

with a producer. Then, if a consumer still has a RaW dependence on a producer, the composition which was earlier postponed may now safely be performed.

Example 17. Consider the procedure in Figure 9(a). The composition of the GPUs for statements 02 and 04 is admissible. However, statement 03 may cause a side-effect by indirectly defining \( y \) (if \( x \) points to \( y \) in the calling context). Thus, \( q \) in statement 04 would point to \( b \) if \( x \) points to \( y \); otherwise it would point to \( a \). If we replace the GPU \( q_1 \rightarrow 04 \) by \( q_1 \rightarrow 04 \) (which is the result...
Consider a GPU $p$ originally blocked by a barrier $b$. After inlining the GPG in its callers and performing reductions in the calling contexts, the following situations could arise:

1. The `indlev` of the source of the indirect GPU ($p$ or $b$) is reduced to 1 thereby eliminating the potential dependence. In this case, $b$ ceases being a barrier and so no longer blocks $p$ leading to the following two situations:
   a. $b$ has a WaW dependence on $p$ and therefore redefines the pointer defined by $p$, killing $p$ thereby obviating the composition $c \circ \tau p$.
   b. $b$ does not have a dependence on $p$ 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.

Example 18. Case 1(a) above could arise if $x$ points to $p$ in the calling context of the procedure in Figure 9(a). As a result, GPU $y \xrightarrow{\frac{10}{02}} a$ is killed by the barrier GPU $y \xrightarrow{\frac{10}{03}} b$ (which is the simplified version of the barrier GPU $x \xrightarrow{\frac{20}{03}} b$) 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 $y$ in the calling context.

In this case, the composition between $q \xrightarrow{\frac{10}{04}} y$ and $y \xrightarrow{\frac{10}{02}} a$ 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 $x \xrightarrow{\frac{20}{03}} b$ continues to block $y \xrightarrow{\frac{10}{02}} a$.

Our measurements (Section 10) show that situation 1(a) rarely arises in practice because it amounts to defining the same pointer multiple times through different aliases in the same context.
Generalized Points-to Graphs: A Precise and Scalable Abstraction for Points-to Analysis

Example 19. To see how reaching GPUs analysis with blocking helps, consider the example in Figure 9(b). The set of GPUs reaching the statement 04 is \( \text{RGIn}_{04} = \{ x_{202}^0 \rightarrow a, y_{103}^1 \rightarrow b \} \). The GPU \( x_{202}^0 \rightarrow a \) is blocked by the barrier GPU \( y_{103}^1 \rightarrow b \) and hence \( \overline{\text{RGIn}}_{04} = \{ y_{103}^1 \rightarrow b \} \). Thus, GPU reduction for \( y_1 : q_{04}^1 \rightarrow x \) (in the context of \( \overline{\text{RGIn}}_{04} \)) computes Red as \( \{ y_1 \} \) because \( y_1 \) cannot be reduced further within the GPG of the procedure. However, \( y_1 \) is still not a points-to edge and can be simplified further after the GPG is inlined in its callers. Hence we postpone the composition of \( y_1 \) with \( p : x_{202}^0 \rightarrow a \) until \( p \) has been simplified.

4.6.2 Data Flow Equations for Computing \( \overline{\text{RGIn}} \) and \( \overline{\text{RGOut}} \). The following GPUs should be blocked as barriers:

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

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

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

\[ \overline{\text{DDep}}(B, I) \iff \text{TDef}(B) \cap (\text{TDef}(I) \cup \overline{\text{TRef}}(I)) \neq \emptyset \]

\[ \text{TDef}(X) := \{ \text{typeof}(x, i) \mid x \xrightarrow{i/j} y \in X \} \]

\[ \text{TRef}(X) := \{ \text{typeof}(x, k) \mid 1 \leq k < i, x \xrightarrow{i/j} y \in X \} \cup \{ \text{typeof}(y, k) \mid 1 \leq k < j, x \xrightarrow{i/j} y \in X \} \]

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

Definition 6. Blocking

\[ \overline{\text{RGIn}}_n := \begin{cases} \{ x \xrightarrow{i/j} x' \mid x \in L, 0 < i \leq \kappa \} & n = \text{Start}, \kappa \text{ is the largest indlev} \\ \bigcup_{m \in \text{pred}(n)} \overline{\text{RGOut}}_m & \text{otherwise} \end{cases} \]

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

\[ \overline{\text{RGGen}}_n := \text{Gen} (\delta_n, \overline{\text{RGIn}}_n) \]

\[ \overline{\text{RGKill}}_n := \text{Kill} (\overline{\text{RGGen}}_n, \overline{\text{RGIn}}_n) \cup \text{Blocked}(\overline{\text{RGIn}}_n, \overline{\text{RGGen}}_n) \]

Note: The definitions of Gen and Kill are same as in Definition 5.
• If $\overline{\text{RGGen}}_n$ does not contain an indirect GPU and is not $\emptyset$, then all indirect GPUs reaching $\delta_n$ that share a data dependence with a GPU in $\overline{\text{RGGen}}_n$ should be blocked.

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

The data flow equations in Definition 7 differ from those in Definition 5) as follows:

- $\overline{\text{RGKill}}_n$ additionally includes blocked GPUs computed using function $\text{Blocked}(I, G)$. The latter examines the GPUs in $\overline{\text{RGGen}}_n$ (argument “$G$” for generated) to identify the GPUs in $\overline{\text{RGIn}}_n$ (argument “$I$” for incoming) that should be blocked using three cases that exhaust all possibilities:
  - Case 1 corresponds to not blocking any GPU because $\overline{\text{RGGen}}_n$ is empty.
  - Case 2 corresponds to blocking some GPUs because $\overline{\text{RGGen}}_n$ contains an indirect GPU.
  - Case 3 corresponds to blocking indirect GPUs because $\overline{\text{RGGen}}_n$ does not contain an indirect GPU and is not $\emptyset$.

- $\overline{\text{RGOut}}_n$ explicitly introduces boundary definitions for the GPUs that are blocked. This is essential for ensuring the completeness of the set of reaching GPUs (Section 4.3) which is necessary for replacing $\delta_n$ by $\overline{\text{RGGen}}_n$. This is possible because of the following property of upwards-exposed versions: any read of $x$ anywhere in the procedure can be replaced by $x'$ without affecting soundness or precision because there is no GPU with $(x', i)$ as its source. Hence a statement $\ast x' = \&a$ does not have a RaW dependence on any statement and the occurrences of $x'$ cannot be simplified any further. After inlining in the caller, $x'$ is replaced by $x$ and hence the statement reverts to its original form.

**Example 20.** For the procedure in Figure 9(b), $\overline{\text{RGIn}}_{02} = \emptyset$ and $\overline{\text{RGGen}}_{02}$ is $\{x_{\text{202}}^{0} \mapsto a\}$. Although $\overline{\text{RGGen}}_{02}$ contains an indirect GPU, since no GPUs reach 02 (because it is the first statement), $\overline{\text{RGOut}}_{02}$ is $\{x_{\text{202}}^{0} \mapsto a\}$ indicating that no GPUs are blocked.

For statement 03, $\overline{\text{RGIn}}_{03} = \{x_{\text{203}}^{0} \mapsto a\}$ and $\overline{\text{RGGen}}_{03} = \{y_{\text{103}}^{0} \mapsto b\}$. $\overline{\text{RGGen}}_{03}$ is non-empty and does not contain an indirect GPU and thus $\overline{\text{RGOut}}_{03} = \{y_{\text{103}}^{0} \mapsto b\}$ according to the third case in the Blocked equation in Definition 7 indicating that the GPU $x_{\text{202}}^{0} \mapsto a$ is blocked and should not be used for composition by the later GPUs. The indirect GPU in $\overline{\text{RGIn}}_{03}$ is excluded from $\overline{\text{RGOut}}_{03}$. Note that the indirect GPU $x_{\text{202}}^{0} \mapsto a$ is blocked by the GPU $y_{\text{103}}^{0} \mapsto b$ because typeof($x$, 2) matches with typeof($p$, 1) indicating a possibility of WaW dependence.

For statement 04, $\overline{\text{RGIn}}_{04} = \{y_{\text{104}}^{0} \mapsto b\}$ and $\overline{\text{RGGen}}_{04} = \{q_{\text{104}}^{0} \mapsto x\}$. For this statement, the composition $(q_{\text{104}}^{0} \mapsto x \circ b \mapsto x_{\text{202}}^{0} \mapsto a)$ is postponed because the GPU $x_{\text{202}}^{0} \mapsto a$ is blocked. In this case, $\overline{\text{RGGen}}_{04}$ does not contain an indirect GPU and $\overline{\text{RGOut}}_{04} = \{y_{\text{103}}^{0} \mapsto b, q_{\text{104}}^{0} \mapsto x\}$. 
In Figure 9(a), the GPU \( y \xrightarrow{\delta_1}{0_2} a \) is blocked by the barrier GPU \( x \xrightarrow{\gamma_0}{0_3} b \) because typeof(y, 1) matches with typeof(x, 2). Hence, the composition \( (q \xrightarrow{\gamma_0}{0_3} y \xrightarrow{\delta_1}{0_2} a) \) is postponed.

In the GPG of procedure \( Q \) (of our motivating example) shown in Figure 4, the GPUs \( r \xrightarrow{\delta_1}{0_1} a \) and \( q \xrightarrow{\delta_0}{0_3} b \) are not blocked by the GPU \( q \xrightarrow{\gamma_0}{0_2} m \) because they have different types. However, the GPU \( e \xrightarrow{\gamma_0}{0_4} p \) blocks the indirect GPU \( q \xrightarrow{\gamma_0}{0_2} m \) because there is a possible WaW data dependence (\( e \) and \( q \) could be aliased in the callers of \( Q \)).

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

**Example 21.** Let the GPUs of the statements 1, 2, and 3 on the right be denoted by \( y_1, y_2, \) and \( y_3 \), respectively. Then, \( y_1 \) is blocked by \( y_2 \) because of potential RaW dependence (if \( p \) points-to \( x \) in the caller). Thus \( \text{RGIn}_3 = \{y_1, y_2\} \). Then the \( \text{RGGen}_3 = \{y \xrightarrow{\delta_0}{0_3} a\} \).

Replacing \( \delta_3 \) by \( \text{RGGen}_3 \) is unsound because if \( p \) points to \( x \) in the caller then \( y \) should also point to \( b \). The problem arises because \( \text{RGIn}_3 \) does not have the source \( (x, 1) \) defined along the path 1-2-3 because of blocking in node 2. This violates the completeness of \( \text{RGIn}_3 \). Explicitly adding the boundary definition \( x \xrightarrow{\gamma_0}{0_3} x' \) in 2 ensures that \( (x, 1) \) is defined along both the paths leading to \( \text{RGGen}_3 = \{y \xrightarrow{\delta_0}{0_3} a, y' \} \). When the resulting GPG is inlined in the caller, \( x' \) is replaced by \( x \) and the original consumer GPU \( c \) representing \( y = x \) is recovered. So, if \( p \) points to \( x \) then, after node 3, \( y \) points to both \( a \) and \( b \).

On the other hand, if \( p \) does not point to \( x \) then, after node 3, \( y \) points to \( a \) as expected.

### 5 DEAD GPU ELIMINATION

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

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

**Example 22.** In procedure \( Q \) of Figure 4, pointer \( q \) is defined in statement 03 but is redefined in statement 05 and hence the GPU \( q \xrightarrow{\delta_0}{0_3} b \) is killed and does not reach the End GPB. Since no composition with the GPU \( q \xrightarrow{\delta_0}{0_3} b \) is postponed, it does not belong to set Queued either. Hence the GPU \( q \xrightarrow{\delta_0}{0_3} b \) is eliminated from the GPB \( \delta_{03} \) as an instance of dead GPU elimination.
Similarly, the GPUs $q_{07}^{110}$ $d$ (in $δ_{07}$) and $e_{04}^{11}$ $c$ (in $δ_{14}$) in the GPG of procedure $R$ (Figure 5) are eliminated from their corresponding GPBs.

**Example 23.** For the procedure in Figure 9(a), the GPU $y_{02}^{110}$ $a$ is blocked by the barrier $x_{03}^{20}$ $b$; hence it is present in $R\text{GOut}_{05}$ but not in $R\text{GOut}_{05}$ (05 is the End GPB). This GPU may be required when the barrier $x_{03}^{20}$ $b$ is reduced after call inlining (and ceases to block $y_{02}^{110}$ $a$). Thus, it is not removed by dead GPU elimination.

To see the need of $R\text{GOut}_{\text{End}}$, observe that $q_{04}^{110}$ $y$ is reduced to $q_{04}^{10}$ $a$ in $R\text{GOut}_{\text{End}}$. Hence $q_{04}^{110}$ $y$ is not contained in $R\text{GOut}_{\text{End}}$. However, it cannot be removed as dead code. It is contained in $R\text{GOut}_{\text{End}}$ which should be additionally used for determining which GPUs are dead.

### 6 CONTROL FLOW MINIMIZATION

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

**Example 24.** In the GPG of procedure $Q$ of Figure 4, the GBP $δ_{03}$ becomes empty after dead GPU elimination. Hence, $δ_{03}$ can be removed by connecting its predecessors to successors. This transforms the back edge $δ_{03} \rightarrow δ_{01}$ to $δ_{02} \rightarrow δ_{01}$. Similarly, the GBP $δ_{07}$ is deleted from the GPG of procedure $R$ in Figure 5.

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

#### 6.1 The Motivation Behind Coalescing

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

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

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

**Example 25.** Let the GPUs of the statements 1, 2, and 3 on the right be denoted by $y_1$, $y_2$, and $y_3$, respectively. Then, $y_1$ cannot be coalesced with $y_2$ because of potential WaW dependence (if $p$ points-to $x$ in the caller). Similarly $y_2$ cannot be coalesced with $y_3$ because of potential WaW dependence (if $p$ points to $y$ in the caller). There is no data dependence between $y_1$ and $y_3$. However, they cannot be coalesced together because doing so will create GPBs $δ = \{y_1, y_3\}$ and $δ' = \{y_2\}$ with control flow edges $δ \rightarrow δ'$ and $δ' \rightarrow δ$ leading to spurious potential data dependences.

The next example illustrates that a non-strict WaW dependence does not constrain coalescing.
6.2 Creating a Coalesced GPG From a Partition

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

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

There is no ‘best’ coalescing operation: given three sequenced GPUs \( \gamma_1, \gamma_2, \gamma_3 \), then \( \gamma_1 \) may coalesce with \( \gamma_2 \) and separately \( \gamma_2 \) may coalesce with \( \gamma_3 \) but the GPUs \( \gamma_1, \gamma_2, \gamma_3 \) do not all coalesce.

Example 26. Let the GPUs of the statements 1, 2, and 3 on the right be denoted by \( \gamma_1, \gamma_2, \) and \( \gamma_3 \), respectively. The WaW dependence between \( \gamma_1 \) and \( \gamma_2 \) is definite but not strict and is not removed by dead GPU elimination because \( \gamma_1 \) is not killed along the path 1,3. Thus both \( \gamma_1 \) and \( \gamma_2 \) reach statement 3. Hence, although there is a WaW dependence between \( \gamma_1 \) and \( \gamma_2 \), they can be coalesced because the semantics of GPB allows both of them to the executed in parallel without any data dependence between them. This enables both of them to reach statement 3.

Example 27. Let the GPUs of the statements 1, 2, and 3 on the right be denoted by \( \gamma_1, \gamma_2, \) and \( \gamma_3 \), respectively. Let the type of pointers \( x \) and \( z \) be ‘int *’ and that of \( y \) be ‘float *’. Then there is no data dependence between \( \gamma_1 \) and \( \gamma_2 \) because \( x \) and \( y \) are guaranteed to point to different locations based on types. Similarly, there is no data dependence between \( \gamma_2 \) and \( \gamma_3 \). However, there is a potential data dependence between \( \gamma_1 \) and \( \gamma_3 \). Thus, \( \gamma_1 \) and \( \gamma_2 \) can be coalesced and so can \( \gamma_2 \) and \( \gamma_3 \); however, all three of them cannot be coalesced.
Given a GPG $\Delta$ and a partition $\Pi$ on its nodes, we obtain a coalesced GPG, written $\Delta/\Pi$, in the following steps:

1. The nodes of $\Delta/\Pi$ (written $\hat{n}$) are sets of the nodes of $\Delta$. More precisely they are the members of $\Pi$ and represent its equivalence classes.

The entry and exit nodes of a part $\hat{n}$ are defined as follows:

\[
\text{entry}(\hat{n}) = \{n \in \hat{n} \mid \exists n' \in \text{pred}(n), n' \notin \hat{n}\}
\]

\[
\text{exit}(\hat{n}) = \{n \in \hat{n} \mid \exists n' \in \text{succ}(n), n' \notin \hat{n}\}
\]

2. $\Delta/\Pi$ has an edge $\hat{n}_1 \rightarrow \hat{n}_2$ if $\hat{n}_1 \neq \hat{n}_2$ and $\exists n_1 \in \hat{n}_1, n_2 \in \hat{n}_2$ such that $n_1 \rightarrow n_2$ is an edge in $\Delta$.

3. The Start and End nodes of $\Delta/\Pi$ are respectively the parts containing the Start and End nodes of $\Delta$.

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

5. The may-definition set $\mu_\hat{n}$ of each node is computed using Definition 8 as follows:

- We identify the GPUs that are not modified in $\hat{n}$ by finding out the GPUs that reach the entry nodes of $\hat{n}$ (represented by the set inGPUs($\hat{n}$)) as well as the exit nodes of $\hat{n}$ (represented by the set outGPUs($\hat{n}$)) but are not generated within $\hat{n}$ (set $\delta_\hat{n}$).
- Set $\mu_\hat{n}$ contains the sources of the above GPUs (represented by the set preservedSources($\hat{n}$)) that are also defined within the GPB $\hat{n}$ (represented by the set definedSources($\hat{n}$)).

In essence we compute $\mu_\hat{n}$ in terms of $\mu_n$ ($n \in \hat{n}$).

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

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

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

### 6.3 What Is a Valid Coalescing Partition

A partition $\Pi$ is valid for coalescing to construct $\Delta/\Pi$ if it preserves the semantic understandings of $\Delta$. Validity is characterized by a set of conditions which ensures:

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

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

- **Soundness.**
\(\Delta\) \hspace{1cm} \Delta/\Pi\) with \(\mu_5 = \mu_6 = \mu_7 = \emptyset\)

\[
\begin{array}{c|c|c|c|c}
\text{Option A} & \text{Option B} & \text{Option C} & \text{Option D} \\
\hline
\begin{array}{c}
Y_1 \rightarrow a, b, *w, *x, *y, *z, **q; \\
y_1: x \mapsto a, y_2: z \mapsto q' \\
y_3: \mapsto z', y_4: y \mapsto b
\end{array} & Y_1 \rightarrow 5 & Y_1 \rightarrow 5 & Y_1 \rightarrow 5 \\
Y_2 \rightarrow 6 & Y_2, Y_3 \rightarrow 6 & Y_3 \rightarrow 6 & Y_2, Y_3, Y_4 \rightarrow 6 \\
Y_4 \rightarrow 7 & Y_4 \rightarrow 7 & Y_4 \rightarrow 7
\end{array}
\]

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

(S1) If there is a control flow path from \(n_1\) to \(n_2\) (\(n_2 \in \text{succ}^*(n_1)\)) and \(n_1\) and \(n_2\) are coalesced (\(n_2 \in \Pi(n_1)\)) then we require, for all GPUs \(y_1 \in \delta_{n_1}\) and \(y_2 \in \delta_{n_2}\), that \(y_1\) and \(y_2\) have no potential RaW dependence between them.

(S2) Consider a definition-free path \(\rho: x_1, n_2, \ldots, n_k\) for source \((x, i)\) in \(\Delta\). Then \(\Delta/\Pi\) must have a corresponding definition-free path \(\rho': \hat{n}_1, \hat{n}_2, \ldots, \hat{n}_l\) such that
- If \(n_1\) is Start, then \(\hat{n}_1 = \Pi(n_1)\); otherwise, \(\hat{n}_1 = \Pi(n_{j+1})\) such that first \(j\) nodes in \(\rho\) belong to \(\Pi(n_0)\) where \(n_0 \in \text{pred}(n_1)\) and \((x, i) \notin \mu_{n_0}\).
- If \(n_k\) is End, then \(\hat{n}_l = \Pi(n_k)\); otherwise, \(\hat{n}_l = \Pi(n_{l-1})\) such that last \(l\) nodes in \(\rho\) belong to \(\Pi(n_{k+1})\) where \(n_{k+1} \in \text{succ}(n_k)\) and \((x, i) \notin \mu_{n_{k+1}}\).

• Precision.

(P1) If there is a control flow path from \(n_1\) to \(n_2\) (\(n_2 \in \text{succ}^*(n_1)\)) and \(n_1\) and \(n_2\) are coalesced (\(n_2 \in \Pi(n_1)\)) then we require, for all GPUs \(y_1 \in \delta_{n_1}\) and \(y_2 \in \delta_{n_2}\), that \(y_1\) and \(y_2\) have no potential strict WaW dependence between them. Observe that definite strict WaW dependences have already been eliminated by dead GPU elimination.

(P2) For every \(\hat{n}\) in \(\Delta/\Pi\) that may-defines source \((x, i)\), there must be a control flow path \(n_1, n_2, \ldots, n_{k-1}, n_k\) in \(\Delta\) such that
- \(n_1\) belongs to a predecessor of \(\hat{n}\), \(n_k\) belongs to a successor of \(\hat{n}\), and
- all nodes from \(n_2\) to \(n_{k-1}\) belong to \(\hat{n}\) and may-defines source \((x, i)\).

(P3) For every control flow edge \(\hat{n}_1 \rightarrow \hat{n}_2\) in \(\Delta/\Pi\), \(\Delta\) must have a control flow path from every \(n_1 \in \hat{n}_1\) to every \(n_2 \in \hat{n}_2\).

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

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

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

Example 28. Consider the GPG in Figure 10 for coalescing and proposed partitions assuming that the may-definition sets are \(\emptyset\). GPU \(y_2\) has a potential RaW dependence on \(y_1\). Option A

violates both soundness and precision whereas options B and D violate soundness. Only option C satisfies all conditions.

- Option A violates condition (S1) by coalescing nodes 1 and 2—if q points to x in the caller, the RaW dependence of y₂ on y₁ is missed. It also violates condition (P3) by creating edge 2 → 3 which creates a RaW dependence of y₃ on y₂ after inlining in the caller (because z’ will be replaced by z). All other conditions are satisfied.
- Options B and D satisfy all conditions except (S2) because the definition-free paths for (w, 1) and (z, 1) are missed.

Two important characteristics of these conditions are:

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

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

6.4 Honoring the Validity Conditions

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

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

1. Given a part Π(n), a node n₁ \notin Π(n) is considered for inclusion in Π(n) only if some predecessor or some successor of n₁ is in Π(n). This ensures that every part Π(n) is a connected subgraph.⁹
2. Node n₁ is included in Π(n) only if there is no potential RaW or WaW dependences¹⁰ between the GPUs n₁ and those of any node n₂ \in Π(n) \cap pred⁺(n₁).
3. Definition-free paths are preserved by maintaining may-definition sets, with µ̃ containing the sources that are may-defined in ̃n.

Example 30. This example illustrates why step (2) above only considers the dependence between n₁ and n₂ \in (Π(n) \cap pred⁺(n₁)) rather than between n₁ and n₂ \in Π(n). In Figure 11(a) nodes n₁, n₂, and n₄ can be included in the same part. Consider node n₃ for inclusion in this part: The GPU in n₃ appears to have RaW dependence with the GPU in node n₄ because variable z’ will be replaced by z after inlining z’. However, there is no control flow from n₄ to n₃. Hence the data dependence of the GPU in n₃ need only be checked with those in n₁ and n₂ and not with those in n₄. Thus, n₃ can also be included in the same part. Similarly, although it appears that there is a WaW dependence between n₂ and n₅, the latter can also be included in the same part.

⁹Note that a part could be singleton too.
¹⁰We use a tighter condition and prohibit all potential WaW dependences and not just strict potential WaW dependences to avoid computing post-dominance information after inlining calls within a procedure.
6.4.1 Ensuring Precision. Define the external predecessors and successors of entry and exit nodes of a part $\hat{n}$ as follows:

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

We now wish to demand that, whenever $n_1$ and $n_2$ are entries of $\hat{n}$ then they have the same set of external predecessors and similarly for their exits and their external successors. Thus we define a partition $\Pi$ to be coherent if for all nodes $\hat{n} \in \Delta/\Pi$ we have

$$n_1, n_2 \in entry(\hat{n}) \Rightarrow xpred(n_1) = xpred(n_2) \land$$
$$n_1, n_2 \in exit(\hat{n}) \Rightarrow xsucc(n_1) = xsucc(n_2)$$

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

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

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

1. First, a data flow analysis (described in Section 6.4.4) identifies whether a node can be merged into a partition containing its predecessors and its successors. It honours the constraints described in Sections 6.4.1 and 6.4.2 except the constraint for coherence which is checked after the analysis (see step (2) below).
As a heuristic, the data flow analysis identifies partitions by accumulating nodes in a part in the "forward" direction. Consider a sequence of nodes $n_1, n_2, n_3$ such that we have control flow edges $n_1 \rightarrow n_2$ and $n_2 \rightarrow n_3$ such that there are two valid partitions: $\{ \{n_1, n_2\}, \{n_3\} \}$ and $\{ \{n_1\}, \{n_2, n_3\} \}$. Our algorithm constructs the first partition.

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

(3) Actual partitions are created.

6.4.4 Data Flow Analysis for Coalescing. We define two interdependent data flow analyses that

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

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

The data flow equations to compute $\text{ColIn}_n / \text{ColOut}_n$ are given in Definition 9. The initialization is $\text{false}$ for all GPBs. Predicate $\text{coalesce}(P, n)$ uses $\text{gpuFlow}(P, n)$ to check if GPUs reaching $P$ from

| $\text{ColIn}_n$ | $\text{false}$ if $n$ is Start  
| \( \bigvee_{m \in \text{pred}(n)} \text{coalesce}(m, n) \) otherwise |
| $\text{ColOut}_n$ | $\text{false}$ if $n$ is End  
| \( \bigvee_{m \in \text{succ}(n)} \text{ColIn}_m \) otherwise |
| $\text{coalesce}(m, n)$ | $\text{false}$ if $n$ is Start  
| $\bigvee_{m \in \text{pred}(n)} \text{gpuFlow}(m, n)$ otherwise |
| $\text{GpuIn}_n$ | $\emptyset$ if $n$ is Start  
| $\bigcup_{m \in \text{pred}(n)} \text{gpuFlow}(m, n)$ otherwise |
| $\text{GpuOut}_n$ | $\emptyset$ if $n$ is Start  
| $\bigcup_{m \in \text{pred}(n)} \text{gpuFlow}(m, n)$ otherwise |
| $\text{gpuFlow}(m, n)$ | $\emptyset$ if $n$ is Start  
| $\bigcup_{m \in \text{pred}(n)} \text{gpuFlow}(m, n)$ otherwise |
| $\text{DDep}(X, Y)$ | $\text{true}$ if $X$ and $Y$ are defined and used together  
| $\text{true}$ otherwise |
| $\text{deref}(X)$ | $\text{true}$ if $X$ is defined and dereferenced  
| $\text{false}$ otherwise |

Definition 9. Data flow equations for Coalescing Analysis.
```plaintext
int m, n, o, s, t; int *p, *q, *v; int **x; int ***u; short **z; float **y;
```

**Fig. 12.** An example demonstrating the effect of coalescing. The loop formed by the back edge $\delta_5 \rightarrow \delta_1$ reduces to a self loop over GPB $\delta_8$ after coalescing which is redundant (Section 6.2) and is removed.

Within $\Pi(P)$ are allowed to flow from $P$ to $n$—if yes, then $P$ and $n$ belong to the same part. If $\text{GpuOut}_P$ is $\emptyset$, they belong to the same part regardless of $\text{gpuFlow}(P, n)$. The presence of $\text{ColOut}_P$ in the equation of coalesce (Definition 9) ensures that GPB $\delta_P$ is considered for coalescing with $\delta_n$ only if $\delta_P$ has not been found to be an exit node of a part.

Unlike the usual data flow equations, the data flow variables $\text{ColIn}_n$ and $\text{ColOut}_n$ for GPB $n$ are independent of each other—$\text{ColIn}_n$ depends only on the $\text{ColOut}$ of its predecessors and $\text{ColOut}_n$ depends only on the $\text{ColIn}$ 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.

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 the GPUs reaching node $n$ in $\Pi(n)$ using data flow variables $\text{GpuIn}_n/\text{GpuOut}_n$. The data flow equations to compute them are given in Definition 9. Function $\text{gpuFlow}(p, n)$ in the equation for $\text{GpuIn}$ computes the set of GPUs reaching $p$ in $\Pi(p)$ that flow from $p$ to $n$ (provided $n$ can be included in $\Pi(p)$). It facilitates step (2) of Sections 6.4.1. If no data dependence exists (i.e., predicate $\text{DDep}$ is $\text{false}$), the GPUs accumulated in $\text{GpuOut}_p$ are propagated to $n$. The presence of $\neg\text{ColIn}_p$ in equation for $\text{gpuFlow}$ ensures that GPUs in $\text{GpuOut}_p$ are propagated to $\delta_n$ only if $\delta_n$ has not been found to be an entry node of $\Pi(n)$.
Names for GPUs. Statement labels play no part here

<table>
<thead>
<tr>
<th>GPB n</th>
<th>TDef (n)</th>
<th>TRef (n)</th>
<th>GpuIn n</th>
<th>GpuOut n</th>
<th>CollIn n</th>
<th>CollOut n</th>
</tr>
</thead>
<tbody>
<tr>
<td>δ1</td>
<td>0</td>
<td>0</td>
<td>{y1, y2, y3, y4}</td>
<td>{y1, y2, y3, y4}</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>δ2</td>
<td>{int*, float*}</td>
<td>{int**, float**}</td>
<td>{y1, y2, y3, y4}</td>
<td>{y1, y2, y3, y4}</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>δ3</td>
<td>{short*, int*}</td>
<td>{short**, int**}</td>
<td>{y1, y2, y3, y4}</td>
<td>{y1, y2, y3, y4}</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>δ4</td>
<td>{int*, float*}</td>
<td>{int**, float**}</td>
<td>{y1, y2, y3, y4}</td>
<td>{y1, y2, y3, y4}</td>
<td>true</td>
<td>true</td>
</tr>
<tr>
<td>δ5</td>
<td>{int**}</td>
<td>{int***}</td>
<td>{y1, y2, y3, y4}</td>
<td>{y1, y2, y3, y4}</td>
<td>true</td>
<td>false</td>
</tr>
<tr>
<td>δ6</td>
<td>{int*}</td>
<td>0</td>
<td>0</td>
<td>{y5}</td>
<td>false</td>
<td>true</td>
</tr>
<tr>
<td>δ7</td>
<td>{int*}</td>
<td>0</td>
<td>{y5}</td>
<td>{y5, y6}</td>
<td>true</td>
<td>false</td>
</tr>
</tbody>
</table>

Fig. 13. The data flow information computed by coalescing analysis for example in Figure 12. The Colln and ColOut values indicate that GPBs δ1, δ2, δ3, δ4, δ5 can be coalesced. Similarly, GPBs δ6 and δ7 can be coalesced. GPBs δ5 and δ6 must remain in different parts.

Example 32. Figure 13 gives the data flow information for the example of Figure 12. GPBs δ1 and δ3 can be coalesced because ColOut1 is true and GpuOut1 is 0. Thus, DDep(δ1, δ2) returns false indicating that types do not match and hence there is no possibility of a data dependence between the GPUs of δ1 and δ2. Similarly, GPBs δ1 and δ3 can be coalesced. Thus Colln1, Colln2, and Colln3 are true. We check the data dependence between the GPUs of GPBs δ5 and δ4 using the type information. However, DDep(δ4, δ4) returns false because the term (GpuOut4 – δ4) is 0. Thus, GPBs δ3 and δ4 belong to the same part and can be coalesced. For GPBs δ3 and δ4, the possibility of data dependence is resolved based on the type information. The term (GpuOut3 – δ4) contains a type mismatch of z on x, which does not match that of the pointers being read in the GPUs in δ4. Thus, GPBs δ3 and δ4 can be coalesced. GPBs δ4 and δ5 both contain a GPU with a dereference, however DDep(δ4, δ5) returns false indicating that there is no type matching and hence no possibility of data dependence, thereby allowing the coalescing of the two GPBs. The DDep(δ5, δ6) returns true (type of source of the GPU x of GpuOut3 matches the source of the GPU p of δ6) indicating a possibility of data dependence in the caller through aliasing and hence the two GPBs cannot be coalesced. Thus, the first part is δ8 = {δ1, δ2, δ3, δ4, δ5}. The loop δ8 → δ1 before coalescing now reduces to self loop over GPB δ9 after coalescing and is eliminated. GPB δ6 becomes the entry of the new part. GPBs δ6 and δ7 can be coalesced as there is no data dependence between their GPUs. Note that the resulting partition is trivially coherent because each part is a single entry and single exit node.

GPU z of δ8 has a definition-free path in δ8 because boundary definition z reaches the exit of part δ8 along the path δ1 → δ2 → δ4 → δ5. No other GPU has a definition-free path.

Observe that some GPUs appear in multiple GPBs of a GPG (before coalescing). This is because we could have multiple calls to the same procedure. Thus, even though the GPBs are renumbered, the statement labels in the GPUs remain unchanged resulting in repetitive occurrence of a GPU. This is a design choice because it helps us to accumulate the points-to information of a particular statement in all contexts.
We explain call inlining by classifying calls into three categories: (a) callee is known and the call
is non-recursive, (b) callee is known and the call is recursive, and (c) callee is not known.

### Call is Known and the Call is Non-Recursive

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

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

---

**Example 33.** In Figure 4, GPBs $\delta_{01}$ and $\delta_{02}$ can be coalesced because $\text{DDep}(\delta_{01}, \delta_{02})$ returns
false indicating that there is no type matching and hence no possible data dependence between
their GPUs. Thus, $\text{ColOut}_{01}$ and $\text{Coll}_n_{02}$ are set to true. The loop formed by the back edge $\delta_{03} \rightarrow
\delta_{01}$ reduces to a self loop over GPB $\delta_{11}$ after coalescing. The self loop is redundant and hence it is
eliminated. For GPBs $\delta_{02}$ and $\delta_{04}$, $\text{DDep}(\delta_{02}, \delta_{04})$ returns true because $\text{typeof}(q, 2)$ (for the GPU
$q \overset{02}{\rightarrow} m$ in $\delta_{02}$) matches $\text{typeof}(p, 2)$ (for the GPU $e \overset{04}{\rightarrow} p$ in $\delta_{04}$) which is int *.
This indicates the possibility of a data dependence between the GPUs of GPBs $\delta_{02}$ and $\delta_{04}$ ($q$ and $p$
could be aliased in the caller) and hence these GPBs cannot be coalesced. Thus, $\text{ColOut}_{02}$ and $\text{Coll}_{n_{04}}$
are set to false. For GPBs $\delta_{04}$ and $\delta_5$, $\text{DDep}(\delta_{04}, \delta_{05})$ returns true because there is no possible data
dependence. Hence $\text{ColOut}_{04}$ and $\text{Coll}_{n_{05}}$ are set to true and the two GPBs can be coalesced.

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

For procedure $R$ (Figure 5), the boundary definition $b \overset{11}{\rightarrow} b'$ reaches the exit of $\Delta_R$ indicating
that $b$ is may-defined. Hence $\mu_{15} = \{(b, 1)\}$. The GPU $q \overset{02}{\rightarrow} m$ reduces to $d \overset{10}{\rightarrow} m$ in $\delta_{13}$ in $\Delta_R$.
Note that $d$ is defined in $\delta_{08}$ also, hence neither $(q, 2)$ nor $(d, 1)$ is contained in $\mu_{15}$.

### Call Inlining

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

#### 7.1 Call is Known and the Call is Non-Recursive

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

We inline the optimized GPGs of the callees at the call sites in the caller procedures by renumber-
ing the GPB nodes and each inlining of a callee gives fresh numbering to the nodes. This process
does not change the statement labels within the GPUs. Besides, the upwards-exposed variable $x'$
occurring in a callee’s GPU inlined in the caller are substituted by the original variable $x$. 
Input: $P, \Delta_p^1, \Delta_p^i$ // A recursive procedure, its first incomplete GPG containing only
      // recursive calls, and its $i^{th}$ GPG in the fixed-point computation
Output: $\Delta_p^{i+1}$ // Optimized $(i + 1)^{th}$ GPG for procedure $P$

01 Refine_GPG ($P, \Delta_p^1, \Delta_p^i$)
02 {
03 $R_{prev} = \text{RGOOut}_\text{End}(\Delta_p^1)$
04 $\overline{R}_{prev} = \overline{\text{RGOOut}}_\text{End}(\Delta_p^1)$
05 Compute $\Delta_p^{i+1}$ by inlining recursive calls in $\Delta_p^i$ with their latest GPGs
06 Perform both variants of reaching GPUs analysis over $\Delta_p^{i+1}$
07 $R_{curr} = \text{RGOOut}_\text{End}(\Delta_p^{i+1})$
08 $\overline{R}_{curr} = \overline{\text{RGOOut}}_\text{End}(\Delta_p^{i+1})$
09 if ($R_{curr} \neq R_{prev}$) ∨ ($\overline{R}_{curr} \neq \overline{R}_{prev}$)
10   Push callers of $P$ on the worklist
11 Perform strength reduction and control flow elimination optimizations over $\Delta_p^{i+1}$
12 return $\Delta_p^{i+1}$
13 }

Definition 10. Computing GPGs for Recursive Procedures by Successive Refinement

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).

7.2 Callee is Known and the Call is Recursive

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

A set of recursive procedures is represented by a strongly connected component in a call graph. We construct GPGs for a set of recursive procedures by visiting the procedures in a post order obtained through a topological sort of the call graph. Because of recursion, the GPGs of some callees of the leaf are not available in the beginning. We handle such situations by using a special GPG $\Delta_T$ that represents the effect of a call when the callee’s GPG is not available. The GPG $\Delta_T$ is the $T$ element of the lattice of all possible procedure summaries. It kills all GPUs and generates none (thereby, when applied, computes the $T$ value—$\emptyset$—of the lattice for may-points-to analysis) [22]. This is consistent with using $T$ value as the initialization for computing the maximum fixed point solution in iterative data flow analysis. Semantically, $\Delta_T$ 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 $\Lambda_T$ as their initial GPGs, until no further inlining is required. Let $\Lambda^1_p$ denote the GPG of procedure $P$ in which all the calls to the procedures that are not part of the strongly connected component are inlined by their respective optimized GPGs. Note that the GPGs of these procedures have already been constructed because of the bottom up traversal over the call graph. The calls to procedures that are part of the strongly connected component are retained in $\Lambda^1_p$. In each step of refinement, the recursive calls in $\Lambda^1_p$ are inlined either

- by $\Lambda_T$ 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 $\Lambda_T$.

Thus we compute a series of GPGs $\Lambda^i_p$, $i > 1$ for every procedure $P$ in a strongly connected component in the call graph until the termination of fixed-point computation. Once $\Lambda^i_p$ is constructed, we decide to construct $\Lambda^i_Q$ for a caller $Q$ of $P$ if the data flow values of the End GPB of $\Lambda^i_p$ differ from those of the End GPB of $\Lambda^{i-1}_p$. This is because the overall effect of a procedure on its callers

---

**Fig. 15.** Series of GPGs of procedures $P$ and $Q$ of Figure 14. They are computed in the order shown in Figure 14(b). See Example 35 for explanation.
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 $\Delta_p^{i-1}$ and $\Delta_p^i$ are same, then they would have identical effect on their callers. Thus, the GPGs are semantically identical as procedure summaries even if they differ structurally. Thus, the convergence of this fixed-point computation differs subtly from the usual fixed-point computation in that it does not depend on matching the values at corresponding nodes (or the structure of the GPGs) across successive iterations. Instead, it depends on matching the data flow values of the End GPB. This process is described in Definition 10.

**Example 35.** In the example of Figure 14, the sole strongly connected component contains procedures $P$ and $Q$. The GPG of procedure $Q$ is constructed first and $\Delta_Q^1$ contains a single call to procedure $P$ whose GPG is not constructed yet and hence the construction of $\Delta_Q^2$ requires inlining of $\Delta_T$. Since $\Delta_T$ represents a procedure call which never returns, the GPG $\text{End}_Q$ becomes unreachable from the rest of the GPBs in $\Delta_Q^2$. The optimized $\Delta_Q^2$ is $\Delta_T$ 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. $\Delta_p^1$ contains a single call to procedure $Q$ whose incomplete GPG $\Delta_Q^2$, which is $\Delta_T$, is inlined during construction of $\Delta_p^2$. The optimized version of $\Delta_p^2$ is shown in Figure 15. Then, $\Delta_p^2$ is used to construct $\Delta_Q^3$. Reaching GPUs analyses with and without blocking are performed on $\Delta_Q^1$ and $\Delta_Q^3$. The data flow values for $\Delta_Q^2$ are $R_{\text{prev}} = \overline{R_{\text{prev}}} = \emptyset$ whereas the data flow values for $\Delta_Q^3$ are $R_{\text{curr}} = \overline{R_{\text{curr}}} = \{ y \frac{10}{01} a \}$. Since the data flow values have changed, caller of $Q$ i.e., $P$ is pushed on the worklist and $\Delta_p^3$ is constructed by inlining $\Delta_Q^3$. The data flow values computed for $\Delta_p^2$ and $\Delta_p^3$ are identical $R_{\text{prev}} = \overline{R_{\text{prev}}} = R_{\text{curr}} = \overline{R_{\text{curr}}} = \{ y \frac{10}{01} a \}$ 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 $\Delta_p^2$ and $\Delta_p^3$ differ and yet we do not construct the GPG $\Delta_Q^3$. This is because $\Delta_Q^4$ constructed by inlining $\Delta_p^3$ will have the same effect as that of $\Delta_Q^3$ constructed by inlining $\Delta_p^2$ since the impact of $\Delta_p^2$ and $\Delta_p^3$ is identical.

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

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

The above situation models call inlining in GPGs. From Section 3.1.3, the set of GPUs is finite and from [10], they form a lattice with $\subseteq_L$ as the partial order. The flow function for a call GPB is initially assumed to be $f_T$ and then the GPB is replaced by the GPG of the callee. The control flow surrounding this call remains same. Let the effect of the callee GPG be described by a flow function $f$. Clearly, $f \subseteq f_T$ because $f_T$ computes $\top$ value. The process of successive refinements for handling recursion replaces call GPBs by the GPGs of the callees repeatedly. Consider a sequence of refinement, $\Delta_Q^1, \Delta_Q^2, \ldots, \Delta_Q^i$. It can be proved by induction on the length of the sequence that

the GPUs reaching the End GPB of the successive GPBs follow a descending chain because the boundary definitions at the Start GPB of every $\Delta^i_Q$ are identical. Since the set of all possible GPUs is finite, this descending chain must contain two successive elements that are identical. Thus there must exist $\Delta^k_Q$ and $\Delta^{k+1}_Q$ such that the GPUs reaching their End GPB are identical.

7.3 Handling Calls through Function Pointers

We model a call through function pointer (say $fp$) at call site $s$ as a use statement with a GPU $u \downarrow_1 \rightarrow s \downarrow fp$ (Section 8). Interleaving of strength reduction and call inlining reduces the GPU $u \downarrow_1 \rightarrow fp$ and provides the pointees of $fp$. This is identical to computing points-to information (Section 8).

Until the pointees become available, the GPU $u \downarrow_1 \rightarrow s \downarrow fp$ acts as a barrier. Once the pointees become available, the indirect call converts to a set of direct calls (see Appendix C for an illustrative example). A naive approach to function pointer resolution would inline an indirect callee first into its immediate callers. This may require as many rounds of GPG construction as the maximum number of indirect calls in any call chain. Instead, we allow inlining directly in a transitive callee when a pointee of the function pointer of an indirect call becomes available. Hence, we can resolve all indirect calls in a call chain in a single round beginning with the indirect call closest to main. This is explained in Appendix C.

8 COMPUTING POINTS-TO INFORMATION USING GPGS

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

Since we also need points-to information for statements that read pointers but do not define them, we model them as use statements. Consider a use of a pointer variable in a non-pointer assignment or an expression. We represent such a use with a GPU whose source is a fictitious node $u$ with indlev 1 and the target is the pointee which is being read. Thus a condition ‘if ($x == \ast y$)’ where both $x$ and $y$ are pointers, is modelled as a GPB $\{ u \downarrow_1 \rightarrow x, u \downarrow_2 \rightarrow y \}$ whereas an integer assignment ‘$\ast x = 5;’$ is modelled as a GPB $\{u \downarrow_1 \rightarrow x\}$.

Example 36. Consider the assignment sequence 01: $x = &a$; 02: $\ast x = 5;$. A client analysis would like to know the pointees of $x$ for statement 02. We model this use of pointee of $x$ as a GPU $u \downarrow_2 \rightarrow x$. This GPU can be composed with $x \downarrow_0 \rightarrow a$ to get a reduced GPU $u \downarrow_2 \rightarrow a$ indicating that pointee of $x$ in statement 2 is $a$.

When a use involves multiple pointers such as ‘if ($x == \ast y$)’, the corresponding GPB contains multiple GPUs. If the exact pointer-pointee relationship is required, rather than just the reduced form of the use (devoid of pointers), we need additional minor bookkeeping to record GPUs and the corresponding pointers that have been replaced by their pointees in the simplified GPUs.

With the provision of a GPU for a use statement, the process of computing points-to information can be seen simply as a process of simplifying consumer GPUs (including those with a use node $u$. The interleaving of strength reduction and call inlining gradually converts a GPU $x \downarrow_1 \rightarrow y$ to
a set of points-to edges \( \{ a \xrightarrow{10} b \mid a \text{ is } \ell^{th} \text{ pointee of } x, b \text{ is } j^{th} \text{ pointee of } y \} \). This is achieved by propagating the use of a pointer (in a pointer assignment or a use statement) and its definitions to a common context. This may require propagating:

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

**Example 37.** The four variants of hoisting \( p \) and \( c \) to a common procedure in the first phase of a bottom-up method are illustrated below with the help of Figure 16. Effectively, they make the second phase redundant.

(a) When \( \Delta_Q \) is inlined in \( P \), \( c : y \xrightarrow{11} x \) from procedure \( Q \) is hoisted to procedure \( P \) that contains GPU \( p : x \xrightarrow{10} a \) thereby propagating the use of pointer \( x \) in procedure \( Q \) to caller \( P \). Strength reduction reduces \( c \) to \( y \xrightarrow{10} a \).

(b) When \( \Delta_R \) is inlined in \( S \), \( p : x \xrightarrow{10} b \) from procedure \( R \) is hoisted to procedure \( S \) that contains \( c : p \xrightarrow{11} x \) thereby propagating the definition of \( x \) in procedure \( R \) to the caller \( S \). Strength reduction reduces \( c \) to \( p \xrightarrow{10} b \).

(c) When \( \Delta_Q \) and \( \Delta_R \) are inlined in \( S \), \( c : y \xrightarrow{11} x \) in procedure \( Q \) and \( p : x \xrightarrow{10} b \) in procedure \( R \) are both hoisted to procedure \( S \) thereby propagating both the use and definition of \( x \) in procedure \( S \). Strength reduction reduces \( c \) to \( y \xrightarrow{10} b \).
This is followed by a series of lemmas proving the soundness of our analyses and operations.

In this section, we prove the soundness and precision of GPG-based points-to analysis by comparing it with a classical top-down flow- and context-sensitive points-to analysis. We first describe our assumptions, review the classical points-to analysis, and then provide the main proof obligation. This is followed by a series of lemmas proving the soundness of our analyses and operations.

(d) Both the definition and use of pointer $z$ are available in procedure $P$ with $c : w \frac{10}{3} z$ and $p : z \frac{10}{3} c$. Strength reduction reduces $c$ to $w \frac{10}{3} z$. Thus, $y$ points to $a$ along the call from procedure $P$ and it points to $b$ along the call from procedure $S$. Thus, the points-to information $\{ y \frac{10}{3} a, y \frac{10}{3} b \}$ represents flow- and context-sensitive information for statement 4.

9 SOUNDNESS AND PRECISION

In this section, we prove the soundness and precision of GPG-based points-to analysis by comparing it with a classical top-down flow- and context-sensitive points-to analysis. We first describe our assumptions, review the classical points-to analysis, and then provide the main proof obligation. This is followed by a series of lemmas proving the soundness of our analyses and operations.
9.1 Assumptions

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

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

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

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

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

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

Our formulation is based on the classical Sharir-Pnueli tabulation method \[35\]. This method maintains pairs \((X, Y)\) of input-output data flow values (hence the name “tabulation method”) for every procedure \(Q\) where \(X\) reaches Start\(_Q\) and \(Y\) is the corresponding value reaching End\(_Q\). The input value \(X\) forms a context for context-sensitive analysis of \(Q\). Every time a call to \(Q\) is seen with the data flow value \(X\) reaching the call, the data flow value \(Y\) is used as the effect of the call. In other words, procedure \(Q\) is reanalyzed only when a data flow \(X'\) not equal \(X\) reaches a call to \(Q\); the corresponding value \(Y'\) reaching End\(_Q\) is then memoized as the pair \((X', Y')\). \[11\] However, since the tabulation method is algorithmic, we use the ideas from value-contexts based method \[30\] for a declarative description using data flow equations.

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

- Context \(X\) is generated at Start\(_Q\) in the second case of the equation for PT\(_{in,n}\). Thus, the data flow value at Start\(_Q\) is a set of pairs \((Y, Y)\).

\[11\] Since all input-output values are memoized, the method requires the lattice of data flow values to be finite. In our case \(X\) and \(Y\) are points-to graphs involving global variables and their lattice is finite.
The context-sensitive data value after a call to some procedure $Q$ is computed by the first case in the equation for $PT_{out}$. This is achieved by extracting the data flow value (from $PT_{out_{end,Q}}$) that corresponds to the context in which the call to $Q$ was made.

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

9.3 Notations used in the Proof

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

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

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

Note that our implementation does not perform TPT or TRG.

9.3.2 Naming the GPBs after Call Inlining. Let procedure $P$ call procedure $Q$. Then, as illustrated in Figure 18, $\Delta^\text{Call}_P$ contains the $\Delta^\text{Opt}_Q$ as a subgraph that is obtained by expanding GBP $\delta_m$ containing the call to $Q$, by connecting the predecessors of $\delta_m$ to the Start GBP of $\Delta^\text{Opt}_Q$ and the End GBP of $\Delta^\text{Opt}_Q$, and to the successors of $\delta_m$.

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

We handle recursion by repeated inlining of recursive procedures. This process constructs for series of GPGs $\Delta^i_P$, $i > 1$ for every procedure $P$ in a cycle of recursion. GPG $\Delta^{i+1}_P$ is constructed
When $P$ is non-recursive, label $\delta_{m \cdot S}$ in $\Delta_P^{\text{Call}}$ denotes the Start of the callee procedure for the call in node $l$ in $P$. When $P$ is self recursive, label $\delta_{m^k \cdot S}$ of a node in $\Delta_P^{\text{Call}}$ of self-recursive procedure $P$ indicates a sequence $m \cdot m \ldots m \cdot S$ of $k$ occurrences of $m$ followed by $S$ indicating the Start node of the procedures reached after $k$ inlinings of the call to $P$ in node $m$.

Fig. 18. Constructing a GPG by call inlining.

by inlining GPGs $\Delta_Q^1$ in $\Delta_P^1$, for all callees $Q$ of $P$. As explained in Section 7.2, this sequence is bounded by some $k$ when $\Delta_P^k$ is equivalent to $\Delta_P^{k+1}$ in terms of the GPUs reaching the End nodes are identical. This converts a recursive GPG into a non-recursive one.

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

9.3.3 Naming the Data Flow Variables in Different Contexts of a Recursive Procedure. The top-down context-sensitive reaching GPUs analysis over $\Delta_P^{\text{Init}}$ computes the values of data flow variables $\text{RGIn}_n/\text{RGOut}_n$ for different contexts reaching node $n$ for different recursion depth. We distinguish
Generalized Points-to Graphs: A Precise and Scalable Abstraction for Points-to Analysis

<table>
<thead>
<tr>
<th>Equivalence of analyses and associated lemmas. X denotes P or its transitive callees.</th>
</tr>
</thead>
<tbody>
<tr>
<td>TPT($\Delta^\text{Init}_P$, X) ⇔ \text{(Claim A, Lemmas 9.2, 9.3)}</td>
</tr>
<tr>
<td>TRG($\Delta^\text{Init}_P$, X) ⇔ \text{(Claim B, Lemmas 9.4, 9.5, 9.6, and 9.7)}</td>
</tr>
<tr>
<td>IRG($\Delta^\text{Call}_P$, X) ⇔ \text{(Claim C, Lemma 9.8)}</td>
</tr>
<tr>
<td>IRG($\Delta^\text{Red}_P$, X) ⇔ \text{(Claim D, Lemma 9.9)}</td>
</tr>
<tr>
<td>IRG($\Delta^\text{Opt}_P$, X)</td>
</tr>
</tbody>
</table>

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

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

between these different values of the same data flow variable by writing RGI\text{nn}_i / RGO\text{out}_i, where i denote the depth of the recursive call. Note that there is some k for which RGI\text{nn}_n = RGI\text{nn}_n and RGO\text{out}_n = RGO\text{out}_n. It follows from the fact that the flow functions are monotonic and the lattice of GPUs is finite because of only a finite number of combinations of indlevs are possible due to the type restrictions in C (as explained in Footnote 7). A formal proof of the convergence of GPG construction for recursive calls can be found in [10].

### 9.4 The Overall Proof

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

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

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

\[
\text{TPT}(\Delta^\text{Init}_{\text{main}}, \text{main}) \Leftrightarrow \text{TRG}(\Delta^\text{Init}_{\text{main}}, \text{main}) \quad \text{(from Lemma 9.2)}
\]

\[
\Leftrightarrow \text{IRG}(\Delta^\text{Call}_{\text{main}}, \text{main}) \quad \text{(from Lemma 9.5)}
\]
\[\Leftrightarrow \IRG(\Delta_{\text{main}}^{\text{SRed}}, \text{main}) \quad \text{(from Lemma 9.8)}\]
\[\Leftrightarrow \IRG(\Delta_{\text{main}}^{\text{Opt}}, \text{main}) \quad \text{(from Lemma 9.9)}\]

For all transitive callees \(Q\) of \(\text{main}\):
\[\TPT(\Delta_{\text{main}}^{\text{Init}}, Q) \Leftrightarrow \TRG(\Delta_{\text{main}}^{\text{Call}}, Q) \quad \text{(from Lemma 9.3)}\]
\[\Leftrightarrow \IRG(\Delta_{\text{main}}^{\text{SRed}}, Q) \quad \text{(from Lemma 9.7)}\]
\[\Leftrightarrow \IRG(\Delta_{\text{main}}^{\text{Opt}}, Q) \quad \text{(from Lemma 9.8)}\]
\[\Leftrightarrow \IRG(\Delta_{\text{main}}^{\text{Opt}}, Q) \quad \text{(from Lemma 9.9)}\]

Hence the theorem. 

\section{9.5 Equivalence of Analyses Over Different Representations of a GPG}

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

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

**Proof.** Since all pointers are defined before their use, GPU reduction computes GPUs of the form \(x \xrightarrow{10} y\). Thus there are no potential dependences hence no blocking. In such a situation, the data flow equations in Definition 5 reduce to those of the classical flow-sensitive points-to analysis. Assuming that the two analyses maintain context sensitivity using the same mechanism, they would compute the same points-to information at the corresponding program points. 

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

**Proof.** Similar to that of Lemma 9.2. 

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

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

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

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

**The IN Equivalence.** If $\delta_m$ contains a call then $\text{RGIn}_m$ in $\Delta_P^{\text{init}}$ is identical in $\text{RGIn}_{m \cdot S}$ in $\Delta_P^{\text{Call}}$; otherwise $\text{RGIn}_m$ in $\Delta_P^{\text{init}}$ is identical in $\text{RGIn}_m$ in $\Delta_P^{\text{Call}}$.

**The OUT Equivalence.** If $\delta_m$ contains a call then $\text{RGOut}_m$ in $\Delta_P^{\text{init}}$ is identical in $\text{RGOut}_{m \cdot E}$ in $\Delta_P^{\text{Call}}$; otherwise $\text{RGOut}_m$ in $\Delta_P^{\text{init}}$ is identical in $\text{RGOut}_m$ in $\Delta_P^{\text{Call}}$.

We prove these equivalences on the number of iteration $i$.

- **Basis of induction on the number of iterations.** The basis is the first iterations (i.e., $i = 1$). By initialization, $\text{RGIn}$ is $\emptyset$ for each node in both $\Delta_P^{\text{init}}$ and $\Delta_P^{\text{Call}}$ (except for $\text{Start}_P$ for which it contains boundary definitions). Thus IN equivalence holds for each $m$ belonging to $P$. For the OUT equivalence, there are two cases:
  
  - (a) $\delta_m$ does not contain a call. These GPBs are identical in $\Delta_P^{\text{init}}$ and $\Delta_P^{\text{Call}}$. For each such GPB $\delta_m$, $\text{RGOut}_m$ trivially contains all GPUs in $\delta_m$ because $\text{RGIn}_m$ is $\emptyset$ and there is no GPU reduction. Thus, the OUT equivalence also holds for such GPBs.
  
  - (b) $\delta_m$ contains a call. By the hypothesis of structural induction, the lemma holds for the callees. Since $\text{TRG}(\Delta_Q^{\text{init}}, Q) \iff \text{IRG}(\Delta_Q^{\text{Call}}, Q)$, for every value of boundary information, it also holds for $\emptyset$ as boundary information. Hence it holds for the End GPB of $Q$. In $\Delta_P^{\text{init}}$, it becomes the value $\text{RGOut}_m$ whereas in $\Delta_P^{\text{Call}}$, it becomes the value $\text{RGOut}_{m \cdot E}$. Hence the OUT equivalence also holds for such GPBs.

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

This completes the proof of the lemma. \qed

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

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

**The IN equivalence.** $\text{RGIn}_m^i$ computed for the GPB $\delta_m$ in $\Delta_P^{\text{init}}$ is identical to $\text{RGIn}_{m \cdot S}$ for the GPB $\delta_m \cdot S$ in $\Delta_P^{\text{Call}}$, and

**The OUT equivalence.** $\text{RGOut}_m^i$ computed for the GPB $\delta_m$ in $\Delta_P^{\text{init}}$ is identical to $\text{RGOut}_{m \cdot E}$ for the GPB $\delta_m \cdot E$ in $\Delta_P^{\text{Call}}$.

If the IN equivalence holds, then the OUT equivalence holds because by Lemma 9.9, the inlined version $\Delta_P^{\text{Opt}}$ is same as $\Delta_P^{\text{Red}}$ which is same as $\Delta_P^{\text{Call}}$ by Lemma 9.8. Thus our proof obligation reduces to showing the IN equivalence which is easy to argue using induction on recursion depth $i$. The base case $i = 1$ represents the first (i.e., “outermost”) recursive call. Since no recursive call has been encountered so far, it is easy to see that $\text{RGIn}_m^1 = \text{RGIn}_{1 \cdot S}$. Assuming that it holds for recursion depth $i$, it also holds for recursion depth $i + 1$ because as explained above, the effect of $\Delta_P^{\text{Opt}}$ is same as $\Delta_P^{\text{Call}}$ by Lemmas 9.9 and 9.8. For $i = k$, since a fixed point has been reached in both $\Delta_P^{\text{init}}$ and $\Delta_P^{\text{Call}}$, the absence of recursive call $k + 1$ in $\Delta_P^{\text{Call}}$ does not matter. \qed
In the following lemma, we need to consider the contexts of the calls within procedure $P$. For our reasoning, the way a context is defined does not matter and we generically denote a context as $\sigma$. We assume that $\sigma$ denotes the full context without any approximation.

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

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

Consider a particular to call to $Q$ with context $\sigma$. We prove the lemma on the length $j$ of the call chain from $P$ to $Q$ for this context. The base case is $j = 1$ representing the situation when $Q$ is a direct callee of $P$. Let this call be in the GPB $\delta_m$. Then, from Lemma 9.4, $\text{RGIn}_{m-S}$ is same as $\text{RGIn}_{m-S}$. TRG analysis of $\Delta_{P}^{\text{Init}}$ will visit $\Delta_{P}^{\text{Init}}$ for the context $\sigma$ with the boundary information $\text{RGIn}_{m}$. On the other hand IRG analysis of $\Delta_{P}^{\text{Call}}$ will analyze the GPG subgraph between $\delta_{m,E}$ to $\delta_{m,S}$ with the data flow reaching $\text{RGIn}_{m-S}$. Since the information reaching the Start GPB of $Q$ is same in both the cases, the data flow values for statements in $Q$ for analysis within this context would be identical in both the analyses proving the base case.

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

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

**Proof.** The proof is essentially along the lines of Lemma 9.5 because all we need to argue is that $\text{RGIn}_{m}$ computed for the GPB $\delta_m$ in $\Delta_{P}^{\text{Init}}$ is identical to $\text{RGIn}_{m-S}$ for the GPB $\delta_{m-S}$ in $\Delta_{P}^{\text{Call}}$.

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

$$\text{IRG}(\Delta_{P}^{\text{Call}}, Q) \Leftrightarrow \text{IRG}(\Delta_{P}^{\text{SRed}}, Q)$$

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

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

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

\[
\forall Q. \text{IRG}(\Delta_{P\text{Red}}^{\text{SRed}}, Q) \leftrightarrow \text{IRG}(\Delta_{P\text{Opt}}^{\text{Opt}}, Q)
\]

**Proof.** GPGs \( \Delta_{P\text{Red}}^{\text{SRed}} \) and \( \Delta_{P\text{Opt}}^{\text{Opt}} \) do not contain any call hence the reasoning below holds for all corresponding statements between them regardless of whether they belong to \( P \) or a transitive callee of \( P \). We prove the equivalence \( \Delta_{P\text{Red}}^{\text{SRed}} \) and \( \Delta_{P\text{Opt}}^{\text{Opt}} \) in three steps for the three optimizations.

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

2. **Empty GPB Elimination.** Since empty GPBs do not influence the reaching GPUs analysis in any way, removing them by connecting their predecessors to their successors does not change anything (because this transformation preserves the paths through the empty GPBs).

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

- **Ensuring Soundness.** As described in Section 6.4.1, soundness condition (S1) is ensured by considering only adjacent nodes whose GPUs do not have any dependence between them (steps 1 and 2 in Section 6.4.1) whereas condition (S2) is ensured by computing may-definition sets sets associated with coalesced GPBs to maintain definition-free paths (step 3 in Section 6.4.1).

- **Ensuring Precision.** Precision condition (P1) is ensured by considering only those nodes whose GPUs do not have any dependence between them (step 2 in Section 6.4.1) whereas conditions (P2) and (P3) are satisfied by ensuring that no spurious control flow paths are created: Only adjacent nodes (step 1 in Section 6.4.1) for coalescing and coherence ensures that there are no “cross-connections” between exits and entries of adjacent parts with multiple entries or exits (Section 6.4.2).

This proves the lemma. □

### 9.6 Soundness and Precision of Analyses and Operations

Recall that the abstract memory computed by a points-to analysis is a relation \( M \subseteq L_P \times L \) where \( L \) denotes the set of locations and \( L_P \subseteq L \) denotes the set of pointers. Given \( M \), the direct pointee of a set of pointers \( X \subseteq L_P \) are computed by the relation application \( M X = \{ y \mid (x, y) \in M, x \in X \} \).

Let \( M^i \) denote a composition of degree \( i \). Then, \( M^i \{ x \} \) discovers the \( i^{th} \) pointee of \( x \) which involves \( i \) pointee reads from \( x \): first \( i - 1 \) addresses are read followed by the content of the last address. By construction, \( M^0 \{ x \} = \{ x \} \). Abstract execution of GPU \( x \xrightarrow{i} y \) in memory \( M \) imposes the constraint \( M^i \{ x \} \supseteq M^i \{ y \} \) on \( M \) with weak updates; with strong updates, the constraint is
stronger: \( M^i \{ x \} = M^i \{ y \} \). Observe that \( M^i \{ x \} \supseteq M^j \{ y \} \Rightarrow M^{i+k} \{ x \} \supseteq M^{i+k} \{ y \}, k \geq 0 \); this also holds for equality (i.e., "\( = \)" instead of "\( \supseteq \)"").

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

**Proof.** Consider the picture on the right. The memory before the execution of \( p \) is \( M \) with no constraint, while the memory obtained after the execution of \( p \) is \( M \) with the constraint \( C_p \). The memory obtained after the execution of \( c \) and \( r \) is \( M \) with the constraints \( C_c \land C_p \) and \( C_r \land C_p \), respectively. Then, the lemma can be proved by showing that \( C_c \land C_p \) and \( C_r \land C_p \) are identical. We first consider \( TS \) composition. Initially assume that \( c \) causes a weak update; this assumption can be relaxed later. Let \( p \) and \( c \) be the GPUs illustrated in the first column of Figure 7. Since no other GPU with the source \( (x, k') \), \( k' \leq k \), reaches \( c \), the constraint \( C_p \) is \( M^k \{ x \} = M^i \{ y \} \). The constraints \( C_c \) is \( M^i \{ z \} \supseteq M^i \{ x \} \) and \( C_r \) is \( M^i \{ z \} \supseteq M^{i+j-k} \{ y \} \).

\[
\begin{align*}
& C_c \land C_p = M^i \{ z \} \supseteq M^i \{ x \} \land M^k \{ x \} = M^i \{ y \} \\
& \Rightarrow M^i \{ z \} \supseteq M^i \{ x \} \land M^{k+j-k} \{ x \} = M^{i+j-k} \{ y \} \land M^k \{ x \} = M^i \{ y \} \quad \text{(adding} \ j \text{ to} \ C_p) \\
& \Rightarrow M^i \{ z \} \supseteq M^i \{ x \} \land M^i \{ x \} = M^{i+j-k} \{ x \} \land M^k \{ x \} = M^i \{ y \} \\
& \Rightarrow M^i \{ z \} \supseteq M^{i+j-k} \{ x \} \land M^k \{ x \} \supseteq M^i \{ y \} \quad \text{(combining the first two terms)} \\
& \Rightarrow C_r \land C_p \\
\end{align*}
\]

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

\[
\begin{align*}
& C_r \land C_p = M^i \{ z \} \supseteq M^{i+j-k} \{ y \} \land M^k \{ x \} = M^i \{ y \} \\
& = M^i \{ z \} \supseteq M^{i+j-k} \{ y \} \land M^i \{ y \} = M^k \{ x \} \\
& \Rightarrow M^i \{ z \} \supseteq M^{i+j-k} \{ y \} \land M^{i+j-k} \{ y \} \land M^k \{ x \} = M^i \{ y \} \quad \text{(adding} \ j \text{ to} \ C_p) \\
& \Rightarrow M^i \{ z \} \supseteq M^{i+j-k} \{ y \} \land M^{i+j-k} \{ y \} \land M^k \{ x \} = M^i \{ y \} \\
& \Rightarrow M^i \{ z \} \supseteq M^i \{ x \} \land M^k \{ x \} = M^i \{ y \} \quad \text{(combining the first two terms)} \\
& \Rightarrow C_c \land C_p \\
\end{align*}
\]

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

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

**Proof.** Recall that Red is the fixed point of function \( \text{GPU\_reduction}(\text{Red}, R) \) (Definition 4) with the initial value \( \text{Red} = \{ c \} \). As explained in Section 4.3, this computation is monotonic and is guaranteed to converge. Hence, this lemma can be proved by induction on step \( i \) in the fixed point iteration that computes \( \text{Red}^i \). The base case is \( i = 0 \) which follows trivially because \( \text{Red}^0 = \{ c \} \).
For the inductive hypothesis, assume that the lemma holds for Red\textsuperscript{i}. For the inductive step, we observe that Red\textsuperscript{i+1} is computed by reducing the GPUs in Red\textsuperscript{i} by composing them with those in \( R \). Consider the composition of a GPU \( \gamma_1 \in \text{Red}^i \) with a GPU \( p \in R \) such that \( \gamma_2 = \gamma_1 \circ p \); then \( \gamma_2 \in \text{Red}^{i+1} \). Let the source of \( \gamma_1 \) be \((x, i)\). Then, from Lemma 9.10, \( \gamma_2 \) can replace \( \gamma_1 \) if \( R \) does not contain any GPU \( \gamma_i' \) with a source \((x, i')\) where \( i' \leq i \). Thus there are two cases to consider:

- There is no GPU \( \gamma'_1 \) in \( R \) with a source \((x, i'), i' \leq i \). Then, \( \text{Red}^{i+1} = (\text{Red}^i - \{\gamma_1\}) \cup \{\gamma_2\} \). Since the execution of the GPUs in \( \text{Red}^i \) is identical to that of \( c \) by the inductive hypothesis, the execution of the GPUs in \( \text{Red}^{i+1} \) is identical to that of \( c \).

- There is a GPU \( \gamma'_1 \) in \( R \) with a source \((x, i')\), \( i' \leq i \). Then \( \gamma_1 \) will be composed with \( \gamma'_1 \) too giving some simplified GPU \( \gamma'_2 \). Assume that \( \gamma'_1 \) is the only such GPU in \( R \), then \( \text{Red}^{i+1} \) is computed as \( \text{Red}^{i+1} = (\text{Red}^i - \{\gamma_1\}) \cup \{\gamma_2, \gamma'_2\} \). Since the execution of the GPUs in \( \text{Red}^i \) is identical to that of \( c \) by the inductive hypothesis, the execution of the GPUs in \( \text{Red}^{i+1} \) is identical to that of \( c \).

Replacement of \( \gamma_1 \) by the simplified GPUs is sound only when it is composed with all possible GPUs with which it has a RaW dependence. This requires us to argue the following:

- The source \((x, i')\) used for reducing \( \gamma_1 \) for computing \( \gamma_2 \) is defined along every path reaching the node. This follows from the property of completeness of \( R \) (Section 4.3) which is trivially ensured by (a) reaching GPUs analysis without blocking (Section 4.5) because of the presence of boundary definitions at the Start, and by (b) reaching GPUs analysis with blocking (Section 4.6) because of the presence of boundary definitions at the Start, and their re-introduction when some GPUs are blocked.

- Reaching GPUs analyses are sound and precise, i.e., no GPU on which \( \gamma_1 \) may have a RaW dependence is missed from \( R \), nor does \( R \) contain a spurious GPU. This follows from Lemmas 9.12 and 9.14.

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

Hence the lemma. \( \square \)

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

**Lemma 9.12 (Soundness of Reaching GPUs Analysis without Blocking).** Consider a GPU \( \gamma: x \xrightarrow{i_1} y \) obtained after the strength reduction of the GPUs in \( \delta_i \) using the simplified GPUs reaching \( \delta_i \). Assume that there is a control flow path from \( \delta_i \) to \( \delta_m \) along which the source \((x, i)\) is not strongly updated. Then, \( \gamma \) reaches \( \delta_m \).

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

For the inductive hypothesis, assume that the lemma holds when there are \( k \) nodes between \( \delta_i \) and \( \delta_m \). To prove that it holds for \( k + 1 \) nodes between them, let the \( k^{th} \) node be \( \delta_n \). Then, \( \gamma \in \text{RGIN}_n \) by the inductive hypothesis. Since \( \delta_n \) does not strongly update the source \((x, i)\), it means that \( \gamma \not\in \text{RGKill}_n \) and thus, \( \gamma \in \text{RGOut}_n \). Since \( \delta_m \) is a successor of \( \delta_n \), it follows that \( \gamma \in \text{RGIN}_m \), proving the inductive step, and hence the lemma. \( \square \)
Lemma 9.13 (Soundness of Reaching GPUs Analysis with Blocking). Let $GPU \gamma : x \xrightarrow{\gamma} y$ be obtained after the strength reduction of the GPUs in $\delta_1$ using the simplified GPUs reaching $\delta_1$. Assume that there is a control flow path from $\delta_1$ to $\delta_m$ along which the source $(x, i)$ is neither strongly updated, nor blocked. Then, $\gamma$ reaches $\delta_m$.

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

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

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

For the inductive hypothesis, assume that the lemma holds for iteration $i$. Consider a GPU $\gamma$ in RGIn$_l$ in iteration $(i + 1)$. Then, $\gamma$ must be in RGOut$_m$ for some predecessor $\delta_m$ of $\delta_l$ in some iteration $i' < i$. If it is generated by GPU reduction, then the lemma is proved. If not, then it must be the case that it reached RGIn$_m$ in iteration $i' < i$ and was neither killed nor blocked in $\delta_m$. By the inductive hypothesis, it is either a boundary definition reaching from Start or there exists some GPB $\delta_n$ that generated it after the reduction. Hence it follows in iteration $i + 1$ that $\gamma$ is either a boundary definition reaching from Start or there exists some GPB $\delta_n$ that generated it after the reduction. This proves the lemma.

10 EMPIRICAL EVALUATION

The main motivation of our implementation was to evaluate the effectiveness of our optimizations in handling the following challenge for practical programs:

A procedure summary for flow- and context-sensitive points-to analysis needs to model the accesses of pointees defined in the callers and needs to maintain control flow. Thus, the size of a summary can be potentially large. Further, the transitive inlining of the summaries of the callee procedures can increase the size of a summary exponentially thereby hampering the scalability of analysis.

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

10.1 Implementation and Experiments

We implemented GPG-based points-to analysis in GCC 4.7.2 using the LTO framework and have carried out measurements on SPEC CPU2006 benchmarks on a machine with 16 GB RAM with eight 64-bit Intel i7-7700 CPUs running at 4.20GHz. The implementation can be downloaded from https://github.com/PritamMG/GPG-based-Points-to-Analysis.

10.1.1 Modelling Language Features. Our method eliminates all non-address-taken local variables using def-use chains explicated by the SSA-form; this generalizes the technique in Section 3.3.1

12 An address-taken variable is a global or stack-allocated variable to which the C address-of operator, ‘&’, is applied.
which removes compiler-added temporaries. If a GPU defining a global variable or a parameter reads a non-address-taken local variable, we identify the corresponding producer GPUs by traversing the def-use chains transitively. This eliminates the need for filtering out the local variables from the GPGs for inlining them in the callers. As a consequence, a GPG of a procedure consists only of GPUs that involve global variables, parameters of the procedure, and the return variable which is visible in the scope of its callers. All address-taken local variables in a procedure are treated as global variables because they can escape the scope of the procedure. However, these variables are not strongly updated because they could represent multiple locations.

We approximate heap memory using context-insensitive allocation-site-based abstraction and by maintaining \(k\)-limited indirection lists of field dereferences for \(k = 3\) (see Appendix B) for heap locations that are live on entry to a procedure. An array is treated index-insensitively. Since there is no kill owing to weak update, arrays are maintained flow-insensitively by our analysis.

For pointer arithmetic involving a pointer to an array, we approximate the pointer being defined to point to every element of the array. For pointer arithmetic involving other pointers, we approximate the pointer being defined to point to every possible location.

### 10.1.2 Variants of Points-to Analysis Implemented

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

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

\[\text{From now on we also regard heap-summary nodes, and address-taken local variables as 'global variables'}.\]

---

### Table 1. Benchmark characteristics relevant to our analysis. Column E omits procedures with a single call.

<table>
<thead>
<tr>
<th>Program</th>
<th>kLoC</th>
<th># of statements involving pointers</th>
<th># of call sites</th>
<th># of procedures</th>
<th>Proc. count for different buckets of # of calls</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>A</td>
<td>B</td>
<td>C</td>
<td>D</td>
</tr>
<tr>
<td>lbm</td>
<td>0.9</td>
<td>370</td>
<td>30</td>
<td>19</td>
<td>5</td>
</tr>
<tr>
<td>mcf</td>
<td>1.6</td>
<td>480</td>
<td>29</td>
<td>23</td>
<td>11</td>
</tr>
<tr>
<td>libquantum</td>
<td>2.6</td>
<td>340</td>
<td>277</td>
<td>80</td>
<td>24</td>
</tr>
<tr>
<td>bzip2</td>
<td>5.7</td>
<td>1650</td>
<td>288</td>
<td>89</td>
<td>35</td>
</tr>
<tr>
<td>milc</td>
<td>9.5</td>
<td>2540</td>
<td>782</td>
<td>190</td>
<td>60</td>
</tr>
<tr>
<td>sjeng</td>
<td>10.5</td>
<td>700</td>
<td>726</td>
<td>133</td>
<td>46</td>
</tr>
<tr>
<td>hmmer</td>
<td>20.6</td>
<td>6790</td>
<td>1328</td>
<td>275</td>
<td>93</td>
</tr>
<tr>
<td>h264ref</td>
<td>36.1</td>
<td>17770</td>
<td>2393</td>
<td>566</td>
<td>171</td>
</tr>
<tr>
<td>gobmk</td>
<td>158.0</td>
<td>212830</td>
<td>9379</td>
<td>2699</td>
<td>317</td>
</tr>
</tbody>
</table>

---

<table>
<thead>
<tr>
<th>Program</th>
<th># of Call Graph Nodes</th>
<th># of Call Graph Edges</th>
<th># of Monomorphic Calls</th>
<th># of Polymorphic Calls</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>GPG</td>
<td>SVF</td>
<td>GPG</td>
<td>SVF</td>
</tr>
<tr>
<td>lbm</td>
<td>19</td>
<td>19</td>
<td>20</td>
<td>20</td>
</tr>
<tr>
<td>mcf</td>
<td>23</td>
<td>23</td>
<td>26</td>
<td>26</td>
</tr>
<tr>
<td>libquantum</td>
<td>80</td>
<td>80</td>
<td>156</td>
<td>156</td>
</tr>
<tr>
<td>bzip2</td>
<td>88</td>
<td>88</td>
<td>140</td>
<td>144</td>
</tr>
<tr>
<td>milc</td>
<td>174</td>
<td>174</td>
<td>434</td>
<td>434</td>
</tr>
<tr>
<td>sjeng</td>
<td>121</td>
<td>121</td>
<td>367</td>
<td>367</td>
</tr>
<tr>
<td>hmmer</td>
<td>263</td>
<td>263</td>
<td>709</td>
<td>723</td>
</tr>
<tr>
<td>h264ref</td>
<td>560</td>
<td>560</td>
<td>1231</td>
<td>1521</td>
</tr>
<tr>
<td>gobmk</td>
<td>2679</td>
<td>2679</td>
<td>8889</td>
<td>8889</td>
</tr>
</tbody>
</table>

Table 2. Call Graph Statistics for the common part of the program as discovered by GCC (for GPG-based analysis) and LLVM (for SVF-based analysis).

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.

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

10.1.4 Comparison with Other Points-to Analysis. We also computed corresponding data for client analyses using static value flow analysis (SVF) [42]. SVF is used for comparison because its implementation is readily available. SVF is a static analysis framework implemented in LLVM that allows value-flow construction and context-insensitive pointer analysis to be performed in an iterative manner (sparse analysis performed in stages, from cheap over-approximate analysis to precise expensive analysis). It uses the points-to information from Andersen’s analysis and constructs an interprocedural memory SSA (Static Single Assignment) form where def-use chains of both top-level (i.e., non-address-taken) and address-taken variables are included. The scalability and precision of the analysis is controlled by designing memory SSA that allows users to partition memory into a set of regions.

Table 3. Mod/Ref Statistics. RR denotes the reduction ratio of GPG-based analysis over SVF-based analysis computed by dividing the counts for GPG-based method by the counts for SVF-based method. A value smaller than 1.00 indicates that GPG-based analysis is more precise than SVF-based analysis and the smaller the value, higher is the precision.

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

We measured the data for the following two categories of evaluations.

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

10.2.1 Comparison of GPG-based and SVF Analyses. We compared the data for mod-ref analysis and call graph for GPG-based points-to analysis with that of SVF. However, the comparison is not straightforward because the two implementations use two differently engineered intermediate representations of programs. The underlying compiler frameworks (GCC and LLVM) use different strategies for function inlining and function cloning (for creating specialized versions of the same functions) leading to a different number of procedures in the call graph for the same benchmark program. Although we suppressed the two optimizations in GCC with the appropriate flags, GCC continues to perform function inlining and cloning at a smaller scale indicating that we do not have a direct control over the IR. We therefore make the comparison only on the common part of the benchmark programs.
<table>
<thead>
<tr>
<th>Program</th>
<th># of Proc.</th>
<th># of Stmts.</th>
<th>FSCS</th>
<th>FICI</th>
<th>FICS</th>
<th>SVF</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbm</td>
<td>19</td>
<td>367</td>
<td>0.05</td>
<td>3.26</td>
<td>2.11</td>
<td>0.21</td>
</tr>
<tr>
<td>mcf</td>
<td>23</td>
<td>484</td>
<td>0.63</td>
<td>8.13</td>
<td>7.39</td>
<td>0.92</td>
</tr>
<tr>
<td>libquantum</td>
<td>80</td>
<td>396</td>
<td>0.12</td>
<td>3.99</td>
<td>2.42</td>
<td>0.28</td>
</tr>
<tr>
<td>bzip2</td>
<td>89</td>
<td>1645</td>
<td>0.18</td>
<td>4.72</td>
<td>2.94</td>
<td>2.44</td>
</tr>
<tr>
<td>milc</td>
<td>190</td>
<td>2467</td>
<td>0.29</td>
<td>3.43</td>
<td>2.87</td>
<td>1.05</td>
</tr>
<tr>
<td>sjeng</td>
<td>133</td>
<td>684</td>
<td>0.42</td>
<td>1.12</td>
<td>1.9</td>
<td>0.39</td>
</tr>
<tr>
<td>hmmer</td>
<td>275</td>
<td>6717</td>
<td>0.07</td>
<td>5.10</td>
<td>1.52</td>
<td>3.44</td>
</tr>
<tr>
<td>h264ref</td>
<td>566</td>
<td>17253</td>
<td>0.49</td>
<td>5.02</td>
<td>3.08</td>
<td>0.41</td>
</tr>
<tr>
<td>gobmk</td>
<td>2699</td>
<td>10557</td>
<td>0.24</td>
<td>2.95</td>
<td>1.39</td>
<td>7.58</td>
</tr>
<tr>
<td>Geometric Mean</td>
<td></td>
<td></td>
<td>0.21</td>
<td>3.74</td>
<td>2.51</td>
<td>0.94</td>
</tr>
</tbody>
</table>

Table 4. Final points-to information – average points-to pairs per procedure. FSCS (flow- and context-sensitive), FICI (flow- and context-insensitive), FICS (flow-insensitive and context-sensitive), SVF (static value flow) analysis.

- **Call graph construction.** Table 2 provides the call graph nodes (number of functions in a program) and the call graph edges (representing the caller-callee relationships between functions). The table also provides the number of monomorphic calls (single callee at a call site) and polymorphic calls (multiple callees at a call site) for indirect calls. An approach $A_1$ is said to be more precise than approach $A_2$ if the number of call graph edges computed by $A_1$ is smaller than that computed by $A_2$ (given that the number of nodes in the call graph are same). Also, $A_1$ is more precise than $A_2$ if it discovers more monomorphic calls than $A_2$. Table 2 shows that lbm, mcf, libquantum, and milc have no function pointers (indicated by zero counts for polymorphic and monomorphic calls for function pointers). Since we compared only common parts in the IRs of both GCC and LLVM, these benchmarks have identical call graphs. Besides, there is no difference in the precision of call graphs for sjeng (one polymorphic indirect call) and gobmk (44 polymorphic indirect calls). However, the data shows that our approach finds a larger number of monomorphic calls in hmmer and h264ref than the SVF method. This reduces the number of edges in the call graph.

- **Mod-ref analysis.** Table 3 provides the number of calls to procedures in which a pointer variable was either modified or referenced. It also gives the number of pointer variables (globals, parameters, and heap locations) that are modified and referenced across all calls. An approach $A_1$ is said to be more precise than approach say $A_2$, if the numbers computed in this table by $A_1$ are smaller than the numbers computed by $A_2$. Table 3 shows that our approach is more precise than SVF method across all benchmarks. The geometric mean of the reduction ratio of GPG over SVF in the number of calls is 0.39 with a geometric standard deviation of 1.81. The same numbers for ref are 0.41 and 1.60, respectively. The geometric mean of the reduction ratio of the number of mods across all calls is 0.27 with a geometric standard deviation of 2.34. The same numbers for ref are 0.27 and 2.43, respectively.

- **Average points-to information.** A smaller value of average points-to pairs per procedure indicates higher precision. Table 4 shows that in general, the average points-to pairs per procedure for GPG-based points-to analysis is substantially smaller than that of SVF. More specifically, the geometric mean of average points-to pairs per procedure in GPG based FSCS points-to analysis is 0.21 with a geometric standard deviation of 2.40. The number of average points-to pairs is larger for mcf, sjeng, and h264ref because they contain a large number
Table 5. Measurement of the quality of procedure summaries. The geometric mean of percentages of procedures with 1 to 3 GPBs is 95.06%, while that of procedures with 0 GPUs and 1 to 10 GPUs, is 77.63% and 18.08%, respectively. Some procedures have zero GPBs because they have an exit node with no successors.

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

- Effectiveness of control flow minimization. The effectiveness of control flow minimization optimizations is presented in Table 6. The data is represented in terms of percentage of dead GPUs, percentage of empty GPBs, percentage of GPBs reduced because of coalescing, and percentage of back edges removed because of coalescing.

We compute the percentage of dead GPUs as follows: Let $x$ and $y$ denote the number of GPUs before and after dead GPU elimination respectively. Then the number of dead GPUs is $d = x - y$ and percentage of dead GPUs is computed as $u = (d/x) \times 100$ (rounded to the nearest integer). We then create five buckets that associate the number of procedures having percentage of dead GPUs within a given range. Similarly, we create buckets for percentage of empty GPBs, percentage of GPBs reduced because of coalescing, and percentage of back edges removed because of coalescing as shown in Table 6. We observe that:

(a) The percentage of dead GPUs is very small and the dead GPU elimination optimization is the least effective of all the optimizations. For most procedures, less than 20% of the GPUs are eliminated as dead. The geometric mean of the percentage of procedures for this bucket is 97.01% with a geometric standard deviation of 1.04 across different benchmarks. The absolute numbers for this optimizations are very small because the number of candidate procedures for this optimization is small—as shown in Table 7, a large number of GPGs have zero GPUs and a small number GPGs are disconnected in that the corresponding CFGs have an exit node with no successors.
### Count of procedures in each bucket of percentages

<table>
<thead>
<tr>
<th>Program</th>
<th>% of Dead GPUs</th>
<th>% of empty GPBs eliminated</th>
<th>% of GPBs reduced because of coalescing</th>
<th>% of back edges reduced because of coalescing</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0-20</td>
<td>21-40</td>
<td>41-60</td>
<td>61-80</td>
</tr>
<tr>
<td></td>
<td>81-100</td>
<td>0-20</td>
<td>21-40</td>
<td>41-60</td>
</tr>
<tr>
<td></td>
<td>61-80</td>
<td>0-20</td>
<td>21-40</td>
<td>41-60</td>
</tr>
<tr>
<td></td>
<td>81-100</td>
<td>0-20</td>
<td>21-40</td>
<td>41-60</td>
</tr>
<tr>
<td></td>
<td>0-20</td>
<td>21-40</td>
<td>41-60</td>
<td>61-80</td>
</tr>
<tr>
<td></td>
<td>81-100</td>
<td>0-20</td>
<td>21-40</td>
<td>41-60</td>
</tr>
<tr>
<td></td>
<td>61-80</td>
<td>0-20</td>
<td>21-40</td>
<td>41-60</td>
</tr>
<tr>
<td></td>
<td>81-100</td>
<td>0-20</td>
<td>21-40</td>
<td>41-60</td>
</tr>
<tr>
<td></td>
<td>0-20</td>
<td>21-40</td>
<td>41-60</td>
<td>61-80</td>
</tr>
<tr>
<td></td>
<td>81-100</td>
<td>0-20</td>
<td>21-40</td>
<td>41-60</td>
</tr>
<tr>
<td></td>
<td>61-80</td>
<td>0-20</td>
<td>21-40</td>
<td>41-60</td>
</tr>
<tr>
<td></td>
<td>81-100</td>
<td>0-20</td>
<td>21-40</td>
<td>41-60</td>
</tr>
</tbody>
</table>

### Table 6. Effectiveness of control flow minimization. The geometric mean has been shown for percentage of procedures in the buckets with the largest numbers. The percentages for dead GPU elimination is computed against a much smaller number of procedures (obtained by omitting the procedures that have 0 GPUs).

<table>
<thead>
<tr>
<th>Program</th>
<th># of Proc.</th>
<th># of Stmts.</th>
<th># of Proc. with 0 GPUs</th>
<th># of Proc. whose GPG is ∆⁻</th>
<th># of Proc. containing back edges in the CFG</th>
<th># of Proc. containing back edges in the GPG</th>
<th># Queued GPUs computed when GPU compositions were postponed</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbm</td>
<td>19</td>
<td>367</td>
<td>15</td>
<td>0</td>
<td>10</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>mcf</td>
<td>23</td>
<td>484</td>
<td>12</td>
<td>0</td>
<td>20</td>
<td>1</td>
<td>115</td>
</tr>
<tr>
<td>bzip2</td>
<td>80</td>
<td>396</td>
<td>70</td>
<td>0</td>
<td>36</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>milc</td>
<td>89</td>
<td>1645</td>
<td>70</td>
<td>8</td>
<td>43</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>sjeng</td>
<td>190</td>
<td>2467</td>
<td>175</td>
<td>3</td>
<td>92</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>hmmer</td>
<td>275</td>
<td>6717</td>
<td>237</td>
<td>5</td>
<td>153</td>
<td>0</td>
<td>9</td>
</tr>
<tr>
<td>h264ref</td>
<td>566</td>
<td>17253</td>
<td>435</td>
<td>3</td>
<td>308</td>
<td>8</td>
<td>15</td>
</tr>
<tr>
<td>gobmk</td>
<td>2699</td>
<td>10557</td>
<td>2146</td>
<td>2</td>
<td>464</td>
<td>45</td>
<td>3</td>
</tr>
</tbody>
</table>

### Table 7. Miscellaneous data.

### Table 8. Time measurements.
A: Procedure count for different buckets of ratio of GPBs/BBs in CFG and GPG after inlining
B: Procedure count for different buckets of ratio of GPBs/BBs in CFG and optimized GPG
C: Procedure count for different buckets of ratio of GPBs in GPG before and after optimizations

<table>
<thead>
<tr>
<th>Program</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>lbm</td>
<td>9</td>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>mcf</td>
<td>14</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td>libquantum</td>
<td>42</td>
<td>14</td>
<td>12</td>
</tr>
<tr>
<td>bzip2</td>
<td>53</td>
<td>16</td>
<td>10</td>
</tr>
<tr>
<td>milc</td>
<td>115</td>
<td>20</td>
<td>14</td>
</tr>
<tr>
<td>sjeng</td>
<td>87</td>
<td>17</td>
<td>7</td>
</tr>
<tr>
<td>hmmr</td>
<td>205</td>
<td>34</td>
<td>18</td>
</tr>
<tr>
<td>h264ref</td>
<td>401</td>
<td>71</td>
<td>49</td>
</tr>
<tr>
<td>gobmk</td>
<td>2336</td>
<td>275</td>
<td>24</td>
</tr>
</tbody>
</table>

| Geo. mean | 63.3% | 79.13% | 69.31% |

Table 9. Relative size of GPGs with respect to corresponding procedures in terms of GPBs and basic blocks. The geometric mean has been shown for the percentages of procedures in buckets with the largest numbers.

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

c) Coalescing was most effective for recursive procedures whose GPGs are constructed by repeated inlinings of recursive calls. Once these GPGs were optimized, the GPGs of the caller procedures did not have much scope for coalescing. In other words, coalescing did not cause uniform reduction across all GPGs but helped in the most critical GPGs. Hence we observe a reduction of 20% to 50% of GPBs for some but not majority of procedures. More specifically, the geometric mean of the percentage of procedures that undergo a reduction of less than 20% is 91.6% with a geometric standard deviation of 1.09. Although this number may look small, it should be noted that without coalescing, the GPGs became too large and our implementation failed to scale. In other words, the transitive effect of coalescing is significant and is presented in the discussion on relative sizes of GPGs before and after optimizations for measuring the quality of procedure summaries. In any case coalescing eliminated almost all back edges as shown in the table. This is significant because most of the inlined GPGs are acyclic and hence analyzing the GPGs of the callers does not require additional iterations in a fixed-point computation.

• Quality of procedure summaries. This data is presented in Tables 1, 5, and 7. We use the following quality metrics on procedure summaries:
  (a) Reusability. The number of calls to a procedure is a measure for the reusability of its summary. The construction of a procedure summary is meaningful only if it is use multiple times. From column E in Table 1, it is clear that most procedures are called from many call sites. We counted only the procedures that were called multiple times, ignoring the procedures that have only one call.
  (b) Compactness of a procedure summary. For scalability of a bottom-up approach, a procedure summary should be as compact as possible. In the worst case, a procedure summary
A: Procedure count for different buckets of ratio of GPUs/stmts in CFG and GPG after inlining

B: Procedure count for different buckets of ratio of GPUs/stmts in CFG and optimized GPG

C: Procedure count for different buckets of ratio of GPBs in GPG before and after optimizations

<table>
<thead>
<tr>
<th>Program</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0-20</td>
<td>21-40</td>
<td>41-60</td>
</tr>
<tr>
<td>lbm</td>
<td>16</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td>mcf</td>
<td>21</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>libquantum</td>
<td>75</td>
<td>4</td>
<td>0</td>
</tr>
<tr>
<td>bzip2</td>
<td>89</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>m1c</td>
<td>189</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td>sjeng</td>
<td>131</td>
<td>0</td>
<td>2</td>
</tr>
<tr>
<td>hmmer</td>
<td>273</td>
<td>0</td>
<td>1</td>
</tr>
<tr>
<td>h264ref</td>
<td>540</td>
<td>12</td>
<td>10</td>
</tr>
<tr>
<td>gobmk</td>
<td>2688</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>Geo. mean</td>
<td>95.59%</td>
<td>99.93%</td>
<td>84.14%</td>
</tr>
</tbody>
</table>

Table 10. Relative size of GPGs with respect to corresponding procedures in terms of GPUs and pointer assignments. The geometric mean has been shown for the percentages of procedures in buckets with the largest numbers.

may be same as the procedure. In such a case, the application of a procedure summary at the call sites in its callers is meaningless because it is as good as visiting the procedure multiple times which is similar to a top-down approach.

Tables 5 and 7 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). More specifically, the geometric mean of the percentage of procedures with 0 GPUs across all benchmarks is 77.63% with a geometric standard deviation of 1.18. The geometric mean of the percentages of procedures with 1 to 10 GPUs is 18.08% with a geometric standard deviation of 1.61. Further, the majority of GPGs have 1 to 3 GPBs; the geometric mean of the percentages of such procedures is 95.06%.

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

- 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 (with a cumulative effect of call inlining), (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 7.2). The number of statements in a CFG is measured only in terms of the pointer assignments.

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

- \( x \) is the number of GPBs/GPUs/control flow edges in a GPG obtained after call inlining
- \( y \) is the number of basic blocks/pointer statements/control flow edges in the CFG after call inlining.
A: Proc. count for different buckets of ratio of control flow edges in CFG and GPG after inlining
B: Proc. count for different buckets of ratio of control flow edges in CFG and optimized GPG
C: Procedure count for different buckets of ratio of GPBs in GPG before and after optimizations

<table>
<thead>
<tr>
<th>Program</th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>0-20</td>
<td>21-40</td>
<td>41-60</td>
</tr>
<tr>
<td>lbm</td>
<td>13</td>
<td>4</td>
<td>2</td>
</tr>
<tr>
<td>mcf</td>
<td>21</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>libquantum</td>
<td>61</td>
<td>8</td>
<td>2</td>
</tr>
<tr>
<td>bzip2</td>
<td>72</td>
<td>9</td>
<td>2</td>
</tr>
<tr>
<td>milc</td>
<td>180</td>
<td>3</td>
<td>5</td>
</tr>
<tr>
<td>sjeng</td>
<td>124</td>
<td>5</td>
<td>1</td>
</tr>
<tr>
<td>hmmer</td>
<td>246</td>
<td>24</td>
<td>3</td>
</tr>
<tr>
<td>h264ref</td>
<td>509</td>
<td>26</td>
<td>13</td>
</tr>
<tr>
<td>gobmk</td>
<td>2572</td>
<td>72</td>
<td>31</td>
</tr>
<tr>
<td>Geo. mean</td>
<td>86.13%</td>
<td>99.8%</td>
<td>89.97%</td>
</tr>
</tbody>
</table>

Table 11. Relative size of GPGs with respect to corresponding procedures in terms of control flow edges. The geometric mean has been shown for the percentages of procedures in buckets with the largest numbers.

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

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

We then create five buckets that associate the number of procedures having the computed ratio within a given range. This data is presented in Table 9 (in terms of GPBs and basic blocks), Table 10 (in terms of GPUs and pointer assignments), and Table 11 (in terms of control flow edges) and is described below:

(a) Column A gives the size of the initial GPG (i.e. II) relative to that of the corresponding CFG (i.e. I). It is easy to see that the reduction is immense: a large number of initial GPGs are in the range 0%-20% of the corresponding CFGs. The geometric mean of percentage of procedures in this bucket for relative size in terms of GPUs and pointer assignments across all benchmarks is 95.59% with geometric standard deviation of 1.09 (Table 10). The same number for relative size in terms of GPBs and basic blocks is 63.3% with a geometric standard deviation of 1.2 (Table 9), and those for relative size in terms of control flow edges is 86.13% with a geometric standard deviation of 1.12 (Table 11).

(b) Column B gives the size of the optimized GPG (i.e. III) relative to that of the corresponding CFG (i.e. I). The number of procedures in the range of 0%-20% is larger in this column than in column A indicating more reduction because of optimizations. The geometric mean of percentage of procedures in this bucket for relative size in terms of GPUs and pointer assignments across all benchmarks is a whopping 99.93% with geometric standard deviation of 1.0 (Table 10). The same number for relative size in terms of GPBs and basic blocks is 79.13% with a geometric standard deviation of 1.18 (Table 10), and those for relative size in terms of control flow edges is a whopping 99.8% with a geometric standard deviation of 1.0 (Table 11). This, we believe is the key to the scalability gain of GPG-based points-to-analysis over top-down context-sensitive points-to-analysis.

(c) Column C gives the size of the optimized GPG (i.e. III) relative to that of the initial GPG (i.e. I). Here the distribution of procedures is different for GPBs, GPUs, and control flow edges. In the case of GPBs, the reduction factor is 50%. For GPUs, the reduction varies
widely. The maximum reduction is found for control flow—a large number of procedures fall in the range 0%-20% and the number is larger than in this range for GPBs or GPUs indicating that the control flow is optimized the most. The geometric mean of percentage of procedures in this bucket for relative size in terms of GPUs and pointer assignments across all benchmarks is 84.14% with geometric standard deviation of 1.18 (Table 10). The same number for relative size in terms of GPBs and basic blocks is 69.31% with a geometric standard deviation of 1.23 (Table 9), and those for relative size in terms of control flow edges is 89.97% with a geometric standard deviation of 1.11 (Table 11).

We also measured the effect of control flow minimization on the number of back edges that get removed because fixed point computation requires a larger number of iterations in the presence of back edges. The data in Table 7 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.

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

We compared the points-to information computed by our approach with flow- and context-insensitive (FICI) and flow-insensitive and context-sensitive (FICS) methods. 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. The summary statistics for this were presented in Section 10.2.1.

10.2.3 Time measurements. We measured the overall time (Table 8). We also measured the time taken by the SVF points-to analysis. We observe that our analysis takes less than 16 minutes on gobmk.445 which is a large benchmark with 158 kLoC. Our current implementation does not scale beyond that. SVF is faster than all the variants of points-to analysis that we implemented. This is expected because SVF is context-insensitive.

10.3 Discussion: Lessons From Our Empirical Measurements

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

10.3.1 Handling Large Size of Context-Dependent Information. Some GPGs had a large amount of context-dependent information (i.e. GPUs with upwards-exposed versions of variables) and the GPGs could not be optimized much. This caused the size of the caller GPGs to grow significantly, threatening the scalability of our analysis. Hence, we devised a heuristic threshold \( t \) representing the number of GPUs containing upwards-exposed versions of variables. This threshold is used as follows: Let a GPG contain \( x \) GPUs containing upwards-exposed versions.

- If \( x < t \) for a GPG, then the GPG is inlined in its callers.
- If \( x \geq t \) for a GPG, then the GPG is not inlined in its callers. Instead its calls are represented symbolically with the GPUs containing upwards-exposed versions. As the analysis proceeds, these GPUs are reduced decreasing the count of \( x \) after which the GPG is inlined.
This keeps the size of the caller GPG small and at the same time, allows reduction of the context-dependent GPUs in the calling context. Once all GPUs are reduced to classical points-to edge, we effectively get the procedure summary of the original callee procedure for that call chain. Since the reduction of context-dependent GPUs is different for different calling contexts, the process needs to be repeated for each call chain. This is similar to the top-down approach where we analyze a procedure multiple times. We used a threshold of 80% context-dependent GPUs in a GPG containing more than 10 GPUs. Thus, 8 context-dependent GPUs from a total of 11 GPUs was below our threshold as was 9 context-dependent GPUs from a total of 9 GPUs.

Note that, in our implementation, we discovered very few cases (and only in large benchmarks) where the threshold actually exceeded. The number of call chains that required multiple traversals are in single digits and they are not very long. The important point to note is that we got the desired scalability only when we introduced this small twist of using symbolic GPG.

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

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

11 RELATED WORK: THE BIG PICTURE

Many investigations reported in the literature have described the popular points-to analysis methods and have presented a comparative study of the methods with respect to scalability and precision [14, 15, 17, 24, 36, 39]. Instead of discussing these methods, we devise a metric of features that influence the precision and efficiency/scalability of points-to analysis. This metric can be used for identifying important characteristic of any points-to analysis at an abstract level.

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

Figure 20 presents our metric. At the top level, we have language features and analysis features. The analysis features have been divided further based on whether their primary influence is on the precision or efficiency/scalability of points-to analysis. The categorization of language features is obvious. Here we describe our categorization of analysis features.

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

- Approximations of data dependence. The approaches that compromise on control flow by using flow-insensitivity or context-insensitivity over-approximate the control flow: flow-insensitivity admits arbitrary orderings of statements whereas context-insensitivity treats call and returns as simple goto statements admitting interprocedurally invalid paths. Observe that honoring control flow in imperative languages preserves data dependence and its over-approximation causes over-approximation of data dependence. This may introduce spurious data dependences causing imprecision.
Fig. 20. Language and analysis features affecting the precision, efficiency, and scalability of points-to analyses. An arrow from feature A to feature B indicates that feature A influences feature B. The features influencing precision, influence efficiency and scalability indirectly.

Note that SSA form also discards control flow but it avoids over-approximation in data dependences by creating use-def chains between renamed variables.

- **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:

<table>
<thead>
<tr>
<th>Feature</th>
<th>Examples</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Language</strong></td>
<td></td>
</tr>
<tr>
<td>Data handling</td>
<td>Addressof (&amp;) operator, type casts, unions, dynamic memory allocation, pointer arithmetic, container objects</td>
</tr>
<tr>
<td>Control flow</td>
<td>Function pointers, receiver objects of calls, virtual calls, concurrency</td>
</tr>
<tr>
<td>Higher order features</td>
<td>Reflection, <code>eval</code> in Javascript</td>
</tr>
<tr>
<td><strong>Analysis</strong></td>
<td></td>
</tr>
<tr>
<td>Approximations of data dependence</td>
<td>Path-sensitivity, flow-sensitivity, context-sensitivity, SSA form</td>
</tr>
<tr>
<td>Data abstractions</td>
<td>Field-sensitivity, object-sensitivity, allocation-site-based or type-based abstraction of heap, heap cloning, summarized access paths, summarization of aggregates</td>
</tr>
<tr>
<td>Relevant points-to information</td>
<td>All pointers (exhaustive analysis), relevant pointers in incremental, demand-driven, staged, level-by-level, or liveness-based analyses</td>
</tr>
<tr>
<td>Order of computing points-to information</td>
<td>Governed by relevance of pointers, or by algorithmic features (e.g. top-down, bottom-up, parallel, or randomized algorithms)</td>
</tr>
<tr>
<td>Specialized data structures</td>
<td>BDDs, bloom filters, disjoint sets (for union-find), points-to graphs with placeholders, GPGs</td>
</tr>
</tbody>
</table>
• **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 (i.e., non-address-taken) pointers whose indirections are then eliminated. This uncovers a different set of pointers as top-level pointers whose points-to information is then computed.

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

• **Specialized data structures.** A method may use specialized data structures for encoding information efficiently (e.g., BDDs or GPUs and GPGs) or may use them for 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 20.

• **Data abstraction influences approximations of data dependence.** An abstract location may be over-approximated to represent a larger set of concrete locations in many situations such as in field-insensitivity, type-based abstraction, 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 [47], points-to information is computed one level at a time. The relevant information to be computed at a given level requires points-to information computed by the higher levels. Thus, in this case the relevance of points-to information influences the order of computation. In LFCPA [21] only the live pointers are relevant. Thus, points-to information is computed only when the liveness of pointers is generated. The order of computing liveness influences the relevant points-to information to be computed.

11.1.4 Our Work in the Context of Big Picture of Points-to Analysis. GPG-based points-to analysis preserves data dependence by being flow- and context-sensitive. It is path-insensitive and uses SSA form for non-address-taken local variables. Unlike the approaches that 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 context-insensitive allocation-site-based abstraction for representing heap locations and use \(k\)-limiting for summarizing the unbounded accesses of heap where allocation sites are not known.

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

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

Most of the top-down approaches to flow- and context-sensitive pointer analysis of C programs have not scaled [8, 21, 32] with the largest program successfully analyzed by them consisting of 35 kLoC [21]. It is no surprise then, that the literature of flow- and context-sensitive points-to analyses is dominated by bottom-up approaches. Our work also belongs to this category and hence we focus on them in this section by classifying them into MTF or STF approach (Section 2.3).

11.2.1 MTF Approach for Bottom-Up Summaries. In this approach [16, 44, 47, 48], 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.

Two approaches that stand out among these from the viewpoint of scalability are bootstrapping [16] and Level-by-level analysis [47]. The bootstrapping approach partitions the pointers using flow-insensitive analyses such that each subset is an equivalence class with respect to alias information and then analyses are performed in a cascaded fashion in a series \( A_0, A_1, \ldots, A_k \) where analysis \( A_i \) uses the points-to information computed by the analysis \( A_{i-1} \). Besides, analysis of different equivalence classes at any level can be performed in parallel. This process involves constructing MTF procedure summaries using a top-down traversal of call graph to compute alias information using FSCI analysis. This may cause some imprecision in the computed summaries. However, it is not clear if the precision loss is significant because there are no formal guarantees of precision nor does the paper provide empirical evaluation of precision—the focus being solely on scalability. The analysis is reported to scale to 128kLoC.

Level-by-level analysis [47] 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. This method is claimed to scale to 238kLoC, however, similar to bootstrapping method, there are no formal guarantees or empirical evaluation of precision.

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

Relevant context inference [5] constructs a procedure summary by inferring the relevant potential aliasing between unknown pointees that are accessed in the procedure. Although, it does not...
use the information from the context, it has multiple versions of the summary depending on the alias and the type context. This analysis could be inefficient if the inferred possibilities of aliases and types do not actually occur in the program. It also over-approximates the alias and the type context as an optimization thereby being only partially context-sensitive.

11.2.2 STF Approach for Bottom-Up Summaries. This approach does not make any assumptions about the calling contexts [4, 6, 23, 25–27, 34, 40, 43, 49] and uses multiple placeholders for distinct accesses of the pointees of the same pointer (Section 2.3). This tends to increase the size of the resulting procedure summaries. This problem is mitigated by choosing carefully where the placeholders are required [40, 43], by employing optimizations that merge placeholders [26], by maintaining restricted control flow [4], by over-approximating the control flow through flow-insensitivity [23], or a combination of the above [27]. In some cases, the over-approximation is only in the application of procedure summaries even though they are constructed flow-sensitively [27]. Many of these approaches scale to millions of lines of code.

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

Lattner et al. [23] proposed a heap-cloning based context-sensitive points-to analysis. For achieving a scalable implementation, several algorithmic and engineering design choices were made in this approach. Some of these choices are: a flow-insensitive and unification-based analysis, and sacrificing context-sensitivity across recursive procedures.

Cheng and Mei [6] proposed a modular interprocedural pointer analysis based on access-paths for C programs. They illustrate that access-paths can reduce the overhead of representing context-sensitive transfer functions. The abstraction of access paths is similar to the indirection lists (indlist) used by our approach. The approach uses allocation-site-based abstraction and cycles in the access paths to bound the length of access paths. This approach is flow-insensitive and hence does not maintain any control flow between these access paths.

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

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

11.2.3 The Hybrid Approach. Hybrid approaches use customized summaries and combine the top-down and bottom-up analyses to construct summaries [48]. This choice is controlled by the number of times a procedure is called. If this number exceeds a fixed threshold, a summary is constructed using the information of the calling contexts that have been recorded for that procedure. A new calling context may lead to generating a new precondition and hence a new summary. If the threshold is set to zero, then a summary is constructed for every procedure and hence we have a pure bottom-up approach. If the threshold is set to a very large number, then we have a pure top-down approach and no procedure summary is constructed.

Additionally, we can set a threshold on the size of procedure summary or the percentage of context-dependent information in the summary or a combination of these choices. In our implementation, we used the percentage of context-dependent information as a threshold—when a procedure has a significant amount of context-dependent information, it is better to introduce a small
touch of top-down analysis (Section 10.3.1). If this threshold is set to 0%, our method becomes 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.

These issues have been handled in past as follows: 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 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 control flow minimization. Together, these optimizations reduce the indirection levels of GPUs, eliminate data dependences between GPUs, and significantly reduce control flow. These optimizations also mitigate the impact of the third issue.

We achieved the above by devising novel data flow analyses such as two variants of reaching GPUs analysis and coalescing analysis. Interleaved call inlining and strength reduction of GPGs facilitated a novel optimization that computes flow- and context-sensitive points-to information in the first phase of a bottom-up approach. This obviates the need for the usual second phase.

Our measurements on SPEC benchmarks show that GPGs are small enough to scale fully flow- and context-sensitive exhaustive points-to analysis to C programs as large as 158 kLoC. Our work differs from most other investigations exploring scalable exhaustive flow- and context-sensitive points-to analysis of C in the following ways:

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

Two important takeaways from our empirical evaluation are:

(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 \[21\] for scalability. It would also be useful to extend the scope of the implementation to C++ and Java programs.

The concept of GPG provides a useful abstraction of memory and memory transformers involving pointers by directly 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. In particular, we expect to integrate this method into an in-house Bounded Model Checking infrastructure being developed at IIT Bombay.

ACKNOWLEDGMENTS

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

We would like to thank Akshat Garg and N Venkatesh for empirical measurements of points-to information computed by SVF analysis as also the data for the client analyses. During the course of this work, Pritam Gharat was partially supported by TCS Research Fellowship.

APPENDIX

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

REFERENCES


