Region-based Selective Flow-Sensitive Pointer Analysis

Sen Ye*, Yulei Sui* and Jingling Xue

Programming Languages and Compilers Group
School of Computer Science and Engineering, UNSW Australia

Abstract. We introduce a new region-based SELective Flow-Sensitive (Selfs) approach to inter-procedural pointer analysis for C that operates on the regions partitioned from a program. Flow-sensitivity is maintained between the regions but not inside, making traditional flow-insensitive and flow-sensitive as well as recent sparse flow-sensitive analyses all special instances of our Selfs framework. By separating region partitioning as an independent concern from the rest of the pointer analysis, Selfs facilitates the development of flow-sensitive variations with desired efficiency and precision tradeoffs by reusing existing pointer resolution algorithms. We also introduce a new unification-based approach for region partitioning to demonstrate the generality and flexibility of our Selfs framework, as evaluated using SPEC2000/2006 benchmarks in LLVM.

1 Introduction

Finding a right balance between efficiency and precision lies at the core of pointer analysis. A flow-insensitive analysis (FI), as formulated for C using Andersen's algorithm [2] in Figure 1(a), is fast but imprecise, because it ignores control flow and thus computes a single solution pt to the entire program. Here, $pt(v)$ gives the points-to set of a variable v. In contrast, a flow-sensitive analysis (FS), as formulated by solving a data-flow problem in Figure 1(b), makes the opposite tradeoff. By respecting control flow (`[S-OUTIN]`, `[S-INOUT1]` and `[S-INOUT2]`), separate solutions $pt[\overline{\ell}]$ and $pt[\underline{\ell}]$ at distinct program points $\overline{\ell}$ and $\underline{\ell}$ (the ones immediately before and after each ℓ-labeled statement) are computed and maintained. Preserving flow-sensitivity this way has two precision benefits. One is to track the values read at a location through the control flow. The other is to enable *strong updates*: if a location is definitely updated by an assignment, then the previous values at the location can be killed. In `[S-ADDROF]`, `[S-COPY]` and `[S-LOAD]`, p at $\underline{\ell}$ is strongly updated since $pt[\overline{\ell}](p)$ is killed. For any $q \neq p$, its points-to information is preserved (`[S-INOUT1]`). In `[S-STORE]`, o at $\underline{\ell}$ is strongly updated if o is a singleton, i.e., a concrete location uniquely pointed by p (`[S-STORE`^{SU}`]`) and weakly updated otherwise (`[S-STORE`^{WU}`]`). For a target o' not pointed by p, its points-to information remains unchanged (`[S-INOUT2]`).

* the first and second author contributed equally

[I-ADDROF]	$\dfrac{p = \&o}{\{o\} \subseteq pt(p)}$	[I-COPY]	$\dfrac{p = q}{pt(q) \subseteq pt(p)}$
[I-STORE]	$\dfrac{*p = q \quad o \in pt(p)}{pt(q) \subseteq pt(o)}$	[I-LOAD]	$\dfrac{p = *q \quad o \in pt(q)}{pt(o) \subseteq pt(p)}$

(a) FI: constraints for Andersen's algorithm (flow-insensitive)

[S-ADDROF]	$\dfrac{\ell\colon p = \&o}{\{o\} \subseteq pt[\underline{\ell}](p)}$	[S-COPY]	$\dfrac{\ell\colon p = q}{pt[\overline{\ell}](q) \subseteq pt[\underline{\ell}](p)}$
[S-STORE$^{SU/\boxed{WU}}$]	$\dfrac{\ell\colon *p = q \quad o \in pt[\overline{\ell}](p)}{pt[\overline{\ell}](q) \subseteq pt[\underline{\ell}](o) \quad \boxed{pt[\overline{\ell}](o) \subseteq pt[\underline{\ell}](o)}}$		
[S-LOAD]	$\dfrac{\ell\colon p = *q \quad o \in pt[\overline{\ell}](q)}{pt[\overline{\ell}](o) \subseteq pt[\underline{\ell}](p)}$	[S-OUTIN]	$\dfrac{v \in \mathcal{V} \quad \ell' \in succ(\ell)}{pt[\underline{\ell}](v) \subseteq pt[\overline{\ell'}](v)}$
[S-INOUT1]	$\dfrac{\ell\colon p = _ \quad q \neq p}{pt[\overline{\ell}](q) \subseteq pt[\underline{\ell}](q)}$	[S-INOUT2]	$\dfrac{\ell\colon *p = _ \quad o' \notin pt[\overline{\ell}](p)}{pt[\overline{\ell}](o') \subseteq pt[\underline{\ell}](o')}$

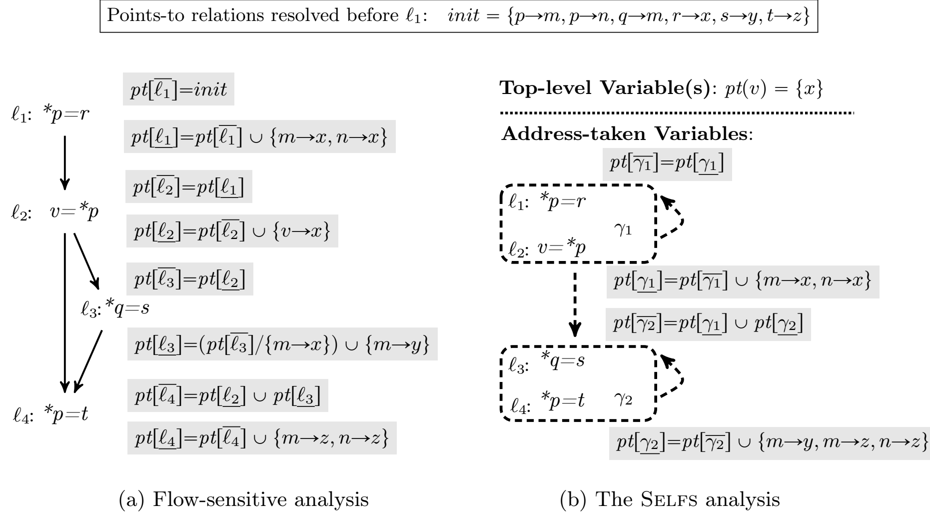
$\mathcal{V}$: set of variables   *succ*: mapping statements to control flow successors

(b) FS: constraints for data-flow (flow-sensitive)

Fig. 1. Two traditional pointer analyses, FI and FS, for C programs.

Flow-sensitivity is beneficial for a wide range of clients such as bug detection [22, 30, 31, 34], program verification [10, 11] and change impact analysis [1, 4]. As the size and complexity of software increases, how to achieve flow-sensitivity exactly or approximately with desired efficiency and precision tradeoffs becomes attractive. The "sparse" approach [14, 23, 35] aims to achieve the same precision as FS but more scalably. The basic idea is to first over-approximate the points-to information in a program with a fast but imprecise pre-analysis and then refine it by propagating the points-to facts *sparsely* only along the pre-computed def-use chains instead of across all program points as FS does. Alternatively, the "strong-update" approach [21] sacrifices the precision of FS in order to gain better efficiency. The basic idea is to proceed flow-sensitively to perform the same strong updates as in FS but falls back to FI otherwise. Despite these recent advances on flow-sensitive analysis, balancing efficiency and precision remains challenging, partly due to the difficulty in orchestrating various algorithms used during the analysis and partly due to the desire to meet different clients' needs.

A program usually exhibits diverse characteristics in its different code regions, which should be handled with different efficiency and precision tradeoffs (to avoid under- or over-analysing). In this paper, we propose a new region-based SELective Flow-Sensitive (SELFS) approach to pointer analysis for C that operates on the regions partitioned from a program rather than individual statements as in [14, 35]. Top-level pointers can be put in SSA form [8] without requiring pointer analysis. To track the value-flows of address-taken variables effectively, we will perform a pre-analysis to enable their sparse analysis as in [14, 23, 35],



(a) Flow-sensitive analysis

(b) The SELFS analysis

Fig. 2. An illustration of SELFS performed on regions γ_1 and γ_2 by preserving the precision of FS with respect to the reads from variables (with further details given in Figure 3 and Examples 1 – 3). The focus is on analysing the points-to relations for the top-level variable v and the two address-taken variables m (a singleton) and n, by assuming that the points-to relations in *init* are given. Here, $pt[\overline{\gamma}]$ ($pt[\underline{\gamma}]$), where γ is a region, is an analogue of $pt[\overline{\ell}]$ ($pt[\underline{\ell}]$), where ℓ is a statement.

but on a region graph with its regions containing loads and stores. Each region is analysed flow-insensitively but flow-sensitivity is maintained across the regions.

Consider Figure 2, where the points-to relations in *init* are known before the code is analysed. Figure 2(a) depicts the points-to relations obtained by applying FS to the code. Note that a strong update on m (assumed to be a singleton) is performed at ℓ_3. Figure 2(b) gives the solution obtained by SELFS on a region graph (with two regions γ_1 and γ_2) to achieve more efficiently the same precision for reads from (but not necessarily for writes into) each variable. As p points to m and n, no strong update is possible at ℓ_1. Instead of flow-sensitively propagating the points-to relations from ℓ_1 to ℓ_2, both can be analysed flow-insensitively in region γ_1 without any precision loss for the reads from m and n at ℓ_2. Interestingly, even if a strong update is performed for m at ℓ_3, the points-to relations from ℓ_2 and ℓ_3 are merged on entry of ℓ_4, making any read from m at ℓ_4 (if any) as precise as if ℓ_3 and ℓ_4 are analysed together in γ_2 flow-insensitively. Note that SELFS has over-approximated the potential target of m at ℓ_3: $m{\to}y$ found by FS in Figure 2(a) with $m{\to}x$, $m{\to}y$ and $m{\to}z$ given in Figure 2(b). As argued in Section 3, preserving the precision for reads from all variables always preserves the alias information (among others). By operating at the granularity of regions rather than statements while maintaining flow-sensitivity across their edges (illustrated further in Figure 3), SELFS is expected to run faster.

Our SELFS analysis is also advantageous in that region partitioning is separated as an independent concern from the rest of the analysis. Different region partitions may lead to different degrees of flow-sensitivity, resulting in different efficiency and precision tradeoffs being made. As discussed in Section 3, the two traditional analyses, FI and FS, given in Figure 1 and some recent sparse flow-sensitive analyses [14, 23, 35] are all special instances of SELFS. As a result, SELFS provides a general framework for designers to develop and evaluate different flow-sensitive variations by reusing existing pointer resolution algorithms.

This paper makes the following contributions:

- We present SELFS that performs inter-procedural flow-sensitive pointer analysis across but not inside the regions partitioned from a C program, allowing different efficiency and precision tradeoffs to be made subject to different region partitioning strategies used (Section 2).
- We introduce a new unification-based region partitioning approach that enables SELFS to achieve nearly the same precision as FS for almost all practical purposes (Section 3) and discuss some heuristics for trading precision for efficiency in future work (Section 6).
- We have implemented SELFS in LLVM (version 3.3) and evaluated it using a total of 14 benchmarks selected from SPEC2000 and SPEC2006 (Section 4). SELFS can accelerate a state-of-the-art sparse yet precision-preserving version [14] of FS by 2.13X on average while maintaining the same precision for reads from variables, i.e., for all alias queries. In addition, the best speedups are observed at `h264ref` (7.45X) and `mesa` (6.08X).

2 The SELFS Analysis Framework

In this section, we present our SELFS analysis on a given region graph created from a program. In the next section, we describe some region partitioning strategies. Section 2.1 makes precise the canonical representation used for analysing C programs. Section 2.2 defines the region graphs operated on by SELFS. Section 2.3 formalises our region-based flow-sensitive pointer analysis.

2.1 Canonical Representation

In the pointer analysis literature, a C program is represented by a CFG (Control-Flow Graph) containing the four types of pointer-manipulating statements shown in Figure 1: $p = \&o$ (ADDROF), $p = q$ (COPY), $p = *q$ (LOAD) and $*p = q$ (STORE). More complex statements are decomposed into these basic ones. Passing arguments into and returning results from functions are modeled by copies. For a given ADDROF statement $p = \&o$, o is either a stack variable with its address taken or an abstract object dynamically created at an allocation site.

For simplicity, we adopt the convention of LLVM by separating the set $\mathcal{V}$ of all variables into two subsets, (1) $\mathcal{A}$ containing all possible targets, i.e., *address-taken variables* of a pointer and (2) $\mathcal{T}$ containing all *top-level variables*, where $\mathcal{V} = \mathcal{T} \cup \mathcal{A}$. For the four types of statements given, we have $p, q \in \mathcal{T}$ and $o \in \mathcal{A}$.

2.2 Region Graph

Our Selfs analysis operates on a region graph created from a program being analysed. Leveraging recent progress on sparse flow-sensitive analysis [14, 23, 35], we will perform a pre-analysis to both guide region partitioning and enable sparse analysis at the granularity of regions rather than individual statements.

To obtain a region graph from a program, top-level and address-taken variables are treated differently. In our Selfs framework, top-level variables are always analysed sparsely since they can be put in SSA without requiring pointer analysis. A top-level pointer that is defined multiple times is split into distinct versions after SSA conversion. All versions of a variable, say $q_{i_1}, \ldots, q_{i_n}$, that reach a joint point at the CFG are combined by introducing a new Phi statement, $q_j = \phi(q_{i_1}, \ldots, q_{i_n})$, where $q_j, q_{i_1}, \ldots, q_{i_n} \in \mathcal{T}$, so that every version is defined once (statically). After SSA conversion, the *(direct) def-use chains* for all top-level variables are readily available. As a result, their points-to sets can be simply obtained flow-sensitively by performing a flow-insensitive analysis.

Unlike top-level variables, address-taken variables are read/written indirectly via top-level pointers at loads/stores and thus harder to analyse sparsely. Sparsity requires points-to information to be propagated along def-use chains but the *(indirect) def-use chains* for address-taken variables can only be computed using points-to information. To break the cycle, we perform a pre-analysis as in [7, 14, 23, 30] to first over-approximate indirect def-use chains and then refine them by performing a data-flow analysis sparsely along such pre-computed def-use chains.

Note that an address-taken variable o accessed at a store represents a potential use of o if a weak update is performed ([S-STORE^{WU}] in Figure 1(b)). Due to space limitation, we refer to [14] on how to over-approximate indirect def-use chains (via a pre-analysis named Aux). The basic idea is to annotate a load $p = *q$ with a potential use of o for every o pointed by q and a store $*p = q$ with a potential use and def of o for every o pointed by p. Then indirect def-use chains can be built by putting all address-taken variables in SSA.

Therefore, Selfs keeps track of value flows for top-level variables in SSA explicitly along their (direct) def-use chains and refines value flows for address-taken variables along their pre-computed (indirect) def-chains in a region graph.

Definition 1 (Region Graph). A *region graph* $\mathcal{G}_{\text{rg}} = (\mathcal{N}_{\text{rg}}, \mathcal{E}_{\text{rg}})$ for a program is a multi-edged directed graph. $\mathcal{N}_{\text{rg}}$ is a partition of the set of its loads and stores into regions. $\mathcal{E}_{\text{rg}}$ contains an edge $\gamma_1 \xrightarrow{o} \gamma_2$ labeled by an address-taken variable $o \in \mathcal{A}$ from γ_1 to γ_2, where γ_1 and γ_2 may be identical, if there is an indirect def-use chain for o from γ_1 to γ_2 computed by the pre-analysis.

Example 1. Consider our example again in Figure 3. Figure 3(b) duplicates the region graph from Figure 2(b) except that its edges are now annotated explicitly with address-taken variables indicating their value flows. By Definition 1, these edges are added based on the statement-level indirect def-use chains obtained by pre-analysis, given in Figure 3(a). The presence of self-loop edge(s) in a region allows naturally the loads/stores inside to be analysed flow-insensitively. □

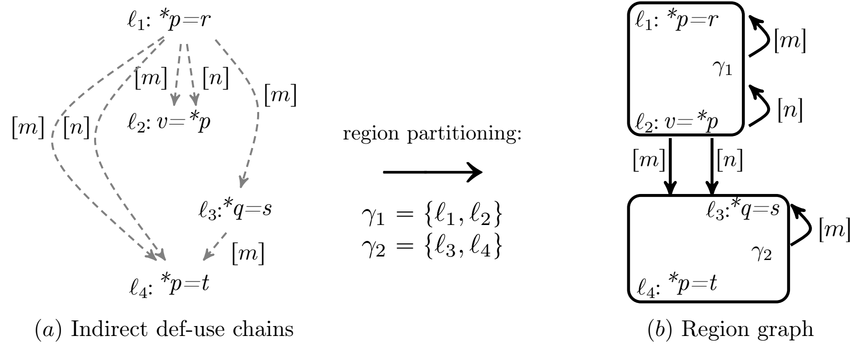


Fig. 3. The region graph in Figure 2(b) redrawn with all indirect def-use edges made explicit. The pre-analysis yields $p \rightarrow m$, $p \rightarrow n$ and $q \rightarrow m$ (included in *init* in Figure 2).

2.3 Region-Based Flow-Sensitivity

Figure 4 gives the inference rules used in our SELFS framework. Top-level variables are analysed as before except that they are now in SSA (`[R-PHI]`). Therefore, analysing the top-level variables in SSA flow-insensitively (`[R-ALLOC]` and `[R-COPY]`) as in FI gives rise to the flow-sensitive precision obtained as in FS. As a result, the points-to sets $pt(p)$ and $pt(q)$ of top-level pointers p and q are directly read off at a load (`[R-LOAD]`), a store (`[R-STORE]`), and in `[R-INOUT]`.

SELFS computes and maintains the points-to relations for address-taken variables sparsely in a region graph. Flow-sensitivity is maintained across the regions (along their indirect region-level def-use edges) but not inside. This implies that all statements in a region γ are handled flow-insensitively if $|\gamma| > 1$ and flow-sensitively otherwise (i.e., if $|\gamma| = 1$). As SELFS operates at the granularity of regions, the notation $pt[\overline{\gamma}]$ ($pt[\underline{\gamma}]$) for a region γ is an analogue of $pt[\overline{\ell}]$ ($pt[\underline{\ell}]$) for a statement ℓ, as already illustrated in Figure 2(b). For a region γ containing multiple statements, $pt[\underline{\gamma}]$ is the solution for all program points inside the region.

Below we explain the four rules, `[R-DU]`, `[R-INOUT]`, `[R-LOAD]` and `[R-STORE]`, used to compute the points-to relations for address-taken variables.

Together with `[R-INOUT]`, `[R-DU]` represents the sparse propagation of points-to relations for address-taken variables $o \in \mathcal{A}$ across their pre-computed def-use chains at the granularity of regions. In contrast, FS propagates such points-to relations blindly across the control flow (`[S-OUTIN]`, `[S-INOUT1]` and `[S-INOUT2]`).

In `[R-LOAD]`, $\gamma = \texttt{selR}(\ell)$ is the region where the load at ℓ resides. No strong update is possible even if γ contains a store since $|\gamma| > 1$ will then hold. Regardless of how many statements that γ contains, the points-to set of o at the entry of γ is propagated into the points-to set of p: $pt[\overline{\gamma}](o) \subseteq pt(p)$, where $pt[\overline{\gamma}](o)$ contains the points-to relations that are (1) either received from its predecessors (`[R-DU]`) or (2) generated inside γ (due to a self-loop edge labeled by o when $|\gamma| > 1$), in which case, all statements inside γ are analysed flow-insensitively.

$$\text{[R-ADDROF]}\ \frac{\ell\colon p = \&o}{\{o\} \subseteq pt(p)} \quad \text{[R-COPY]}\ \frac{\ell\colon p = q}{pt(q) \subseteq pt(p)} \quad \text{[R-PHI]}\ \frac{\ell\colon p = \phi(_, q, _)}{pt(q) \subseteq pt(p)}$$

$$\text{[R-STORE}^{SU/\boxed{WU}}\text{]}\ \frac{\ell\colon *p = q \quad o \in pt(p) \quad \gamma = \texttt{selR}(\ell)}{pt(q) \subseteq pt[\underline{\gamma}](o) \qquad \boxed{pt[\overline{\gamma}](o) \subseteq pt[\underline{\gamma}](o)}}$$

$$\text{[R-LOAD]}\ \frac{\ell\colon p = *q \quad o \in pt(q) \quad \gamma = \texttt{selR}(\ell)}{pt[\overline{\gamma}](o) \subseteq pt(p)} \qquad \text{[R-DU]}\ \frac{o \in \mathcal{A} \quad \gamma \xrightarrow{o} \gamma'}{pt[\underline{\gamma}](o) \subseteq pt[\overline{\gamma'}](o)}$$

$$\text{[R-INOUT]}\ \frac{\gamma \in \mathcal{N}_{\text{rg}} \quad o' \notin \{o \in pt(p) \mid (*p = q) \in \gamma\}}{pt[\overline{\gamma}](o') \subseteq pt[\underline{\gamma}](o')}$$

Fig. 4. Inference rules for SELFS (with top-level variables in SSA).

[R-STORE] is similar except that the points-to set of o indirectly accessed at a store is updated at the end of the region γ that contains the store. [R-STORE^{SU}], which is explained earlier in Section 1, comes into play only when $|\gamma| = 1$. Recall that SELFS only analyses single-statement regions flow-sensitively.

Example 2. Let us apply our inference rules to the region graph in Figure 3(b) to obtain the points-to relations given in Figure 2(b). As $|\gamma_1| = |\gamma_2| = 2$, [R-STORE^{SU}] cannot be applied. When γ_1 is processed, applying [R-STORE^{WU}] to ℓ_1 adds $m \to x$ and $n \to x$ to $pt[\underline{\gamma_1}]$ and applying [R-LOAD] to ℓ_2 yields $pt(v) = x$. Applying [R-DU] to the two self-loop edges $[m]$ and $[n]$ around γ_1 gives rise to $pt[\overline{\gamma_1}] = pt[\underline{\gamma_1}]$. Applying [R-DU] to the two edges $[m]$ and $[n]$ from γ_1 to γ_2, we obtain $pt[\overline{\gamma_2}] = \{m \to x, n \to x\}$.

When γ_2 is analysed, [R-STORE^{WU}] is applied to each store. So the points-to relations in $pt[\overline{\gamma_2}]$ are preserved in $pt[\underline{\gamma_2}]$. Then we add $m \to y$ generated at ℓ_3 and $m \to z$ and $n \to z$ at ℓ_4 to $pt[\underline{\gamma_2}]$. Next, applying [R-DU] to the self-loop $[m]$ on γ_2 causes $m \to y$ and $m \to z$ to be added to $pt[\overline{\gamma_2}]$. Finally, we obtain $pt[\overline{\gamma_2}] = \{m \to x, m \to y, m \to z, n \to x\}$ and $pt[\underline{\gamma_2}] = pt[\overline{\gamma_2}] \cup \{n \to z\}$. □

Theorem 1 (Soundness). SELFS *is sound if the pre-analysis used is.*

Proof Sketch. A sound pre-analysis over-approximates the indirect def-use chains used for constructing the edges in a region graph $\mathcal{G}_{\text{rg}}$. Essentially, SELFS combines FI and FS to refine such pre-computed def-use chains flow-sensitively ([R-DU] and [R-INOUT]) by performing strong updates ([R-STORE^{SU}]). □

Theorem 2 (Precision). *Suppose* FI *is used as the pre-analysis, Then* SELFS *lies between* FI *and* FS *in terms of precision.*

Proof Sketch. We can show that SELFS is no more precise than FS (with respect to each variable's points-to set) by observing the following facts: (1) performing the pre-analysis with FI gives rise to over-approximated indirect def-use chains (Theorem 1), (2) both analyses handle top-level pointers in exactly the same way

except that Selfs does it sparsely in SSA (`[R-ADDROF]`, `[R-COPY]` and `[R-PHI]`) and FS takes a data-flow approach (`[S-ADDROF]` and `[S-COPY]`), and (3) Selfs applies FS only to a region that contains one load or one store and FI to handle the remaining regions flow-insensitively. Thus, for every variable, the points-to set obtained by Selfs is no smaller than that obtained by FS.

To see that Selfs is no less precise than FI, we note that Selfs works by refining the points-to sets produced by FI (as the pre-analysis) through performing strong updates and maintaining inter-region flow-sensitivity. □

Finally, some prior representative analyses are special instances of Selfs, with the following changes made to Selfs (mainly to region partitioning):

FI in Figure 1(a): All loads and stores are in one region (and top-level variables are not in SSA if they are to be analysed also flow-insensitively).
[14]: Each region contains one load or one store (same precision as FS).
FS in Figure 1(b): Each region contains one statement and each inter-region edge represents control flow, labeled by all variables.
[21]: Each store is in its own region if it can be strongly updated and all the other stores and all the loads are in another region (less precise than FS).

3 Instantiating the Selfs Analysis

Selfs is sound (Theorem 1) and can easily achieve a precision between FI and FS on an arbitrary region graph $\mathcal{G}_{\text{rg}}$ (Theorem 2). Ideally, we should use a region graph $\mathcal{G}_{\text{rg}}$ that allows Selfs to attach the precision of FS at the efficiency of FI.

In this section, we introduce a new unification-based approach that allows Selfs to preserve the precision of FS with respect to the reads from all variables, thus making it nearly as precise as FS in practice. We also discuss how to relax this so-called strict load-precision-preserving approach to tolerate some precision loss in future work in Section 6. Our focus is on demonstrating the generality and flexibility of Selfs in allowing efficiency and precision tradeoffs to be made subject to region partitioning strategies used.

3.1 Load-Precision-Preserving Partitioning

As discussed in Section 2, Selfs degenerates into the sparse analysis [14] if Selfs operates on a region graph, denoted $\mathcal{G}_{\text{one}}$, such that each of its regions contains one load or one store. In this important special case, Selfs is significantly faster than FS while achieving the same precision as FS, but can still be costly for large programs, especially when field-sensitivity is considered. By merging small regions into larger ones, Selfs can run faster at some possible precision loss.

We introduce a partitioning strategy that works by unifying two regions into a larger one successively, starting from any given region graph, say $\mathcal{G}_{\text{one}}$. Some unification steps are applied online if they require the knowledge about whether a strong or weak update is performed at a store and some can otherwise be applied offline. Our rules are *load-precision-preserving* in the sense that every

load behaves identically before and after each unification. Thus, for every load $\cdots = *q$, the points-to set of every target o pointed by q remains unchanged.

For almost all practical purposes, making all the loads precise is as good as making SELFS as precise as FS. First of all, the points-to set of every top-level pointer remains unchanged (since it is overwritten from either an ADDROF, a LOAD statement, or possibly via a sequence of COPY or PHI assignments). Thus, the precision for reads of q in $\cdots = q$ and $\cdots = *q$ is preserved. In addition, the alias information remains unchanged. This is because in our LLVM-like canonical representation, all aliases must be tested between top-level pointers. Similarly, all function pointers are reserved in the same way as they are all top-level.

However, some stores can be imprecise, but only when they are not read from later, as is the case of $*q = s$ illustrated in Figure 2(b) and revisited later in Example 3. Such stores are dead code and can thus be eliminated.

3.2 Unification

The following lemma gives a sufficient condition to make SELFS load-precision-preserving and motivates the development of our unification approach.

Lemma 1. *Let $\mathcal{G}_{\text{rg}}$ be a region graph with its two regions identified by γ_i and γ_j. Let $\mathcal{G}'_{\text{rg}}$ the resulting graph after γ_i and γ_j are unified (i.e., merged) into a new region $\gamma_{i,j}$. Let L be the set of all regions in $\mathcal{G}_{\text{rg}}$ such that each contains at least one load. Let L' be similarly defined for $\mathcal{G}'_{\text{rg}}$. Let $\pi : L \mapsto L'$ be defined such that $\pi(\gamma) = \gamma$ if $\gamma \notin \{\gamma_i, \gamma_j\}$ and $\pi(\gamma) = \gamma_{i,j}$ otherwise. If $\forall \gamma \in L : \text{pt}[\overline{\gamma}] = \text{pt}[\overline{\pi(\gamma)}]$ before and after the unification, then SELFS is load-precision-preserving.*

Proof. No strong update can be performed in a region γ that contains a load since that would imply $|\gamma| > 1$. For every region $\gamma \in L$, if $pt[\overline{\gamma}] = pt[\overline{\pi(\gamma)}]$, then $pt[\overline{\gamma}][o] = pt[\overline{\pi(\gamma)}][o]$ for every load $p = *q$ that appears in both γ and $\pi(\gamma)$, where $o \in pt(q)$ (`[R-LOAD]`). So SELFS is load-precision-preserving. □

This lemma is expensive to apply during the analysis. Guided by the basic idea behind, we have developed a conservative but simple unification approach. Each unification step operates on a small neighbourhood of the two regions being unified. Our approach is promising as it can be relaxed to allow different efficiency and precision tradeoffs to be made, as discussed in Section 6.

Definition 2 (Region Types). Given a region γ, $\tau(\gamma) = \mathsf{S}$ if a strong update can be performed inside (implying that γ contains a single store), $\tau(\gamma) = \mathsf{W}$ if a weak update can be performed inside, and $\tau(\gamma) = \mathsf{L}$ if γ contains loads only.

When unifying a region γ with another region in a region graph $\mathcal{G}_{\text{rg}} = (\mathcal{N}_{\text{rg}}, \mathcal{E}_{\text{rg}})$, we can identify the *potential* points-to relations generated by γ and the *potential* uses for the points-to relations generated by the two regions being unified directly from $\mathcal{G}_{\text{rg}}$. Below we write *rsucc* (*rpred*) to relate a region to its set of successors (predecessors) in a region graph.

- $GEN(\gamma) = \{o \mid \gamma \xrightarrow{o} \gamma', \gamma' \in \mathcal{N}_{\mathrm{rg}}\}$ contains the address-taken variables potentially defined in γ ([R-DU]), which implies $GEN(\gamma) = \varnothing$ if $\tau(\gamma) = \mathsf{L}$.
- $USE(\gamma) = \{\gamma' \mid \gamma' \in rsucc(\gamma)\} \cup \{\gamma \mid \gamma \text{ contains a load}\}$ gives the set of potential uses for the points-to relations generated by γ and the region to be unified together. Note that $rsucc(\gamma) \supseteq \{\gamma\}$ if γ contains a store that produces values used by some other stores or some loads also contained in γ (Figure 3).
- $PRD(\gamma) = rpred(\gamma)$ gives the set of all potential defs for the points-to relations used in γ.

When merging two regions, we need to reason about the value flows for the address-taken variables potentially defined inside these two regions.

Definition 3 (Value-Flow Reachability). Let γ_i and γ_j be two regions in $\mathcal{G}_{\mathrm{rg}} = (\mathcal{N}_{\mathrm{rg}}, \mathcal{E}_{\mathrm{rg}})$. We say that γ_j is *o-reachable* from γ_i, where $o \in \mathcal{A}$, and write $\gamma_i \xhookrightarrow{o} \gamma_j$ if there is either (1) an edge $\gamma_i \xrightarrow{o} \gamma_j \in \mathcal{E}_{\mathrm{rg}}$ (*directly reachable*) or (2) a path $\gamma_i \xrightarrow{o} \gamma_1, \ldots, \gamma_n \xrightarrow{o} \gamma_j$ in $\mathcal{G}_{\mathrm{rg}}$ (*indirectly reachable*) via one or more regions, $\gamma_1, \ldots, \gamma_n$, where $\tau(\gamma_k) = W$, for weakly updating o.

When two regions γ_i and γ_j are unified, all loads and stores in the resulting region are resolved flow-insensitively (since $|\gamma_i \cup \gamma_j| > 1$), even though a strong update is possible in either region before. The points-to relations flowing into both γ_i and γ_j from $PRD(\gamma_i)$ and $PRD(\gamma_j)$ are merged, preserved and propagated together with the points-to relations generated inside γ_i and γ_j to their uses in $USE(\gamma_i)$ and $USE(\gamma_j)$. In order to preserve the precision for loads, we can ensure conservatively that the same propagation happens before and after each unification (for loads). The presence of a strong update in γ_i or γ_j can be unification-preventing only when the killed values in γ_i or γ_j cannot already reach their uses in $USE(\gamma_i)$ and $USE(\gamma_j)$ before the unification.

Let us introduce some notational shorthand, where $R, R' \subseteq \mathcal{N}_{\mathrm{rg}}$ and $O \subseteq \mathcal{A}$:

$$R \xhookrightarrow{O} R' =_{\mathrm{def}} \forall o \in O : \forall \gamma \times \gamma' \in R \times R' : \gamma \xhookrightarrow{o} \gamma'$$

Theorem 3 (Load-Precision-Preserving Unification). *Unifying γ_i and γ_j will make* SELFS *load-precision-preserving if all the following conditions hold:*

C1 $\{\gamma_i\} \xhookrightarrow{GEN(\gamma_i)} USE(\gamma_j)$;
C2 $PRD(\gamma_i) \xhookrightarrow{GEN(\gamma_i)} USE(\gamma_i) \cup USE(\gamma_j)$;
C3 $\{\gamma_j\} \xhookrightarrow{GEN(\gamma_j)} USE(\gamma_i)$; *and*
C4 $PRD(\gamma_j) \xhookrightarrow{GEN(\gamma_j)} USE(\gamma_i) \cup USE(\gamma_j)$.

Proof Sketch. By unifying γ_i and γ_j, only the points-to relations reaching the regions in $USE(\gamma_i) \cup USE(\gamma_j)$ are affected. As is standard, SELFS is monotonic, implying that no strong update at a store is possible after the store has been weakly updated. Therefore, any indirect reachable path (Definition 3), once established, will remain unchanged. For reasons of symmetry, let us consider C1 and C2 only. C1 says that whatever γ_i generates (along its def-use edges) must

be used not only by $USE(\gamma_i)$ (by construction) but also by $USE(\gamma_j)$. C2 says that even if some values are killed in γ_i due to a strong update, the killed values will still reach both $USE(\gamma_i)$ and $USE(\gamma_j)$ via a different path (without going through γ_i), rendering the values non-killable (effectively).

Let π be defined in Lemma 1. If C1 – C4 hold, then $\forall \gamma \in USE(\gamma_i) \cup USE(\gamma_j)$: $pt[\overline{\gamma}] = pt[\overline{\pi(\gamma)}]$. By Lemma 1, SELFS is load-precision-preserving. □

Example 3. Let us apply Theorem 3 to the example given in Figure 3, assuming initially that each statement is in its own region: $\gamma_1 = \{\ell_1\}$, $\gamma_2 = \{\ell_2\}$, $\gamma_3 = \{\ell_3\}$ and $\gamma_4 = \{\ell_4\}$. Let us try to unify γ_1 and γ_2. According to the region graph given in Figure 3(a), we have $GEN(\gamma_1) = \{m, n\}$, $GEN(\gamma_2) = \varnothing$. $USE(\gamma_1) = \{\gamma_2, \gamma_3, \gamma_4\}$, $USE(\gamma_2) = \{\gamma_2\}$, $PRD(\gamma_1) = \varnothing$ and $PRD(\gamma_2) = \{\gamma_1\}$. By Theorem 3, C1 – C4 are satisfied. So γ_1 and γ_2 are unifiable. We can also choose to unify γ_3 and γ_4 instead. Then $GEN(\gamma_3) = \{m\}$, $GEN(\gamma_4) = \varnothing$. $USE(\gamma_3) = \{\gamma_4\}$, $USE(\gamma_4) = \varnothing$, $PRD(\gamma_3) = \{\gamma_1\}$ and $PRD(\gamma_4) = \{\gamma_1, \gamma_3\}$. Note that $GEN(\gamma_4) = \varnothing$ because there are no outgoing def-use chains from ℓ_4. Again, C1 – C4 are satisfied, making γ_3 and γ_4 unifiable. By proceeding in either order, we will obtain the region graph shown in Figure 3(b).

Note that unifying γ_3 and γ_4 makes SELFS lose the precision at ℓ_3 as explained earlier. However, in this example, both ℓ_3 and ℓ_4 are dead code. If we add a load $\ell_5 : w = *q$ immediately after ℓ_3, which is in a new region $\gamma_5 = \{\ell_5\}$, there will be a new indirect def-use $\ell_3 \xrightarrow{m} \ell_5$ in Figure 3(a). In this case, γ_3 and γ_4 are no longer unifiable since $USE(\gamma_3) = \{\gamma_4, \gamma_5\}$. By treating γ_3 as γ_i in Theorem 3, C2 is violated since there is only one path from γ_1 to γ_5: $\gamma_1 \xrightarrow{m} \gamma_3 \xrightarrow{m} \gamma_5$, where a strong update is performed on m in γ_3. So γ_5 is not m-reachable from γ_1. In fact, merging γ_3 and γ_4 will cause the load $\ell_5 : w = *q$ to lose precision since the store at ℓ_3 will only be weakly updated afterwards. □

Figure 5 illustrates our unification approach further, by assuming that all indirect def-use chains are related to one address-taken variable, o. In Figure 5(d), if W_3 is S_3 (with a strong update to o inside), then the unification cannot be performed as L_{45} is not o-reachable from S_1 (the predecessor of S_2 and S_3). Otherwise, L_{45} may receive spurious points-to relations propagated from S_1. In Figure 5(h), S_1 and W_{2345} cannot be unified further because the predecessors of S_1 reach W_{2345} via only S_1 (where a strong update to o is performed).

4 Evaluation

We demonstrate the effectiveness of SELFS under our unification-based region partitioning strategy. The baseline is a state-of-the-art sparse yet precision-preserving version [14], denoted SFS, of FS given in Figure 1(b). We have selected 14 large C programs (totalling 672 KLOC) from SPEC CPU2000/CPU2006, with their characteristics given in Figure A.1. Our platform is a 2.00GHz Intel Xeon 32-core CPU running Ubuntu Linux with 64GB memory.

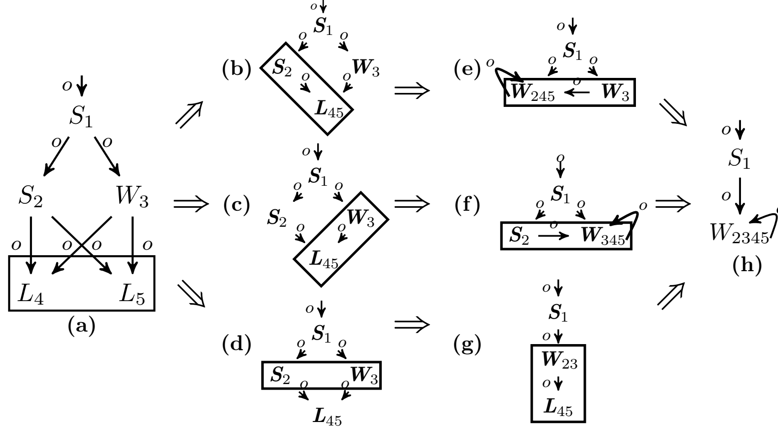


Fig. 5. Some possible unification sequences illustrated for a part of a region graph, by assuming that all indirect-use edges are related to one single address-taken variable, o. The type of a region is identified by S, W or L (Definition 2).

4.1 Methodology

As discussed in Section 2, SFS works on a region graph such that each region contains one single load or store. To apply our load-precision-preserving partitioning in SELFS, we start with a region graph such that each load or store is in its own region. We then apply our unification-based approach to form larger regions. As a result, SELFS is load-precision-preserving (Theorem 3), resulting in the same precision for alias queries as SFS (among others). We repeat a process of picking a region randomly and trying to unify it with one of its predecessors, successors, and siblings in that order until no more unification is possible.

For efficiency reasons, we verify the four conditions in Theorem 3 during the SELFS analysis by restricting ourselves to the o-reachable paths (Definition 3) such that each of its intermediate nodes is one of the two regions to be unified. As a result, starting with Figure 5(a), we will accept Figures 5(c) and (d) but reject Figure 5(b). Finally, some unification steps are performed offline rather than online during the analysis if they do not require the knowledge about whether a store in a region (containing that store only) can be strongly updated or not.

4.2 Implementation

We have implemented SELFS in LLVM (version 3.3). The source files of each benchmark are compiled into bit-code files using clang and then linked together using llvm-link, with mem2reg being applied to promote memory into registers. We use FI, i.e., Andersen's analysis (using the constraint resolution techniques from [25, 27]) as pre-analysis for building indirect def-use chains [14, 30, 31].

Program	Size KLOC	Analysis Times (Secs)			Selfs' Regions (of L, S and W Types)				
		SFS	Selfs	Speedup	#L	#S	#W	#Avg	#Max
ammp	13.4	0.31	0.31	1.00	538	0	187	1.53	16
crafty	21.2	0.30	0.31	0.97	377	0	328	1.95	276
gcc	230.4	826.34	408.93	2.02	10690	294	6867	2.23	401
h264ref	51.6	40.84	5.48	7.45	3523	159	1460	1.94	128
hmmer	36.0	0.42	0.52	0.81	1170	59	487	1.59	56
mesa	61.3	1096.63	180.37	6.08	3801	0	2211	1.40	50
milc	15.0	0.28	0.26	1.08	566	8	253	1.69	66
parser	11.4	3.80	2.31	1.65	820	11	931	1.47	110
perlbmk	87.1	407.86	143.25	2.85	7514	513	4451	1.78	189
sjeng	13.9	0.07	0.19	0.37	463	0	524	1.50	14
sphinx3	25.1	1.16	1.23	0.94	953	14	598	1.98	42
twolf	20.5	1.07	1.02	1.05	1798	1	494	3.51	184
vortex	67.3	86.01	37.92	2.27	2369	198	2061	2.11	830
vpr	17.8	0.30	0.27	1.11	768	0	305	2.36	39

Fig. 6. Comparing SFS and Selfs.

Selfs is field-sensitive. Each field of a struct is treated as a separate object, but arrays are considered monolithic. Positive weight cycles (*PWC*s) that arise from processing fields of struct objects are detected and collapsed [24]. Distinct allocation sites are modeled by distinct abstract objects as in [14, 30, 31].

We have implemented SFS differently from that in [14]. In this paper, a program's call graph is built on the fly and points-to sets are represented using sparse bit vectors, for both SFS and Selfs, which are implemented in LLVM 3.3. In [14], implemented in LLVM 2.5, a program's call graph is pre-computed and points-to sets are represented using binary decision diagrams (BDDs).

4.3 Results and Analysis

As shown in Figure 6, Selfs is 2.13X faster than SFS on average under our load-precision-preserving partitioning strategy while maintaining the same precision for reads, i.e., for all alias queries in all the functions from a benchmark. The best speedups are achieved at `h264ref` (7.45X) and `mesa` (6.08X). Note that Selfs can go faster, approaching eventually the efficiency of FI, if increasingly larger regions are used. The analysis time of a benchmark, which excludes the time spent on pre-analysis, is the average of three runs.

Let us look at the results of the two analyses in more detail. SFS spends 2465.39 seconds to analyse all the benchmarks but Selfs finishes in 782.37 seconds. In Columns 6 – 10, we see the number of regions of each type as well as the average and maximum region sizes. The average region sizes range from 1.40 (`mesa`) to 3.51 (`twolf`). In `gcc`, `perlbmk` and `vortex`, each largest region ends up with 150+ loads or stores being resolved flow-insensitively, with the precision for all reads being preserved. This shows the great potential promised by Selfs

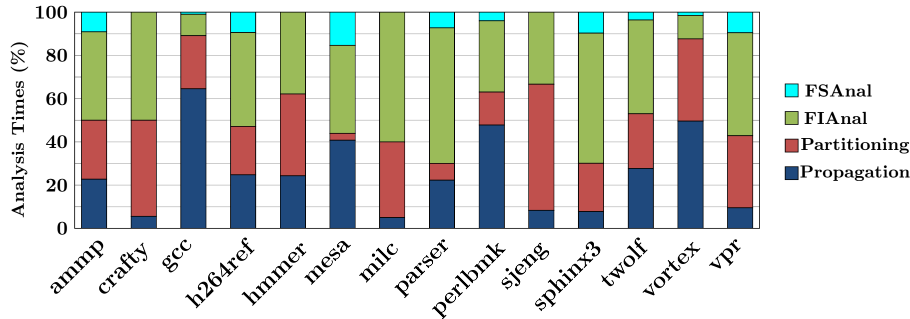


Fig. 7. Percentage distributions of SELFS' analysis time in a benchmark over "FSAnal" (the time for its flow-sensitive analysis), "FIAnal" (the time for its flow-insensitive analysis), "Partitioning" (the time on region partitioning), and "Propagation" (the time for propagating points-to facts across the indirect def-use chains in the region graph).

in achieving load-precision-preserving flow-sensitive analysis in a region-based manner. With better tuned unification rules, better speedups are expected.

For relatively small program, such as `ammp`, `hmmer`, `milc` and `sjeng`, SELFS yields little or no performance benefits due to the overhead on region partitioning, as illustrated in Figure 7. However, for relatively larger ones, which contain more objects and more dense def-use chains to be dealt with flow-sensitively (Figure A.1), such as `gcc`, `mesa`, `perlbmk` and `vortex`, SELFS is beneficial. The best speedups are observed at `h264ref` (7.45X) and `mesa` (6.08X), because the times for propagating points-to facts across indirect def-use chains have been significantly reduced by 22.3X and 10.24X, respectively (Figure 8).

5 Related Work

Sparse Pointer Analysis Sparse analysis, a recent improvement over the classic iterative data-flow approach, can achieve flow-sensitivity more efficiently by propagating points-to facts sparsely across pre-computed def-use chains [14, 15, 23, 32, 35]. Initially, sparsity was experimented with in [16, 17] on a Sparse Evaluation Graph [5, 26], a refined CFG with irrelevant nodes being removed. On various SSA form representations (e.g., factored SSA [6], HSSA [7] and partial SSA [20]), further progress was made later. The def-use chains for top-level pointers, once put in SSA, can be explicitly and precisely identified, giving rise to a so-called semi-sparse flow-sensitive analysis [15]. Recently, the idea of staged analysis [11, 14] that uses pre-computed points-to information to bootstrap a later more precise analysis has been leveraged to make pointer analysis full-sparse for both top-level and address-taken variables [14, 23, 35].

Hybrid Analysis The aim of hybrid-sensitive pointer analysis is to find a right balance between efficiency and precision. As a well-known example for mixing different flow-insensitive analyses, one-level approach [9] lies between Steensgaard's and Andersen's analyses (in terms of precision) by not applying its

Program	Propagation Times (Secs)		Speedup
	SFS	SELFS	
ammp	0.13	0.05	2.60
crafty	0.02	0.01	2.00
gcc	741.42	270.32	2.74
h264ref	25.20	1.13	22.30
hmmer	0.11	0.09	1.22
mesa	719.97	70.31	10.24
milc	0.02	0.01	2.00
parser	2.65	0.43	6.16
perlbmk	285.90	68.96	4.15
sjeng	0.02	0.01	2.00
sphinx3	0.53	0.08	6.63
twolf	0.42	0.23	1.83
vortex	77.33	18.47	4.19
vpr	0.05	0.02	2.50

Fig. 8. Propagation times of SFS and SELFS for analysing the address-taken variables across their indirect def-use chains.

unification process to top-level pointers. For context-sensitivity, hybrid analysis has been used in Java to pick up the benefits of both call-site sensitivity and object sensitivity [19]. In [21], strong updates are performed for only singleton objects on top of a flow- and field-insensitive Andersen's analysis. Earlier [12, 29], how to adjust the analysis precision according to clients' needs is discussed.

Region-based Analysis Region-based analysis, which partitions a program into different compilation units, was commonly used to explore locality and gain more opportunities for compiler optimisations, such as inlining [13, 33], partial dead code elimination [3], and just-in-time optimisation [28]. In [36, 37], programs are decomposed into different regions according to the alias relations and each region is solved independently. Same partition strategy was also adopted by [18] to speed up a flow- and context-sensitive pointer analysis.

6 Conclusion

We introduce a new region-based flow-sensitive pointer analysis, called SELFS, that allows efficiency and precision tradeoffs to be made subject to region partitioning strategies used. We have implemented SELFS in LLVM and demonstrated its effectiveness with a unification-based region partitioning strategy, by comparing it with a state-of-the-art flow-sensitive analysis. In addition, our unification-based approach is interesting in its own right as it leads to a particular analysis that is as precise as FS for almost all practical purposes.

In future work, we will develop a range of partitioning strategies to relax our unification-based approach. There is a lot of freedom in performing a precision-loss partitioning (Theorems 1 and 2). In order to be tunable and client-specific, a relaxed strategy can be designed along the following directions (among others). First, the user can identify parts of a program that require flow-sensitive

analysis based on client analyses' needs (e.g., hot functions and major changes made during software development). Second, the user may request customised flow-sensitivity for some selected variables. Third, some stores can always be weakly updated (to enable more offline unification steps, for example). Finally, our unification approach can be relaxed to enable more regions to be merged without having to preserve the precision for all the loads.

7 Acknowledgments

The authors wish to thank the anonymous reviewers for their valuable comments. This work is supported by Australian Research Grants, DP110104628 and DP130101970, and a generous gift by Oracle Labs.

References

1. M. Acharya and B. Robinson. Practical change impact analysis based on static program slicing for industrial software systems. In *ICSE '11*, pages 746–755.
2. L. O. Andersen. Program analysis and specialization for the C programming language. *PhD Thesis, DIKU, University of Copenhagen*, 1994.
3. Q. Cai, L. Gao, and J. Xue. Region-based partial dead code elimination on predicated code. In *CC '04*, pages 150–166.
4. M. Ceccarelli, L. Cerulo, G. Canfora, and M. Di Penta. In *ICSE '10*, pages 163–166.
5. J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL '91*, pages 55–66.
6. J.-D. Choi, R. Cytron, and J. Ferrante. On the efficient engineering of ambitious program analysis. *IEEE Transactions on Software Engineering*, 20(2):105–114, 1994.
7. F. Chow, S. Chan, S. Liu, R. Lo, and M. Streich. Effective representation of aliases and indirect memory operations in SSA form. In *CC '96*, pages 253–267.
8. R. Cytron, J. Ferrante, B. Rosen, M. Wegman, and F. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 13(4):451–490, 1991.
9. M. Das. Unification-based pointer analysis with directional assignments. In *PLDI '00*, pages 35–46.
10. M. Das, S. Lerner, and M. Seigle. ESP: Path-sensitive program verification in polynomial time. In *PLDI '02*, pages 57–68.
11. S. J. Fink, E. Yahav, N. Dor, G. Ramalingam, and E. Geay. Effective typestate verification in the presence of aliasing. *ACM Transactions on Software Engineering and Methodology*, 17(2):1–34, 2008.
12. S. Z. Guyer and C. Lin. Client-driven pointer analysis. In *SAS'03*, pages 214–236.
13. R. E. Hank, W.-M. W. Hwu, and B. R. Rau. Region-based compilation: An introduction and motivation. In *MICRO '95*, pages 158–168.
14. B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO '11*, pages 289–298.
15. B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *POPL '09*, pages 226–238.

16. M. Hind, M. Burke, P. Carini, and J.-D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, 1999.
17. M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *SAS '98*, pages 57–81.
18. V. Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In *PLDI '08*, pages 249–259.
19. G. Kastrinis and Y. Smaragdakis. Hybrid context-sensitivity for points-to analysis. In *PLDI '13*, pages 423–434.
20. C. Lattner and V. Adve. LLVM: A compilation framework for lifelong program analysis & transformation. In *CGO '04*, pages 75–86.
21. O. Lhoták and K.-C. A. Chung. Points-to analysis with efficient strong updates. In *POPL '11*, pages 3–16.
22. V. B. Livshits and M. S. Lam. Tracking pointers with path and context sensitivity for bug detection in c programs. In *FSE '03*, pages 317–326.
23. H. Oh, K. Heo, W. Lee, W. Lee, and K. Yi. Design and implementation of sparse global analyses for C-like languages. In *PLDI '12*, pages 229–238.
24. D. Pearce, P. Kelly, and C. Hankin. Efficient field-sensitive pointer analysis of C. *ACM Transactions on Programming Languages and Systems*, 30(1), 2007.
25. F. Pereira and D. Berlin. Wave propagation and deep propagation for pointer analysis. In *CGO '09*, pages 126–135.
26. G. Ramalingam. On sparse evaluation representations. *Theoretical Computer Science*, 277(1):119–147, 2002.
27. R. R. Rick Hank, Loreena Lee. Implementing next generation points-to in Open64. In *Open64 Developers Forum*.
28. T. Suganuma, T. Yasue, and T. Nakatani. A region-based compilation technique for a Java just-in-time compiler. In *PLDI '03*, pages 312–323.
29. Y. Sui, Y. Li, and J. Xue. Query-directed adaptive heap cloning for optimizing compilers. In *CGO '13*, pages 1–11.
30. Y. Sui, D. Ye, and J. Xue. Static memory leak detection using full-sparse value-flow analysis. In *ISSTA '12*, pages 254–264.
31. Y. Sui, D. Ye, and J. Xue. Detecting memory leaks statically with full-sparse value-flow analysis. *IEEE Transactions on Software Engineering*, 40(2):107–122, 2014.
32. Y. Sui, S. Ye, J. Xue, and P. Yew. SPAS: Scalable path-sensitive pointer analysis on full-sparse SSA. In *APLAS '11*, pages 155–171.
33. S. Triantafyllis, M. J. Bridges, E. Raman, G. Ottoni, and D. I. August. A framework for unrestricted whole-program optimization. In *PLDI '06*, pages 61–71.
34. D. Ye, Y. Sui, and J. Xue. Accelerating dynamic detection of uses of undefined variables with static value-flow analysis. In *CGO '14*, pages 154–164.
35. H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow-and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO '10*, pages 218–229.
36. S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step toward practical analyses. In *FSE '96*, pages 81–92.
37. S. Zhang, B. G. Ryder, and W. A. Landi. Experiments with combined analysis for pointer aliasing. In *PASTE '98*, pages 11–18.

A Appendix

Program	Size	#Statement					#Ptrs	#Object				
	KLOC	AddrOf	Copy	Load	Store	Total		Glob.	Heap	Stk	Func	Total
ammp	13.4	702	6875	925	187	8689	29499	49	42	76	209	376
crafty	21.2	1632	9603	1011	367	12613	44744	457	33	70	147	707
gcc	230.4	8934	135332	32498	7832	184596	399377	1400	154	1018	2273	4845
h264ref	51.6	1829	27845	8324	1635	39633	114221	374	209	284	287	1154
hmmer	36	1195	7635	2083	581	11494	32415	42	376	89	155	662
mesa	61.3	2691	45447	6112	2298	56548	136415	35	322	465	1130	1952
milc	15	1104	8591	1138	263	11096	26437	92	63	203	270	628
parser	11.4	1417	6045	1626	964	10052	23417	174	114	42	353	683
perlbmk	87.1	4366	39602	17096	5154	66218	148231	415	28	458	1168	2069
sjeng	13.9	926	5579	848	632	7985	29624	214	14	119	173	520
sphinx3	25.1	1898	10169	2482	622	15171	36588	69	59	122	421	671
twolf	20.5	1371	14390	7526	506	23793	62430	304	192	116	212	824
vortex	67.3	6636	20408	6577	3185	36806	104218	739	29	1864	961	3593
vpr	17.8	1195	5703	2222	310	9430	28405	101	6	101	303	511

Fig. A.1. Program characteristics.