공학박사학위논문

필요한 부위, 시점, 상황 선별을 통한 프로그램 전체 분석의 비용 절감 기술

Spatial, Temporal, and Contextual Localization Techniques for Scalable Global Static Analysis

2012년 2월

서울대학교 대학원

컴퓨터공학부

오 학 주

필요한 부위, 시점, 상황 선별을 통한 프로그램 전체 분석의 비용 절감 기술

지도교수 이 광 근

이 논문을 공학박사학위논문으로 제출함

2011년 10월

서울대학교 대학원

컴퓨터공학부

오 학 주

오 학 주의 박사학위논문을 인준함

2011년 12월

위 원 장	문 병 로	(인)
부 위 원 장	이 광 근	(인)
위 원	김 지 홍	(인)
위 원	이 제 희	(인)
위 원	류 석 영	(인)

Abstract

Spatial, Temporal, and Contextual Localization Techniques for Scalable Global Static Analysis

Hakjoo Oh
School of Computer Science and Engineering
College of Engineering
Seoul National University

In this dissertation we present general methods for achieving sound, precise, and also scalable global static analyzers. We first use the abstract interpretation framework to have a sound and precise global staic analyzer whose scalability is not yet attended. Then, we add our localization techniques to improve its scalability while preserving the precision and soundness of the underlying analysis. We present three localization techniques that allow the analysis to concentrate on relevant portions of spatial, temporal, and contextual states, respectively. The techniques are successfully applied to an industrial-strength C static analyzer to scale to analyze up to one million lines of C programs.

Keywords : Programming Languages, Static Program Analysis, Abstract Interpretation, Global Analysis, Localization
Student Number : 2007-30226

Contents

List of Tables

List of Figures

Chapter 1

Overview

In this thesis, we show that achieving scalable global static analysis is feasible. We present localization techniques that allow semantics-based global static analysis to scale up to million lines of C programs without compromising soundness and precision.

1.1 Goal

Precise, sound, scalable yet global static analyzers have been unachievable in general. Other than almost syntactic properties, once the target property becomes slightly deep in semantics it's been a daunting challenge to achieve the four goals in a single static analyzer. This situation explains why, for example, in the static error detection tools for full C, there exists a clear dichotomy: either "bug-finders" that risk being unsound yet scalable or "verifiers" that risk being unscalable yet sound. No such tools are scalable to globally analyze million lines of C code while being sound and precise enough for practical use.

In this dissertation, we present general methods to achieve scalable global static analyzers that are still precise and sound. Our approach first leverages the abstract interpretation [12,16] to have a sound, global, and yet arbitrarily precise static analyzer whose scalability is unattended. Upon this sound static analyzer, we add a set of cost-reduction techniques to improve its scalability while preserving the

precision and soundness of the underlying analysis. Note that while abstract interpretation framework provides a theoretical knob to control the analysis precision without violating its correctness, our techniques provide knobs to control the resulting analyzer's scalability preserving its precision.

1.2 Approach

In order to achieve the goal, we propose to collectively use various kinds of localization techniques. By localization, we refer to general static analysis techniques that somehow embody the idea of local reasoning [52] (equivalently, "framing" in separation logic [58]), by which the analysis concentrates on the relevant portion of analysis states rather than the entire global state. In this dissertation, we propose three localizations for global static analysis:

- **Spatial localization**: When analyzing a code block with an input state, it is typical that only a tiny subset of the whole memory space is actually accessed and the most are irrelevant. For example, consider statement `x:=y`. When we analyze the statement, the entire input state is irrelevant except for `x` and `y`. Spatial localization enables the analysis to concentrate on the relevant portion of memory used by the current code block, potentially avoiding reanalyses of the block when the memory is changed only for those outside the accessible portion.

- **Temporal localization**: In static analysis, semantic dependence among statements are usually sparse and thus the output memory of statements are not immediately used but only after a number of irrelevant statements. For example, suppose we analyze code `y:=x;t:=1;z:=y`. The updated value of `y` is used in the last statement after analyzing the semantically-irrelevant, second statement. Temporal localization enables the analysis to concentrate on the relevant next statements after analyzing the current statement, avoiding unnecessary value propagation.

- **Contextual localization**: In global interprocedural analysis, it is inevitable to follow some multiple, spurious return paths. Such spurious paths spawn serious performance problem because the multiple returns often lead to a single large fixpoint cycle involving almost all parts of the program. Contextual localization enables the analysis to concentrate on the relevant call-context of procedures, which then justifies single returns of procedures and mitigates the problem of large flow cycles.

Note that the localization techniques do not compromise the precision of the underlying analysis (Section 3.3.2, 4.2.5, 5.3.2); the techniques sometimes even improve the analysis precision (Section 3.5.4, 5.3.2).

1.3 Performance

Throughout this dissertation, the proposed techniques are corroborated by experiments with Airac, an industrial-strength global static analyzer for full C [29, 31, 32, 48–51]. Table 1.1 summarizes the achieved performance. From the baseline analyzer ($\mathsf{Airac}_{\mathsf{Base}}$), we implemented its localized version ($\mathsf{Airac}_{\mathsf{Local}}$) that is fully equipped with the three localization techniques. The results clearly show the effectiveness of our techniques: $\mathsf{Airac}_{\mathsf{Base}}$ only scales to analyze up to 35 KLOC but $\mathsf{Airac}_{\mathsf{Local}}$ scales to 1.4 MLOC. For the first five benchmarks that they both completes, $\mathsf{Airac}_{\mathsf{Local}}$ shows 181–4,008 x faster performance than $\mathsf{Airac}_{\mathsf{Base}}$. It also significantly saves memory costs (by 73–97 %).[1]

1.4 Contributions

The major challenge of localization is that finding relevant portions of states for an analysis is generally impossible unless we actually perform the analysis. Thus,

[1] Note that the baseline analyzer in Table 1.1 is already a highly tuned one. In previous works [31, 48–51], we reported the performance of the early versions of the analyzer, and, compared to them, the current baseline is much more efficient. For example, in [48], the baseline analyzer took 88,221s in analyzing make-3.76.1 but the current baseline takes 24,240s.

Program	**LOC**	**Baseline**		**Sparse**		**Spd↑**	**Mem↓**
		Time	**Mem**	**Time**	**Mem**		
gzip-1.2.4a	7 K	772	240	3	63	257 x	74 %
bc-1.06	13 K	1,270	276	7	75	181 x	73 %
less-382	23 K	9,561	1,113	33	127	289 x	86 %
make-3.76.1	27 K	24,240	1,391	21	114	1,154 x	92 %
wget-1.9	35 K	44,092	2,546	11	85	4,008 x	97 %
a2ps-4.14	64 K	∞	N/A	40	353	N/A	N/A
sendmail-8.13.6	130 K	∞	N/A	744	678	N/A	N/A
nethack-3.3.0	211 K	∞	N/A	16,373	5,298	N/A	N/A
emacs-22.1	399 K	∞	N/A	37,830	7,795	N/A	N/A
python-2.5.1	435 K	∞	N/A	11,039	5,535	N/A	N/A
linux-3.0	710 K	∞	N/A	33,618	20,529	N/A	N/A
gimp-2.6	959 K	∞	N/A	3,874	3,602	N/A	N/A
ghostscript-9.00	1,363 K	∞	N/A	14,814	6,384	N/A	N/A

Table 1.1: Effectiveness of the proposed techniques on various open-source benchmarks: time (in seconds) and peak memory consumption (in megabytes) of the baseline analyzer ($\mathsf{Airac}_{\mathsf{Base}}$) and its localized version ($\mathsf{Airac}_{\mathsf{Local}}$). ∞ means the analysis ran out of time (exceeded 24 hour time limit). **Spd↑** is the speed-up of $\mathsf{Airac}_{\mathsf{Local}}$ over $\mathsf{Airac}_{\mathsf{Base}}$. **Mem↓** shows the memory savings of $\mathsf{Airac}_{\mathsf{Local}}$ over $\mathsf{Airac}_{\mathsf{Base}}$. All experiments were done on a Linux 2.6 system running on a single core of Intel 3.07 GHz box with 24 GB of main memory.

any localization techniques need to somehow approximate the required information. The main contribution of this dissertation is the invention of new approximation methods that are more effective than existing techniques. Specifically, we make the following contributions:

- **An access-based technique for spatial localization:** Most previous approaches for spatial localization take reachability-based techniques [11, 22, 28, 39, 44, 59, 60, 71, 72]: abstract memories are localized in such a way that unreachable entities from the current environment are removed. We show that the reachability-based approach is sometimes too conservative in practice, and propose a novel, access-based approach for spatial localization. We also show that the access-based technique more tightly localizes abstract memories than the reachability-based technique does.

- **A formal framework for temporal localization:** Temporal localization has been heavily studied in contexts of sparse analysis techniques [8, 19, 24, 25, 55, 66, 68]. Previous sparse analysis techniques, however, are mostly algorithmic and tightly coupled with particular analyses. Thus, the sparse techniques are not general enough to be used for an arbitrarily complicated semantic analysis. We generalize the sparse analysis ideas on top of the abstract interpretation framework. As a result, we bridge the gap between the abstract interpretation and sparse analysis, shedding light on correct sparse analysis designs for arbitrarily complicated analysis.

- **An algorithmic technique for contextual localization:** Previously, contextual localization has been hardly studied. We report, for the first time, that the lack of contextual localization leads to significant performance degradation of global analysis caused by large spurious interprocedural cycles. In addition, we present a technique to mitigate the problem, which is purely algorithmic and directly applicable without changing the analysis' underlying abstract semantics.

- **Experimental evaluations in a realistic setting:** The effectiveness of the proposed techniques are proven by experiments with an industrial-strength global static analyzer [29, 31, 32, 48–51]. We show that, in concert with the three proposed techniques, the semantics-based global analysis can scale to analyze up to million lines of C programs, which itself represents a significant departure from existing general-purpose global static analyzers [2, 3, 31, 74].

1.5 Outline

The rest of this dissertation is organized as follows:

- Chapter 2 describes the baseline analyzer, on top of which we will develop our localization techniques in later chapters.
- Chapter 3 develops the access-based spatial localization technique. On top of the baseline analyzer in Chapter 2, we formalize the conventional reachability-based approach and our access-based technique. In addition, we present two extensions, block-level localization and bypassing, that further enhances its performance. We present experimental results regarding spatial localization in detail.
- Chapter 4 presents the temporal localization technique. We first formally present our framework and prove its correctness. Then, we show how to design sparse version of our baseline analyzer within the framework. We also show a set of implementation techniques that turn out to be critical to economically support the sparse analysis process.
- Chapter 5 presents our contextual localization technique. We show how spurious return paths in global static analysis create large spurious cycles. We present an algorithmic technique that mitigates the performance problem caused by the spurious flow cycles.
- Chapter 6 discusses related work and Chapter 7 concludes the dissertation.

Chapter 2

Setting: Baseline Analyzer

We develop our localization techniques on top of Airac [29, 31, 48–51], an interval domain-based abstract interpreter for full set of C programs. In this chapter, we describe the underlying abstract semantics of the analyzer. We present the program representation (Section 2.2), collecting semantics (Section 2.3), and abstract semantics (Section 2.4). Because the abstract semantics is derived by fairly standard abstractions, we do not formally present the concrete domain and semantics of programs, nor a soundness proof of the analysis.

2.1 Notation

Throughout this dissertation, we use the following notations. Given a function $f : A \to B$ and a set $X \subseteq A$, $f|_X$ denotes function restriction: $f|_X(x) = f(x)$ if $x \in X$, otherwise $f|_X(x) = \bot$. We write $f\backslash_X$ for the restriction of f to the domain $\mathsf{dom}(f) - X$. We abuse the notation $f|_a$ and $f\backslash_a$ for the domain restrictions on singleton set $\{a\}$ and $\mathsf{dom}(f) - \{a\}$, respectively. We write $a \mapsto b$ for a function $(a \mapsto b)(x) = b$ if $a = x$, otherwise $(a \mapsto b)(x) = \bot$. We write $f[a \mapsto b]$ to mean the function got from function f by changing the value for a to b. For all of the domains, we assume an implicit and appropriate $\top$ and $\bot$ element for domains that need them. We write $f[a_1 \mapsto b_1, \cdots, a_n \mapsto b_n]$ for $f[a_1 \mapsto b_1] \cdots [a_n \mapsto b_n]$. We write $f[\{a_1, \cdots, a_n\} \overset{w}{\mapsto} b]$ for $f[a_1 \mapsto f(a_1) \sqcup b, \cdots, a_n \mapsto f(a_n) \sqcup b]$ (weak update). When

X is a tuple, $X.n$ indicates the nth component of the tuple. We write R^+ and $R^\star$ for the transitive and reflexive-transitive closure of binary relation R. Finally, $\mathbb{N}$ represents the set of natural numbers $\{0, 1, 2, \ldots\}$.

2.2 Programs

A program is a tuple $\langle \mathbb{C}, \hookrightarrow \rangle$ where $\mathbb{C}$ is a finite set of control points and $\hookrightarrow \subseteq \mathbb{C} \times \mathbb{C}$ is a relation that denotes control dependencies of the program; $c' \hookrightarrow c$ indicates that c is a next control point of c'. Each control point c is associated with command $\mathsf{cmd}(c)$. Command c has one of the following five types:

$$lv := e \mid lv := \mathsf{alloc}(a) \mid \mathsf{assume}(x < e) \mid \mathsf{call}(f_x, e) \mid \mathsf{return}_f$$

where expression e, l-value expression lv, and allocation expression a are defined as follows:

$$\begin{array}{lrcl} \text{expression} & e & \rightarrow & n \mid e \texttt{ + } e \mid lv \mid \&lv \\ \text{l-value} & lv & \rightarrow & x \mid \texttt{*}e \mid e\texttt{[}e\texttt{]} \mid e\texttt{.}x \\ \text{allocation} & a & \rightarrow & \texttt{[}e\texttt{]}_l \mid \texttt{\{}x\texttt{\}}_l \end{array}$$

An expression may be a constant integer (n), a binary operation (e + e), an l-value expression (lv), or an address-of expression ($\&lv$). An l-value may be a variable (x), a pointer dereference ($\texttt{*}e$), an array access ($e\texttt{[}e\texttt{]}$), or a field access ($e\texttt{.}x$). Expressions and l-value expressions have no side-effects. All program variables, including formal parameters, have unique names. Command $lv := e$ assigns the value of e into the location of lv. Command $lv := \mathsf{alloc}(a)$ allocates an array $\texttt{[}e\texttt{]}_l$ or a structure $\texttt{\{}x\texttt{\}}_l$, where e is the size of the array, x is the field name, and the subscript l is the label for the allocation site. For simplicity, we consider structures with one field only. Each call-site for a procedure is represented by two control points: a call-point and its corresponding return-point. A call-point is associated with command $\mathsf{call}(f_x, e)$, which indicates that procedure f, whose formal parameter is x, is called with actual parameter e. When c is a call-point (resp., return-point), $\mathsf{callof}(c)$ (resp., $\mathsf{rtnof}(c)$) denotes the corresponding return-point (resp., call-

point). For simplicity, we assume that there are no function pointers[1] and consider procedures with one parameter only. Command return_f denotes the return statement of procedure f.

2.3 Collecting Semantics

Collecting semantics of program P is an invariant $[\![P]\!] \in \mathbb{C} \to 2^{\mathbb{S}}$ that represents a set of reachable states at each control point, where the concrete domain of states, $\mathbb{S}$, is defined as follows:

$$\mathbb{S} = \mathbb{L} \to \mathbb{V}$$

Concrete state $s \in \mathbb{S}$ is a map from locations to values, and a value is either integer ($\mathbb{Z}$) or location ($\mathbb{L}$). The collecting semantics is characterized by the least fixpoint of semantic function $F \in (\mathbb{C} \to 2^{\mathbb{S}}) \to (\mathbb{C} \to 2^{\mathbb{S}})$ such that,

$$F(X) = \lambda c \in \mathbb{C}. f_c(\bigcup_{c' \hookrightarrow c} X(c')). \tag{2.1}$$

where $f_c \in 2^{\mathbb{S}} \to 2^{\mathbb{S}}$ is a semantic function at control point c. We leave out the standard definition of the concrete semantic function.

2.4 Abstract Semantics

2.4.1 Abstract Domain

We abstract the collecting semantics of program P by the following Galois connection:

$$\mathbb{C} \to 2^{\mathbb{S}} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{C} \to \hat{\mathbb{S}} \tag{2.2}$$

[1] In implemetation, we prior resolve all function pointers with a pre-analysis.

where α and γ are pointwise liftings of abstract and concretization function $\alpha_{\mathbb{S}}$ and $\gamma_{\mathbb{S}}$ (such that $2^{\mathbb{S}} \underset{\alpha_{\mathbb{S}}}{\overset{\gamma_{\mathbb{S}}}{\leftrightarrows}} \hat{\mathbb{S}}$), respectively. That is, we abstract the set of reachable states by a single abstract state. Abstract memory state

$$\hat{\mathbb{S}} \quad = \quad \hat{\mathbb{L}} \to \hat{\mathbb{V}}$$

denotes a finite map from abstract locations ($\hat{\mathbb{L}}$) to abstract values ($\hat{\mathbb{V}}$).

$$\begin{array}{rcl} \hat{\mathbb{L}} & = & \mathit{Var} + \mathit{AllocSite} + \mathit{AllocSite} \times \mathit{FieldName} \\ \hat{\mathbb{V}} & = & \hat{\mathbb{Z}} \times 2^{\hat{\mathbb{L}}} \times 2^{\mathit{AllocSite} \times \hat{\mathbb{Z}} \times \hat{\mathbb{Z}}} \times 2^{\mathit{AllocSite} \times 2^{\mathit{FieldName}}} \\ \hat{\mathbb{Z}} & = & \{[l,u] \mid l,u \in \mathbb{Z} \cup \{-\infty, +\infty\} \wedge l \leq u\} \cup \{\bot\} \end{array}$$

An abstract location may be a program variable (Var), an allocation site ($\mathit{AllocSite}$), or a structure field ($\mathit{AllocSite} \times \mathit{FieldName}$). All elements of an array allocated at allocation site l are collectively represented by l. The abstract location for field x of a structure allocated at l is represented by $\langle l, x \rangle$. An abstract value is a quadruple. Numeric values are tracked by the interval values ($\hat{\mathbb{Z}}$). Points-to information is kept by the second component ($2^{\hat{\mathbb{L}}}$): it indicates pointer targets an abstract locations may point to. Allocated arrays of memory locations are represented by array blocks ($2^{\mathit{AllocSite} \times \hat{\mathbb{Z}} \times \hat{\mathbb{Z}}}$): an array block $\langle l, o, s \rangle$ consists of abstract base address (l), offset (o), and size (s). A structure block $\langle l, \{x\} \rangle \in 2^{\mathit{AllocSite} \times 2^{\mathit{FieldName}}}$ abstracts structure values that are allocated at l and have a set of fields $\{x\}$.

2.4.2 Abstract Semantic Function

Abstract semantics is characterized by the least fixpoint of abstract semantic function $\hat{F} \in (\mathbb{C} \to \hat{\mathbb{S}}) \to (\mathbb{C} \to \hat{\mathbb{S}})$ defined as,

$$\hat{F}(\hat{X}) = \lambda c \in \mathbb{C}. \hat{f}_c(\bigsqcup_{c' \hookrightarrow c} \hat{X}(c')). \tag{2.3}$$

where $\hat{f}_c \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ is a semantic function at control point c.

$$\hat{f}_c(\hat{s}) =$$
$$\begin{cases} \hat{s}[\hat{\mathcal{L}}(lv)(\hat{s}) \overset{w}{\mapsto} \hat{\mathcal{V}}(e)(\hat{s})] & \mathsf{cmd}(c) = lv := e \\ \hat{s}[\hat{\mathcal{L}}(lv)(\hat{s}) \overset{w}{\mapsto} \langle \bot, \bot, \{\langle l, [0,0], \hat{\mathcal{V}}(e)(\hat{s}).1 \rangle\}, \bot \rangle] & \mathsf{cmd}(c) = lv := \mathsf{alloc}(\texttt{[}e\texttt{]}_l) \\ \hat{s}[\hat{\mathcal{L}}(lv)(\hat{s}) \overset{w}{\mapsto} \langle \bot, \bot, \bot, \{\langle l, \{x\} \rangle\} \rangle] & \mathsf{cmd}(c) = lv := \mathsf{alloc}(\{x\}_l) \\ \hat{s}[x \mapsto \langle \hat{s}(x).1 \sqcap [-\infty, \mathsf{u}(\hat{\mathcal{V}}(e)(\hat{s}).1)], \hat{s}(x).2, \hat{s}(x).3, \hat{s}(x).4 \rangle] & \mathsf{cmd}(c) = \mathsf{assume}(x < e) \\ \hat{s}[x \mapsto \hat{\mathcal{V}}(e)(\hat{s})] & \mathsf{cmd}(c) = \mathsf{call}(f_x, e) \\ \hat{s} & \mathsf{cmd}(c) = \mathsf{return}_f \end{cases}$$

Auxiliary functions $\hat{\mathcal{V}}(e)(\hat{s})$ and $\hat{\mathcal{L}}(lv)(\hat{s})$ computes abstract values for e and abstract locations for lv, respectively, under $\hat{s}$. The effect of node $lv := e$ is to (weakly) update the abstract value of e into abstract locations $\hat{\mathcal{L}}(lv)(\hat{s})$.[2] The array allocation command $lv := \mathsf{alloc}(\texttt{[}e\texttt{]}_l)$ creates a new array block with offset 0 and size e. The structure block command $lv := \mathsf{alloc}(\{x\}_l)$ creates a new structure block. In both cases, we use the allocation site l as the base address, by which many (possibly infinite) concrete locations are summarized by finite abstract locations. Assume $\mathsf{assume}(x < e)$ confines the value of x so that the resulting memory state satisfies the condition $(\mathsf{u}([a,b]) = b)$. The call command $\mathsf{call}(f_x, e)$ binds the formal parameter x to the value of actual parameter e. Note that the output of the call node is the memory state that flows into the body of the called procedure, not the memory state returned from the call. The abstract semantics for procedure calls show that our analysis is context-insensitive: it ignores the calling context in which procedures are invoked.[3]

Lemma 1 (Soundness) *If* $\alpha \circ F \sqsubseteq \hat{F} \circ \alpha$, *then,* $\alpha(\mathbf{lfp}F) \sqsubseteq \mathbf{lfp}\hat{F}$.

[2]For brevity, we consider only weak updates. Applying strong updates is orthogonal to our localization techniques.

[3]Extention to context-sensitivity is presented in Chapter 5

We now define $\hat{\mathcal{V}}$ and $\hat{\mathcal{L}}$, which compute abstract values and locations, respectively. Given expression e and abstract state $\hat{s}$, $\hat{\mathcal{V}}(e)(\hat{s})$ evaluates the abstract value of e under $\hat{s}$.

$$\begin{aligned}
\hat{\mathcal{V}}(e) &\in \hat{\mathbb{S}} \to \hat{\mathbb{V}} \\
\hat{\mathcal{V}}(n)(\hat{s}) &= \langle [n,n], \bot, \bot, \bot \rangle \\
\hat{\mathcal{V}}(e_1 \texttt{ + } e_2)(\hat{s}) &= \hat{\mathcal{V}}(e_1)(\hat{s}) \hat{+} \hat{\mathcal{V}}(e_2)(\hat{s}) \\
\hat{\mathcal{V}}(lv)(\hat{s}) &= \bigsqcup \{ \hat{s}(a) \mid a \in \hat{\mathcal{L}}(lv)(\hat{s}) \} \\
\hat{\mathcal{V}}(\&lv)(\hat{s}) &= \langle \bot, \hat{\mathcal{L}}(lv)(\hat{s}), \bot, \bot \rangle
\end{aligned}$$

$\hat{\mathcal{V}}$ is inductively defined for each type of expression. Integer n evaluates to its corresponding interval value $[n,n]$. Expressions involving binary expression are inductively evaluated. For l-values lv, we first find the abstract locations that lv denotes and then look up the abstract values associated with the locations. $\&lv$ evaluates to the abstract locations that lv denotes. We skip the conventional definition of the abstract binary ($\hat{+}$) and join ($\sqcup$) operations.

Similarly, Given l-value expression lv and abstract memory state $\hat{s}$, $\hat{\mathcal{L}}(lv)(\hat{s})$ evaluates the set of abstract locations that lv denotes under $\hat{s}$.

$$\begin{aligned}
\hat{\mathcal{L}}(lv) &\in \hat{\mathbb{S}} \to 2^{\hat{\mathbb{L}}} \\
\hat{\mathcal{L}}(x)(\hat{s}) &= \{x\} \\
\hat{\mathcal{L}}(*e)(\hat{s}) &= \hat{\mathcal{V}}(e)(\hat{s}).2 \cup \{ l \mid \langle l, o, s \rangle \in \hat{\mathcal{V}}(e)(\hat{s}).3 \} \\
&\quad \cup \{ \langle l, x \rangle \mid \langle l, \{x\} \rangle \in \hat{\mathcal{V}}(e)(\hat{s}).4 \} \\
\hat{\mathcal{L}}(e_1 \texttt{[}e_2\texttt{]})(\hat{s}) &= \{ l \mid \langle l, o, s \rangle \in \hat{\mathcal{V}}(e_1)(\hat{s}).3 \} \\
\hat{\mathcal{L}}(e\,.\,x)(\hat{s}) &= \{ \langle l, x \rangle \mid \langle l, \{x\} \rangle \in \hat{\mathcal{V}}(e)(\hat{s}).4 \}
\end{aligned}$$

The abstract location for variable x is represented by x. When $*e$ is used as l-value, it denotes all the abstract locations accessible from e, including arrays and structure fields. Array access $e_1\texttt{[}e_2\texttt{]}$ refers to the location of arrays e_1 denotes. In our analysis, all of the array elements are smashed into a single element, and hence, the definition of $\hat{\mathcal{L}}(e_1\texttt{[}e_2\texttt{]})$ does not involve e_2. The abstract location for structure field $e\,.\,x$ is represented by a pair of allocation site and field name.

2.5 Fixpoint Algorithm

We compute the least fixpoint of abstract semantic function $\hat{F}$ (4.3) by a worklist-based fixpoint algorithm in Figure 2.1. Compared to the naive fixpoint algorithm (Figure 2.2), which directly iterates $\hat{F}^i$, the worklist algorithm evaluates only the program points whose abstract states had changed before. The worklist, W, consists of those control points whose abstract states are not yet stabilized. For each iteration of the loop, a control point is selected (choose) and evaluated. When the new output memory state $\hat{s}$ is changed, next control points of c are added to the worklist and the updated information is stored. Because our abstract domain is of infinite height, we need to apply an widening operator during the fixpoint computation. We use the conventional widening for the lattice of intervals [13], which is applied to heads of flow cycles. We use weak topological ordering for worklist management [6].

$W \in \mathit{Worklist} = 2^{\mathbb{C}}$
$\hat{X} \in \mathbb{C} \to \hat{\mathbb{S}}$
$\hat{f}_c \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$

$W := \mathbb{C}$
$\hat{X} := \lambda c.\bot$
repeat
  $c := \mathsf{choose}(W)$
  $\hat{s} := \hat{f}_c(\bigsqcup_{c' \hookrightarrow c} X(c'))$
  if $\hat{s} \not\sqsubseteq \hat{X}(c)$
    $W := W \cup \{c' \in \mathbb{C} \mid c \hookrightarrow c'\}$
    $\hat{X}(c) := \hat{X}(c) \sqcup \hat{s}$
until $W = \emptyset$

Figure 2.1: The worklist-based fixpoint computation algorithm. For brevity, we omit the widening operation, which is necessary for analysis' termination.

$\hat{X}, \hat{X}' \in \mathbb{C} \to \hat{\mathbb{S}}$
$\hat{f}_c \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$

$\hat{X} := \hat{X}' := \lambda c.\bot$
repeat
  $\hat{X}' := \hat{X}$
  for all $c \in \mathbb{C}$ **do**
    $\hat{X}(c) := \hat{f}_c(\bigsqcup_{c' \hookrightarrow c} X(c'))$
until $\hat{X} \sqsubseteq \hat{X}'$

Figure 2.2: A naive fixpoint algorithm.

Chapter 3

Spatial Localization

On-the-fly spatial localization, also known as abstract memory localization, is vital for economical abstract interpretation of imperative programs. Such localization is sometimes called "abstract garbage collection" or "framing". In this chapter we present a new memory localization technique that is more effective than the conventional reachability-based approach. Our technique is based on a key observation that collecting the reachable memory parts is too conservative and the accessed parts are usually tiny subsets of the reachable. Our technique first estimates, by an efficient pre-analysis, which parts of input states will be accessed during the analysis of each code block. Then the main analysis uses the access-set results to trim the memory entries before analyzing code blocks.

In experiments with an industrial-strength global C static analyzer, the technique is applied right before analyzing each procedure's body and reduces the average analysis time and memory by 92.1% and 71.2%, respectively, without sacrificing the analysis precision. Localizing more frequently such as at loop bodies and basic blocks as well as procedure bodies, the generalized localization additionally reduces analysis time by an average of 31.8%.

3.1 Introduction

In global abstract interpretation of imperative programs, memory localization ("framing" in separation logic) is vital for reducing analysis cost [9,22,39,44,72]. Memory localization, when analyzing a code block, attempts to remove the irrelevant parts of input memory states, which will not be used during the analysis of the block. Not to mention the immediate benefit of the reduced memory footprint, localization has other important impact on cost reduction. In flow-sensitive, semantically-dense global abstract interpretation, code blocks such as procedure bodies are repeatedly analyzed (often needlessly) with different input memory states. Localization makes input memory states smaller, which results in more general summaries for the blocks. More general summaries reduce re-analysis of blocks by increasing the chance of reusing the previously computed analysis results. For example, consider a code `x=0;f();x=1;f();` and assume that `x` is not used inside `f`. Without localization, `f` is analyzed twice because the input state to `f` is changed at the second call. If `x` is removed from the input state (localization), the analysis result of `f` for the first call can be reused for the second call without re-analyzing the procedure.

Realization of effective localization is not trivial. The problem is that it is not always possible to know in advance the to-be-used parts of the input state unless we actually analyze the procedure. For example, we cannot exactly determine the access information for indirect reference `*p` before starting analysis. Hence, any localization technique estimates the usable memory parts conservatively: the localized state is guaranteed to contain all the parts that will be used but may contain spurious entities (that will not be actually used) as well.

The conventional estimation methods are reachability-based [9,22,39,44,59,60, 71, 72]. They collect memory parts that are reachable (by pointer chains) from the current environment, and the memory parts that can no longer be reached (hence, obviously cannot be used anymore) are removed from the current states. When applied to procedure bodies, the technique allows a procedure to be analyzed with only memory portions that are reachable from actual parameters or

Program	LOC	accessed / reachable		memory	
spell-1.0	2,213	5	/	453	(1.1%)
httptunnel-3.3	6,174	10	/	673	(1.5%)
gzip-1.2.4a	7,327	22	/	1,002	(2.2%)
jwhois-3.0.1	9,344	28	/	830	(3.4%)
parser	10,900	75	/	1,787	(4.2%)
bc-1.06	13,093	24	/	824	(2.9%)
less-290	18,449	86	/	1,546	(5.6%)

Table 3.1: Reachability-based approach is too conservative. The table shows a comparison of accessed and reachable (abstract) memory portions during the analyses of open-source programs. For each a/b (r%), a is the average number of memory entries accessed in the called procedures, b is the average size of the reachable input state, and r is their ratio. The reachable- and accessed-memory ratio is an average over all procedures. We ran the reachability-based analysis and recorded, for every analysis of procedures, the sizes of localized memory and its accessed portion. We averaged the size ratio over the total number of analyses of procedures.

global variables. Because the reachable is often smaller than the entire state, the method is popular in various kinds of program analysis: for example, in shape analysis [11, 22, 39, 60, 71] and higher-order flow analysis [9, 44].

However, reachability is just a crude approximation and sometimes too conservative in practice. This is mainly because large parts of the reachable portion of input states are not actually accessed, i.e. the values are neither read nor written during the analysis. For example, Table 3.1 shows, given a reachability-based localized input state to a procedure, how much is actually accessed inside the (directly or transitively) called procedures. The table shows a comparison of accessed and reachable (abstract) memory portions during the analyses of open-source programs. The results show that only few reachable memory entries were actually accessed: procedures accessed only 1.1%–5.6% of reachable memory states. Nonetheless, the reachability-based approach propagates all the reachable parts to procedures. It

is therefore possible for a procedure body to be needlessly recomputed for input memory states whose only differences lie in the reachable-but-non-accessed portions. This means that the reachability-based approach can be too conservative for real C programs and hence is inefficient in both time and memory cost.

In this chapter, we present a memory localization technique that is more aggressive than the reachability-based approach. In addition to excluding unreachable memory entries from the localized state, we also exclude some memory entries that are reachable but will not be accessed. We attack the problem of localization by staging: (1) the set of abstract locations that will be used during the analysis of a code block is conservatively estimated by a pre-analysis; (2) then, the actual analysis uses the information and trims input memories before analyzing each block. The pre-analysis is derived by applying conservative abstractions to the abstract semantics of the original analysis and quickly finds an over-approximation of resources that the actual analysis requires. By reducing the sizes of localized memory states, our technique saves analysis time and memory more than the reachability-based approach does.

The time savings by our new localization method are significant: when applied to each procedure's body, our access-based localization reduces the analysis time by on average 92.1% over reachability-based localization. We implemented our approach inside an industrial-strength interval-domain-based abstract interpreter [29, 31,32,48–51]. In experiments, the technique reduces analysis time by 78.5%–98.5%, on average 92.1%, and peak memory consumption by 33.0–81.2%, on average 71.2%, over the reachability-based approach for a variety of open-source C benchmarks (2K–100KLOC). Moreover, our technique enables the largest four programs of our benchmarks to be analyzed, which could not be analyzed with the reachability-based approach because of the analysis running out of memory.

We generalize the idea of localization at procedure entries to localization of arbitrary code blocks. When applying localization to such smaller code blocks, we have to carefully select localization targets because localizing operations introduce performance overhead. We present a block selection strategy that is flexible to bal-

ance actual cost reduction against the overhead. The generalized localization reduces the analysis time by 8.5–53.7%, on average 31.8%, on top of the procedure-level localization.

This chapter makes the following contributions.

- We present a new memory localization technique, access-based localization. We employ a pre-analysis that is a conservative abstraction of the abstract semantics of the actual analysis. As far as we know published program analyzers do not perform access-based localization: previous analyses use pure reachability-based techniques (e.g., [22,44,59]) or their variants (e.g., [9,43]).
- We present a generalized localization algorithm that applies to arbitrary code blocks as well as procedure bodies. To the best of our knowledge, there is no published program analyzers that apply localization to arbitrary code blocks.

Example Consider the C code in Figure 3.1 and an interval analysis of the code. The analysis begins with an empty memory state ($\lambda x.\bot$). The abstract memory state right before calling `f` at line 7 (after parameter bound) is represented by Figure 3.1(a). Here, s denotes a structure with fields $\{a, b\}$ allocated at line 5. The abstract locations of each field are represented by $\langle l_5, a\rangle$ and $\langle l_5, b\rangle$, which initially have bottom values. p is a parameter of `f` and g is a global variable.

Reachability-based localization collects all reachable memory entries: global variable g, parameter p, and structure fields $\langle l_5, a\rangle$ and $\langle l_5, b\rangle$ that are reachable by dereferencing p. Figure 3.1(b) shows the resulting localized memory.

Our approach additionally filters the memory entries for $\langle l_5, b\rangle$ and g. Our pre-analysis infers that only the abstract locations $\{p, \langle l_5, a\rangle\}$ could be accessed during actual analysis of `f`. The actual analysis uses the results and trims memory entries, resulting in the memory state shown in Figure 3.1(c). Note that, because the localized memory (Figure 3.1(c)) does not contain $\langle l_5, b\rangle$, the update to location $\langle l_5, b\rangle$ at line 8 does not cause `f` to be re-analyzed at the subsequent call to `f` (line 8). On the other hand, with reachability-based localization, `f` will be analyzed again at the second call.

```
1  struct S { int a; int b; }
2  int g = 0;
3  void f (S* p) { p->a = 0; }
4  void main() {
5      S *s = (S*)malloc(sizeof S);
6      s->a = 0;
7      s->b = 0; f(s);   // first call to f
8      s->b = 1; f(s);  // second call to f
9  }
```

$s \mapsto \langle l_5, \{a, b\}\rangle$	$p \mapsto \langle l_5, \{a, b\}\rangle$	$p \mapsto \langle l_5, \{a, b\}\rangle$
$p \mapsto \langle l_5, \{a, b\}\rangle$	$\langle l_5, a\rangle \mapsto [0, 0]$	$\langle l_5, a\rangle \mapsto [0, 0]$
$\langle l_5, a\rangle \mapsto [0, 0]$	$\langle l_5, b\rangle \mapsto [0, 0]$	
$\langle l_5, b\rangle \mapsto [0, 0]$	$g \mapsto [0, 0]$	
$g \mapsto [0, 0]$		

(a) Non-localized memory (b) Reachability-based (c) Access-based

Figure 3.1: Example code and abstract memories (non-localized, reachability-based localized memory, access-based localized memory) right before the first call-site.

Outline Section 3.2 formulates the conventional reachability-based localization technique. Section 5.3 develops our approach on top of the reachability-based technique. Section 3.5 shows experimental results.

3.2 Reachability-Based Localization

In this section, we define the reachability-based localization of abstract memories. We formalize the technique on top of our baseline analyzer Airac. We call our analyzer with the technique $\mathsf{Airac}_{\mathsf{Reach}}$. Before discussion, we need to clarify the meaning of 'accessed'. We say an abstract location $a \in \hat{\mathbb{L}}$ is accessed if an abstract value is read by referencing the abstract location or an abstract value is written to a.

Given an input abstract memory to a procedure, unless we actually analyze the procedure with the input memory, it is generally impossible to infer the exact set of abstract locations that will be accessed during the analysis of the proce-

dure. Instead, the conventional localization technique uses a reachability heuristic, which is based on the simple intuition that unreachable abstract locations cannot be accessed in the future of the analysis. With reachability-based localization, procedures are analyzed as follows:

1. Before entering the procedure, the analysis stops and searches the input memory to collect abstract locations that are reachable from abstract locations for global variables or actual parameters.

2. Then, the input memory is restricted on the collected, reachable abstract locations.

3. The called procedure is analyzed with only the reachable portion of the input memory, not the entire input memory.

Formally, given a call $\mathsf{call}(f_x, e)$ and its input abstract memory state $\hat{s}$ (parameter-bound), $\mathsf{Airac}_{\mathsf{Reach}}$ computes the following set of abstract locations (let $\mathsf{Globals}$ be the set of abstract locations that represent global variables in the program):

$$\mathcal{R}(f_x, \hat{s}) = \mathsf{Reach}(\mathsf{Globals}, \hat{s}) \cup \mathsf{Reach}(\{x\}, \hat{s})$$

which is the union of abstract locations that are reachable from global variables ($\mathsf{Globals}$) and those that are reachable from the parameter (x). We use $\mathsf{Reach}(X, \hat{s})$ to denote the set of abstract locations in $\hat{s}$ that are (directly or transitively) reachable from the location set X.

$$\mathsf{Reach}(X, \hat{s}) = \mathbf{lfp}(\lambda Y.X \cup \mathsf{DirectReach}(Y, \hat{s}))$$

$\mathsf{DirectReach}(X, \hat{s})$ is the set of locations that are directly reachable from X:

$$\mathsf{DirectReach}(X, \hat{s}) = \bigcup_{x \in X} \hat{s}(x).2 \cup \{l \mid \langle l, o, s\rangle \in \hat{s}(x).3\} \cup \{\langle l, f\rangle \mid \langle l, \{f\}\rangle \in \hat{s}(x).4\}$$

Given an input memory $\hat{s}$ to a call-point c (such that $\mathsf{cmd}(c) = \mathsf{call}(f_x, e)$), the definition of the semantic function $\hat{f}_c$ is changed as follows: (Note that the output

from the semantic function is the memory that flows into the body of the called procedure, not the memory that is returned from the body.)

$$\hat{f}_c(\hat{s}) = \hat{s}'|_{\mathcal{R}(f_x, \hat{s}')} \text{ where } \hat{s}' = \hat{s}[x \mapsto \hat{\mathcal{V}}(e)(\hat{s})]$$

That is, when a procedure is being analyzed, its input abstract memory is restricted to reachable abstract locations.

We also have to consider procedure returns. When a procedure returns to a return-point, in order to recover the local information from the corresponding call-point, we combine the returned memory with the memory parts that were not propagated to called procedures from the call-point. Note that the issue of cut-points [59] is not involved in our analysis because our semantics is store-based and hence every object is represented by a fixed location.

3.3 Access-Based Localization

This section describes our access-based localization technique. Our technique more tightly localizes abstract memories than the reachability-based technique does: whereas the reachability-based approach only strips away unreachable abstract locations, our method additionally filters out the abstract locations that are reachable but definitely not to-be-accessed during the analysis. We call our approach access-based localization.

In order to compute such a tighter information, we take a "static" approach. That is, with our technique, all the abstract locations that will be accessed during the analysis are already prepared from the beginning of the analysis. On the other hand, the reachability-based localization is "dynamic" in a sense that the estimation process, which collect possibly accessed abstract locations, is simultaneously performed during the analysis. We separate the entire analysis into two phases:

1. **Pre-analysis**: the set of abstract locations that are accessed during the actual analysis of each procedure are conservatively estimated.

2. **Actual analysis**: the actual analysis uses the access-information and filters out memory entries that will definitely not be accessed by called procedures.

A careful design of pre-analysis is important for both safety and efficiency of our approach. To be safe, the pre-computed access information should be conservative with respect to the abstract locations that would be accessed during the actual analysis. To be useful, it should be efficient enough to compensate for the extra burden of running pre-analysis once more.

In Section 3.3.1, we design a pre-analysis and prove its safety. Its practical efficiency is experimentally proven in Section 3.5. Section 3.3.2 describes the actual analysis. We write $\mathsf{Airac}_{\mathsf{ProcAcc}}$ for the analyzer that performs access-based localization for procedures.

3.3.1 Pre-analysis

The pre-analysis aims to compute a map $\mathsf{access} \in \mathbb{C} \to 2^{\hat{\mathbb{L}}}$ that, for each control point, conservatively estimates a set of abstract locations that are possibly accessed during the actual analysis of the point. In order to find such a map, we use the following strategy:

- We define an 'access function' that computes access information of each control point. The access function is a summarized version of semantics function $\hat{f}_c$ that focuses only on the access behaviors instead of describing full semantics.
- We define a conservative abstraction of the original analysis.
- By using the access function and over-approximated analysis results, the accessed abstract locations for each control point are conservatively estimated.

Access Function We define the access function, $\mathcal{A} \in \mathbb{C} \to \hat{\mathbb{S}} \to 2^{\hat{\mathbb{L}}}$, that computes which abstract locations are accessed during the analysis of a control point on an input memory state. The intuition is that, given control point c and its input abstract memory $\hat{s}$, $\mathcal{A}(c)(\hat{s})$ computes the set of abstract locations that are

accessed during the evaluation of $\hat{f}_c(\hat{s})$. Note that an abstract location a is accessed during the evaluation of $\hat{f}_c(\hat{s})$ if a is referenced (i.e., $\hat{s}(a)$ appears in the definition of $\hat{f}_c$) or a value v is written to a (i.e., $\hat{s}[a \mapsto v]$ appears in the definition) during the evaluation.

In order to define function $\mathcal{A}$, we first define two sub-functions $\hat{\mathcal{AV}} \in e \to \hat{\mathbb{S}} \to 2^{\hat{\mathbb{L}}}$ and $\hat{\mathcal{AL}} \in lv \to \hat{\mathbb{S}} \to 2^{\hat{\mathbb{L}}}$. Given an expression e and a memory state $\hat{s}$, $\hat{\mathcal{AV}}(e)(\hat{s})$ computes abstract locations that are accessed during the evaluation of $\hat{\mathcal{V}}(e)(\hat{s})$. Similarly, $\hat{\mathcal{AL}}(lv)(\hat{s})$ computes abstract locations that are accessed during the evaluation of $\hat{\mathcal{L}}(lv)(\hat{s})$. $\hat{\mathcal{AV}}$ and $\hat{\mathcal{AL}}$ are defined as follows.

$$
\begin{array}{rcl}
\hat{\mathcal{AV}}(e) & \in & \hat{\mathbb{S}} \to 2^{\hat{\mathbb{L}}} \\
\hat{\mathcal{AV}}(n)(\hat{s}) & = & \emptyset \\
\hat{\mathcal{AV}}(e_1 \texttt{ + } e_2)(\hat{s}) & = & \hat{\mathcal{AV}}(e_1)(\hat{s}) \cup \hat{\mathcal{AV}}(e_2)(\hat{s}) \\
\hat{\mathcal{AV}}(lv)(\hat{s}) & = & \hat{\mathcal{AL}}(lv)(\hat{s}) \cup \hat{\mathcal{L}}(lv)(\hat{s}) \\
\hat{\mathcal{AV}}(\&lv)(\hat{s}) & = & \hat{\mathcal{AL}}(lv)(\hat{s})
\end{array}
\qquad
\begin{array}{rcl}
\hat{\mathcal{AL}}(lv) & \in & \hat{\mathbb{S}} \to 2^{\hat{\mathbb{L}}} \\
\hat{\mathcal{AL}}(x)(\hat{s}) & = & \emptyset \\
\hat{\mathcal{AL}}(*e)(\hat{s}) & = & \hat{\mathcal{AV}}(e)(\hat{s}) \\
\hat{\mathcal{AL}}(e_1\texttt{[}e_2\texttt{]})(\hat{s}) & = & \hat{\mathcal{AV}}(e_1)(\hat{s}) \\
\hat{\mathcal{AL}}(e\texttt{.}x)(\hat{s}) & = & \hat{\mathcal{AV}}(e)(\hat{s})
\end{array}
$$

We easily notice that these definitions are naturally derived from the definitions of semantic functions $\hat{\mathcal{V}}$ and $\hat{\mathcal{L}}$. Consider $\hat{\mathcal{AV}}$ (defined in the left column) first. When $e = n$, we see that the definition of $\hat{\mathcal{V}}$ (in Section 2.4.2) does not read (nor write to) any location, and hence there are no accessed locations ($\emptyset$). When $e = e_1$ + e_2, accessed locations are just collected recursively. When an l-value lv is used as an r-value (the third case), from the definition of $\hat{\mathcal{V}}(lv)(\hat{s})$, we see that abstract locations of lv and abstract locations that are accessed during the evaluation of $\hat{\mathcal{L}}(lv)(\hat{s})$ are accessed, which are collected by $\hat{\mathcal{L}}(lv)(\hat{s})$ and $\hat{\mathcal{AL}}(lv)(\hat{s})$, respectively. When an l-value lv is used as an address-of expression (the last case), from the definition of $\hat{\mathcal{V}}(\&lv)(\hat{s})$, we see that abstract locations that lv denotes ($\hat{\mathcal{L}}(lv)(\hat{s})$) are not accessed during the evaluation of $\hat{\mathcal{V}}(\&lv)(\hat{s})$ and hence the fourth case only includes $\hat{\mathcal{AL}}(lv)(\hat{s})$. Similarly, the definition of $\hat{\mathcal{AL}}$ (defined in the right column) is derived from the definition of $\hat{\mathcal{L}}$. $\hat{\mathcal{AL}}(x)(\hat{s})$ is $\emptyset$ because $\hat{\mathcal{L}}(x)(\hat{s})$ just produces a location x but does not read (resp., write) any value from (resp., to) x. The second case holds because computing $\hat{\mathcal{L}}(*e)(\hat{s})$ only the accesses locations that are

accessed during the evaluation of e. Likewise, the fourth case holds. Lastly, the third case holds: we collect only the locations accessed during the evaluation of e_1 because $\hat{\mathcal{L}}(e_1\,\texttt{[}e_2\texttt{]})(\hat{s})$ does not involve the evaluation of e_2 (we smash all the array elements into one memory cell). Formally, the following lemmas show that $\hat{\mathcal{AV}}$ and $\hat{\mathcal{AL}}$ are correct and monotone.

Lemma 2 *For all expression e, lv, and input memory $\hat{s}$,*

$$a \in \hat{\mathcal{AV}}(e)(\hat{s}) \Leftrightarrow a \textit{ is accessed during the evaluation of } \hat{\mathcal{V}}(e)(\hat{s})$$
$$a \in \hat{\mathcal{AL}}(lv)(\hat{s}) \Leftrightarrow a \textit{ is accessed during the evaluation of } \hat{\mathcal{L}}(lv)(\hat{s})$$

Proof By structural induction on e and lv. For example, consider the case for $e = e_1 \texttt{ + } e_2$:

$$\begin{array}{lll} & a \in \hat{\mathcal{AV}}(e_1 \texttt{ + } e_2)(\hat{s}) & \\ \Leftrightarrow & a \in \hat{\mathcal{AV}}(e_1)(\hat{s}) \cup \hat{\mathcal{AV}}(e_2)(\hat{s}) & \cdots \text{ def. of } \hat{\mathcal{AV}} \\ \Leftrightarrow & a \text{ is accessed in } \hat{\mathcal{V}}(e_1)(\hat{s}) \text{ or } \hat{\mathcal{V}}(e_2)(\hat{s}) & \cdots \text{ induction hypothesis} \\ \Leftrightarrow & a \text{ is accessed in } \hat{\mathcal{V}}(e_1 \texttt{ + } e_2)(\hat{s}) & \cdots \text{ def. of } \hat{\mathcal{V}}(e_1 \texttt{ + } e_2)(\hat{s}) \end{array}$$

Other cases are proved similarly. □

Lemma 3 *For all expression e, l-value lv and abstract memory states m and m',*

$$m \sqsubseteq m' \Rightarrow \hat{\mathcal{AV}}(e)(m) \sqsubseteq \hat{\mathcal{AV}}(e)(m')$$
$$m \sqsubseteq m' \Rightarrow \hat{\mathcal{AL}}(lv)(m) \sqsubseteq \hat{\mathcal{AL}}(lv)(m')$$

Proof By structural inductions on e and lv. □

Now we define the access function $\mathcal{A}$ that computes abstract locations that are accessed during the analysis of each command:

$$\mathcal{A}(c)(\hat{s}) = \begin{cases} \hat{\mathcal{A}}\hat{\mathcal{L}}(lv)(\hat{s}) \cup \hat{\mathcal{A}}\hat{\mathcal{V}}(e)(\hat{s}) \cup \hat{\mathcal{L}}(lv)(\hat{s}) & \mathsf{cmd}(c) = lv := e \\ \hat{\mathcal{A}}\hat{\mathcal{L}}(lv)(\hat{s}) \cup \hat{\mathcal{A}}\hat{\mathcal{V}}(e)(\hat{s}) \cup \hat{\mathcal{L}}(lv)(\hat{s}) & \mathsf{cmd}(c) = lv := \mathsf{alloc}([e]_l) \\ \hat{\mathcal{A}}\hat{\mathcal{L}}(lv)(\hat{s}) \cup \hat{\mathcal{L}}(lv)(\hat{s}) & \mathsf{cmd}(c) = lv := \mathsf{alloc}(\{x\}_l) \\ \hat{\mathcal{A}}\hat{\mathcal{V}}(e)(\hat{s}) \cup \{x\} & \mathsf{cmd}(c) = \mathsf{assume}(x < e) \\ \hat{\mathcal{A}}\hat{\mathcal{V}}(e)(\hat{s}) \cup \{x\} & \mathsf{cmd}(c) = \mathsf{call}(f_x, e) \\ \emptyset & \mathsf{cmd}(c) = \mathsf{return}_f \end{cases}$$

The following lemmas show that $\mathcal{A}$ is correct and monotone.

Lemma 4 *For all $c \in \mathbb{C}, \hat{s} \in \hat{\mathbb{S}}, a \in \hat{\mathbb{L}}$,*

$$a \in \mathcal{A}(c)(\hat{s}) \Leftrightarrow a \text{ is accessed during the evaluation of } \hat{f}_c(\hat{s})$$

Proof By case analysis on type of $\mathsf{cmd}(c)$. For example, consider the case for $\mathsf{cmd}(c) = lv := e$:

$$\begin{array}{lll} & a \in \mathcal{A}(c)(\hat{s}) & \\ \Leftrightarrow & a \in \hat{\mathcal{A}}\hat{\mathcal{L}}(lv)(\hat{s}) \cup \hat{\mathcal{A}}\hat{\mathcal{V}}(e)(\hat{s}) \cup \hat{\mathcal{L}}(lv)(\hat{s}) & \cdots \text{ def. of } \mathcal{A} \\ \Leftrightarrow & a \text{ is accessed in } \hat{\mathcal{L}}(lv)(\hat{s}) \text{ or } \hat{\mathcal{V}}(e)(\hat{s}) \text{ or } a \in \hat{\mathcal{L}}(lv)(\hat{s}) & \cdots \text{ lemma 2} \\ \Leftrightarrow & a \text{ is accessed in } \hat{f}_c(\hat{s}) & \cdots \text{ def. of } \hat{f} \end{array}$$

Other cases are proved similarly. □

Lemma 5 *$\mathcal{A}$ is monotone, i.e., $\forall \hat{s}, \hat{s}' \in \hat{\mathbb{S}}$ and $\forall c \in \mathbb{C}$,*

$$\hat{s} \sqsubseteq \hat{s}' \Rightarrow \mathcal{A}(c)(\hat{s}) \sqsubseteq \mathcal{A}(c)(\hat{s}')$$

Proof By case analysis on type of $\mathsf{cmd}(c)$ and lemma 3.

□

Using the access function $\mathcal{A}$, we can estimate accessed locations. Suppose $\hat{X}$ is the analysis results for the original analysis, i.e., $\hat{X} = \mathbf{lfp}(\hat{F})$. Then, because $\mathcal{A}$ is

monotone, all the abstract locations that are accessed at c throughout the analysis are captured by $\mathcal{A}(c)(\hat{s})$, where $\hat{s} = \bigsqcup_{c' \hookrightarrow c} \hat{X}(c')$ is the input abstract memory at fixpoint. However, because $\hat{X}$ itself is computed from the original analysis ($\mathbf{lfp}(\hat{F})$), the accessed-locations-estimation phase would take at least as the same time as the actual analysis. We have to find the accessed locations in a more efficient way. We do this by computing $\hat{X}'$ that is more approximate than $\hat{X}$, i.e., $\hat{X} \sqsubseteq \hat{X}'$.

Deriving a Further Abstraction We define a pre-analysis that computes such a $\hat{X}'(\sqsupseteq \hat{X})$. To this end, we apply a conservative abstraction to the original analysis. The abstract domain $\mathbb{C} \to \hat{\mathbb{S}}$ and semantic function $\hat{F} \in (\mathbb{C} \to \hat{\mathbb{S}}) \to (\mathbb{C} \to \hat{\mathbb{S}})$ for the original (actual) analysis was defined as follows (the following is just a repetition, for convenience, of the definition in Section 2.4) :

$$\hat{F}(\hat{X}) = \lambda c \in \mathbb{C}.\hat{f}_c(\bigsqcup_{c' \hookrightarrow c} \hat{X}(c')). \tag{3.1}$$

We apply a simple abstraction that ignores the orders of program statements (flow-insensitivity). The abstract domain is obtained by defining a Galois connection:

$$\mathbb{C} \to \hat{\mathbb{S}} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \hat{\mathbb{S}}$$

such that,

$$\alpha = \lambda \hat{X}. \bigsqcup_{c \in \mathbb{C}} \hat{X}(c)$$
$$\gamma = \lambda \hat{s}.\lambda c \in \mathbb{C}.\hat{s}$$

The semantic function $\hat{F}_p : \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ is defined as follows:

$$\hat{F}_p = \lambda \hat{s}.(\bigsqcup_{c \in \mathbb{C}} \hat{f}_c(\hat{s}))$$

The following lemma shows that the pre-analysis is a conservative approximation of the original analysis.

Lemma 6 $\mathbf{lfp}(\hat{F}) \sqsubseteq \gamma(\mathbf{lfp}(\hat{F}_p))$

Proof By the help of the fixpoint transfer theorem [12], showing the following two properties

- α and γ are related by Galois connection
- $\alpha \circ \hat{F} \sqsubseteq \hat{F}_p \circ \alpha$

corresponds to showing $\mathbf{lfp}\hat{F} \sqsubseteq \gamma(\mathbf{lfp}(\hat{F}_p))$.

For the first part (Galois connection), we have to show $\forall x \in \mathbb{C} \to \hat{\mathbb{S}}, y \in \hat{\mathbb{S}}.\ \alpha(x) \sqsubseteq y \Leftrightarrow x \sqsubseteq \gamma(y)$.

- $\alpha(x) \sqsubseteq y \Rightarrow x \sqsubseteq \gamma(y)$:

$$\begin{array}{lll} & \alpha(x) \sqsubseteq y & \\ \Rightarrow & \bigsqcup_{c \in \mathbb{C}} x(c) \sqsubseteq y & \cdots \text{ def. of } \alpha \\ \Rightarrow & \forall c \in \mathbb{C}. x(c) \sqsubseteq y & \cdots \forall c \in \mathbb{C}. x(c) \sqsubseteq \bigsqcup_{c' \in \mathbb{C}} x(c') \\ \Rightarrow & x \sqsubseteq \lambda c.y & \cdots \text{ def. of } \sqsubseteq \text{ on } \mathbb{C} \to \hat{\mathbb{S}} \\ \Rightarrow & x \sqsubseteq \gamma(y) & \cdots \text{ def. of } \gamma \end{array}$$

- $\alpha(x) \sqsubseteq y \Leftarrow x \sqsubseteq \gamma(y)$:

$$\begin{array}{lll} & x \sqsubseteq \gamma(y) & \\ \Rightarrow & x \sqsubseteq \lambda c.y & \cdots \text{ def. of } \gamma \\ \Rightarrow & \forall c. x(c) \sqsubseteq y \ (= (\lambda c.y)(c)) & \cdots \text{ def. of } \sqsubseteq \text{ on } \mathbb{C} \to \hat{\mathbb{S}} \\ \Rightarrow & \bigsqcup_{c \in \mathbb{C}} x(c) \sqsubseteq y & \cdots \ y \text{ is an upper bound of } \{x(c) \mid c \in \mathbb{C}\} \\ \Rightarrow & \alpha(x) \sqsubseteq y & \cdots \text{ def. of } \alpha \end{array}$$

Next, we show $\alpha \circ \hat{F} \sqsubseteq \hat{F}_p \circ \alpha$, i.e., $\forall d \in \mathbb{C} \to \hat{\mathbb{S}}. \alpha(\hat{F}(d)) \sqsubseteq \hat{F}_p(\alpha(d))$.

$$\begin{array}{lll} \alpha(\hat{F}(d)) = & \alpha(\lambda c. \hat{f}_c(\bigsqcup_{c' \hookrightarrow c} d(c'))) & \cdots \text{ def. of } \hat{F} \\ = & \bigsqcup_{c \in \mathbb{C}} \hat{f}_c(\bigsqcup_{c' \hookrightarrow c} d(c')) & \cdots \text{ def. of } \alpha \\ \sqsubseteq & \bigsqcup_{c \in \mathbb{C}} \hat{f}_c(\bigsqcup_{c' \in \mathbb{C}} d(c')) & \cdots \forall c \in \mathbb{C}. \hat{f}_c \text{ is mono., } \{c' \mid c' \hookrightarrow c\} \subseteq \mathbb{C} \\ = & \bigsqcup_{c \in \mathbb{C}} \hat{f}_c(\alpha(d)) & \cdots \text{ def. of } \alpha \\ = & \hat{F}_p(\alpha(d)) & \cdots \text{ def. of } \hat{F}_p \end{array}$$

$old, new \in \hat{\mathbb{S}}$

$Preliminary\ (old, new) =$
$new := \bot_{\hat{\mathbb{S}}}$
repeat
  $old := new$
  for all $c \in \mathbb{C}$ **do**
    $new := \hat{f}_c(new)$
until $new \sqsubseteq old$
return new

Figure 3.2: The fixpoint algorithm for our pre-analysis. For brevity, we omit the widening operation, which is necessary for analysis' termination.

□

In general, the pre-analysis can be derived using any conservative abstraction of the original analysis. However, we chose the above two abstractions because it is efficient enough in practice and it is precise enough to track reachability along the dynamically allocated locations and structure fields. We believe that filtering out not only unused variables but also unused allocated locations and fields is vital for the performance of our localization technique.

Figure 3.2(b) shows the fixpoint algorithm for the pre-analysis. It is based on a flow-insensitive fixpoint computation. The analysis starts with a bottom memory state ($\bot_{\hat{\mathbb{S}}}$). The state is iteratively updated by flow functions for all control points in the program until the resulting state is subsumed by the state of the previous iteration. Let $\hat{s}_{pre}$ be the analysis results from the pre-analysis.

3.3.2 Actual Analysis

The actual analysis is the same as Airac except that now we use the access information ($\mathcal{A}$) and localizes memory states. Given an input memory state $\hat{s}$ to a

call-point c (such that $\mathsf{cmd}(c) = \mathsf{call}(f_x, e)$), the semantic function $\hat{f}_c$ for the call statement is changed as follows:

$$\hat{f}_c(\hat{s}) = \hat{s}'|_{\mathsf{access}(f)} \text{ where } \hat{s}' = \hat{s}[x \mapsto \hat{\mathcal{V}}(e)(\hat{s})]$$

After parameter bound ($\hat{s}'$), the memory state is restricted to the set of accessed locations $\mathsf{access}(f)$ that represents the set of abstract locations that are accessed by procedure f:

$$\mathsf{access}(f) = \bigcup_{g \in \mathsf{callees}(f)} (\bigcup_{c \in \mathsf{control}(g)} \mathcal{A}(c)(\hat{s}_{pre}))$$

where $\mathsf{callees}(f)$ denotes the set of procedures, including f, that are reachable from f via the call-graph and $\mathsf{control}(f)$ the set of control points in procedure f, and $\hat{s}_{pre}$ is the analysis result from the pre-analysis. The following theorem ensures the safety of the localization.

Theorem 1 (Safety of Access-based Localization) *For all procedure f, $\mathsf{access}(f)$ conservatively estimates abstract locations that are accessed during the original analysis of f.*

Proof Abstract location a is accessed inside procedure f if and only if it is accessed either in the body of f or in the bodies of procedures that are called by (reachable via call-graph from) f, which is the definition of access. Moreover, because $\hat{s}_{pre}$ conservatively approximates the abstract memories of all program points (lemma 6) and $\mathcal{A}$ is monotone (lemma 4), $\mathcal{A}(n)(\hat{s}_{pre})$ contains all the abstract locations that would be accessed in actual analysis. Thus, access is a safe estimation of accessed locations.

□

Access-based localization can be used in combination with the reachability-based approach to localize memory states more aggressively. Given an input memory state $\hat{s}$ to a call point c such that $\mathsf{cmd}(c) = \mathsf{call}(f_x, e)$, reachable locations $\mathcal{R}(f_x, \hat{s})$,

and accessed locations $\mathsf{access}(f)$, the semantic function $\hat{f}$ for the call statement $\mathsf{call}(f_x, e)$ is changed as follows:

$$\hat{f}_c(\hat{s}) = (\hat{s}'|_{\mathcal{R}(f_x,\hat{s}')})|_{\mathsf{access}(f)} \text{ where } \hat{s}' = \hat{s}[x \mapsto \hat{\mathcal{V}}(e)(\hat{s})]$$

After parameter binding ($\hat{s}'$) the memory is first restricted to the reachable locations ($\mathcal{R}(f_x, \hat{s}')$) and then the resulting memory is restricted to $\mathsf{access}(f)$. The reason why we restrict the memory to $\mathcal{R}(f_x, \hat{s}') \cap \mathsf{access}(f)$ is that $\mathsf{access}(f)$ may have locations that are unreachable, i.e., not contained in $\mathcal{R}(f_x, \hat{s}')$, because $\hat{s}_{pre}$ is computed by the less precise pre-analysis but $\mathcal{R}(f_x, \hat{s}')$ is computed during the more precise actual analysis. Hence, the memory states localized by the combination of reachability- and access-based approach are always smaller than those localized by the reachability-based approach.

3.4 Extensions of Access-based Localization

Access-based approach to localization allows two meaningful extensions. In Section 3.4.1, we extends the access-based localization technique to support arbitrary code blocks, and, in Section 3.4.2, we refine the technique to support efficient handling of recursive call cycles.

3.4.1 Localization for Arbitrary Code Blocks

Generalizing the idea, we can apply the access-based localization technique for code blocks smaller than procedures. Given a code block, it is straightforward to collect accessed locations for the block because our pre-analysis provides access information ($\mathcal{A}$) for each node in the control flow graph. We localize the input memories to the block according to the access information for the block, and analyze the block with the localized memory state, which avoids re-analyses of blocks and speeds up memory operations. We select localization target blocks before starting the actual analysis.

For effectiveness, we have to carefully select blocks to apply localization. Localization improves the analysis performance, but at the same time, introduces a performance overhead. At the entry of a selected block, additional set-operations to localize the input memory state have to be performed and at the exit of the block, non-localized memory portions of the input memory have to be merged with the output of the block. In order to balance against the localization overhead, we select code blocks $\langle entry, exit, B\rangle$ that consists of one *entry* node, one *exit* node, and a selected block B that satisfy the following properties:

- the *entry* (respectively, the *exit*) node strictly dominates (respectively, post-dominates) all nodes in B, and B contains all nodes that are strictly dominated and post-dominated by the *entry* and *exit*, respectively
- code block size $|B| \geq k$ for parameter k

Using the parameter k, we are able to find a balance between actual reduction and overhead introduced by localizing operations. The above selection strategy is applied recursively: a block satisfying the requirements can be selected inside another selected block.

Example 1 *Consider the control flow graph of a procedure in Figure 3.3 . The dashed nested boxes are the selected blocks by our algorithm when* $k = 2$*. First, the entire body of the procedure is selected (*B_1*). Inside* B_1*, the algorithm selects* B_2, B_3, B_4, B_5 *and* B_6 *recursively. As an example, consider block* B_4 *whose entry and exit are node 5 and 14, respectively. We localize* B_4*'s input memory (the output memory of node 5) according to the set of abstract locations accessed by* B_4 *(the set of nodes* $6, 7, 8, 9, 10, 11, 12, 13$*). And the non-localized memory portions at the entry (node 5) are merged with output memory of* B_4 *at the exit (node 14).*

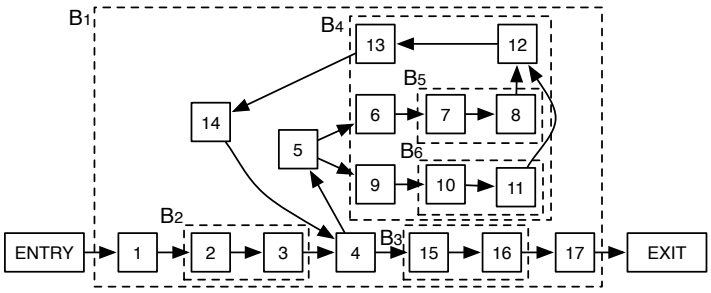


Figure 3.3: An example for access-based localization for arbitrary code blocks.

3.4.2 Access-based Localization with Bypassing

Any localization technique has a source of inefficiency in handling procedure calls. In access-based localization[1], the localized input state for a procedure involves not only the abstract locations that are accessed by the called procedure but also those locations that are accessed by transitively called procedures. For instance, when procedure f calls g, the localized state for f contains abstract locations that are accessed by g as well as abstract locations accessed by f. Those locations that are exclusively accessed by g are, however, irrelevant to the analysis of f because they are not used in analyzing f. Even so, those locations are involved in the localized state (for f), which sometimes leads to unnecessary computational cost (due to re-analyses of procedure body).

Such inefficiency is especially exacerbated with recursive call cycles. Consider a recursive call cycle $f \to g \to h \to f \to \cdots$. Because of the cyclic dependence among procedures, every procedure receives input memories that contain all abstract locations accessed by the whole cycle. That is, access-based localization does not help any more inside call cycles. Moreover, recursive cycles (even large ones) are common in real C programs. For example, in GNU open source code, we noticed that a number of programs have large recursive cycles and a single cycle some-

[1] In fact, any localization techniques suffers from similar problems. Here, we discuss the problem in the context of access-based localization.

times contains more than 40 procedures. This is the main performance bottleneck of access-based localization in practice (Section 3.5.2).

In this section, we extend access-based localization technique so that the aforementioned inefficiency can be relieved. With our technique, localized states for a procedure contains only the abstract locations that are accessed by the procedure and does not contain other locations that are exclusively accessed by transitively called procedures. Those excluded abstract locations are "bypassed" to the transitively called procedures, instead of passing through the called procedure. In this way, analysis of a procedure involves only the memory parts that the procedure directly accesses (even inside recursive cycles), which results in more tight localization and hence reduces analysis cost more than access-based localization does. The following example illustrates how our technique saves cost.

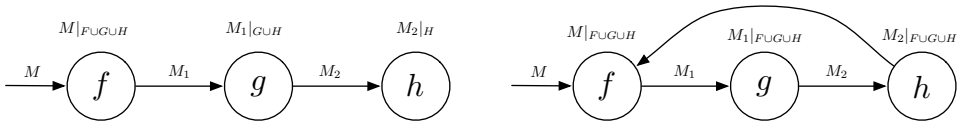
Example 2 *Consider the following code.*

```
1  int a=0, b=0;
2  void g() { b++; }
3  void f() { a++; g(); }
4  int main () {
5     b=1; f();       // first call to f
6     b=2; f();       // second call to f
7  }
```

Procedure `main` *calls* `f`*, and* `f` *calls* `g`*. Procedures* `f` *and* `g` *update the value of* `a` *and* `b`*, respectively. Procedure* `main` *calls* `f` *two times with the value of* `b` *changed.*

- *With access-based localization: Both* `f` *and* `g` *are analyzed two times. The localized input memory for* `f` *at the first call (line 5) contains locations* `a` *and* `b` *because both are (directly/indirectly) accessed while analyzing* `f`*. The localized state at the second call (line 6) contains the same locations. Because the value of* `b` *is changed,* `f` *(as well as* `g`*) is re-analyzed at the second call.*

- *With our technique:* `f` *is analyzed only once (though* `g` *is analyzed twice). Localized memories for procedure* `f` *contain only the location that* `f` *directly accesses, i.e.,* `a`*. The value of* `a` *is not changed and the body of* `f` *is not re-*



(a) Non-recursive call chain (b) Recursive call cycle

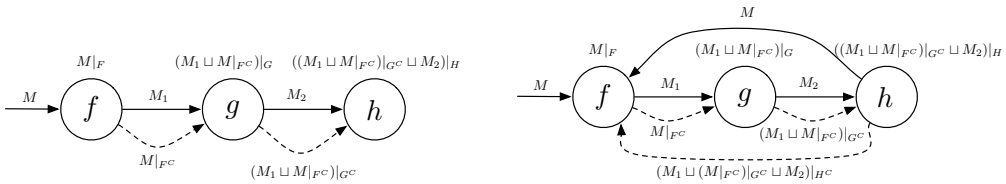
Figure 3.4: Problem of localization. F (respectively, G and H) denotes the set of abstract locations that procedure f (respectively, g and h) directly accesses. $M|_F$ denotes the memory state M with projected on abstract locations F.

> *analyzed at the second call. However, procedure* `g` *is re-analyzed because we propagate the changed value of* `b` *to the entry of* `g`*.*

We illustrate how our technique works with examples. Figure 3.4 shows example call graphs. There are three procedures: f, g and h. Suppose F (respectively, G and H) denotes the set of abstract locations that procedure f (respectively, g and h) directly accesses. We describe how the problem occurs and then how to overcome the problem.

Access-based localization has inefficient aspects in analyzing procedure calls. We first consider the case for non-recursive call chains (Figure 3.4(a)). With the localization, the input memory M to f is localized so that the procedure f is analyzed only with a subpart $M|_{F\cup G\cup H}$ (M with projected on locations set $F\cup G\cup H$) rather than the entire input memory. Similarly, the input memory M_1 to g is localized to $M|_{G\cup H}$, and h's input memory M_2 is localized to $M_2|_H$. The inefficiency comes from the fact that not the entire localized memory is accessed by each procedure. For example, abstract locations $G \cup H$ are not necessary in analyzing the body of f.

The problem becomes severe when analyzing recursive call cycles. Consider Figure 3.4(b). As in the previous case, the input memory M to f is localized to $M|_{F\cup G\cup H}$. However, in this case, the input memory M_1 to g is also projected on $F \cup G \cup H$, not on $G \cup H$, because f can be called from g through the recursive cycle. Similarly, input memory M_2 to h is localized to $M|_{F\cup G\cup H}$. In summary, localization does not work any more inside the cycle.



(a) Non-recursive call chain (b) Recursive call cycle

Figure 3.5: Illustration of our technique. With our technique, each procedure is analyzed with its respective directly accessed locations, and others are bypassed (dashed line) to the subsequent procedure.

Figure 3.5 illustrates how our technique works. We first consider non-recursive call case (Figure 3.5(a)). Instead of restricting f's input memory to $F \cup G \cup H$, we localize it with respect to only the directly accessed locations, i.e., F. Thus, f is analyzed with $M|_F$. The non-localized memory part ($M|_{F^C}$) is directly bypassed (dashed line) to g. Then, the output memory M_1 from f and the bypassed memory $M|_{F^C}$ are joined to prepare input memory $M_1 \sqcup M|_{F^C}$ for procedure g. The input memory is localized to $(M_1 \sqcup M|_{F^C})|_G$ and g is analyzed with the localized memory. Again, the non-localized parts $(M_{F^C} \sqcup M_1)|_{G^C}$ are bypassed to the subsequent procedure h. In this way, each procedure is analyzed only with abstract locations that the procedure directly accesses.

The technique is naturally applicable to recursive cycles (Figure 3.5(b)). With our technique, even procedures inside recursive call cycles are analyzed with memory parts that are directly accessed by each procedure. Hence, in Figure 3.5(b), the localized memory for f (resp., g and h) only contains locations F (resp., G and H).

3.5 Experiments

We check the performance of our new localization techniques by experiments with Airac, a global abstract interpretation engine in an industrialized bug-finding analyzer [29, 31, 48–51].

3.5.1 Setting

From our baseline analyzer Airac, which does not use any localization technique, we have made four analyzers $\mathsf{Airac}_{\mathsf{Reach}}$, $\mathsf{Airac}_{\mathsf{ProcAcc}}$, $\mathsf{Airac}_{\mathsf{GenAcc}}$, and $\mathsf{Airac}_{\mathsf{Bypass}}$ that respectively use procedure-level reachability-based, procedure-level access-based, generalized access-based localization, and access-based localization with bypassing. Those analyzers differ from Airac only in their respective localization schemes. Hence, performance differences, if any, are solely attributed to the different localization methods. Note that $\mathsf{Airac}_{\mathsf{GenAcc}}$ subsumes $\mathsf{Airac}_{\mathsf{ProcAcc}}$: $\mathsf{Airac}_{\mathsf{GenAcc}}$ implements procedure-level localization (Section 3.3) as well as block-level localization (Section 3.4.1). We set the minimum block size k to 6 for $\mathsf{Airac}_{\mathsf{GenAcc}}$, which was shown to be most efficient in our setting. The analyzers are written in OCaml.

We have analyzed 15 software packages. Table 3.2 shows our benchmark programs. `parser` and `twolf` are from SPEC2000 benchmarks, and the others from GNU open-source programs. The entire program is analyzed starting from the `main` procedure. All experiments were done on a Linux 2.6 system running on a Pentium4 3.2 GHz box with 4 GB of main memory.

We use three performance measures: (1) *#iters* is the total number of iterations during the worklist algorithm (the number of iterations of the outside loop in Table 3.2(a); (2) *time* is the CPU time spent; (3) *MB* is the peak memory consumption.

Program	**LOC**	**Proc**	**Blocks**	Airac (w.o. localization)			$\text{Airac}_{\text{Reach}}$			
				#iters	*time*(sec)	*MB*	*#iters*	*time*(sec)	*MB*	$Save_1$
spell-1.0	2,213	31	782	37,085	46.1	29	23,249	53.0	23	-15.0%
barcode-0.96	4,460	57	2,634	38,742	105.7	291	17,997	92.6	125	12.4%
httptunnel-3.3	6,174	110	2,757	444,354	2808.9	284	201,046	1383.2	154	50.8%
gzip-1.2.4a	7,327	132	6,271	1,327,464	12,756.2	886	393,338	2,866.6	333	77.5%
jwhois-3.0.1	9,344	73	5,147	428,584	3,424.5	633	198,249	1,185.4	254	65.4%
parser	10,900	325	9,298	5,707,185	196,318.8	2,917	2,327,303	60,577.8	1,048	69.1%
bc-1.06	13,093	132	4,924	5,677,277	87,988.5	767	800,474	13,879.2	335	84.2%
twolf	19,700	222	14,610	∞	∞	∞	2,375,894	27,230.3	1,199	N/A
tar-1.13	20,258	221	10,800	6,244,121	157,545.0	2,916	3,819,726	113,061.4	1,797	28.2%
less-382	23,822	382	10,056	7,654,188	148,015.7	2,445	2,998,969	137,827.3	1,480	6.9%
make-3.76.1	27,304	19	11,061	6,162,145	126,908.8	2,757	4,013,647	142,325.6	1,954	-12.1%
wget-1.9	35,018	433	16,544	∞	∞	∞	∞	∞	∞	N/A
screen-4.0.2	44,734	588	31,792	∞	∞	∞	∞	∞	∞	N/A
bison-2.4	56,361	1,203	20,781	∞	∞	∞	∞	∞	∞	N/A
bash-2.05a	105,174	955	28,675	∞	∞	∞	∞	∞	∞	N/A

Table 3.2: Lines of code (**LOC**) are given before preprocessing. The number of basic blocks (**Blocks**) is given after preprocessing. $Save_1$ shows time savings of $\text{Airac}_{\text{Reach}}$ against Airac. *MB* is peak memory consumption in megabytes. Entries with ∞ mean missing data because of the analysis running out of memory.

Program	$\mathsf{Airac}_{\mathsf{ProcAcc}}$				$\mathsf{Airac}_{\mathsf{GenAcc}}(k=6)$			
	#iters	*time*:total(pre)	*MB*	$Save_2$	*#iters*	*time*:total(pre)	*MB*	$Save_3$
spell-1.0	5,512	2.4 (0.2)	5	**95.4%**	4,857	2.1 (0.3)	5	**13.5%**
barcode-0.96	9,433	9.4 (0.6)	25	**89.8%**	7,213	5.5 (1.4)	21	**41.6%**
httptunnel-3.3	17,072	31.4 (1.3)	36	**97.7%**	14,824	21.6 (2.4)	24	**31.0%**
gzip-1.2.4a	78,471	94.8 (1.3)	73	**96.7%**	47,454	54.5 (8.0)	67	**42.5%**
jwhois-3.0.1	99,815	254.8 (1.2)	170	**78.5%**	73,000	188.1 (18.5)	148	**26.2%**
parser	206,173	890.0 (3.8)	224	**98.5%**	173,285	617.0 (9.2)	245	**30.7%**
bc-1.06	146,407	730.9 (4.1)	106	**94.7%**	123,398	542.7 (8.5)	116	**25.8%**
twolf	520,561	1,037.7 (7.5)	332	**96.2%**	337,837	480.4 (20.0)	260	**53.7%**
tar-1.13	360,009	2,524.0 (6.0)	338	**97.8%**	219,109	1,638.3 (15.9)	351	**35.1%**
less-382	1,223,535	26,817.6 (40.7)	466	**80.5%**	833,643	18,766.7 (95.0)	528	**30.0%**
make-3.76.1	1,149,151	19,015.2 (39.4)	580	**86.6%**	894,843	17,405.7 (75.9)	740	**8.5%**
wget-1.9	526,975	6,735.8 (20.8)	609	N/A	366,051	3,823.3 (48.5)	623	**43.2%**
screen-4.0.2	6,402,974	340,849.0 (281.5)	2458	N/A	4,699,777	274,280.3 (667.1)	2958	**19.5%**
bison-2.4	305,988	2,487.3 (13.1)	301	N/A	234,751	1,696.6 (37.7)	302	**31.8%**
bash-2.05a	379,429	2,011.3 (20.2)	439	N/A	251,175	1,142.7 (59.5)	416	**43.2%**

Table 3.3: Performance of our access-based localization. *time* (sec) for $\mathsf{Airac}_{\mathsf{ProcAcc}}$ and $\mathsf{Airac}_{\mathsf{GenAcc}}$ is the total time that includes pre-analysis time. The pre-analysis time is shown inside parentheses. $Save_2$ shows time savings of $\mathsf{Airac}_{\mathsf{ProcAcc}}$ against $\mathsf{Airac}_{\mathsf{Reach}}$. $Save_3$ shows time savings of $\mathsf{Airac}_{\mathsf{GenAcc}}$ against $\mathsf{Airac}_{\mathsf{ProcAcc}}$. Entries with ∞ mean missing data because of the analysis running out of memory.

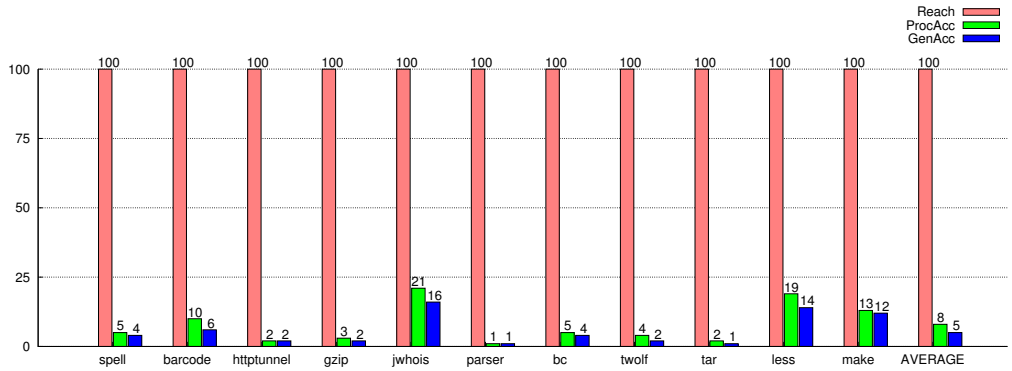


Figure 3.6: Comparison of analysis time among reachability-based, procedure-level access-based, and block-level access-based localizations.

3.5.2 Comparison of Analysis Time

We compare the performance of Airac, $\text{Airac}_{\text{Reach}}$, $\text{Airac}_{\text{ProcAcc}}$, and $\text{Airac}_{\text{GenAcc}}$. Comparison of analysis time among $\text{Airac}_{\text{Reach}}$, $\text{Airac}_{\text{ProcAcc}}$, and $\text{Airac}_{\text{GenAcc}}$ is summarized in Figure 3.6. The analysis results for Airac and $\text{Airac}_{\text{Reach}}$ are shown in Table 3.2. The results for $\text{Airac}_{\text{ProcAcc}}$ and $\text{Airac}_{\text{GenAcc}}$ are shown in Table 3.3. The number of procedures (**Proc**), basic blocks in the program (**Blocks**) in programs are given after preprocessing. *time* for $\text{Airac}_{\text{ProcAcc}}$ and $\text{Airac}_{\text{GenAcc}}$ is the total time that includes pre-analysis time. The pre-analysis time is shown inside parentheses. $Save_1$ shows time savings of $\text{Airac}_{\text{Reach}}$ against Airac. $Save_2$ shows time savings of $\text{Airac}_{\text{ProcAcc}}$ against $\text{Airac}_{\text{Reach}}$. $Save_3$ shows time savings of $\text{Airac}_{\text{GenAcc}}$ against $\text{Airac}_{\text{ProcAcc}}$. Entries with ∞ mean missing data because of the analysis running out of memory.

Airac vs. $\text{Airac}_{\text{Reach}}$ The results show that the reachability-based localization reduces the analysis time and memory for most programs. $\text{Airac}_{\text{Reach}}$ consistently reduces *#iters* of Airac by 54.9% on average and reduces the analysis time by 36.7% on average. The effectiveness is clear from the fact that $\text{Airac}_{\text{Reach}}$ reduces analysis time of Airac more than 50% for programs `httptunnel`, `gzip`, `jwhois`, `parser` and

`bc`. And the peak memory consumption is reduced by on average 46.5%, enabling $\mathsf{Airac}_{\mathsf{Reach}}$ to analyze `twolf` that cannot be analyzed by Airac because of memory cost.

However, the results also show some limitations of the reachability-based localization. First, for two programs (`spell` and `make`), $\mathsf{Airac}_{\mathsf{Reach}}$ took more time than Airac even though $\mathsf{Airac}_{\mathsf{Reach}}$ always reduced *#iters*. This is mainly because of the overhead of localizing operations at procedure calls. Second, it cannot analyze the largest four programs (`wget`, `screen`, `bison`, `bash`) because of running out of memory. Yang et al. [71,72] report similar observations that shape analysis with reachability-based localization is insufficient for practical performance, and another technique (in their case, a new join operator) is required for more scalability.

$\mathsf{Airac}_{\mathsf{Reach}}$ vs. $\mathsf{Airac}_{\mathsf{ProcAcc}}$ Overall, $\mathsf{Airac}_{\mathsf{ProcAcc}}$ saved 78.5%–98.5%, on average 92.1%, of the analysis time of $\mathsf{Airac}_{\mathsf{Reach}}$. The analysis time of $\mathsf{Airac}_{\mathsf{ProcAcc}}$ includes pre-analysis time. The pre-analysis takes just a small portion of the total analysis time.

The big time savings are thanks to the synergy between reduction in the number of iterations (*#iters*) and improved speed of memory operations. First, $\mathsf{Airac}_{\mathsf{ProcAcc}}$ reduces *#iters* by on average 74.3%: more general (weaker) procedural summaries computed by $\mathsf{Airac}_{\mathsf{ProcAcc}}$ let the analyzer avoid re-computation of procedure bodies at different call-sites than $\mathsf{Airac}_{\mathsf{Reach}}$. Second, $\mathsf{Airac}_{\mathsf{ProcAcc}}$ improves speed (*#iters*/*time*) by on average 4.0 times: each procedure is analyzed with smaller memory states.

During the experiments, we observed that the improved performance is quite sensitive to the localization ratio (the size of accessed memory entries/the size of the reachable memory). For example, in Table 3.3, we see that, analyzing `screen`, $\mathsf{Airac}_{\mathsf{ProcAcc}}$ takes much more time than other programs. This is because `screen` has a bigger localization ratio (on average 38.5%) than other programs (e.g., the ratio for `bash` was 4.6%). So, we believe that localizing the input memory as possible as we can is important for saving analysis time. This is why we designed a pre-analysis that considers dynamically allocated locations and structure fields as well other than global variables (Section 3.3).

Moreover, our technique noticeably saves peak memory consumption by on average 71.2%. The reduction enabled $\mathsf{Airac}_{\mathsf{ProcAcc}}$ to analyze the largest four programs (`wget`, `screen`, `bison`, `bash`) that cannot be analyzed by $\mathsf{Airac}_{\mathsf{Reach}}$.

3.5.3 Performance of Extensions

Block-level Localization $\mathsf{Airac}_{\mathsf{GenAcc}}$ additionally saved 8.5%–53.7%, on average 31.8%, of the analysis time of $\mathsf{Airac}_{\mathsf{ProcAcc}}$. For example, for `gzip`, $\mathsf{Airac}_{\mathsf{ProcAcc}}$ took 2011.3 s but $\mathsf{Airac}_{\mathsf{GenAcc}}$ took 1142.7 reducing the analysis time by 43.2%. Memory costs between them is nearly the same: $\mathsf{Airac}_{\mathsf{GenAcc}}$ reduced peak memory consumption by on average 1.9%. Memory costs for $\mathsf{Airac}_{\mathsf{GenAcc}}$ sometimes increase (e.g., `parser`), because we cache accessed-locations sets for each localization block. The pre-analysis time for $\mathsf{Airac}_{\mathsf{GenAcc}}$ increases because of additional computation selecting localization blocks.

Access-based Localization with Bypassing Figure 3.7 compares the *time* of $\mathsf{Airac}_{\mathsf{ProcAcc}}$ and $\mathsf{Airac}_{\mathsf{Bypass}}$.[2] Overall, $\mathsf{Airac}_{\mathsf{Bypass}}$ saved 8.9%–78.5%, on average 42.1%, of the analysis time of $\mathsf{Airac}_{\mathsf{ProcAcc}}$. There are some noteworthy points.

- Some programs contain large recursive call cycles. One common belief for C programs is that it does not largely use recursion in practice. However, our finding from the benchmark programs is that some programs extensively use recursion and large recursive cycles unexpectedly exist in a number of real C programs. For example, from Table 3.7, note that program less, make, and screen have recursive cycles (scc) that contain more than 40 procedures.

- $\mathsf{Airac}_{\mathsf{ProcAcc}}$ is extremely inefficient for those programs. For other programs that have small (or no) recursive cycles, the analysis with access-based localization is quite efficient. For example, analyzing bash (the largest one in our benchmark) takes 1,637s. However, analyzing those programs that have large

[2] The performance numbers of $\mathsf{Airac}_{\mathsf{ProcAcc}}$ in Figure 3.7 is little bit different fromt those in Table 3.3, because the analyzer had been more tuned since the time when experiments in Table 3.3 was conducted.

recursive cycles takes much more time: less and make take more than 10,000s and screen takes more than 310,000s to finish the analysis, even though they are not the largest programs.

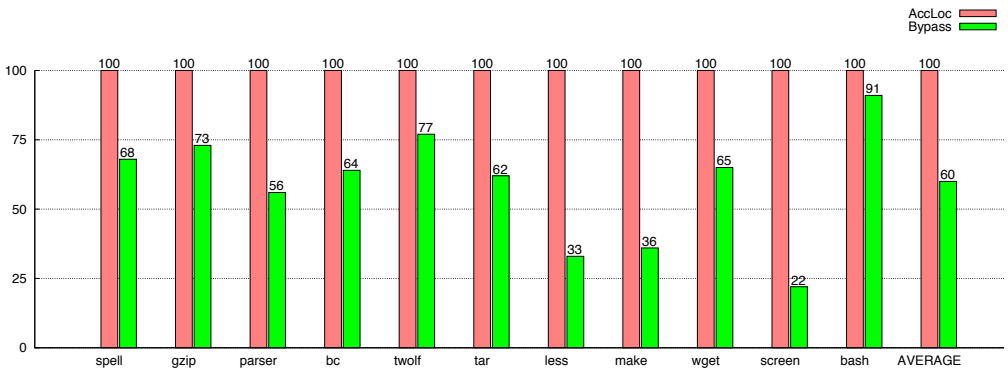
- $\mathsf{Airac}_{\mathsf{Bypass}}$ is especially effective for those programs. For programs less, make, and screen that contain large recursive cycles, our technique reduces the analysis time by 66.7%, 64.3%, and 78.5%, respectively.

- $\mathsf{Airac}_{\mathsf{Bypass}}$ is also noticeably effective for other programs. For programs, which have small cycles (consisting of less than 20 procedures), $\mathsf{Airac}_{\mathsf{Bypass}}$ saved 8.9%–44.1% of the analysis time of $\mathsf{Airac}_{\mathsf{ProcAcc}}$. For example, in analyzing parser, $\mathsf{Airac}_{\mathsf{ProcAcc}}$ took 572 seconds but $\mathsf{Airac}_{\mathsf{Bypass}}$ took 319 seconds.

The bypassing technique is also likely to reduce peak memory cost. Because the bypassing technique localizes memory states more aggressively than the original access-based localization, the peak memory consumption must be reduced. However, in the experiments, memory cost for analyzing smaller programs (gzip, parser, bc, twolf, tar) slightly increased. This is because $\mathsf{Airac}_{\mathsf{Bypass}}$ additionally keeps bypassing information on memory. But, for larger programs (less, make, wget, screen, bash), the results show that our technique reduces memory costs. For example, $\mathsf{Airac}_{\mathsf{ProcAcc}}$ required 2,228 MB in analyzing screen but $\mathsf{Airac}_{\mathsf{Bypass}}$ required 1,875 MB.

3.5.4 Discussion on Analysis Precision

In principle, more aggressive localization improves precision of our analysis because unnecessary memory entries are not passed to procedures, avoiding needless widening to them (more discussed in Section 3.5.4). In order to simply compare the precision, we first joined all the memories, ignoring flow-sensitivity, associated with each program point and then counted the number of constant intervals ($\#const$), finite intervals ($\#finite$), intervals with one infinity ($\#open$), and intervals with two infinities ($\#top$) from contained in the joined memory. The constant interval and top interval indicate the most precise and imprecise values, respectively. Table 3.4 shows the results for some programs. (This result is just to show that our lo-



Program	**LOC**	**Proc**	**LRC**	$\mathsf{Airac}_{\mathsf{ProcAcc}}$		$\mathsf{Airac}_{\mathsf{Bypass}}$		*Save*
				time(sec)	*MB*	*time*(sec)	*MB*	(*time*)
spell-1.0	2,213	31	0	2.4	10	1.6	10	31.6%
gzip-1.2.4a	7,327	135	2	51.9	65	37.7	64	27.4%
parser	10,900	325	3	571.6	206	319.4	245	44.1%
bc-1.06	13,093	134	1	496.9	131	318.4	165	35.9%
twolf	19,700	192	1	509.5	212	389.9	212	23.5%
tar-1.13	20,258	222	13	2,407.9	294	1,503.2	338	37.6%
less-382	23,822	382	46	14,720.8	490	4,906.4	427	66.7%
make-3.76.1	27,304	191	61	14,681.9	695	5,248.0	549	64.3%
wget-1.9	35,018	434	13	6,717.5	544	4,383.4	552	34.7%
screen-4.0.2	44,734	589	77	310,788.0	2,228	66,920.6	1,875	78.5%
bash-2.05a	105,174	959	4	1,637.6	272	1,492.4	265	8.9%

Figure 3.7: Performance of bypassing techniques. Lines of code (**LOC**) are given before preprocessing. The number of procedures (**Proc**) is given after preprocessing. **LRC** represents the size of largest recursive call cycle contained in each program. *time* shows analysis time in seconds. *MB* shows peak memory consumption in megabytes. $\mathsf{Airac}_{\mathsf{ProcAcc}}$ uses access-based localization for procedure calls and $\mathsf{Airac}_{\mathsf{Bypass}}$ uses our technique. *time* for $\mathsf{Airac}_{\mathsf{ProcAcc}}$ and $\mathsf{Airac}_{\mathsf{Bypass}}$ are the total time that includes pre-analysis time. *Save* shows time savings in percentage of $\mathsf{Airac}_{\mathsf{Bypass}}$ against $\mathsf{Airac}_{\mathsf{ProcAcc}}$.

Program	Analysis	*#const*	*#finite*	*#open*	*#top*
spell-1.0	$\mathsf{Airac}_{\mathsf{Reach}}$	665	101	33	142
	$\mathsf{Airac}_{\mathsf{ProcAcc}}$	665	102	32	142
barcode-0.96	$\mathsf{Airac}_{\mathsf{Reach}}$	2343	595	221	515
	$\mathsf{Airac}_{\mathsf{ProcAcc}}$	2347	597	218	512

Table 3.4: Precision comparison between reachability- and access-based localization. For each analysis, the table reports the number of constant intervals (*#const*), finite intervals (*#finite*), intervals with one infinity (*#open*), and intervals with two infinities (*#top*).

calization technique does not sacrifice (in fact, it increases) analysis precision, *not* to show how accurate our analyzer is. For simplicity of the comparison, we turned off many precision-improving techniques: flow-sensitivity, context pruning, and narrowing.)

Why does the precision improve? When the analysis uses widening, localization improves the analysis precision, which is best explained by an example. Consider analyzing the following code with interval domain.

```
1  int g = 0;
2  void f () {    // assume g is not used inside f
3    int i = 0;
4    while (i++) { if (...) break; }  // loop bound is unknown
5  }
6  void main() {
7    g = 0; f();  // first call to f
8    g = 1; f();  // second call to f
9  }
```

In order to analyze the above program, we need widening. Widening is a speed-up technique that accelerates fixpoint iterations and guarantees analysis' termination. In the above example, we apply widening at the head of the loop at line 4, and consequently all the changed values inside the `while` loop is widened: value of `i` keeps changing at the loop head $[0,0] \to [0,1] \to \cdots$ and extrapolated to $[0,+\infty]$ by widening. With $[0,+\infty]$ the loop reaches a (post-)fixpoint.

With reachability-based localization, the widening is also applied to the value of `g`. This is because the localized input states contain the value for `g`, because `g` is a global variable and always reachable. At the first call (line 7), `g` has value $[0, 0]$ and it is not modified inside `f`. Hence, `g` is not widened at this time. But, at the second call, `g`'s value is changed to $[1, 1]$, which in turn flows to the body of `f`. Note that we apply widening operation to changed values at the loop head at line 4. Because the value for `g` is changed $[0, 0] \rightarrow [0, 1]$, the value becomes widened to $[0, +\infty]$. Hence, after the analysis finishes, `g` has value $[0, \infty]$ at line 9.

With access-based localization, the widening does not affect to the value for `g`. Because global variable `g` is not used inside procedure `f`, the input memory states for procedure `f` does not contain the value for `g`. Hence, the widening operation at line 4 is not applied to `g`. After the analysis finishes, `g` has $[1, 1]$ at line 9.

Chapter 4

Temporal Localization

This chapter presents a formal framework for temporal localization. Previously, temporal localization has been known as "sparse analysis" [24, 25] or "sparse evaluation" [10, 54] in dataflow analysis. We generalize the sparse analysis techniques on top of the abstract interpretation framework [12, 16] to support arbitrarily complicated semantics properties for C-like languages. We first use the abstract interpretation framework to have a global static analyzer that is conventional but not yet sparse. Upon this underlying sound static analyzer, we add our generalized sparse analysis techniques to improve its scalability while preserving the precision of the underlying analysis. Our framework determines what to prove to guarantee that the resulting sparse version should preserve the precision of the underlying analyzer.

We formally present our framework; we present that existing sparse analyses are all restricted instances of our framework; we show more semantically elaborate design examples of sparse static analyses; we present their implementation results that scales to analyze up to one million lines of C programs. We also show a set of implementation techniques that turn out to be critical to economically support the sparse analysis process.

4.1 Introduction

Sparse analysis is a promising technique for optimizing global static analysis [8, 10, 18, 19, 30, 53–55, 66, 68]. The key observation behind the sparse analysis is that semantic functions are typically sparse; analyzing a statement typically updates only a tiny part of memory states at each point of analysis and simply propagates them to the other parts. To exploit the sparsity, sparse analysis techniques use def-use chains to guide the propagation of abstract values. The technique makes the analysis only propagate the values that are meaningfully generated to the use points, thereby significantly improves analysis' scalability in time and memory.

It is not obvious, however, how to apply existing sparse analysis techniques for general static analysis. Though def-use dependence is a semantic property of analysis, previous sparse analysis design was not semantics-based, but rather algorithmic. Thus, application of sparse analysis has been mostly limited to simple settings where def-use dependence is syntactically identifiable [24]. Recently, a sparse analysis with semantically identifiable def-use dependence was proposed [25] but the analysis is still algorithmic and tightly coupled with a specific analysis, which is far from obvious how to generalize the algorithm to general, arbitrarily complicated static analysis and how to formally prove the correctness of the sparse analysis in connection with the original analysis.

In this chapter, we present a general sparse analysis framework. Our approach generalizes the sparse analysis ideas on top of the abstract interpretation framework. Since the abstract interpretation framework [16,17] guides us to design sound yet arbitrarily precise static analyzers for any target language, we first use the framework to have a sound and precise global static analyzer. Upon this underlying non-sparse static analyzer, we add our generalized sparse analysis techniques to improve its scalability while preserving the precision of the underlying analysis. Our framework determines what to prove to guarantee that the resulting sparse version should preserve the precision of the underlying analyzer.

Our method bridges the gap between the two existing technologies – abstract interpretation and sparse analysis – towards the design of sound, yet scalable global

static analyzers. Note that while abstract interpretation framework provides a theoretical knob to control the analysis precision without violating its correctness, the framework does not provide a knob to control the resulting analyzer's scalability preserving its precision. On the other hand, existing sparse analysis techniques [8, 10, 18, 19, 24, 25, 30, 53–55, 66, 68] achieve scalability, but they are mostly algorithmic and tightly coupled with particular analyses. The sparse techniques are not general enough to be used for an arbitrarily complicated semantic analysis.

In this article we formally present our framework; we present that existing sparse analyses are all restricted instances of our framework; we show more semantically elaborate design examples of sparse static analyses; we present their implementation results that scales to analyze up to one million lines of C programs.

Contributions Our contributions are as follows.

- We propose a general framework for sparse static analysis. Our framework is semantic-based and precision-preserving. We prove that our framework yields a correct sparse analysis that has the same precision as the original.
- Within the framework, we design a sparse analysis which is still general as itself. We can instantiate the design with arbitrary non-relational abstract domains.
- We prove the practicality of our framework by experimentally demonstrating the achieved speedup of sparse analysis versions. We show the derived sparse analysis can analyze programs up to 1 million lines of C code with interval domain.

Outline The rest of this paper is organized as follows. Section 4.2 explains our sparse analysis framework. Section 4.3 and 4.4 present design examples within the framework. Section 4.5 discusses several issues involved in the implementations. Section 4.6 presents the experimental studies.

4.2 Sparse Analysis Framework

4.2.1 Baseline Abstraction

Our framework considers a particular family of abstractions.

Program As in Chapter 2, a program is a tuple $\langle \mathbb{C}, \hookrightarrow \rangle$ where $\mathbb{C}$ is a finite set of control points and $\hookrightarrow \subseteq \mathbb{C} \times \mathbb{C}$ is a relation that denotes control dependencies of the program; $c' \hookrightarrow c$ indicates that c is a next control point of c'.

Collecting Semantics Collecting semantics of program P is an invariant $[\![P]\!] \in \mathbb{C} \to 2^{\mathbb{S}}$ that represents a set of reachable states at each control point, where the concrete domain of states, $\mathbb{S}$, is defined as follows:

$$\mathbb{S} = \mathbb{L} \to \mathbb{V}$$

Concrete state $s \in \mathbb{S}$ is a map from locations to values, and a value is either integer ($\mathbb{Z}$) or location ($\mathbb{L}$). The collecting semantics is characterized by the least fixpoint of semantic function $F \in (\mathbb{C} \to 2^{\mathbb{S}}) \to (\mathbb{C} \to 2^{\mathbb{S}})$ such that,

$$F(X) = \lambda c \in \mathbb{C}. f_c(\bigcup_{c_p \hookrightarrow c} X(c_p)). \tag{4.1}$$

where $f_c \in 2^{\mathbb{S}} \to 2^{\mathbb{S}}$ is a semantic function at control point c.

Abstract Semantics We abstract the collecting semantics of program P by the following Galois connection

$$\mathbb{C} \to 2^{\mathbb{S}} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \mathbb{C} \to \hat{\mathbb{S}} \tag{4.2}$$

where α and γ are a pointwise lifting of abstract and concretization function $\alpha_{\mathbb{S}}$ and $\gamma_{\mathbb{S}}$ (such that $2^{\mathbb{S}} \underset{\alpha_{\mathbb{S}}}{\overset{\gamma_{\mathbb{S}}}{\leftrightarrows}} \hat{\mathbb{S}}$), respectively.

We consider a particular, but general and practical, family of abstract domains where abstract state $\hat{\mathbb{S}}$ is map $\hat{\mathbb{L}} \to \hat{\mathbb{V}}$ where $\hat{\mathbb{L}}$ is a finite set of abstract loca-

tions, and $\hat{\mathbb{V}}$ is a (potentially infinite) set of abstract values. All non-relational abstract domains are members of this family. Furthermore, the family covers some numerical, relational domains. Practical relational analyses exploit *packed* relationality [5, 14, 46, 67], where the abstract domain is of form $Packs \to \hat{\mathbb{R}}$ in which *Packs* is a set of variable groups that are selected to be related together. $\hat{\mathbb{R}}$ denotes numerical constraints among variables in those groups. In such relational analysis, each variable pack is treated as an abstract location ($\hat{\mathbb{L}}$) and numerical constraints amount to abstract values ($\hat{\mathbb{V}}$). Examples of the numerical constraints are domain of octagons [46] and polyhedrons [15].

Abstract semantics is characterized as the least fixpoint of abstract semantic function $\hat{F} \in (\mathbb{C} \to \hat{\mathbb{S}}) \to (\mathbb{C} \to \hat{\mathbb{S}})$ defined as,

$$\hat{F}(\hat{X}) = \lambda c \in \mathbb{C}.\hat{f}_c(\bigsqcup_{c' \hookrightarrow c} \hat{X}(c')). \tag{4.3}$$

where $\hat{f}_c \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ is an abstract semantic function at control point c. Note that, given $\hat{X} \in \mathbb{C} \to \hat{\mathbb{S}}$, the input state of control point c is denoted by $\bigsqcup_{c' \hookrightarrow c} \hat{X}(c')$. For example, the input state at fixpoint is denoted by $\bigsqcup_{c' \hookrightarrow c} \mathcal{S}(c')$, where $\mathcal{S} = \mathbf{lfp}\hat{F}$. The set of input states that potentially occur during the course of the analysis at control point c are defined by $\{\hat{s} \mid \hat{s} \sqsubseteq \bigsqcup_{c' \hookrightarrow c} \mathcal{S}(c')\}$. We use these two notions in Section 4.4.2. The soundness of abstract semantics is followed by fixpoint transfer theorem [17].

Lemma 7 (Soundness) *If* $\alpha \circ F \sqsubseteq \hat{F} \circ \alpha$, *then,* $\alpha(\mathbf{lfp}F) \sqsubseteq \mathbf{lfp}\hat{F}$.

4.2.2 Sparse Analysis by Eliminating Unnecessary Propagation

The abstract semantic function given in (4.3) propagates some abstract values unnecessarily. For example, suppose that we analyze statement $x := y$ using a non-relational domain, like interval domain [16]. We know for sure that the abstract semantic function for the statement *defines* a new abstract value only at variable x and *uses* only the abstract value of variable y. Thus, it is unnecessary to propagate

the whole abstract states. However, the function given in (4.3) blindly propagates the whole abstract states of all predecessors c' to control point c.

To make the analysis sparse, we need to eliminate this unnecessary propagation by making the semantic function propagate abstract values along data dependency, not control dependency. Formally, our goal is to transform the original semantic function $\hat{F}$ (4.3) to the following sparse semantic function $\hat{F}_s$:

$$\hat{F}_s(\hat{X}) = \lambda c \in \mathbb{C}.\hat{f}_c(\bigsqcup_{c_d \overset{l}{\rightsquigarrow} c} \hat{X}(c_d)|_l). \tag{4.4}$$

The meaning of data dependency $c_d \overset{l}{\rightsquigarrow} c$ is that we propagate the value of l from c_d directly to c, avoiding unnecessary propagations along control points c_i such that $c_d \hookrightarrow^+ c_i \hookrightarrow^+ c$. As this definition is only different in that it is defined over data dependency ($\rightsquigarrow$), we can reuse abstract semantic function $\hat{f}_c$, and its soundness result, from the original analysis design. In the rest of this section, we explain how to define such data dependency ($\rightsquigarrow$) that make the analysis sparse while preserving soundness and precision.

4.2.3 Definition and Use Set

We first need to precisely define what "definitions" and "uses" are. In order for sparse analysis to be correct and general, they must be defined in terms of abstract semantics, i.e., abstract semantic function $\hat{f}_c$. Otherwise, designing sparse analysis is a error-prone task, as illustrated in the following example.

Example 3 *Consider statement $*p = 1$, and suppose p points to location l. What are definitions and uses for the statement? Glancing at the statements, one might answer that location l is defined and p is used. However, the correct answer cannot be made without consulting abstract semantics of interest. For example, if we're considering an analysis that performs a weak update for above statement, then l must be included as uses as well. More complicated, if we consider relational analysis*

that updates values of other related locations, definitions must include those related locations also.

Given abstract state $\hat{s}$, we say an abstract location l is defined at control point c if the location is assigned a new abstract value by abstract semantics function $\hat{f}_c$, i.e., $\hat{f}_c(\hat{s})(l) \neq \hat{s}(l)$. Given abstract state $\hat{s}$ and abstract location d that is defined at control point c, we say abstract location l is used for definition d if l is essential in generating the value for d, i.e., $\hat{f}_c(\hat{s})(d) \neq \hat{f}_c(\hat{s}\backslash_l)(d)$. In below, we define the set of abstract locations that are defined and used during the course of the analysis at control point c. Note that the notions of definitions and uses are semantic one. For example, consider statement `x:=x`. Syntactically, variable `x` is defined and used at the statement, but, semantically, the statement has no effect and no variable is defined nor used according to our definition.

Definition 4.2.1 (Definition set) *Let $\mathcal{S}$ be the least fixpoint $\mathbf{lfp}\hat{F}$ of the original semantic function $\hat{F}$. Definition set $\mathsf{D}(c)$ is the set of abstract locations that are defined, at control point c, during the course of the fixpoint computation, i.e.,*

$$\mathsf{D}(c) \triangleq \{l \in \hat{\mathbb{L}} \mid \exists \hat{s} \sqsubseteq \bigsqcup_{c' \hookrightarrow c} \mathcal{S}(c').\hat{f}_c(\hat{s})(l) \neq \hat{s}(l)\}.$$

Note that $\bigsqcup_{c' \hookrightarrow c} \mathcal{S}(c')$ denotes the final input state (i.e., the input state at fixpoint) at control point c and quantified variable $\hat{s}$ ranges over the set of all input states that potentially occur during the analysis.

In fact, $\mathsf{D}(c)$ denotes abstract locations that *may* be defined at control point c. We also define the notion of *must*-definitions that denote the set of abstract locations whose values are always changed during the analysis.

Definition 4.2.2 (Must-Definition set) *Let $\mathcal{S}$ be the least fixpoint $\mathbf{lfp}\hat{F}$ of the original semantic function $\hat{F}$. Must-definition set $\mathsf{D}_{\mathsf{must}}(c)$ is the set of abstract lo-*

cations that are must-defined, at control point c, during the course of the fixpoint computation, i.e.,

$$\mathsf{D}_{\mathsf{must}}(c) \triangleq \{l \in \hat{\mathbb{L}} \mid \forall \hat{s} \sqsubseteq \bigsqcup_{c' \hookrightarrow c} \mathcal{S}(c').\hat{f}_c(\hat{s})(l) \neq \hat{s}(l)\}.$$

Definition 4.2.3 (Use set) *Let $\mathcal{S}$ be the least fixpoint $\mathbf{lfp}\hat{F}$ of the original semantic function $\hat{F}$. Use set $\mathsf{U}(c)$ is the set of abstract locations that are used, at control point c, during the course of the fixpoint computation, i.e.,*

$$\mathsf{U}(c) \triangleq \{l \in \hat{\mathbb{L}} \mid \exists \hat{s} \sqsubseteq \bigsqcup_{c' \hookrightarrow c} \mathcal{S}(c').\hat{f}_c(\hat{s})|_{\mathsf{D}(c)} \neq \hat{f}_c(\hat{s}\backslash_l)|_{\mathsf{D}(c)}\}.$$

The definition means that if the value of abstract location l will be used by abstract semantic function $\hat{f}_c$, the function yields a different result once we remove l from the domain of the input. Note that we only compare the values of abstract locations that are supposed to be changed by the semantic function; i.e. definition set.

Example 4 *Consider the following simple subset of C:*

$$x := e \mid *x := e \ \textit{where}\ e \to x \mid \&x \mid *x.$$

The meaning of each statement and each expression is fairly standard. We design a pointer analysis for this as follows:

$$\begin{array}{rcl}
\hat{s} \in \hat{\mathbb{S}} & = & \mathit{Var} \to 2^{\mathit{Var}} \\
\hat{f}_c(\hat{s}) & = & \begin{cases}
\hat{s}[x \mapsto \hat{\mathcal{E}}(e)(\hat{s})] & \mathsf{cmd}(c) = x := e \\
\hat{s}[y \mapsto \hat{\mathcal{E}}(e)(\hat{s})] & \mathsf{cmd}(c) = *x := e \\
 & \textit{and } \hat{s}(x) = \{y\} \\
\hat{s}[\hat{s}(x) \overset{w}{\mapsto} \hat{\mathcal{E}}(e)(\hat{s})] & \mathsf{cmd}(c) = *x := e
\end{cases} \\
\hat{\mathcal{E}}(e)(\hat{s}) & = & \begin{cases}
\hat{s}(x) & e = x \\
\{x\} & e = \&x \\
\bigcup_{y \in \hat{s}(x)} \hat{s}(y) & e = *x
\end{cases}
\end{array}$$

Now suppose that we analyze program ${}^{10}x := \&y$; ${}^{11}{*p} := \&z$; ${}^{12}y := x$; *(superscripts are control points). Suppose that points-to set of pointer* p *is* $\{x, y\}$ *at control point* 10 *according to the fixpoint. Definition set and use set at each control point are as follows.*

$$\begin{array}{lllllllll} \mathsf{D}(10) & = & \{x\} & \mathsf{D}_{\mathsf{must}}(10) & = & \{x\} & \mathsf{U}(10) & = & \varnothing \\ \mathsf{D}(11) & = & \{x, y\} & \mathsf{D}_{\mathsf{must}}(11) & = & \emptyset & \mathsf{U}(11) & = & \{x, y\} \\ \mathsf{D}(12) & = & \{y\} & \mathsf{D}_{\mathsf{must}}(12) & = & \{x\} & \mathsf{U}(12) & = & \{x\} \end{array}$$

Note that $\mathsf{U}(11)$ *contains* $\mathsf{D}(11)$ *because of the weak update* ($\overset{w}{\mapsto}$).

4.2.4 Data Dependencies

Once identifying definition set and use set at each control point, we can discover data dependencies of abstract semantic function $\hat{F}$ between control points. However, the notion of data dependencies must be defined carefully so that the sparse semantic function $\hat{F}_s$ (4.4) is correct with respect to the original semantic function $\hat{F}$ (4.3). In this section, we define our notion of data dependencies.

Intuitively, if the abstract value of abstract location l defined at control point c_d is used at control point c_u, there is a data dependency between c_d and c_u on l. The formal definition is given below:

Definition 4.2.4 (Data dependency) *Let* c_d *and* c_u *be control points and* l *be an abstract location. Data dependency is ternary relation* $\rightsquigarrow$ *defined as follows:*

$$\begin{array}{rcl} c_d \overset{l}{\rightsquigarrow} c_u & \triangleq & l \in \mathsf{D}(c_d) \\ & \wedge & l \in \mathsf{U}(c_u) \\ & \wedge & \exists c_1, \ldots, c_n \in \mathbb{C}. c_d \hookrightarrow c_1 \hookrightarrow \cdots \hookrightarrow c_n \hookrightarrow c_u \Longrightarrow \forall i \in [1, n]. l \notin \mathsf{D}(c_i) \end{array}$$

The definition means that the value of abstract location l can be defined at c_d and used at c_u and there exists a path from c_d to c_u whose intermediate control point c_i does not change the value of l, then we can directly propagate the value of l from c_d to c_u.

Example 5 *In the program presented in Example 4, we can find two data dependencies,* $10 \overset{x}{\rightsquigarrow} 11$ *and* $11 \overset{x}{\rightsquigarrow} 12$.

The following lemma states that the analysis result with sparse abstract semantic function is the same as the one of original analysis.

Lemma 8 (Correctness) *Let* $\mathcal{S}$ *and* $\mathcal{S}_s$ *be* $\mathbf{lfp}\hat{F}$ *and* $\mathbf{lfp}\hat{F}_s$. *Then,*

$$\forall c \in \mathbb{C}. \forall l \in \mathsf{D}(c). \mathcal{S}_s(c)(l) = \mathcal{S}(c)(l).$$

Proof (Sketch) We prove the lemma by showing that the fixpoint equation of $\hat{F}_s$ is equivalent to the one of $\hat{F}$ up to the definitions $\mathsf{D}(c)$ for each $c \in \mathbb{C}$. For simplicity, we only consider the case with the following assumptions: x is an abstract location and $c_1, \cdots, c_n$ are control points such that $c_1 \hookrightarrow \cdots \hookrightarrow c_n$, $x \in \mathsf{D}(c_1)$, $x \in \mathsf{U}(c_n)$, $x \in \mathsf{D}(c_n)$, $x \notin \mathsf{D}(c_i)$ for all $2 \leq i < n$, and c_i is the only predecessor of c_{i+1} for all $1 \leq i < n$ (we can easily extend this proof to the general case). What we have to show are (1) $\hat{F}_s$ is derived from $\hat{F}$; and (2) $\hat{F}$ is derived from $\hat{F}_s$. We show the first case. By the assumptions and definition of $\rightsquigarrow$, $c_1 \overset{x}{\rightsquigarrow} c_n$. The fixpoint equations of $\hat{F}$ are as follows:

$$\begin{array}{rcl} \mathcal{S}(c_2) & = & \hat{f}_{c_2}(\mathcal{S}(c_1)) \\ & \vdots & \\ \mathcal{S}(c_n) & = & \hat{f}_{c_n}(\mathcal{S}(c_{n-1})). \end{array} \tag{4.5}$$

We can transform these into the fixpoint equation of $\hat{F}_s$ as follows:

$$\begin{array}{rll} \mathcal{S}(c_n)(x) = \hat{f}_{c_n}(\mathcal{S}(c_{n-1}))(x) & & (\because (4.5)) \\ = \hat{f}_{c_n}(\mathcal{S}(c_{n-1})|_x)(x) & (\because \text{ Def. of } \mathsf{U} \text{ and } \mathsf{U}(c_n) = \{x\}) & \\ = \hat{f}_{c_n}(\mathcal{S}(c_1)|_x)(x) & (\because \text{ Def. of } \rightsquigarrow \text{ and } c_1 \overset{x}{\rightsquigarrow} c_n) & \end{array}$$

Note that $c_1 \overset{x}{\rightsquigarrow} c_n \Rightarrow S(c_i)(x) = \hat{f}_{c_i}(S(c_{i-1}))(x) = S(c_{i-1})(x)$ where $1 < i < n$. The fixpoint equation of $\hat{F}_s$ is $\mathcal{S}_s(c_n)(x) = \hat{f}_{c_n}(\mathcal{S}_s(c_1)|_x)(x)$ and this is equivalent to the one derived above.

$$\therefore \mathcal{S}(c_n)(x) = \mathcal{S}_s(c_n)(x)$$

□

The lemma guarantees that the sparse analysis result is identical to the original result only up to the entries that exist in the sparse analysis result. This is fair since the sparse analysis result does not contain the entries unnecessary for its computation.

Comparison with Def-use Chains Note that our notion of data dependency is different from the conventional notion of def-use chains. The def-use chain relation $\rightsquigarrow_{\mathsf{du}}$ is defined as follows:

$$\begin{array}{rcl} c_d \overset{l}{\rightsquigarrow}_{\mathsf{du}} c_u & \triangleq & l \in \mathsf{D}(c_d) \\ & \wedge & l \in \mathsf{U}(c_u) \\ & \wedge & \exists c_1, \ldots, c_n \in \mathbb{C}. c_d \hookrightarrow c_1 \hookrightarrow \ldots \hookrightarrow c_n \hookrightarrow c_u \Longrightarrow \forall i \in [1, n]. l \notin \mathsf{kill}(c_i) \end{array}$$

where $\mathsf{kill}(c_i) = \mathsf{D}_{\mathsf{must}}(c_i)$. The definition means that if the value of abstract location l can be defined at c_d and used at c_u, and there exists a path from c_d to c_u whose intermediate control point c_i does not kill the value of l, then the definition of l at c_d is potentially propagated to its use points c_u.

Example 6 *We can find three def-use chains,* $10 \overset{x}{\rightsquigarrow}_{\mathsf{du}} 11$, $10 \overset{x}{\rightsquigarrow}_{\mathsf{du}} 12$, *and* $11 \overset{x}{\rightsquigarrow}_{\mathsf{du}} 12$ *in Example 4.*

The reason why we use our notion of data dependencies instead of def-use chains becomes evident in Section 4.2.5, where we discuss the approximations of them.

4.2.5 Sparse Analysis with Approximated Data Dependency

Sparse analysis designed until now might not be practical since we can decide definition set D and use set U only with the original fixpoint $\mathbf{lfp}\hat{F}$ computed.

To design a practical sparse analysis, we can approximate data dependency using an approximated definition set $\hat{\mathsf{D}}$ and use set $\hat{\mathsf{U}}$.

Definition 4.2.5 (Approximated Data Dependency) *Let c_d and c_u be control points and l be an abstract location. Approximated data dependency is ternary relation $\rightsquigarrow_a$ defined as follows:*

$$\begin{array}{rcl} c_d \overset{l}{\rightsquigarrow}_a c_u & \triangleq & l \in \hat{\mathsf{D}}(c_d) \\ & \wedge & l \in \hat{\mathsf{U}}(c_u) \\ & \wedge & \exists c_1, \ldots, c_n \in \mathbb{C}. c_d \hookrightarrow c_1 \hookrightarrow \cdots \hookrightarrow c_n \hookrightarrow c_u \implies \forall i \in [1,n]. l \notin \hat{\mathsf{D}}(c_i) \end{array}$$

The definition is the same with the previous definition (4.2.4) except that it is defined using $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$. The derived sparse analysis is to compute the fixpoint of the following abstract semantic function:

$$\hat{F}_a(\hat{X}) = \lambda c \in \mathbb{C}. \hat{f}_c(\bigsqcup_{c_d \overset{l}{\rightsquigarrow}_a c} \hat{X}(c_d)|_l).$$

One thing to note is that not all $\hat{\mathsf{D}}$, $\hat{\mathsf{D}}_{\mathsf{must}}$, and $\hat{\mathsf{U}}$ make the derived sparse analysis compute the same result as the original. First, both $\hat{\mathsf{D}}(c)$ and $\hat{\mathsf{U}}(c)$ at each control point should be an over-approximation of $\mathsf{D}(c)$ and $\mathsf{U}(c)$, respectively (we can easily show that the analysis computes different result if one of them is an under-approximation). Next, all spurious definitions that are included in $\hat{\mathsf{D}}$ but not in D should be also included in $\hat{\mathsf{U}}$. The following example illustrates what happens when there exists an abstract location which is a spurious definition but is not included in the approximated use set.

Example 7 *Consider the same program presented in Example 4. except that we now suppose the points-to set of pointer p being* $\{y\}$*. Then, definition set and use set at each control point are as follows:*

$$\begin{array}{lllllll} \mathsf{D}(10) & = & \{x\} & \mathsf{U}(10) & = & \varnothing \\ \mathsf{D}(11) & = & \{y\} & \mathsf{U}(11) & = & \varnothing \\ \mathsf{D}(12) & = & \{y\} & \mathsf{U}(12) & = & \{x\}. \end{array}$$

Note that $\mathsf{U}(11)$ *does not contain* $\mathsf{D}(11)$ *because of strong update. The following is one example of unsafe approximation.*

$$\begin{array}{lllllll} \hat{\mathsf{D}}(10) & = & \{x\} & \hat{\mathsf{U}}(10) & = & \varnothing \\ \hat{\mathsf{D}}(11) & = & \{x, y\} & \hat{\mathsf{U}}(11) & = & \varnothing \\ \hat{\mathsf{D}}(12) & = & \{y\} & \hat{\mathsf{U}}(12) & = & \{x\}. \end{array}$$

This approximation is unsafe because spurious definition $\{x\}$ *at control point* 11 *is not included in approximated use set* $\hat{\mathsf{U}}(11)$*. With this approximation, abstract value of* x *at 10 is not propagated to 12, while it is propagated in the original analysis (*$10 \overset{x}{\rightsquigarrow} 12$*, but* $10 \not\overset{x}{\rightsquigarrow}_a 12$*). The following is one example of safe approximation.*

$$\begin{array}{lllllll} \hat{\mathsf{D}}(10) & = & \{x\} & \hat{\mathsf{U}}(10) & = & \varnothing \\ \hat{\mathsf{D}}(11) & = & \{x, y\} & \hat{\mathsf{U}}(11) & = & \varnothing \\ \hat{\mathsf{D}}(12) & = & \{y\} & \hat{\mathsf{U}}(12) & = & \{x\}. \end{array}$$

Because $\{x\} \subseteq \hat{\mathsf{U}}(11)$*, the abstract value will be propagated through two data dependency,* $10 \overset{x}{\rightsquigarrow}_a 11$ *and* $11 \overset{x}{\rightsquigarrow}_a 12$*. Note that* x *is not defined at* 11*, thus the propagated abstract value for* x *is not modified at* 11*.*

We can formally define safe approximation of definition set and use set as follows:

Definition 4.2.6 (Approximated definition set and use set) *Set* $\hat{\mathsf{D}}(c)$ *and* $\hat{\mathsf{U}}(c)$ *are a safe approximation of definition set* $\mathsf{D}(c)$ *and use set* $\mathsf{U}(c)$*, respectively, if and only if*

(1) $\hat{\mathsf{D}}(c) \supseteq \mathsf{D}(c) \wedge \hat{\mathsf{U}}(c) \supseteq \mathsf{U}(c)$*; and*

(2) $\hat{\mathsf{D}}(c) - \mathsf{D}(c) \subseteq \hat{\mathsf{U}}(c)$.

The remaining thing is to prove that the safe approximation $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ yields the correct sparse analysis, which the following lemma states:

Lemma 9 (Correctness of Safe Approximation) *Suppose sparse abstract semantic function $\hat{F}_a$ is derived by the safe approximation $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$. Let $\mathcal{S}$ and $\mathcal{S}_a$ be $\mathbf{lfp}\hat{F}$ and $\mathbf{lfp}\hat{F}_a$. Then,*

$$\forall c \in \mathbb{C}. \forall l \in \hat{\mathsf{D}}(c). \mathcal{S}_a(c)(l) = \mathcal{S}(c)(l).$$

Proof (Sketch) We can prove the lemma by showing the equivalence of the fixpoint equations up to the definitions $\mathsf{D}(c)$ for each $c \in \mathbb{C}$, as we did for Lemma 8. We make the same assumptions as in the proof of Lemma 8 (we can generalize the proof easily). Consider the case when $x \in \hat{\mathsf{D}}(c_m) - \mathsf{D}(c_m)$ for some m such that $1 < m < n$. By the definition of the safe approximation, $x \in \hat{\mathsf{U}}(x)$ and we now have two data dependencies $c_1 \overset{x}{\rightsquigarrow}_a c_m$ and $c_m \overset{x}{\rightsquigarrow}_a c_n$ instead of $c_1 \overset{x}{\rightsquigarrow} c_n$. The fixpoint equations of $\hat{F}_a$ are as follows:

$$\begin{aligned} \mathcal{S}_a(c_m)(x) &= \hat{f}_{c_m}(\mathcal{S}_a(c_1)|_x)(x) \\ \mathcal{S}_a(c_n)(x) &= \hat{f}_{c_n}(\mathcal{S}_a(c_m)|_x)(x). \end{aligned}$$

The fixpoint equations of $\hat{F}$ we derived in the proof of Lemma 8 are as follows:

$$\begin{aligned} S(c_i)(x) &= S(c_{i-1})(x) \text{ where } 1 < i < n \\ \mathcal{S}(c_n)(x) &= \hat{f}_{c_n}(\mathcal{S}(c_1)|_x)(x). \end{aligned}$$

Since x is spurious definition at c_m, we have $S(c_m)(x) = S(c_{m-1})(x) = \hat{f}_{c_m}(S(c_{m-1}))(x) = \hat{f}_{c_m}(S(c_1)|_x)(x)$, which proves that two sets of fixpoint equations are equivalent. Therefore, $\mathcal{S}_a(c_n)(x) = \mathcal{S}(c_n)(x)$. □

Precision Loss with Conservative Def-use Chains While approximated data dependency does not degrade the precision of an analysis, conservative def-use chains from approximated definition set and use set make the analysis less precise even if the approximation is safe. The following example illustrates the case of imprecision.

Example 8 *Consider the same setting of Example 7. Approximated definition set and use set establish the following three def-use chains:* $10 \overset{x}{\rightsquigarrow}_{\mathsf{du}} 11$, $11 \overset{x}{\rightsquigarrow}_{\mathsf{du}} 12$, *and* $10 \overset{x}{\rightsquigarrow}_{\mathsf{du}} 12$ *(we assume here that relation* $\rightsquigarrow_{\mathsf{du}}$ *is similarly modified as in Definition 4.2.6). With these conservative def-use chains, the points-to set of* x *propagated to control point 12 is* $\{y\} \cup \{z\}$*, which is bigger set than* $\{y\}$*, the one that appears in the original analysis.*

4.2.6 Designing Sparse Analysis within the Framework

In summary, the design of sparse analysis within our framework is done in the following two steps:

(1) Design a static analysis based on abstract interpretation framework [16]. Note that the abstract domain should be a member of the family explained in Section 4.2.1.

(2) Design a method to find a safe approximation $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ of definition set D and use set U (Definition 4.2.6).

Once the safe approximation is found in step (2), our framework guarantees that the derived sparse analysis is correct; that is, the sparse analysis is sound and has the same precision as the original analysis designed in step (1).

4.3 Example: Sparse Non-Relational Analysis

As a concrete example, we show how to design sparse non-relational analyses within our framework. Following Section 4.2.6, we proceed in two steps: (1) We design a conventional non-relational analysis based on abstract interpretation. Relying on

the abstract interpretation framework [12,16], we can flexibly design a static analysis of our interest with soundness guaranteed. However, the analysis is not yet sparse. (2) We design a method to find $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ and prove that they are safe approximations (Definition 4.2.6). We assume $\mathsf{D}_{\mathsf{must}}(c) = \emptyset$ because it is enough in practice.

For brevity, we restrict our presentation to the following simple subset of C, where a variable has either an integer value or a pointer (i.e. $\mathbb{V} = \mathbb{Z} + \mathbb{L}$):

$$x := e \mid *x := e \mid \mathsf{assume}(x < n)$$
$$\text{where} \quad e \to n \mid x \mid \&x \mid *x \mid e \ + \ e$$

Assignment $x := e$ corresponds to assigning the value of expression e to variable x. Store $*x := e$ performs indirect assignments; the value of e is assigned to the location that x points to. Assume command $\mathsf{assume}(x < n)$ makes the program continues only when the condition evaluates to true.

4.3.1 Step 1: Designing Non-sparse Analysis

Abstract Domain From the baseline abstraction (in Section 4.2.1), we consider a family of state abstractions

$$2^{\mathbb{S}} \underset{\alpha_{\mathbb{S}}}{\overset{\gamma_{\mathbb{S}}}{\leftrightarrows}} \hat{\mathbb{S}}$$

such that, (Because it is standard, we omit the definition of $\alpha_{\mathbb{S}}$.)

$$\begin{aligned} \hat{\mathbb{S}} &= \hat{\mathbb{L}} \to \hat{\mathbb{V}} \\ \hat{\mathbb{L}} &= \mathit{Var} \\ \hat{\mathbb{V}} &= \hat{\mathbb{Z}} \times \hat{\mathbb{P}} \\ \hat{\mathbb{P}} &= 2^{\hat{\mathbb{L}}} \end{aligned}$$

Abstract state $\hat{\mathbb{S}}$ maps abstract locations $\hat{\mathbb{L}}$ to abstract values $\hat{\mathbb{V}}$. An abstract location is a program variable. An abstract value is a pair of an abstract integer $\hat{\mathbb{Z}}$ and an abstract pointer $\hat{\mathbb{P}}$. A set of integers is abstracted to an abstract integer ($2^{\mathbb{Z}} \underset{\alpha_{\mathbb{Z}}}{\overset{\gamma_{\mathbb{Z}}}{\leftrightarrows}} \hat{\mathbb{Z}}$). Note that the abstraction is generic so we can choose any non-

relational numeric domains of our interest, such as intervals ($\hat{\mathbb{Z}} = \{[l,u] \mid l,u \in \mathbb{Z} \cup \{-\infty,+\infty\} \wedge l \leq u\}_\perp$) or constant propagation domain ($\hat{\mathbb{Z}} = \{\perp, \ldots, -1, 0, 1, \ldots, \top\}$). For simplicity, we do not abstract pointers (because they are finite): pointer values are kept by a points-to set ($\hat{\mathbb{P}} = 2^{\hat{\mathbb{L}}}$). Other pointer abstractions are also orthogonally applicable.

Abstract Semantics The abstract semantics is defined by the least fixpoint of semantic function (4.3), $\hat{F}$, where the abstract semantic function $\hat{f}_c \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ is defined as follows:

$$\hat{f}_c(\hat{s}) = \begin{cases} \hat{s}[x \mapsto \hat{\mathcal{E}}(e)(\hat{s})] & \mathsf{cmd}(c) = x := e \\ \hat{s}[\hat{s}(x).\hat{\mathbb{P}} \overset{w}{\mapsto} \hat{\mathcal{E}}(e)(\hat{s})] & \mathsf{cmd}(c) = *x := e \\ \hat{s}[x \mapsto \langle \hat{s}(x).\hat{\mathbb{Z}} \sqcap_{\hat{\mathbb{Z}}} \alpha_{\mathbb{Z}}(\{z \in \mathbb{Z} | z < n\}), \hat{s}(x).\hat{\mathbb{P}} \rangle] & \mathsf{cmd}(c) = \mathsf{assume}(x < n) \end{cases}$$

Auxiliary function $\hat{\mathcal{E}}(e)(\hat{s})$ computes abstract value of e under $\hat{s}$. Assignment $x := e$ updates the value of x. Store $*x := e$ weakly[1] updates the value of abstract locations that $*x$ denotes. $\mathsf{assume}(x < n)$ confines the interval value of x according to the condition. $\hat{\mathcal{E}} \in e \to \hat{\mathbb{S}} \to \hat{\mathbb{V}}$ is defined as follows:

$$\begin{array}{rcl} \hat{\mathcal{E}}(n)(\hat{s}) & = & \langle \alpha_{\mathbb{Z}}(\{n\}), \perp \rangle \\ \hat{\mathcal{E}}(x)(\hat{s}) & = & \hat{s}(x) \\ \hat{\mathcal{E}}(\&x)(\hat{s}) & = & \langle \perp, \{x\} \rangle \\ \hat{\mathcal{E}}(*x)(\hat{s}) & = & \bigsqcup \{\hat{s}(a) \mid a \in \hat{s}(x).\hat{\mathbb{P}}\} \\ \hat{\mathcal{E}}(e_1 \texttt{ + } e_2)(\hat{s}) & = & \langle v_1.\hat{\mathbb{Z}} \hat{+}_{\hat{\mathbb{Z}}} v_2.\hat{\mathbb{Z}}, v_1.\hat{\mathbb{P}} \cup v_2.\hat{\mathbb{P}} \rangle \\ & & \text{where } v_1 = \hat{\mathcal{E}}(e_1)(\hat{s}), v_2 = \hat{\mathcal{E}}(e_2)(\hat{s}) \end{array}$$

Note that the above analysis is parameterized by an abstract numeric domain $\hat{\mathbb{Z}}$ and sound operators $\hat{+}_{\hat{\mathbb{Z}}}$ and $\sqcap_{\hat{\mathbb{Z}}}$.

Lemma 10 (Soundness) *If* $\alpha_{\mathbb{S}} \circ f_c \sqsubseteq \hat{f}_c \circ \alpha_{\mathbb{S}}$, $\alpha(\mathbf{lfp}F) \sqsubseteq \mathbf{lfp}\hat{F}$.

[1] For brevity, we consider only weak updates. Applying strong update is orthogonal to sparse analysis design.

4.3.2 Step 2: Finding Definitions and Uses

The second step is to find a safe approximations of definitions and uses. The framework provides a mathematical definitions regarding correctness but does not provide how to find safe $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$. In the rest part of this section, we present a semantics-based, systematic way to find them.

We propose to find $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ from a conservative approximation of $\hat{F}$. We call the approximated analysis by pre-analysis. Let $\hat{\mathbb{D}}_p$ and $\hat{F}_p$ be the domain and semantic function of such a pre-analysis, which satisfies the following two conditions.

$$\mathbb{C} \to \hat{\mathbb{S}} \underset{\alpha_p}{\overset{\gamma_p}{\leftrightarrows}} \hat{\mathbb{D}}_p$$

$$\alpha_p \circ \hat{F} \sqsubseteq \hat{F}_p \circ \alpha_p$$

By abstract interpretation framework [12, 16], such a pre-analysis is guaranteed to be conservative, i.e., $\alpha_p(\mathbf{lfp}\hat{F}) \sqsubseteq \mathbf{lfp}\hat{F}_p$. As an example, in experiments (Section 4.6), we use a simple abstraction as follows:

$$\begin{array}{rcl} & & \mathbb{C} \to \hat{\mathbb{S}} \underset{\alpha_p}{\overset{\gamma_p}{\leftrightarrows}} \hat{\mathbb{S}} \\ \alpha_p & = & \lambda \hat{X}. \bigsqcup \{\hat{X}(c) \mid c \in \mathsf{dom}(\hat{X})\} \\ \hat{F}_p & = & \lambda \hat{s}. \bigsqcup_{c \in \mathbb{C}} \hat{f}_c(\hat{s}) \end{array}$$

The abstraction ignores the control dependences among control points and computes a single global invariant (a.k.a., flow-insensitivity).

We now define $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ by using pre-analysis. Let $\hat{\mathcal{T}}_p \in \mathbb{C} \to \hat{\mathbb{S}}$ be the pre-analysis result in terms of original analysis, i.e., $\hat{\mathcal{T}}_p = \gamma_p(\mathbf{lfp}\hat{F}_p)$. The definitions of $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ are naturally derived from the semantic definition of $\hat{f}_c$.

$$\hat{\mathsf{D}}(c) = \begin{cases} \{x\} & \mathsf{cmd}(c) = x := e \\ \hat{\mathcal{T}}_p(c)(x).\hat{\mathbb{P}} & \mathsf{cmd}(c) = *x := e \\ \{x\} & \mathsf{cmd}(c) = \mathsf{assume}(x < n) \end{cases}$$

$\hat{\mathsf{D}}$ is defined to include locations whose values are potentially defined (changed). In the definition of $\hat{f}_c$ for $x := e$ and $\mathsf{assume}(x < n)$, we notice that abstract location x may be defined. In $*x := e$, we see that $\hat{f}_c$ may define locations $\hat{s}(x).\hat{\mathbb{P}}$ for a given input state $\hat{s}$ at program point c. Here, we use the pre-analysis: because we cannot have the input state $\hat{s}$ prior to the analysis, we instead use its conservative abstraction $\hat{\mathcal{T}}_p(c)$. Such $\hat{\mathsf{D}}$ satisfies the safe approximation condition (Definition 4.2.6), because we collect all potentially defined locations, pre-analysis is conservative, and $\hat{f}$ is monotone.

Before defining $\hat{\mathsf{U}}$, we define an auxiliary function $\mathcal{U} \in e \to \hat{\mathbb{S}} \to 2^{\hat{\mathbb{L}}}$. Given expression e and state $\hat{s}$, $\mathcal{U}(e)(\hat{s})$ finds the set of abstract locations that are referenced during the evaluation of $\hat{\mathcal{E}}(e)(\hat{s})$. Thus, $\mathcal{U}$ is naturally derived from the definition of $\hat{\mathcal{E}}$.

$$\begin{aligned}
\mathcal{U}(n)(\hat{s}) &= \emptyset \\
\mathcal{U}(x)(\hat{s}) &= \{x\} \\
\mathcal{U}(\&x)(\hat{s}) &= \emptyset \\
\mathcal{U}(*x)(\hat{s}) &= \{x\} \cup \hat{s}(x).\hat{\mathbb{P}} \\
\mathcal{U}(e_1 \ + \ e_2)(\hat{s}) &= \mathcal{U}(e_1)(\hat{s}) \cup \mathcal{U}(e_2)(\hat{s})
\end{aligned}$$

When e is either n or $\&x$, $\hat{\mathcal{E}}$ does not refer any abstract location. Because $\hat{\mathcal{E}}(x)(\hat{s})$ references abstract location x, $\mathcal{U}(x)(\hat{s})$ is defined by $\{x\}$. $\hat{\mathcal{E}}(*x)(\hat{s})$ references location x and each location $a \in \hat{s}(x)$, thus the set of referenced locations is $\{x\} \cup \hat{s}(x).\hat{\mathbb{P}}$. $\hat{\mathsf{U}}$ is defined as follows: (For brevity, let $\hat{s}_c = \hat{\mathcal{T}}_p(c)$)

$$\hat{\mathsf{U}}(c) = \begin{cases} \mathcal{U}(e)(\hat{s}_c) & \mathsf{cmd}(c) = x := e \\ \{x\} \cup \hat{s}_c(x).\hat{\mathbb{P}} \cup \mathcal{U}(e)(\hat{s}_c) & \mathsf{cmd}(c) = *x := e \\ \{x\} & \mathsf{cmd}(c) = \mathsf{assume}(x < n) \end{cases}$$

Using $\hat{\mathcal{T}}_p$ and $\mathcal{U}$, we collect abstract locations that are potentially used during the evaluation of e. Because $\hat{f}_c$ is defined to refer to abstract location x in $*x := e$ and $\mathsf{assume}(x < n)$, $\mathcal{U}$ additionally includes x.

Lemma 11 *$\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ are safe approximations.*

Proof Note that $\hat{\mathsf{D}}(c) - \mathsf{D}(c)$ is not empty only when $\mathsf{cmd}(c) = *x := e$. Then,

$$\hat{\mathsf{D}}(c) = \hat{\mathcal{T}}_p(c)(x).\hat{\mathbb{P}} \subseteq \{x\} \cup \hat{\mathcal{T}}_p(c)(x).\hat{\mathbb{P}} \cup \mathcal{U}(e)(\hat{s}_c) = \hat{\mathsf{U}}(c)$$

Thus, $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ are safe approximations □

Sparse Pointer Analysis as Instances We can instantiate this design of non-relational analysis to the recent two successful scalable sparse analysis presented in [24, 25].

Semi-sparse analysis [24] applies sparse analysis only for top-level variables whose addresses are never taken. We do the same thing by designing pre-analysis which computes a fixpoint $\hat{\mathcal{T}}_p$ such that $\hat{\mathcal{T}}_p(c)(x).\hat{\mathbb{P}} = \hat{\mathbb{L}}$ for all xs that are not top-level variables.

Staged Flow-Sensitive Analysis [25] uses auxiliary flow-insensitive pointer analysis to get an over-approximation of def-use information on pointer variables. By coincidence, our sparse non-relational analysis already does the same analysis for pointer variables except it also tracks numeric constraints of variables. We can design pre-analysis whose precision is incomparable to the original one, as in [25], although the framework cannot guarantee the correctness anymore.

4.4 Example: Sparse Relational Analysis

As another example, we show how to design sparse relational analyses. As before, we design the sparse analysis within our framework: (1) We design a relational analysis by a conventional abstract interpretation; (2) We define $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ that safely approximates definitions and uses of the analysis.

We consider relational analyses with variable packings. A pack is a set of variables selected to be related together. In the rest of this section, we assume a set of variable packs, $\mathit{Packs} \subseteq 2^{\mathit{Var}}$ such that $\bigcup \mathit{Packs} = \mathit{Var}$, are given by users or a pre-analysis [14, 46]. In a packed relational analysis, abstract states ($\hat{\mathbb{S}}$) map variable packs (Packs) to a relational domain ($\hat{\mathbb{R}}$), i.e., $\hat{\mathbb{S}} = \mathit{Packs} \to \hat{\mathbb{R}}$, which is a

member of the abstract domains that our framework deals with (Section 4.2.1). In fact, the packed relational domain ($\hat{\mathbb{S}}$) is a generalized version of $\hat{\mathbb{R}}$: $\hat{\mathbb{R}}$ is a special case of $\hat{\mathbb{S}}$ in which *Packs* is the full set of variables ($Packs = \{Var\}$). However, such full relational analysis is impractical [14]. Thus, realistic relational analyzers usually exploit small packs of variables [5, 14, 67].

The distinguishing feature of sparse relational analysis is that definition sets and use sets are defined in terms of variable packs. Consider code x:=1. Which abstract location is defined? In non-relational analysis, variable x may be defined. In relational analysis, because an abstract location is a pack, variable packs that contain x may be defined. For example, when $Packs = \{\langle\langle x, y\rangle\rangle, \langle\langle x, z\rangle\rangle, \langle\langle y, z\rangle\rangle\}$, definitions are $\{\langle\langle x, y\rangle\rangle, \langle\langle x, z\rangle\rangle\}$.

In Section 4.4.1, we define relational analysis with variable packing. In Section 4.4.2, we define safe $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$.

4.4.1 Step 1: Designing Non-sparse Analysis

For brevity,we consider the following pointer-free language: ($\mathbb{S} = Var \to \mathbb{Z}$).

$$cmd \to x := e \mid \mathsf{assume}(x < n) \quad \text{where } e \to n \mid x \mid e \ + \ e$$

Including pointers in the language does not require novelty but verbosity. We focus only on the key differences between non-relational and relational sparse analysis designs.

Abstract Domain From the baseline abstraction (in Section 4.2.1), we consider a family of state abstractions $2^{\mathbb{S}} \underset{\alpha_{\mathbb{S}}}{\overset{\gamma_{\mathbb{S}}}{\leftrightarrows}} \hat{\mathbb{S}}$ such that, ($\alpha_{\mathbb{S}}$ is defined using $\alpha_{\hat{\mathbb{R}}}$ such that $2^{\mathbb{S}} \underset{\alpha_{\hat{\mathbb{R}}}}{\overset{\gamma_{\hat{\mathbb{R}}}}{\leftrightarrows}} \hat{\mathbb{R}}$.)

$$\begin{aligned} \hat{\mathbb{S}} &= \hat{\mathbb{L}} \to \hat{\mathbb{V}} \\ \hat{\mathbb{L}} &= Packs \\ \hat{\mathbb{V}} &= \hat{\mathbb{R}} \end{aligned}$$

Note that the abstraction is generic so we can choose any relational domain for $\hat{\mathbb{R}}$, such as octagon [46].

Abstract Semantics In packed relational analysis, we sometimes need to know actual values (such as ranges) of variables. For example, suppose we analyze a:=b with $Packs = \{\langle\langle a, c\rangle\rangle, \langle\langle b, c\rangle\rangle\}$. Analyzing the statement amounts to updating the abstract value associated with pack $\langle\langle a, c\rangle\rangle$. However, because variable b is not contained in the pack, we need to obtain the value of b from the abstract value associated with $\langle\langle b, c\rangle\rangle$. Here, the value for b is obtained by projecting the relational domain elements associated with $\langle\langle b, c\rangle\rangle$ into a non-relational value, such as intervals. Thus, we consider abstract semantic function $\mathcal{R} \in cmd^r \to \hat{\mathbb{R}} \to \hat{\mathbb{R}}$ for relational domain $\hat{\mathbb{R}}$ is defined over the following internal language:

$$cmd^r \to x := e^r \mid \mathsf{assume}(x < \hat{\mathbb{Z}}) \text{ where } e^r \to \hat{\mathbb{Z}} \mid x \mid e^r \texttt{ + } e^r$$

where $\hat{\mathbb{Z}}$ is an (non-relational) abstract integer ($2^{\mathbb{Z}} \underset{\alpha_{\mathbb{Z}}}{\overset{\gamma_{\mathbb{Z}}}{\leftrightarrows}} \hat{\mathbb{Z}}$), such as an interval. This language is not for program codes, but for our semantics definition.

We now define the semantics of the packed relational analysis. The abstract semantics is defined by the least fixpoint of semantic function (4.3), where the abstract semantic function $\hat{f}_c$ is defined as follows:

$$\hat{f}_c(\hat{s}) = \hat{s}[p_1 \mapsto \mathcal{R}(cmd_1)(\hat{s}(p_1)), \ldots, p_k \mapsto \mathcal{R}(cmd_k)(\hat{s}(p_k))]$$

where

$$\begin{array}{rcl} \{p_1, \ldots, p_k\} & = & \begin{cases} \mathsf{pack}(x) & \mathsf{cmd}(c) = x := e \\ \mathsf{pack}(x) & \mathsf{cmd}(c) = \mathsf{assume}(x < n) \end{cases} \\ cmd_i & = & \mathscr{T}(p_i)(\hat{s})(\mathsf{cmd}(c)) \end{array}$$

For variable x, $\mathsf{pack}(x)$ returns the set of packs that contains x, i.e., $\mathsf{pack}(x) = \{p \in Packs \mid x \in p\}$. For both $x := e$ and $\mathsf{assume}(x < n)$, we update only the packs that include x. $\rightsquigarrow$ is the function that transforms cmd into cmd^r. Given a variable

pack p, state $\hat{s}$, and command cmd, $\rightsquigarrow (p)(\hat{s}) \in cmd \to cmd^r$ returns transformed command for a given command.

$$\begin{aligned} \mathscr{T}(p)(\hat{s})(x := e) &= x := \mathscr{T}_e(p)(\hat{s})(e) \\ \mathscr{T}(p)(\hat{s})(\mathsf{assume}(x < n)) &= \mathsf{assume}(x < \mathscr{T}_e(p)(\hat{s})(n)) \end{aligned}$$

where $\mathscr{T}_e(p)(\hat{s}) \in e \to e^r$ transforms expressions:

$$\begin{aligned} \mathscr{T}_e(p)(\hat{s})(n) &= \alpha_{\mathbb{Z}}(\{n\}) \\ \mathscr{T}_e(p)(\hat{s})(x) &= \begin{cases} x & \text{if } x \in p \\ \pi_x(\hat{s}) & \text{otherwise} \end{cases} \\ \mathscr{T}_e(p)(\hat{s})(e_1 \ + \ e_2) &= \mathscr{T}_e(p)(\hat{s})(e_1) \ + \ \mathscr{T}_e(p)(\hat{s})(e_2) \end{aligned}$$

where $\pi_x \in \hat{\mathbb{S}} \to \hat{\mathbb{Z}}$ is a function that projects a relational domain element onto variable x to obtain its abstract integer value. To be safe, π_x should satisfies the following condition:

$$\forall \hat{s} \in \hat{\mathbb{S}}.\pi_x(\hat{s}) \sqsupseteq \alpha_{\hat{\mathbb{Z}}}(\{s(x) | s \in \gamma_{\hat{\mathbb{R}}}(\sqcap_{p \in \mathsf{pack}(x)} \hat{s}(p))\})$$

Lemma 12 (Soundness) *If* $\alpha_{\mathbb{S}} \circ f_c \sqsubseteq \hat{f}_c \circ \alpha_{\mathbb{S}}$, $\alpha(\mathbf{lfp}F) \sqsubseteq \mathbf{lfp}\hat{F}$.

4.4.2 Step 2: Finding Definitions and Uses

We now approximate $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$. In the previous section, we already presented a general, semantics-based method to safely approximate $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ for a given abstract semantics. Because our language in this section is pointer-free, simple syntactic method is enough for our purpose.

The distinguishing feature of sparse relational analysis is that the entities that are defined and used are variable packs, not each variable. From the definition of $\hat{f}_c$, we notice that packs $\mathsf{pack}(x)$ are potentially defined both in assignment and assume:

$$\hat{\mathsf{D}}(c) = \begin{cases} \mathsf{pack}(x) & \mathsf{cmd}(c) = x := e \\ \mathsf{pack}(x) & \mathsf{cmd}(c) = \mathsf{assume}(x < n) \end{cases}$$

$\hat{\mathsf{U}}$ is defined depending on the definition of π_x. On the assumption that *Packs* contains singleton packs of all program variables, we may define π_x as follows:

$$\pi_x(\hat{s}) = \pi_x^{rel}(\hat{s}(\{x\}))$$

where $\pi_x^{rel} \in \hat{\mathbb{R}} \to \hat{\mathbb{Z}}$ which project a relational domain element onto variable x to obtain its abstract integer value, which is supplied by each relational domain, e.g., see [46]. In section 3.5, we implement our analyzer based on this definition of π_x.

Now, we define $\hat{\mathsf{U}}$ as follows:

$$\hat{\mathsf{U}}(c) = \begin{cases} \mathsf{pack}(x) \cup \{\{l\} \mid l \in \mathcal{U}(e)\} & \mathsf{cmd}(c) = x := e \\ \mathsf{pack}(x) & \mathsf{cmd}(c) = \mathsf{assume}(x < n) \end{cases}$$

where we $\mathcal{U}(e)$ denotes the set of variables that appear inside expression e.

Lemma 13 $\hat{\mathsf{D}}$ *and* $\hat{\mathsf{U}}$ *are safe approximations.*

4.5 Implementation Techniques

So far, we have mainly discussed about how to design a correct sparse analysis. In this section, we discuss practical issues.

Implementing sparse analysis presents unique challenges regarding construction and management of data dependencies. The key different feature of sparse analysis from conventional analysis is that sparse analysis uses data dependencies instead of control flows of programs. Because, data dependencies for realistic programs are very complex, it is the key to practical sparse analyzers to generate data dependencies efficiently in space and time. We describe the basic algorithm we used for dependency generation, and discuss two performance issues that we experienced in the implementation of sparse analyzers (Section 4.6).

Generation of Data Dependencies We use the standard SSA algorithm to generate data dependencies. Because our notion of data dependencies ($\rightsquigarrow$, Definition 4.2.4) equals to def-use chains with $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$ being treated as must-definitions

and must-uses, in fact, any def-use chain generation algorithms (e.g., reaching definition analysis, SSA algorithm) can be used. We use SSA generation algorithm because it is fast and reduces size of def-use chains [68].

Interprocedural Extension During the implementation, we noticed that the ways of handling procedure calls significantly influence the efficiency of data dependency generation. Because we are interested in global interprocedural analysis, the data dependencies should describe inter-dependencies of data across procedure boundaries. There are two possible ways for the interprocedural extension.

The first approach is to compute global data dependencies for the entire program, but was not scalable in our case. The main problem was due to unexpected spurious dependencies among procedures. Consider the following code and suppose we compute data dependencies for global variable `x`.

```
int f() { x=0;[1] h(); a=x;[2]}
int h() { ... } // does not use variable x
int g() { x=1;[3] h(); b=x;[4]}
```

Data dependencies for `x` not only include $\texttt{1} \overset{\texttt{x}}{\rightsquigarrow} \texttt{2}$ and $\texttt{3} \overset{\texttt{x}}{\rightsquigarrow} \texttt{4}$ but also include spurious dependencies $\texttt{1} \overset{\texttt{x}}{\rightsquigarrow} \texttt{4}$ and $\texttt{3} \overset{\texttt{x}}{\rightsquigarrow} \texttt{2}$, because there are control flow paths from `1` to `4` (as well as `3` to `2`) via the common procedure calls to `h`. In real C programs, thousands of global variables exist and procedures are called from many different callsites, which generates overwhelming number of spurious dependencies. In our experiments, such spurious dependencies made this approach hardly scalable. Staged pointer analysis algorithm [25] takes this approach but no performance problem was reported; we guess that this is because pointer analysis typically ignores non-pointer statements or locations–for example, number of global pointer variables are just small subset of the entire globals. However, in full semantics analyses, we must consider all aspects of the entire program. The needs for careful handling of spurious interprocedural flows were also reported previously [48], where spurious paths

were shown to be a major reason for performance problems of (non-sparse) global static analysis.

Thus, we generate data dependencies separately for each procedure. In this approach, we need to specially handle procedure calls; we treat a procedure call as a definition (resp., use) of all abstract locations defined (resp., used) by the callee. After generating dependencies inside each procedure, we connect inter-dependences among procedures. For the inter-connection, we use the flow-insensitive pre-analysis (defined in Section 4.3.2) to resolve function pointers. Because the pre-analysis is fairly precise[2], the precision loss caused by this approximation of the callgraph would be reasonably small in practice [45]. With this approach, above spurious dependencies reduces, because variable `x` is not propagated to procedure `h` (because `h` does not use `x`). However, there is a problem with this method; data dependencies are not fully sparse. For example, consider a call chain $\texttt{f} \rightarrow \texttt{g} \rightarrow \texttt{h}$ and suppose `x` is defined in procedure `f` and used in procedure `h`. Even when `x` is not used inside `g`, value of `x` is propagated to `g`. In our experiments, this incomplete sparseness of the analysis could not make the resulting sparse analysis scalable enough. We solved the problem by applying an optimization to the data dependencies ($\rightsquigarrow$) until convergence. Suppose $a \overset{l}{\rightsquigarrow} b$, $b \overset{l}{\rightsquigarrow} c$, and that l is not defined nor used in b, then we remove those two dependencies and add $a \overset{l}{\rightsquigarrow} c$. This optimization is clearly sound, and makes the approach more sparse, leading to significant speed up.

Using BDDs in Representing Data Dependencies The second practical issue is memory consumption of data dependencies. Analyzing real C programs must deal with hundreds of thousands of statements and abstract locations. Thus, naive representations for the data dependencies immediately makes memory problems. For example, in analyzing ghostscript-9.00 (the largest benchmark in Table 4.1), the data dependencies consist of 201 K abstract locations spanning over 2.8 M state-

[2] The pointer abstraction of our pre-analysis is basically the same with inclusion-based pointer analysis, which is the most precise form of flow-insensitive pointer analysis [23]. In addition, our pre-analysis combines numeric analysis and pointer analysis, which further enhances the precision of the pointer analysis [3, 14].

ments. Thus, storing dependency relation $\rightsquigarrow$ by a naive set-based implementation, which keeps a map ($\in \mathbb{C} \times \mathbb{C} \to 2^{\hat{\mathbb{L}}}$) , did not work for such large programs (It only worked for programs of moderate sizes less than 150 KLOC). Fortunately, the dependency relation is highly redundant, making it a good application of BDDs. For example, $\langle c_1, c_3, l\rangle \in (\rightsquigarrow)$ and $\langle c_2, c_3, l\rangle$ are different but share the common suffix, and $\langle c_1, c_2, l_1\rangle$ and $\langle c_1, c_2, l_2\rangle$ are different but share the common prefix. BDDs can effectively share such common suffixes and prefixes. We treat each relation $\langle c_1, c_2, l\rangle$, by bit-encoding each control point and abstract location, as a boolean function that is naturally represented by BDDs. This way of using BDDs greatly reduced memory costs. For example, for vim60 (227 KLOC), set-based representation of data dependencies required more than 24 GB of memory but BDD-implementation just required 1 GB. No particular dynamic variable ordering was necessary in our case.

4.6 Experiments

In this section, we evaluate sparse analyses designed in Section 4.3. For the non-relational analysis, we use the interval domain [16], a representative non-relational domain that is widely used in practice [2, 3, 5, 14, 31].

We have analyzed 18 software packages. Table 4.1 shows characteristics of our benchmark programs. The benchmarks are various open-source applications, and most of them are from GNU open-source projects. The linux kernel includes only a few drivers (keyboard, power management, block device, and terminal) but includes many other modules such as file system, memory management, x86 architecture, and so on. The analyses were performed globally (whole-program analysis); the entire program is analyzed starting from procedure main (for linux, start_kernel). Standard library calls are summarized using handcrafted function stubs. For other unknown procedure calls to external code, we assume that the procedure returns arbitrary values and has no side-effect. Procedures that are unreachable from the main procedure, such as callbacks, are made to be explicitly called from the main

procedure. All experiments were done on a Linux 2.6 system running on a single core of Intel 3.07 GHz box with 24 GB of main memory.

4.6.1 Setting

The baseline analyzer, Baseline, is a global abstract interpretation engine in an industrialized verification tool for C programs. The analysis consists of a frontend and backend. The frontend parses C code and transforms it into an intermediate language (IL) and the backend performs the analysis. The analyzer handles all aspects of C. The abstract domain of the analysis is an extension of the one defined in Section 4.3 to support additional C features such as arrays and structures. The analysis abstracts an array by a set of tuples of base address, offset, and size. Abstraction of dynamically allocated array is similarly handled except that base addresses are abstracted by their allocation-sites. A structure is abstracted by a tuple of base address and set of field locations (the analysis is field-sensitive). The fixpoint is computed by a worklist algorithm using the conventional widening operator [16] for interval domain.

The baseline analyzer is not a straw-man but much engineering effort has been put to its implementation. It adopts a set of well-known cost reduction techniques in static analysis such as efficient worklist/widening strategies [6] and selective memory operators [5]. In particular, the analysis aggressively exploits the technique of localization [49, 59, 71]. We use access-based localization [49]: before entering a code block such as procedure body, its input memory state is tightly localized so that the block is analyzed with only the to-be-accessed parts of the input state. The memory parts that will be accessed by the procedure is safely, yet precisely, estimated by an efficient auxiliary analysis. The analysis with the access-based localization technique is faster by up to 50x than reachability localization-based analysis [49].

From the baseline, we made two analyzers: Vanilla and Sparse. Vanilla is identical to Baseline except that Vanilla does not perform the access-based localization. We compare the performance between Vanilla and Baseline just to check that our

baseline analyzer is not a straw-man. Sparse is the sparse version derived from the baseline. The sparse analysis consists of three steps: pre-analysis (to approximate def-use sets), data dependency generation, and actual fixpoint computation. As defined in Section 4.3, we use a flow-insensitive pre-analysis. Data dependencies are generated as described in Section 4.5. The fixpoint of sparse abstract transfer function is computed by a worklist-based fixpoint algorithm. The analyzers are written in OCaml. We use the BuDDy library [38].

4.6.2 Results

Table 4.2 gives the analysis time and peak memory consumption of the three analyzers. Because three analyzers share a common frontend, we report only the time for backend (analysis). For Baseline, the time includes auxiliary analysis time for access-based localization [49] as well as actual analysis time. For Sparse, **Dep** includes times for pre-analysis and generation of data dependencies. **Fix** represent the time for fixpoint computation.

The results show that Baseline already has a competitive performance: it is faster than Vanilla by 8–55x, saving peak memory consumption by 54–85%. Vanilla scales to 35 KLOC before running out of time limit (24 hours). In contrast, Baseline scales to 111 KLOC. For the first six benchmarks that they both complete, Baseline is on average 27x faster than Vanilla, and uses on average 71% less memory.

Sparse is faster than Baseline by 5–110x and saves memory by 3–92%. In particular, the analysis' scalability has been remarkably improved: Sparse scales to 1.4M LOC, which is an order of magnitude larger than that of Baseline.

There are some counterintuitive results. First, the analysis time for Sparse does not strictly depend on program sizes. For example, analyzing emacs-22.1 (399 KLOC) requires 10 hours, taking six times more than analyzing ghostscript-9.00 (1,363 KLOC). This is mainly due to the fact that some real C programs have unexpectedly large recursive call cycles [36,73]. Column **maxSCC** in Table 4.1 reports the sizes of the largest recursive cycle (precisely speaking, strongly connected component) in pro-

grams. Note that some programs (such as nethack-3.3.0, vim60, and emacs-22.1) have a large cycle that contains hundreds or even thousands of procedures. Such non-trivial SCCs markedly increase analysis cost because the large cyclic dependencies among procedures make data dependencies much more complex. Thus, the analysis for gimp-2.6 (959 KLOC) or ghostscript-9.00 (1,363 KLOC), which have few recursion, is even faster than python-2.5.1 (435 KLOC) or nethack-3.3.0 (211 KLOC), which have large recursive cycles.

Second, data dependency generation takes much longer time than actual fixpoint computation. For example, data dependency generation for ghostscript-9.00 takes 14,116 s but the fixpoint is computed in 698 s. In fact, this phenomenon paradoxically shows the effectiveness of our pre-analysis. Finding data dependencies of programs is not an easy work but their exact computation requires the full analysis (Baseline). Instead, the pre-analysis finds an approximation with small cost (compared to Baseline). Our pre-analysis is effective because the approximated data dependencies are shown to be precise enough to make our sparse analysis efficient. On the other hand, the seemingly unbalanced timing results are partly because of the uses of BDDs in dependency construction. While BDD dramatically saves memory costs, set operations for BDDs such as addition and removal are noticeably slower than usual set operations. Especially, large programs are more influenced by this characteristic because their data dependency generation is more complex and much more BDD-operations are involved. However, thanks to the space-effectiveness of BDDs, our sparse analysis does not steeply increase memory consumption as program sizes increase.

4.6.3 Discussion on Sparsity

We discuss the relation between performance and sparsity. Column $\hat{\mathsf{D}}(c)$ and $\hat{\mathsf{U}}(c)$ in Table 4.2 show how many abstract locations are defined and used for each basic block on average. It clearly shows the key observation to sparse analysis in real programs; only a few abstract locations are defined and used in each program point. In interval domain-based analysis, 2.4–285.3 abstract locations are defined (Avg.

$\hat{\mathsf{D}}(c)$) and 2.5–285.5 are used (Avg. $\hat{\mathsf{U}}(c)$) in average. For example, a2ps-4.14 defines and uses only 0.1% of all abstract locations in one program point. Similarly, 2.3–15.9 (resp., 2.5–16.0) variable packs per program point are defined (resp., used) in octagon domain-based analysis. By exploiting this sparsity of analysis, we could achieve orders of magnitude speed up compared to the baseline possible.

One interesting observation from the experiment results is that the analysis performance is more dependent on the sparsity than the program size. As an extreme case, consider two programs, emacs-22.1 and ghostscript-9.00. Even though ghostscript-9.00 is 3.5 times bigger than emacs-22.1 in terms of LOC, ghostscript-9.00 takes 2.6 times less time to analyze. Behind this phenomenon, there is a large difference of sparsity; average $\hat{\mathsf{D}}(c)$ size (and $\hat{\mathsf{U}}(c)$ size) of emacs-22.1 is 30 times bigger than the one of ghostscript-9.00.

Program	LOC	Functions	Statements	Blocks	maxSCC	AbsLocs
gzip-1.2.4a	7K	132	6,446	4,152	2	1,784
bc-1.06	13K	132	10,368	4,731	1	1,619
tar-1.13	20K	221	12,199	8,586	13	3,245
less-382	23K	382	23,367	9,207	46	3,658
make-3.76.1	27K	190	14,010	9,094	57	4,527
wget-1.9	35K	433	28,958	14,537	13	6,675
screen-4.0.2	45K	588	39,693	29,498	65	12,566
a2ps-4.14	64K	980	86,867	27,565	6	17,684
bash-2.05a	105K	955	107,774	27,669	4	17,443
lsh-2.0.4	111K	1,524	137,511	27,896	13	31,164
sendmail-8.13.6	130K	756	76,630	52,505	60	19,135
nethack-3.3.0	211K	2,207	237,427	157,645	997	54,989
vim60	227K	2,770	150,950	107,629	1,668	40,979
emacs-22.1	399K	3,388	204,865	161,118	1,554	66,413
python-2.5.1	435K	2,996	241,511	99,014	723	51,859
linux-3.0	710K	13,856	345,407	300,203	493	139,667
gimp-2.6	959K	11,728	1,482,230	286,588	2	190,806
ghostscript-9.00	1,363K	12,993	2,891,500	342,293	39	201,161

Table 4.1: Benchmarks: lines of code (**LOC**) is obtained by running `wc` on the source before preprocessing and macro expansion. **Functions** reports the number of functions in source code. **Statements** and **Blocks** report the number of statements and basic blocks in our intermediate representation of programs (after preprocessing). **maxSCC** reports the size of the largest strongly connected component in the callgraph. **AbsLocs** reports the number of abstract locations that are generated during the analysis.

Programs	Vanilla		Baseline		**Spd↑$_1$**	**Mem↓$_1$**	Sparse						**Spd↑$_2$**	**Mem↓$_2$**
	Time	**Mem**	**Time**	**Mem**			**Dep**	**Fix**	**Total**	**Mem**	**D**	**U**		
gzip-1.2.4a	772	240	14	65	**55 x**	**73 %**	2	1	3	63	2.4	2.5	**5 x**	**3 %**
bc-1.06	1,270	276	96	126	**13 x**	**54 %**	4	3	7	75	4.6	4.9	**14 x**	**40 %**
tar-1.13	12,947	881	338	177	**38 x**	**80 %**	6	2	8	93	2.9	2.9	**42 x**	**47 %**
less-382	9,561	1,113	1,211	378	**8 x**	**66 %**	27	6	33	127	11.9	11.9	**37 x**	**66 %**
make-3.76.1	24,240	1,391	1,893	443	**13 x**	**68 %**	16	5	21	114	5.8	5.8	**90 x**	**74 %**
wget-1.9	44,092	2,546	1,214	378	**36 x**	**85 %**	8	3	11	85	2.4	2.4	**110 x**	**78 %**
screen-4.0.2	∞	N/A	31,324	3,996	N/A	N/A	724	43	767	303	53.0	54.0	**41 x**	**92 %**
a2ps-4.14	∞	N/A	3,200	1,392	N/A	N/A	31	9	40	353	2.6	2.8	**80 x**	**75 %**
bash-2.05a	∞	N/A	1,683	1,386	N/A	N/A	45	22	67	220	3.0	3.0	**25 x**	**84 %**
lsh-2.0.4	∞	N/A	45,522	5,266	N/A	N/A	391	80	471	577	21.1	21.2	**97 x**	**89 %**
sendmail-8.13.6	∞	N/A	∞	N/A	N/A	N/A	517	227	744	678	20.7	20.7	N/A	N/A
nethack-3.3.0	∞	N/A	∞	N/A	N/A	N/A	14,126	2,247	16,373	5,298	72.4	72.4	N/A	N/A
vim60	∞	N/A	∞	N/A	N/A	N/A	17,518	6,280	23,798	5,190	180.2	180.3	N/A	N/A
emacs-22.1	∞	N/A	∞	N/A	N/A	N/A	29,552	8,278	37,830	7,795	285.3	285.5	N/A	N/A
python-2.5.1	∞	N/A	∞	N/A	N/A	N/A	9,677	1,362	11,039	5,535	108.1	108.1	N/A	N/A
linux-3.0	∞	N/A	∞	N/A	N/A	N/A	26,669	6,949	33,618	20,529	76.2	74.8	N/A	N/A
gimp-2.6	∞	N/A	∞	N/A	N/A	N/A	3,751	123	3,874	3,602	4.1	3.9	N/A	N/A
ghostscript-9.00	∞	N/A	∞	N/A	N/A	N/A	14,116	698	14,814	6,384	9.7	9.7	N/A	N/A

Table 4.2: Performance of interval analysis: time (in seconds) and peak memory consumption (in megabytes) of the various versions of analyses. ∞ means the analysis ran out of time (exceeded 24 hour time limit). **Dep** and **Fix** reports the time spent during data dependency analysis and actual analysis steps, respectively, of the sparse analysis. **Spd↑$_1$** is the speed-up of Baseline over Vanilla. **Mem↓$_1$** shows the memory savings of Baseline over Vanilla. **Spd↑$_2$** is the speed-up of Sparse over Baseline. **Mem↓$_2$** shows the memory savings of Sparse over Baseline. $\hat{\mathsf{D}}(c)$ and $\hat{\mathsf{U}}(c)$ show the average size of $\hat{\mathsf{D}}(c)$ and $\hat{\mathsf{U}}(c)$, respectively.

Chapter 5

Contextual Localization

In this chapter, we present a contextual localization technique to mitigate substantial performance degradation caused by spurious interprocedural cycles. Spurious interprocedural cycles are, in a *realistic* setting, key reasons for why approximate call-return semantics in both context-sensitive and -insensitive static analysis can make the analysis much slower than expected.

In the call-strings-based context-sensitive analysis, because the number of distinguished contexts is finite, multiple call-contexts are inevitably joined at the entry of a procedure and the output at the exit is propagated to multiple return-sites. We found that these multiple returns frequently create a single large cycle (we call it "butterfly cycle") covering almost all parts of the program and such a spurious cycle makes analyses very slow and inaccurate.

Our simple algorithmic technique (within the fixpoint iteration algorithm) enables procedures to be analyzed in a contextually localized way: each procedure call is exclusively processed so that pruning spurious return flows is safe. The technique's effectiveness is proven by experiments with a realistic C analyzer to reduce the analysis time by 7%-96%. Since the technique is *algorithmic*, it can be easily applicable to existing analyses without changing the underlying abstract semantics, it is orthogonal to the underlying abstract semantics' context-sensitivity, and its correctness is obvious.

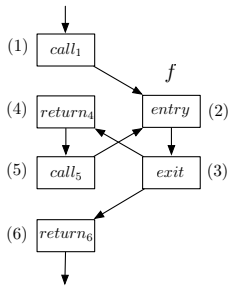
5.1 Introduction

In the approximate context-sensitive analysis, it is inevitable to follow some spurious (unrealizable or invalid) return paths. When the analysis uses a limited context information in which the number of distinguished contexts is finite, multiple call-contexts are inevitably joined at the entry of a procedure and the output at the exit are propagated to multiple return-sites. For example, in a conventional way of avoiding invalid return paths by distinguishing a finite $k \geq 0$ call-sites to each procedure [64], the analysis is doomed to still follow spurious paths if the input program's nested call-depth is larger than the k. Increasing the k to remove more spurious paths quickly hits a limit in practice because of the increasing analysis cost in memory and time.

In this chapter, we present the following:

- in a realistic setting, these multiple returns often create a single large flow cycle (we call it "butterfly cycle") covering almost all parts of the program,
- such a big spurious cycle makes the approximate call-strings method [64] that distinguishes the last k call-sites very slow and inaccurate,
- this performance problem can be relieved by a simple extension of the call-strings method,
- our extension is an algorithmic technique within the worklist-based fixpoint iteration routine, without redesigning the underlying abstract semantics part
- the algorithmic technique works regardless of the underlying abstract semantics' context-sensitivity (the k), and
- the technique also works regardless of the existing worklist ordering strategies of the fixpoint algorithm. The technique consistently saves the analysis time, without sacrificing (or with even improving) the analysis precision.

Performance Degradation by Spurious Interprocedural Cycles Static analysis' spurious paths make spurious cycles across procedure boundaries in global



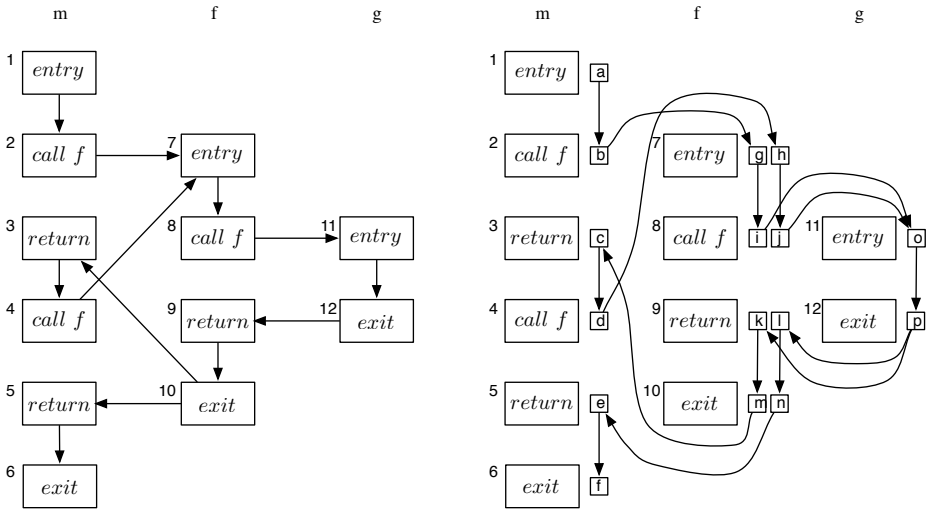
$$
\begin{array}{llll}
call_1 & = & \cdots & \cdots \ (1) \\
entry & = & call_1 \sqcup call_5 & \cdots \ (2) \\
exit & = & \hat{f}(entry) & \cdots \ (3) \\
return_4 & = & exit & \cdots \ (4) \\
call_5 & = & return_4 & \cdots \ (5) \\
return_6 & = & exit & \cdots \ (6)
\end{array}
$$

Figure 5.1: Spurious dependence cycles $(2) \to (3) \to (4) \to (5) \to (2) \to \cdots$ caused by abstract procedure calls and returns.

analysis. For example, consider the semantic equations in Figure 5.1 that (context-insensitively ($k = 0$)) abstract two consecutive calls to a procedure. The system of equations says to evaluate equation (4) and (6) for every return-site after analyzing the called procedure body (equation (3)). Thus, solving the equations follows a cycle: $(2) \to (3) \to (4) \to (5) \to (2) \to \cdots$.

Spurious cycles can be also created when $k \geq 1$. The k length suffix method can be understood by applying intraprocedural analysis algorithm to the extended supergraph [40]. An extended supergraph is a directed graph whose nodes are defined by pairs of nodes and their possible contexts and the edges explicitly shows the propagation paths of abstract values in context-sensitive manner. Figure 5.2(a) shows an example of supergraph where procedure `f` is called twice from procedure `m` and `g` is called once from `f`. Figure 5.2(b) shows its extended supergraph for $k = 1$. In Figure 5.2(b), since `f` is called by two times, each node of `f` has two separate contexts. But, since `g` is called only once, each node of `g` has only one context. Note that, even though the procedure `g` returns to a single return node (node 9), there are two paths which flow to the two different contexts, k and l: these two contexts are due to the two different call sites (node 2 and 4). So the analysis follows a spurious cycle $m \to c \to d \to h \to j \to o \to p \to k \to m \to \cdots$.

Such spurious cycles degrade the analysis performance both in precision and speed. Spurious cycles exacerbate the analysis imprecision because they model spu-



(a) a supergraph (b) the extended supergraph for $k = 1$

Figure 5.2: Spurious cycles in context-sensitive analysis ($k = 1$).

rious information flow. Spurious cycles degrade the analysis speed too because solving cyclic equations repeatedly applies the equations in vain until a fixpoint is reached.

The performance degradation becomes dramatic when the involved interprocedural spurious cycles cover a large part of the input program. This is indeed the case in reality. In analyzing real C programs, we observed that the analysis follows (Section 5.2) a single large cycle that spans almost all parts of the input program. Such spurious cycles size can also be estimated by just measuring the strongly connected components (scc) in the "lexical"[1] control flow graphs. Table 5.1 shows the sizes of the largest scc in some open-source programs.[2] In most programs, such cycles cover most (80-90%) parts of the programs. Hence, globally analyzing a program is likely to compute a fixpoint of a function that describes almost all parts of

[1] One node per lexical entity, ignoring function pointers.

[2] We measured the sizes of all possible cycles in the flow graphs. Note that interprocedural cycles happen because of either spurious returns or recursive calls. Because recursive calls in the test C programs are immediate or spans only a small number of procedures, large interprocedural cycles are likely to be spurious ones.

the input program. Even when we do the call-strings-based context-sensitive analysis ($k > 0$), large spurious cycles are likely to remain (Section 5.2).

An Algorithmic Mitigation Without Redesigning the Analysis We present a simple algorithmic technique inside a worklist-based fixpoint iteration procedure that, without redesigning the abstract semantics part, can effectively relieve the performance degradation caused by spurious interprocedural cycles in both call-strings-based context-sensitive ($k > 0$) and -insensitive ($k = 0$) analysis. For the moment, we consider context-insensitive case only. We extend it to context-sensitive analysis in Section 5.3.

While solving flow equations, the algorithmic technique simply forces procedures to return to their corresponding called site, in order not to follow the last edge (edge (3) $\rightarrow$ (4) in Figure 5.1) of the "butterfly" cycles. In order to enforce this, we control the equation-solving orders so that each called procedure is analyzed exclusively for its one particular call-site. To be safe, we apply our algorithm to only non-recursive procedures.

Consider the equation system in Figure 5.1 again and think of a middle of the analysis (equation-solving) sequence, $\cdots \rightarrow (5) \rightarrow (2) \rightarrow (3)$, which indicates that the analysis of procedure f is invoked from (5) and is now finished. After the evaluation of (3), a classical worklist algorithm inserts all the equations, (4) and (6), that depend on (3). But, if we remember the fact that f has been invoked from (5) and the other call-site (1) has not invoked the procedure until the analysis of f finishes, we can know that continuing with (4) is useless, because the current analysis of f is only related to (5), but not to other calls like (1). So, we process only (6), pruning the spurious sequence $(3) \rightarrow (4) \rightarrow \cdots$.

There are two noticeable things. First, the technique prunes some steps of fixpoint computation and its result may not be a fixpoint. However, because the technique prunes only spurious computations that definitely do not happen in the real executions, it is still a sound approximation of program's concrete semantics. Second, the technique is designed only for the express purpose of relieving performance degradation of spurious cycles, not aiming to eliminate all invalid-paths. It is the

Program	Procedures in the largest cycle	Basic-blocks in the largest cycle
spell-1.0	24/31(77%)	751/782(95%)
gzip-1.2.4a	100/135(74%)	5,988/6,271(95%)
sed-4.0.8	230/294(78%)	14,559/14,976(97%)
tar-1.13	205/222(92%)	10,194/10,800(94%)
wget-1.9	346/434(80%)	15,249/16,544(92%)
bison-1.875	410/832(49%)	12,558/18,110(69%)
proftpd-1.3.1	940/1,096(85%)	35,386/41,062(86%)
apache-2.2.2	1,364/2,075(66%)	71,719/95,179(75%)

Table 5.1: The sizes of the largest strongly-connected components in the "lexical" control flow graphs of real C programs.

multiple return paths, not the multiple calls to a procedure, that make the performance problem. Given a k-limiting context-sensitive analysis, our technique helps the analysis not to return to spurious return paths.

We demonstrate the effectiveness of our technique in a *realistic* setting. We implemented the algorithm inside an industry-strength abstract-interpretation-based C static analyzer [29,31,32,48–51] and tested its performance on open-source benchmarks. We have saved 7%-96% of the analysis time for context-insensitive or -sensitive global analysis.

Contributions This chapter makes the following contributions.

- We present an extension of the approximate call-strings approach, which effectively reduces the inefficiency caused by large, inevitable, spurious interprocedural cycles. We prove the effectiveness of the technique by experiments with an industry-strength C static analyzer [29, 31, 32, 48–51] in globally analyzing medium-scale open-source programs.

- The technique is meaningful in three ways.

  (1) The technique aims to alleviate one major reason (spurious interprocedural cycles) for substantial inefficiency in global static analysis.

(2) It is purely an algorithmic technique inside the worklist-based fixpoint iteration routine. So, it can be directly applicable without changing the analysis' underlying abstract semantics, regardless of whether the semantics is context-sensitive or not. The technique's correctness is obvious enough to avoid the burden of a safety proof that would be needed if we newly designed the abstract semantics.

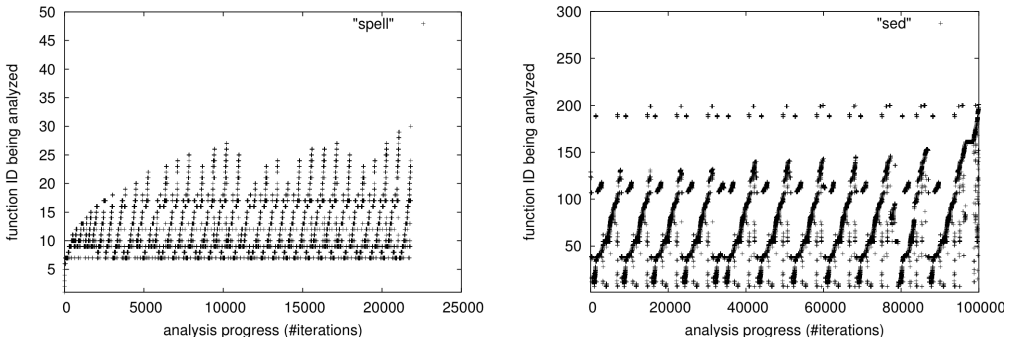
(3) The technique not only reduces the analysis time but also improves the analysis precision. This is because (1) our technique removes some (worklist-level) computations that occur along invalid return paths (Section 5.3.2); (2) when the underlying analysis uses widening, the technique reduces the number of widening points (Section 5.3.2).

- We report one key reason (spurious interprocedural cycles) for why less accurate context-sensitivity actually makes the analyses very slow. Though it is well-known folklore that less precise analysis does not always have less cost [41, 61, 63], there haven't been realistic experiments about their explicit reason.

Organization Section 5.2 discusses the performance problem of the traditional call-strings-based context-sensitive or -insensitive interprocedural analysis. Section 5.3 presents our solution to mitigate the problem. We first describe the approximate call-strings approach and then present our extension of the original method. Section 5.4 presents experimental results that compare the performance of our algorithm with the traditional algorithm. Section 5.5 concludes the paper.

5.2 Performance Problems by Large Spurious Cycles

In this section, we show that large spurious cycles are frequently created during (both context-insensitive and -sensitive) global static analysis, and that they drastically degrade the analysis performance. The approximate call-strings-based context-sensitive abstract semantics cannot effectively eliminate such large spurious cycles.



spell-1.0 (total #procs:31) (2,213 LOC, > 30 repetitions)

sed-4.0.8 (total #procs:294) (26,807 LOC, > 150 repetitions)

Figure 5.3: Global analysis iterates along large spurious interprocedural cycles.

Interprocedural Spurious Cycles Reach Far In Real C Programs If a spurious cycle is created by multiple calls to a procedure f, then all the procedures that are reachable from f or that reach f via the call-graph belong to the cycle because of call and return flows. For example, consider a call-chain $\cdots f_1 \to f_2 \cdots$. If f_1 calls f_2 multiple times, creating a spurious butterfly cycle $f_1 \bowtie f_2$ between them, then fixpoint-solving the cycle involves all the nodes of procedures that reach f_1 or that are reachable from f_2. This situation is common in C programs. For example, in GNU software, the `xmalloc` procedure, which is in charge of memory allocation, is called from many other procedures, and hence generates a butterfly cycle. Then every procedure that reaches `xmalloc` via the call-graph is trapped into a fixpoint cycle.

In conventional context-sensitive analysis that distinguishes the last k call-sites [64], if there are call-chains of length l ($> k$) in programs, it's still possible to have a spurious cycle created during the first $l - k$ calls. This spurious cycle traps the last k procedures into a fixpoint cycle by the above reason.

One spurious cycle in a real C program can trap as many as 80-90% of basic blocks of the program into a fixpoint cycle. Figure 5.3 shows this phenomenon. In the figures, the x-axis represents the execution time of the analysis and the y-axis represents the procedure name, which is mapped to unique integers. During the

analysis, we draw the graph by plotting the point (t, f) if the analysis' worklist algorithm visits a node of procedure f at the time t. For brevity, the graph for sed-4.0.8 is shown only up to 100,000 iterations among more than 3,000,000 total iterations. From the results, we first observe that similar patterns are repeated and each pattern contains almost all procedures in the program. And we find that there are much more repetitions in the case of a large program (sed-4.0.8, 26,807 LOC) than a small one (spell-1.0, 2,213 LOC): more than 150 repeated iterations were required to analyze sed-4.0.8 whereas spell-1.0 needed about 30 repetitions.

5.3 Our Algorithmic Mitigation Technique

In this section, we present our extension of the approximate call-strings-based approach, aiming to mitigate performance problems caused by the large spurious cycles. Our technique is purely algorithmic: the technique does not depend on the underlying abstract semantics but is a simple addition to the existing worklist-based fixpoint algorithm.

We first describe the traditional call-strings-based analysis algorithm (section 5.3.1) as well as the representation of programs (section 2.2). Then we present our algorithmic extension of the classical algorithm (section 5.3.2).

5.3.1 A Normal Call-Strings-Based Analysis Algorithm

Call-strings are sequences of call points. To make them finite, we only consider call-strings of length at most k for some fixed integer $k \geq 0$. We write $\mathit{CallNode}^{\leq k} \stackrel{\text{let}}{=} \Delta$ for the set of call-strings of length $\leq k$, where *CallNode* refers to the set of call points in the program. We write $[c_1, c_2, \cdots, c_i]$ for a call-string of call sequence $c_1, c_2, \cdots, c_i$. Given a call-string δ and a call point c, $[\delta, c]$ denotes a call-string obtained by appending c to δ. Similarly, $[c, \delta]$ denotes a call-string obtained by concatenating $[c]$ and δ. In the case of context-insensitive analysis ($k = 0$), we use $\Delta = \{\epsilon\}$, where the empty call-string ϵ means no context-information.

Figure 5.4.(a) shows the worklist-based fixpoint iteration algorithm that performs call-strings(Δ)-based context-sensitive (or insensitive, when $k = 0$) analysis.

The algorithm computes a table $\hat{X} \in \mathbb{C} \to \hat{\mathbb{S}}$ which associates each node with its input state $\hat{\mathbb{S}} = \Delta \to \hat{\mathbb{M}}$, where $\hat{\mathbb{M}}$ denotes abstract memory, which is a map from abstract locations to abstract values. That is, call-strings are tagged to the abstract memories and are used to distinguish the memories propagated along different interprocedural paths, to a limited extent (the last k call-sites). The worklist W consists of node and call-string pairs. The algorithm chooses a work-item $(c, \delta) \in \mathbb{C} \times \Delta$ from the worklist and evaluates the command $\mathsf{cmd}(c)$ with the flow function $\hat{f}_c$. Next work-items to be inserted into the worklist are defined by function $\mathcal{N} \in \mathbb{C} \times \Delta \to 2^{\mathbb{C} \times \Delta}$:

$$\mathcal{N}(c, \delta) = \begin{cases} \{(\mathsf{rtnof}(c), \delta') \mid \delta = \lceil \delta', c \rceil_k \wedge \delta' \in \mathsf{dom}(\hat{X}(c))\} & \mathsf{cmd}(c) = \mathsf{return}_f \\ \{(c', \lceil \delta, c \rceil_k)) \mid c \hookrightarrow c'\} & \mathsf{cmd}(c) = \mathsf{call}(f_x, e) \\ \{(c', \delta) \mid c \hookrightarrow c'\} & \text{otherwise} \end{cases}$$

where $\mathsf{dom}(f)$ denotes the domain of map f and $\lceil \delta, c \rceil_k$ denotes the call-string $[\delta, c]$ but possibly truncated so as to keep at most the last k call-sites.

The algorithm can follow spurious return paths if the input program's nested call-depth is larger than the k. The mapping δ to $\lceil \delta, c \rceil_k$ is a homomorphism; it is not necessarily one-to-one and $\mathcal{N}$ possibly returns many work-items at an exit node. The following example illustrates this situation.

Example 9 *Let $k = 2$ and suppose call-strings $[c_1, c_3]$ and $[c_2, c_3]$ are tagged to states at call point c. Suppose c calls g under the call-string $[c_1, c_3]$. By the definition of $\mathcal{N}$, the call-string at the entry of g, $entry_g$ is $\lceil c_1, c_3, c \rceil_2 = [c_3, c]$. After the analysis of g, the call-string at the exit is also $[c_3, c]$. When g returns, since the call-string at the exit equals to both $\lceil c_1, c_3, c \rceil_2$ and $\lceil c_2, c_3, c \rceil_2$, $\mathcal{N}$ returns two work-items $(\mathsf{rtnof}(c), [c_1, c_3])$ and $(\mathsf{rtnof}(c), [c_2, c_3])$. The return to $(\mathsf{rtnof}(c), [c_2, c_3])$ is spurious because g was called under the context $[c_1, c_3]$.*

We call the above analysis algorithm $\mathsf{Normal}_k (k = 0, 1, 2, \dots)$. Normal_0 performs context-insensitive analysis, Normal_1 performs context-sensitive analysis that distinguishes the last 1 call-site, and so on.

5.3.2 Our Algorithm

Before discussing our technique, we define the call-context that will be used throughout this section. When a procedure g is called from a call-point c under context δ, we say that (c, δ) is the call-context for that procedure call. Since each call-point c has a unique return-point, we interchangeably write $(\mathsf{rtnof}(c), \delta)$ and (c, δ) for the same call-context.

Our return-site-sensitive (RSS) technique is simple. When calling a procedure at a call-site, the call-context for that call is remembered until the procedure returns. The bookkeeping cost is limited to only one memory entry per procedure. This is possible by the following strategies:

(1) **Single return**: Whenever the analysis of a procedure g is started from a call point c in f under call-string δ, the algorithm remembers its call-context $(\mathsf{rtnof}(c), \delta)$, consisting of the corresponding return-point $\mathsf{rtnof}(c)$ and the call-string δ.[3]. And upon finishing analyzing g's body, after evaluating the exit of g, the algorithm inserts only the remembered return-point and its call-string $(\mathsf{rtnof}(c), \delta)$ into the worklist. Multiple returns are avoided. For correctness, this single return should be allowed only when the other call-points that call g are not analyzed until the analysis of g from (c, δ) completes.

Example 10 *Consider the situation of Example 9 again. When g is called from c under the context $[c_1, c_3]$, our algorithm remembers g's call-context $(\mathsf{rtnof}(c), [c_1, c_3])$. And at the exit of g, under its context $[c_3, c]$, our algorithm inserts only the remembered $(\mathsf{rtnof}(c), [c_1, c_3])$ into the worklist. The spurious return to $(\mathsf{rtnof}(c), [c_2, c_3])$ is avoided.*

(2) **One call per procedure, exclusively**: We implement the single return policy by using one memory entry per procedure to remember the call-context.

[3]In fact, it is possible to remember just (r, δ_1), where δ_1 is the first element, instead of (r, δ), because last $k-1$ elements of the call-context will be also remembered by each work of the callee. Here, we naively save full context to make presentation simple.

This is possible if we can analyze each called procedure exclusively for its one particular call-context. If a procedure is being analyzed from a call-point c with a call-string δ, processing of other call-points that call the same procedure should wait until the analysis of the procedure from (c, δ) is completely finished. This one-exclusive-call-per-procedure policy is enforced by not selecting from the worklist call-points that (directly or transitively) call the procedures that are currently being analyzed.

Example 11 *Suppose procedure g was called from c under the context $[c_1, c_3]$ and our algorithm has remembered the call-context $(\mathsf{rtnof}(c), [c_1, c_3])$. Suppose also the current worklist $W = \{(c, [c_2, c_3]), \cdots\}$ which contains a call-point that invokes g. In this situation, our algorithm does not select $(c, [c_2, c_3])$ as a next work-item unless the analysis of g is completely finished.*

(3) **Recursion handling**: The algorithm gives up the single return policy for recursive procedures. This is because we cannot finish analyzing a recursive procedure's body without considering another call (recursive call) in it. Recursive procedures are handled in the same way as the normal worklist algorithm.

We call our algorithm $\mathsf{Normal}_k/\mathsf{RSS}$. The algorithm does not follow spurious return paths regardless of the program's nested call-depth. While Normal_k starts losing its power when a call chain's length is larger than k, $\mathsf{Normal}_k/\mathsf{RSS}$ does not. The following example shows this difference between Normal_k and $\mathsf{Normal}_k/\mathsf{RSS}$.

Example 12 *Consider a program that has the following call-chain (where $f_1 \overset{c_1,c_2}{\rightarrow} f_2$ denotes that f_1 calls f_2 at call-sites c_1 and c_2) and suppose $k = 1$:*

$$f_1 \overset{c_1,c_2}{\rightarrow} f_2 \overset{c_3,c_4}{\rightarrow} f_3$$

- *Normal_1: The analysis results for f_2 are distinguished by $[c_1]$ and $[c_2]$ hence no butterfly cycle happens between f_1 and f_2. Now, when f_3 is called from f_2 at*

c_3, we have two call-contexts $(c_3, [c_1])$ and $(c_3, [c_2])$ but analyzing f_3 proceeds with context $[c_3]$ (because $k = 1$). That is, Normal_k forgets the call-context for procedure f_3. Thus the result of analyzing f_3 must flow back to all call-contexts with return site c_3, i.e., to both the call-contexts $(c_3, [c_1])$ and $(c_3, [c_2])$.

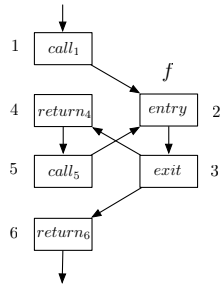
- *$\mathsf{Normal}_1/\mathsf{RSS}$: The results for f_2 and f_3 are distinguished in the same way as Normal_1. But, $\mathsf{Normal}_1/\mathsf{RSS}$ additionally remembers the call-contexts for every procedure call. If f_3 was called from c_3 under context $[c_1]$, our algorithmic technique forces Normal_k to remember the call-context $(c_3, [c_1])$ for that procedure call. And finishing analyzing f_3's body, f_3 returns only to the remembered call-context $(c_3, [c_1])$. This is possible by the one-exclusive-call-per-procedure policy.*

We ensure the one-exclusive-call-per-procedure policy by prioritizing a callee over call-sites that (directly or transitively) invoke the callee. The algorithm always analyzes the control points of the callee g first prior to any other call-points that invoke g: before selecting a work-item as a next job, we exclude from the worklist all the call-point to g if the worklist contains any of procedure h that can be reached from g along some call-chain $g \to \cdots \to h$, including the case of $g = h$. After excluding such call-points, the algorithm chooses a work-item in the same way as a normal worklist algorithm, i.e., after the exclusion, our algorithm relies on the existing worklist ordering strategy in selecting the next work-item.

Figure 5.4(b) shows our algorithmic technique that is applied to the normal worklist algorithm of Figure 5.4(a). To transform Normal_k into $\mathsf{Normal}_k/\mathsf{RSS}$, only shaded lines are inserted; other parts remain the same. $\mathcal{R}$ is a map to record a single return site information (return node and context pair) per procedure. Lines 15-16 are for remembering a single return when encountering a call-site. The algorithm checks if the current control point is a call and its target procedure is non-recursive (the recursive predicate decides whether the procedure is recursive or not), and if so, it remembers its single return-site information for the callee. Lines 17-21 handle procedure returns. If the current point is an exit of a non-recursive procedure, only the remembered return for that procedure is used as a next work-

item, instead of all possible next (successor, context) pairs (line 23). Prioritizing callee over call nodes is implemented by delaying call nodes to procedures now being analyzed. To do this, in line 12-13, the algorithm excludes the call points that invoke non-recursive procedures whose bodies are already contained in the current worklist. $\mathsf{callees}(g, h)$ is true if there is a path in the call graph from g to h.

Example 13 *Analyzing the program in the left-hand side of the figure below proceeds as shown in the right-hand side table. (Assume that $k = 0$, the* choose *function in Figure 5.4 arbitrarily chooses an element from the given worklist, and the initial worklist is $\{1, 4\}$).*



Iters	W	W'	$W \setminus \mathcal{S}$	$\mathcal{R}$	*updated* W
1	$\{1, 4\}$	$\{\}$	$\{\bar{1}\}$	$\{f \mapsto 4\}$	$\{2, 4\}$
2	$\{2, 4\}$	$\{\}$	$\{2, \bar{4}\}$	$\{f \mapsto 4\}$	$\{2, 5\}$
3	$\{2, 5\}$	$\{5\}$	$\{\bar{2}\}$	$\{f \mapsto 4\}$	$\{3, 5\}$
4	$\{3, 5\}$	$\{5\}$	$\{\bar{3}\}$	$\{f \mapsto 4\}$	$\{4, 5\}$

For each iteration of the algorithm, the table shows the contents of the current worklist (W), call-points that are excluded at this iteration (W'), return site information ($\mathcal{R}$), and the updated worklist (W). $\bar{n}$ represents the chosen work-item for each iteration. When the algorithm processes call-point 1 *at the first iteration, f remembers its corresponding return-point* 4*. At the 3rd and 4th iterations, node* 5 *was excluded, because it is another call to f and the worklist contains the bodies of f at both iterations. At the exit of f (when processing node* 3 *at the 4th iteration), only $\mathcal{R}(f) = 4$ is inserted into the worklist instead of* succof$(f) = \{4, 6\}$.

Correctness & Precision

One noticeable thing of $\mathsf{Normal}_k/\mathsf{RSS}$ is that the result is not a fixpoint of the given flow equation system, but still a sound approximation of the program semantics. Since the algorithm prunes some computation steps during worklist algorithm (at exit nodes of non-recursive procedures), the result of the algorithm may not be a fixpoint of the original equation system. However, because the algorithm prunes

```
(01) : δ ∈ Context = Δ
(02) : w ∈ Work = ℂ × Δ
(03) : W ∈ Worklist = 2^Work
(04) : 𝒩 ∈ ℂ × Δ → 2^(ℂ×Δ)
(05) : Ŝ = Δ → M̂
(06) : X̂ ∈ ℂ → Ŝ
(07) : f̂_c ∈ M̂ → M̂

(09) : FixpointIterate (W, X) =

(11) : repeat

(13) :       (n, δ) := cmd(W)
(14) :       m := f̂_c(X̂(c)(δ))

(23) :       for all (c', δ') ∈ 𝒩(c, δ) do
(24) :           if m ⋢ X̂(c')(δ')
(25) :               W := W ∪ {(c', δ')}
(26) :               X̂(c')(δ') := X(c')(δ') ⊔ m
(27) : until W = ∅
```

(a) a normal worklist algorithm Normal_k

```
(01) : δ ∈ Context = Δ
(02) : w ∈ Work = ℂ × Δ
(03) : W ∈ Worklist = 2^Work
(04) : 𝒩 ∈ ℂ × Δ → 2^(ℂ×Δ)
(05) : Ŝ = Δ → M̂
(06) : X̂ ∈ ℂ → Ŝ
(07) : f̂_c ∈ M̂ → M̂
(08) : ℛ ∈ ProcName → Work

(09) : FixpointIterate (W, X) =
(10) :     ℛ := ∅
(11) : repeat
(12) :     W' := {(c, _) ∈ W | cmd(c) = call(g_x, e)∧
           ∃(c', _) ∈ W.callees(g, procof(c')) ∧ ¬recursive(g) }
(13) :     (n, δ) := cmd( W \ W' )
(14) :     m := f̂_C(X̂(c)(δ))
(15) :     if cmd(c) = call(g_x, e) ∧ ¬recursive(g) then
(16) :         ℛ(g) := (rtnof(c), δ)
(17) :     if cmd(c) = return_g ∧ ¬recursive(g) then
(18) :         (rtnof(c), δ') := ℛ(g)
(19) :         if m ⋢ X(rtnof(c))(δ')
(20) :             W := W ∪ {(rtnof(c), δ')}
(21) :             X̂(rtnof(c))(δ') := X̂(rtnof(c))(δ') ⊔ m
(22) :     else
(23) :         for all (c', δ') ∈ 𝒩(c, δ) do
(24) :             if m ⋢ X̂(c')(δ')
(25) :                 W := W ∪ {(c', δ')}
(26) :                 X̂(c')(δ') := X̂(c')(δ') ⊔ m
(27) : until W = ∅
```

(b) our algorithm Normal_k/RSS

Figure 5.4: A normal context-sensitive worklist algorithm and its RSS modification that avoids spurious interprocedural paths.

only spurious returns that definitely do not happen in the real executions of the program, our algorithm does not miss any information flow of real executions. In other words, our algorithm does not necessarily produce a maximal fixpoint solution but something below it and still above the real semantics.

For any f and any arbitrary call-context (c, δ), the single return to (r, δ) after analyzing f is correct if the state from (c, δ) is implied by the input state used in the analysis of f and its result is guaranteed to be returned to (r, δ). The state from every call-context flows into f (abstract semantics). Our single-return policy does not miss returning f's analysis result to its corresponding call-context[4] because (1) we remember the context at each call; (2) for every different call, modulo the underlying context-sensitivity, we exclusively analyze f. Because we cannot enforce this exclusivity for recursive calls, we do not apply the algorithm to recursive procedures.

$\mathsf{Normal}_k/\mathsf{RSS}$ is always at least as precise as Normal_k. Because $\mathsf{Normal}_k/\mathsf{RSS}$ prunes some (worklist-level) computations that occur along invalid return paths, it is likely to have an effect of avoiding propagations of information along invalid return paths. Hence, $\mathsf{Normal}_k/\mathsf{RSS}$ gives more precise (or at least the same) results than Normal_k. The actual precision of $\mathsf{Normal}_k/\mathsf{RSS}$ varies depending on the existing worklist order of Normal_k.

Example 14 *Consider the program in Example 13 again, and suppose the current worklist is* $\{1, 5\}$. *When analyzing the program with* Normal_0, *the fixpoint-solving follows both spurious return paths, regardless of the worklist order,*

$$1 \to 2 \to 3 \to 6 \tag{5.1}$$

$$5 \to 2 \to 3 \to 4 \tag{5.2}$$

because of multiple returns from node 3. When analyzing with $\mathsf{Normal}_0/\mathsf{RSS}$, *there are two possibilities, depending on the worklist order:*

[4]Here, we ignore the cases where the callee never returns (e.g., it calls exit()). However, even though that happens, we can enforce the return of callee by always inserting the exit node of a procedure when inserting the entry node of the procedure into the worklist.

(1) When $\mathsf{Normal}_0/\mathit{RSS}$ *selects node 1 first: Then the fixpoint iteration sequence may be* $1;2;3;4;5;2;3;6$*. This sequence involves the spurious path (1) (because the second visit to node 2 uses the information from node 1 as well as from node 5), but not (2).* $\mathsf{Normal}_0/\mathit{RSS}$ *is more precise than* Normal_0*.*

(2) When $\mathsf{Normal}_0/\mathit{RSS}$ *selects node 5 first: Then the fixpoint iteration sequence may be* $5;2;3;6;1;2;3;4;5;2;3;6$*. This computation involves both spurious paths (1) and (2). With this iteration order,* Normal_0 *and* $\mathsf{Normal}_0/\mathit{RSS}$ *have the same precision.*

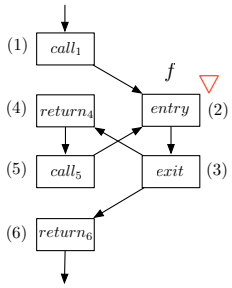
Less Widening Points

Widening [16] is a speed-up technique that have been designed to safely approximate least fixpoints of semantic function. In abstract-interpretation-based static analysis, programs invariants are characterized as least fixpoints of (abstract) semantic functions over abstract domain. For finite height domains, the fixpoints are computed by using a classical iterative algorithm. But the iterative algorithm does not terminate or has unacceptable costs for domains with infinite height or very large height. For infinite or very large height domains such as lattice of intervals, the widening technique [16] is used to guarantee or accelerate the analysis' termination. With widening, the iterative algorithm does not necessarily compute least fixpoints but finds a safe (upper) approximation of the least fixpoint.

Our technique reduces cycles, hence obviously reduces the number of widening points. Because applying widening means losing analysis precision, the widening operation should be carefully applied to as small as possible subset of the entire program points. A common way of selecting such widening points is to apply widening to every heads of loops in program [6], including ones that are interprocedurally created by calling a procedure multiple times. $\mathsf{Normal}_k/\mathsf{RSS}$ can reduce the number of widening points more. $\mathsf{Normal}_k/\mathsf{RSS}$ need not apply widenings at interprocedural loop-heads that are created by non-recursive procedure calls. This is because $\mathsf{Normal}_k/\mathsf{RSS}$ does not follow such interprocedural cycles.

Example 15 *Consider the following code and interval-domain-based analysis of the code.*

```
int g = 0;
int f() { g++; }
int main() {
    f();
    f();
}
```



Since procedure `f` *is called twice from procedure* `main`*, a spurious interprocedural cycle* $(5) \to (2) \to (3) \to (4) \to (5) \cdots$ *will be created during the analysis. Iterating through the cycle continually increases the value of the global variable* `g`*:* $[0,0] \to [0,1] \to [0,2] \to \cdots$*. In order to terminate the analysis, a widening should be applied at the entry of procedure* `f`*. Hence,* Normal_k *computes* `g` $= [0, +\infty]$ *at the end of procedure* `main`*. However,* $\mathsf{Normal}_k/\mathsf{RSS}$ *does not apply the widening at the entry of procedure* `f` *(since* `f` *is non-recursive and* $\mathsf{Normal}_k/\mathsf{RSS}$ *does not follow the spurious return paths* $(5) \to (2) \to (3) \to (4)$*), computing* `g` $= [0,2]$ *at the end of procedure* `main`*.*

5.3.3 A Simple Strategy for Implementation

In practice, if the worklist algorithm uses a particular worklist ordering strategy, the RSS algorithm can be implemented more easily.

Assume that the worklist algorithm uses a partial order RevTop between control points in the program and retrieves the point with the highest order from the worklist. The order RevTop between nodes is defined as a reverse topological order between procedures on the call graph: a control point c of a procedure f precedes a control point c' of a procedure g if f precedes g in the reverse topological order in the call graph. If f and g are the same procedure, the order between the control points is defined by the weak topological order [6] on the control flow graph of that procedure. Note that there can be two or more control points that have the

highest order, for example of each branch of conditional statements. In this case, the algorithm arbitrarily chooses one among them.

Without recursive procedures, the order RevTop guarantees the one-exclusive-call-per-procedure policy. This is because the order means that a callee is always analyzed first rather than its caller. For instance, think of two procedures f and g where f precedes g in reverse topological order on the call graph. It means that f is called by some call sites in g. Then the worklist algorithm selects a control point of f first from the worklist rather than ones in g unless the worklist does not contain anyone of f, which means that all the other calls to f inside g wait until the analysis of f is completely finished. Recursive procedures are handled in the same way as the normal worklist algorithm.

To implement the technique inside the algorithm Figure 5.4.(b), lines 12 is removed and line 13 is replaced by the following:

$$(c, \delta) := \mathsf{choose}_{\mathsf{RevTop}}(W)$$

where $\mathsf{choose}_{\mathsf{RevTop}}$ chooses the work-item that has the highest RevTop order from the worklist (W).

5.4 Experiments

We implemented our algorithm inside a realistic C analyzer [29, 31, 32, 48–51]. Experiments with open-source programs show that $\mathsf{Normal}_k/\mathsf{RSS}$ for any k is very likely faster than Normal_k, and that even $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ can be faster than Normal_k.

5.4.1 Setting

Normal_k is our underlying worklist algorithm, on top of which our industry-strength static analyzer [29, 31, 32, 48–51] for C is installed. The analyzer is an interval-domain-based abstract interpreter. The analyzer performs by default flow-sensitive and call-strings-based context-sensitive global analysis on the supergraph of the

input program: it computes $\hat{X} = \mathbb{C} \to \hat{\mathbb{S}}$ where $\hat{\mathbb{S}} = \Delta \to \hat{\mathbb{M}}$. $\hat{\mathbb{M}}$ denotes abstract memory $\hat{\mathbb{M}} = \hat{\mathbb{L}} \to \hat{\mathbb{V}}$, where $\hat{\mathbb{L}}$ and $\hat{\mathbb{V}}$ are defined in Chapter 2.

We evaluated our algorithm in two ways. First, we measured the net effects of avoiding spurious interprocedural cycles. Since our algorithmic technique changes the existing worklist order, performance differences between Normal_k and $\mathsf{Normal}_k/\mathsf{RSS}$ could be attributed not only to avoiding spurious cycles but also to the changed worklist order. In order to measure the net effects of avoiding spurious cycles, we applied the same worklist order RevTop, defined in Section 5.3.3, to both Normal_k and $\mathsf{Normal}_k/\mathsf{RSS}$. Note that this ordering itself contains the "prioritize callees over call-sites" feature and we don't explicitly need the delaying call technique (lines 12-13 in Figure 5.4.(b)) in $\mathsf{Normal}_k/\mathsf{RSS}$. Hence the worklist order for Normal_k and $\mathsf{Normal}_k/\mathsf{RSS}$ are the same.[5] For this evaluation, we compare analysis time and precision between Normal_k and $\mathsf{Normal}_k/\mathsf{RSS}$.

We also evaluated our algorithm when our technique interferes with the existing worklist order. Because our technique interferes with (i.e., changes) the existing worklist order of Normal_k, it is necessary to check whether our technique works well regardless of the existing worklist order strategies or not. To see what happens in this case, we applied our technique to Normal_k that uses the following worklist order, called Arbitrary; the order between nodes in different procedures is determined by a random order that is fixed before the analysis and the order between nodes in the same procedure is defined by the weak topological order. Note that the worklist order does not contain the "prioritize callees over call-sites" because the order randomly chooses a procedure regardless of call relationship.

We have analyzed 11 open-source and SPEC2000 software packages. Table 5.2 shows our benchmark programs. Lines of code (**LOC**) are given before preprocessing. The number of nodes in the supergraph(#**nodes**) is given after preprocessing. **k** denotes the size of call-strings used for the analysis. Entries with ∞ means missing data because of our analysis running out of memory. All experiments were

[5] In fact, the order described here is the one our analyzer uses by default, which consistently shows better performance than naive worklist management scheme (BFS/DFS) or simple "wait-at-join" techniques (e.g., [31]).

Program	**LOC**	**#nodes**	**k-call-strings**	***#iters***		***time***	
				Normal	Normal/RSS	Normal	Normal/RSS
spell-1.0	2,213	782	0	33,864	5,800	60.98	8.49
			1	31,933	10,109	55.02	13.35
			2	57,083	15,226	102.28	19.04
barcode-0.96	4,460	2,634	0	22,040	19,556	93.22	84.44
			1	33,808	30,311	144.37	134.57
			2	40,176	36,058	183.49	169.08
httptunnel-3.3	6,174	2,757	0	442,159	48,292	2020.10	191.53
			1	267,291	116,666	1525.26	502.59
			2	609,623	251,575	5983.27	1234.75
gzip-1.2.4a	7,327	6,271	0	653,063	88,359	4601.23	621.52
			1	991,135	165,892	10281.94	1217.58
			2	1,174,632	150,391	18263.58	1116.25
jwhois-3.0.1	9,344	5,147	0	417,529	134,389	4284.21	1273.49
			1	272,377	138,077	2445.56	1222.07
			2	594,090	180,080	8448.36	1631.07
parser	10,900	9,298	0	3,452,248	230,309	61316.91	3270.40
			1	∞	∞	∞	∞
bc-1.06	13,093	4,924	0	1,964,396	412,549	23515.27	3644.13
			1	3,038,986	1,477,120	44859.16	12557.88
			2	∞	∞	∞	∞
less-290	18,449	7,754	0	3,149,284	1,420,432	46274.67	20196.69
			1	∞	∞	∞	∞
twolf	19,700	14,610	0	3,028,814	139,082	33293.96	1395.32
			1	∞	∞	∞	∞
tar-1.13	20,258	10,800	0	4,748,749	700,474	75013.88	9973.40
			1	∞	∞	∞	∞
make-3.76.1	27,304	11,061	0	4,613,382	2,511,582	88221.06	44853.49
			1	∞	∞	∞	∞

Table 5.2: Benchmark programs and their raw analysis results when using RevTop worklist order.

done on a Linux 2.6 system running on a Pentium4 3.2 GHz box with 4 GB of main memory. `parser` and `twolf` are from SPEC2000 benchmarks and the others are open-source software.

We use two performance measures: (1) *#iters* is the total number of iterations during the worklist algorithm. The number directly indicates the amount of computation; (2) *time* is the CPU time spent during the analysis. Though *time* is roughly proportional to *#iters*, it is subject to change because of different implementations and test environments.

5.4.2 The Net Effects of Avoiding Spurious Cycles

Reduced Analysis Time Figure 5.5.(a) compares *#iters* between $\mathsf{Normal}_k/\mathsf{RSS}$ and Normal_k for $k = 0, 1, 2$ using RevTop worklist order, which shows the net effects of avoiding spurious cycles. In this comparison, $\mathsf{Normal}_k/\mathsf{RSS}$ reduces the number of iterations of Normal_k by on average 72%.

When $k = 0$ (context-insensitive), $\mathsf{Normal}_0/\mathsf{RSS}$ has reduced *#iters* by, on average, about 72% against Normal_0. For most programs, the analysis time has been reduced by more than 50%. There is one exception: `barcode`. The amount of computation has been reduced by 11%. This is because `barcode` has unusual call structures: it does not call a procedure many times, but calls many different procedures one by one. So, the program contains few butterfly cycles.

When $k = 1$, $\mathsf{Normal}_1/\mathsf{RSS}$ has reduced *#iters* by, on average, about 53% against Normal_1. Compared to the context-insensitive case ($k = 0$), for all programs, cost reduction ratios have been slightly decreased. As an example, for `spell`, the reduction ratio when $k = 0$ is 83% and the ratio when $k = 1$ is 68%. This is mainly because, in our analysis, Normal_0 costs more than Normal_1 for most programs (`spell`, `httptunnel`, `jwhois`). For `httptunnel`, in Table 5.2, the analysis time (2020.10 s) for $k = 1$ is less than the time (1525.26 s) for $k = 0$. This means that performance problems by butterfly cycles is much more severe when $k = 0$ than that of $k = 1$, because by increasing context-sensitivity some spurious paths

Program	**Analysis**	*#const*	*#finite*	*#open*	*#top*
spell-1.0	Normal_0	345	88	33	143
	$\mathsf{Normal}_0/\mathsf{RSS}$	345	89	35	140
barcode-0.96	Normal_0	2136	588	240	527
	$\mathsf{Normal}_0/\mathsf{RSS}$	2136	589	240	526
httptunnel-3.3	Normal_0	1337	342	120	481
	$\mathsf{Normal}_0/\mathsf{RSS}$	1345	342	120	473
gzip-1.2.4a	Normal_0	1995	714	255	1214
	$\mathsf{Normal}_0/\mathsf{RSS}$	1995	716	255	1212
jwhois-3.0.1	Normal_0	2740	415	961	1036
	$\mathsf{Normal}_0/\mathsf{RSS}$	2740	415	961	1036

Table 5.3: Comparison of precision between Normal_0 and $\mathsf{Normal}_0/\mathsf{RSS}$.

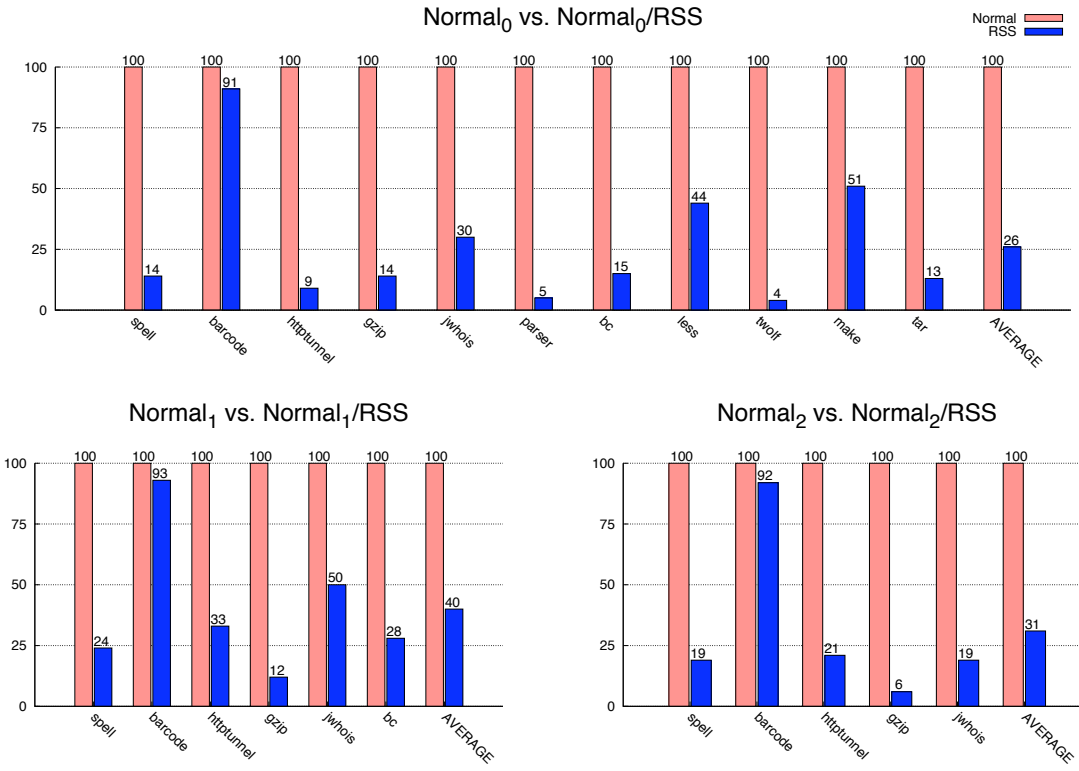
can be removed. However, by using our algorithm, we can still reduce the cost of Normal_1 by 53%.

When $k = 2$, $\mathsf{Normal}_2/\mathsf{RSS}$ has reduced *#iters* by, on average, 60% against Normal_2. Compared to the case of $k = 1$, the cost reduction ratio has been slightly increased for most programs. For example, the ratio for `spell` has changed from 68% to 73%. In the analysis of Normal_2, since the equation system is much larger than that of Normal_1, our conjecture is that the size of butterfly cycles is likely to get larger. Since larger butterfly cycles causes more serious problems (Section 5.2), our RSS algorithm is likely to greater reduce useless computation.

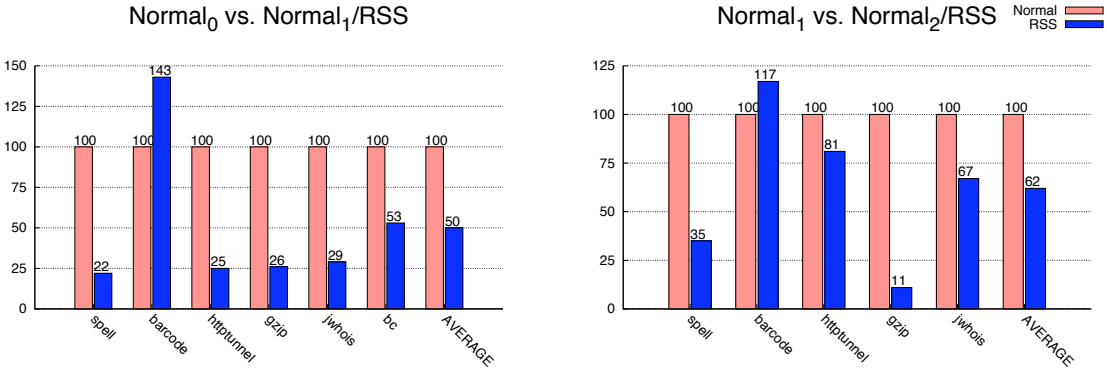
Figure 5.5.(b) compares the performance of $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ against Normal_k for $k = 0, 1$. The result shows that, for all programs except `barcode`, even $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ is faster than Normal_k. Since $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ can be even faster than Normal_k, if memory cost permits, we can consider using $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ instead of Normal_k.

Increased Analysis Precision Table 5.3 compares the precision between Normal_0 and $\mathsf{Normal}_0/\mathsf{RSS}$.[6] In order to measure the increased precision, we first joined all

[6]We compared the precision for the case of $k = 0$ and for the first five programs in Table 5.2 because we need more memory to do the precision comparison (we should keep two analysis results of Normal_0 and $\mathsf{Normal}_0/\mathsf{RSS}$ at the same time).

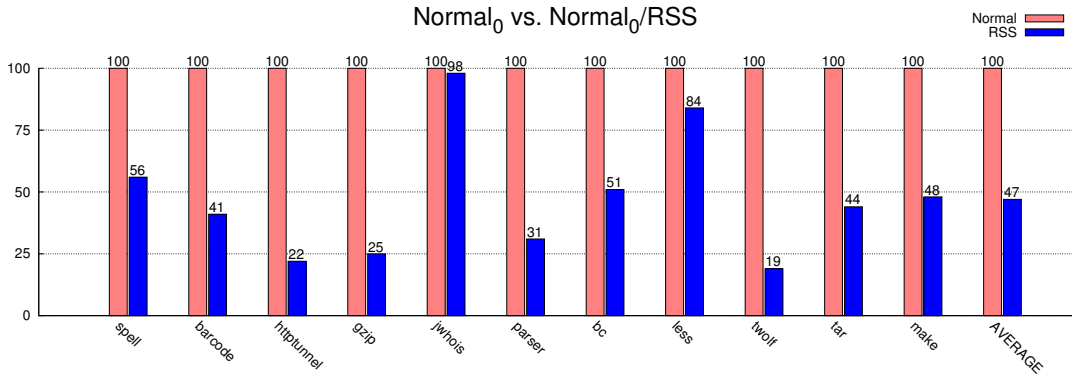


(a) Comparison of *#iters* between Normal_k and $\mathsf{Normal}_k/\mathsf{RSS}$, for $k = 0, 1, 2$.



(b) Comparison of *#iters* between Normal_k and $\mathsf{Normal}_{k+1}/\mathsf{RSS}$, for $k = 0, 1$.

Figure 5.5: Net effects of avoiding spurious cycles



(a) Comparison of of *#iters* between Normal$_0$ and Normal$_0$/RSS, for $k = 0$.

Program	***#iters***		***time***	
	Normal	Normal/RSS	Normal	Normal/RSS
spell-1.0	36,272	20,377	99.19	43.66
barcode-0.96	71,342	29,574	534.9	154.36
httptunnel-3.3	591,030	132,668	4132.21	730.95
gzip-1.2.4a	804,240	204,553	6844.31	1299.36
jwhois-3.0.1	777,867	761,117	5518.04	4664.2
parser	3,500,035	1,095,194	70248.32	24249.95
bc-1.06	2,231,064	1,138,847	23136.25	14240.14
less-290	3,118,068	2,613,384	53152.72	66329.59
twolf	3,347,610	645,922	52372.78	7179.93
tar-1.13	5,310,745	2,334,886	92637.58	78013.96
make-3.76.1	4,415,305	2,110,272	70553.14	43381.18

(b) Benchmark programs and their raw analysis results.

Figure 5.6: The analysis results when using Arbitrary worklist order.

the memories associated with each program point ($\mathbb{C}$). Then we counted the number of constant intervals (*#const*, e.g., $[1,1]$), finite intervals (*#finite*, e.g., $[1,5]$), intervals with one infinity (*#open*, e.g., $[-1,+\infty)$ or $(-\infty,1]$), and intervals with two infinity (*#top*, $(-\infty,+\infty)$) from interval values ($\hat{\mathbb{Z}}$) and array blocks ($2^{AllocSite\times\hat{\mathbb{Z}}\times\hat{\mathbb{Z}}}$) contained in the joined memory. The constant interval and top interval indicate the most precise and imprecise values, respectively. The results show that $\mathsf{Normal}_0/\mathsf{RSS}$ is more precise (`spell`, `barcode`, `httptunnel`, `gzip`) than Normal_0 or the precision is the same (`jwhois`).

5.4.3 Speed Up When Interfering the Existing Worklist Order

Figure 5.6.(a) compares *#iters* between Normal_k and $\mathsf{Normal}_k/\mathsf{RSS}$ for $k=0$ using $\mathsf{Arbitrary}$ worklist order. In the comparison, $\mathsf{Normal}_k/\mathsf{RSS}$ reduces the computation cost of Normal_k by on average 53%. From this results, we can find that the interference does not significantly affect the overall performance differences: the reduction ratio has been decreased by 19% from the case of net effects of avoiding spurious cycles (72%). Hence, the technique is likely to relieve the problems of spurious cycles regardless of the existing worklist ordering strategies.

5.5 Implementation Guideline for Global Analysis

Our technique suggests the following implementation guideline in tuning a global semantic analysis. Suppose we develop an analyzer that uses call-strings of size k for context-sensitivity with the Normal_k algorithm. Suppose further that we cannot increase the call-strings size more than k because of either the time or memory cost. In this situation, our algorithmic technique has the following usages.

- When we cannot increase the call-strings size more than k because of the memory cost: then use $\mathsf{Normal}_k/\mathsf{RSS}$ instead of Normal_k. This is because (1) $\mathsf{Normal}_k/\mathsf{RSS}$ is empirically faster than Normal_k (Section 5.4.1 and Figure 5.5.(a),5.6); (2) $\mathsf{Normal}_k/\mathsf{RSS}$ is in principle more accurate or at least does not sacrifice the precision of Normal_k (Section 5.3.2, 5.3.2 and Table 5.3); (3)

$\mathsf{Normal}_k/\mathsf{RSS}$ requires in extra just as many memory entities as the number of procedures.

- When we cannot increase the call-strings size more than k because of the time cost: then, if memory permits, consider using $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ instead. This is because (1) $\mathsf{Normal}_{k+1}/\mathsf{RSS}$ can be even faster than Normal_k (Section 5.4.1 and Figure 5.5.(b)); (2) it requires in extra just as many entities as the number of procedures.

Though tuning the accuracy of static analysis can in principle be controlled solely by redesigning the underlying abstract semantics, our algorithmic technique is a simple and orthogonal leverage to effectively shift the analysis cost/accuracy balance for the better. The technique's correctness is obvious enough to avoid the burden of safety proof of otherwise a newly designed abstract semantics.

Chapter 6

Related Work

6.1 Scalable Global Static Analyzers

The global static analyzer we report in this dissertation achieves higher scalability (up to 1 MLOC) than previous general-purpose global analyzers. Zitser et al. [74] report that PolySpace C Verifier [42], a commercial tool for detection of runtime errors, cannot analyze programs with moderate sizes (>100 KLOC) because of scalability problem. Previous versions of Airac, a general-purpose interval domain-based global static analyzer, scale only to 30 KLOC in global analysis [31,48]. Recently, a significant progress has been reported [49], but it still does not scale over 120 KLOC. Other similar (interval domain-based) analyzers are also not scalable to large code bases [2,3]. Nevertheless, there have been scalable domain-specific static analyzers, like Astrée [5,14] and CGS [67], which scale to hundreds of thousands lines of code. However, Astrée targets on programs that do not have recursion and backward gotos, which enables a very efficient interpretation-based analysis [14], and CGS is not fully flow-sensitive [67]. There are other summary-based approaches [20,21] for scalable global analysis, which are independent of our abstract interpretation-based approach.

6.2 Spatial Localization

In static program analysis, localization is a well-known idea for reducing analysis cost, however, research has been mainly focused on reachability-based approach [11, 22, 28, 39, 44, 59, 60, 71, 72]. For example, in shape analysis, reachability-based localization has been used to improve the scalability of interprocedural analysis [11, 22, 39, 59, 60, 71, 72]. Rinetzky et al. [59, 60] define a shape analysis in which called procedures are only passed with reachable parts of the heap. Marron et al. [39] reformulate the idea of [59] for graph-based heap models. In separation-logic-based program verification (both by-hand and automatic checking [4]), one typically reasons about a command with respect to its footprint (memory cells that the command accesses) in isolation. However, (even) in separation-logic-based program analysis, the framing, which is expressed in accessibility in logic, is conventionally implemented based on reachability [22, 71, 72]: Gotsman et al. [22] and Yang et al. [71, 72] split states based on reachability. Similar reachability-based techniques are also popular in higher-order flow analyses [26, 28, 44]. Jagannathan et al. [28] use "an abstract form of garbage collection" that removes unreachable bindings. Might et al. [44] formalize the abstract-garbage-collecting control-flow analysis and show that removing unreachable cells significantly improves the analysis performance. However, reachability is, in fact, just one of crude but safe approximation method for estimating "live" abstract resources. In this paper, we present a new non-reachability-based approach, access-based localization.

Chen et al. [9] use a mixture of reachability- and access-based localization, but, their approach is more restricted than ours. During reachability-based localization, they try to infer accessed locations by evaluating expressions two times. However, because input states are not at a fixpoint during the course of the analysis, the accessed locations cannot be completely determined. By contrast our approach makes access-based localization always possible.

Might et al. [43] observes that reachability is overly conservative, and presents a refined localization technique that is orthogonal to our method. From the reachability-based localized state, they additionally exclude some resources that are governed

by unsatisfiable conditions. The resulting localized state may contain non-accessed resources that are not governed by such conditions, which could be filtered by our technique. And, since our technique does not consider unsatisfiable conditions, their technique can improve ours.

It is also a well-known idea to scale an analysis by using an efficient pre-analysis. For example, flow-insensitive pre-analysis has been used in dataflow analysis [1] and pointer analysis [70]. Our work is an instance of these lines of research: we use a pre-analysis to localize memory states in actual analysis. In addition, we show some properties that the pre-analysis must satisfy, which is necessary for the safety of the localization.

6.3 Temporal Localization

Scalable sparse analysis techniques have been mainly reported in the pointer analysis literature. As examples, Hardekopf et al. [24] presented the semi-sparse pointer analysis algorithm that scales to large code bases (up to 474 KLOC) for the first time, and after that, flow-sensitive pointer analysis becomes scalable even to millions of lines of code via sparse analysis technique [25, 37]. However, the sparse techniques are not general and tightly coupled with particular poiter analysis algorithms.

Sparse analysis techniques are first pioneered for optimizing dataflow analysis, which is a simpler setting than the one postulated in our framework. They are mostly in an algorithmic form, thus not applicable to general static analysis problems. Reif and Lewis [55] developed a sparse analysis algorithm for constant propagation and Wegman et al. [68] extended it to conditional constant propagation. Dhamdhere et al. [19] showed how to sparse partial redundancy elimination. These algorithms are fully sparse in that precise def-use chains are syntactically identifiable and values are always propagated along to def-use chains (in an SSA form). Sparse evaluation technique [10, 18, 27, 54], which exploits only a limited form of sparsity, is to remove statements that are irrelevant to the analysis from control flow graphs. For example, we can remove the statements for numerical computa-

tion when we do typical pointer analysis. Sparse evaluation technique is not effective when the underlying analysis does not have many irrelevant statements, which is the case of static analysis that considers the full semantics of programs.

Our sparse analysis framework allows us to design a correct sparse static analysis that exploits more sparsity than localization techniques [44, 49, 59, 71]. Localization is a static analysis technique that exploits a limited form of sparsity; when analyzing code blocks such as procedure bodies, localization attempts to remove irrelevant parts of abstract states that will not be used during the analysis. Still, localization cannot avoid unnecessary propagation of abstract values along to control flow. Localization has been widely used for cost reduction in many semantics-based static analysis, such as shape analysis [59,71], higher-order flow analysis [44], and numeric abstract interpretation [49].

6.4 Contextual Localization

We compare, on the basis of their applicability to general semantic-based static analyzers[1], our method with other approaches that eliminate invalid paths.

The approximate call-strings approach [64] is popular in practice but its precision is not enough to mitigate large spurious cycles. Sharir and Pnueli [64] presented an approximate call-strings approach in which the last k call-sites are remembered for the calling contexts to each procedure. The k length suffix method is an approximation of the full call-strings approach [33, 35, 64] and has been used as a feasible alternative in practice [3, 40, 41]. Moreover, it is actually one of very few options available for semantic-based global static analysis that uses infinite domains and non-distributive flow functions (e.g., [3,31]). However, the k length suffix method induces a large spurious cycle because it permits multiple returns of procedures. Our algorithm is an extension of the k length suffix method and adds extra

[1] For example, such analyzers include octagon-based analyzers (e.g., [5]), interval-based analyzers (e.g., [29, 31, 32, 48–51]), value set analysis [3], and program analyzer generators (e.g, [40]), which usually use infinite (height) domains and non-distributive flow functions.

precision that relieves the performance problem from spurious interprocedural cycles.

Another approximate call-strings method that uses full context-sensitivity for non-recursive procedures has been shown to be practical for points-to analysis [65, 69] but, the method is too costly for more general semantic-based analyses. The method is approximate because it does not distinguish the calling contexts for recursive calls. Whaley and Lam [69] used BDDs to efficiently encode the calling contexts and showed that full context-sensitivity is feasible for non-recursive procedures. Their analysis is fully context-sensitive for non-recursive procedures and does not suffer from large spurious cycles caused by non-recursive procedures. Sridharan and Bodík [65] presented an approximation, called regular-reachability, of the CFL(context-free language)-reachability [57]. They transform the analysis problem into graph reachability problem [56] and only consider execution paths where calls and returns are properly matched for programs without recursive procedures. Since the set of calling contexts that they consider is finite (because they do not consider recursion), the set of calling contexts can be described by a regular language instead of context-free languages. Though these approaches are more precise than k length suffix method, it is unknown whether the BDD-based method [69] or regular-reachability [65] are also applicable in practice to general semantic-based analyzers rather than pointer analysis. Our algorithm can be useful for analyses for which these approaches hit a cost limit in practice and k length suffix method should be used instead.

Full call-strings approaches [33, 35, 64] and functional approaches [64] do not suffer from spurious cycles but are limited to restricted classes of data flow analysis problems. The original full call-strings method [64] prescribes the domain to be finite and its improved algorithms [33,35] are also limited to bit-vector problems or finite domains. For infinite domains, these algorithms can possibly generate infinite number of call-strings and hence may not terminate. Khedker et al.'s algorithm [35] supports infinite domains only after unfolding cyclic call chains by a fixed number. A functional approach [64] builds the summary flow functions for each procedure

in a context independent way and these functions are used as flow functions of call statements. Because using the summary functions does not require traversing the called procedure's bodies, functional approaches also do not suffer from spurious cycles problem. However, computing summary flow functions requires efficient representation of function compositions and meets and hence is applicable to only a restricted data flow analysis problems.

Reps et al.'s algorithms [57,62] to avoid unrealizable paths are limited to analysis problems that can be expressed only in their graph reachability framework. These algorithms are variants of the iterative functional approach [64] that require the flow functions to be distributive. So, their algorithm cannot handle prevalent yet non-distributive analyses. For example, our analyzer that uses the interval domain [16] with non-distributive flow functions does not fall into either their IFDS [57] or IDE [62] problems. Meanwhile, our algorithm is independent of the underlying abstract semantic functions. The regular-reachability [65], which is a restricted version of Reps et al.'s algorithm [57], also requires the analysis problem to be expressed in graph reachability problem.

Chambers et al.'s technique [7] is similar to ours but entails a relatively large change to an existing worklist order. Their technique analyzes each procedure intraprocedurally, and at call-sites continues the analysis of the callee. It returns to analyze the nodes of the caller only after finishing the analysis of the callee. Our worklist prioritizes the callee only over the call nodes that invoke the callee, not the entire caller, which is a relatively smaller change than Chambers et al.'s. In addition, they assume worst case results for recursive calls, but we do not degrade the analysis precision for recursive calls.

The idea of remembering immediate calling context is first proposed by Myers [47] and we extend it to call-strings method. By remembering the immediate calling context only, Myers' algorithm is context-sensitive for bit-vector frameworks [34]. Unfortunately, Myers' formulation is applicable only to bit-vector problems and hard to extend it to general call-strings-based analysis. This paper can

be understood as an extension of Myers' algorithm for general call-strings-based static analysis.

Chapter 7

Conclusion

We showed that practical scalability of global analysis can be achieved by localization techniques. We applied our techniques to an industrial-strength global static analyzer, Airac, and improved its performance by more than hundreds times. When Airac was initially developed [29,31] for practical use, it could be globally used only for small (less than 10 KLOC) programs. Thus, Airac was almost always used for separate (file-by-file) analysis. However, because sound separate analysis is often too imprecise, the analyzer sacrificed the soundness to recover the precision. Now, with the localization techniques presented in this dissertation, Airac scales to globally analyze up to one million lines of C programs without compromising soundness and precision.

Technically, we come to the following conclusions:

- Precise spatial localization (also known as, memory localization) is a key to the higher scalability. The conventional reachability-based localization technique can be too conservative and does not much help in practice. More accurate approach is necessary for practical use. We proposed the access-based approach to spatial localization and showed that the technique leads to much higher scalability than expected.

- Temporal localization (a.k.a., sparse analysis techniques) greatly increases the scalability of abstract interpretation. In abstract interpretation-based general

static analysis, the sparsity had been exploited only in limited ways ("localization" or "sparse evaluation"). We showed that fine-grained, SSA-like sparse analysis techniques can be generalized so that a class of abstract interpretation can use the idea to improve their scalability.

- The contextual localization is also a key to scalable interprocedural analysis. Without the technique, global analysis suffers from large spurious interprocedural cycles, which degrades not only analysis speed but also precision. Traditional context-sensitive analysis techniques are not appropriate for this use. We presented a simple algorithmic solution that mitigates the problem.

Bibliography

[1] Stephen Adams, Thomas Ball, Manuvir Das, Sorin Lerner, Sriram K. Rajamani, Mark Seigle, and Westley Weimer. Speeding up dataflow analysis using flow-insensitive pointer analysis. In *Proceedings of the International Symposium on Static Analysis*, pages 230–246, 2002.

[2] Xavier Allamigeon, Wenceslas Godard, and Charles Hymans. Static analysis of string manipulations in critical embedded C programs. In *Proceedings of the International Symposium on Static Analysis*, pages 35–51, 2006.

[3] Gogul Balakrishnan and Thomas Reps. Analyzing memory accesses in x86 binary executables. In *Proceedings of the International Conference on Compiler Construction*, pages 5–23, 2004.

[4] Josh Berdine, Cristiano Calcagno, Peter W. O'Hearn, Peter W. O'hearn, and Queen Mary. Symbolic execution with separation logic. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, pages 52–68. Springer, 2005.

[5] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *Proceedings of the ACM SIGPLAN-SIGACT Conference on Programming Language Design and Implementation*, pages 196–207, 2003.

[6] Francois Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, pages 128–141, 1993.

[7] Craig Chambers, Jeffrey Dean, and David Grove. Frameworks for intra- and interprocedural dataflow analysis. Technical report, Department of Computer Science and Engineering, University of Washington, 1996.

[8] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, pages 296–310, 1990.

[9] Li-Ling Chen and Williams Ludwell Harrison, III. An efficient approach to computing fixpoints for complex program analysis. In *Proceedings of the 8th international conference on Supercomputing*, pages 98–106, 1994.

[10] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 55–66, 1991.

[11] Stephen Chong and Radu Rugina. Static analysis of accessed regions in recursive data structures. In *Proceedings of the International Symposium on Static Analysis*, pages 463–482. Springer, 2003.

[12] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *Conference Record of the Sixth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 269–282, San Antonio, Texas, 1979. ACM Press, New York, NY.

[13] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In *Proceedings of the International Workshop Programming Language Implementation and Logic Programming, PLILP '92,*, pages 269–295, 1992.

[14] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, December 2009.

[15] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–97, Tucson, Arizona, 1978. ACM Press, New York, NY.

[16] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 238–252, 1977.

[17] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[18] Ron K. Cytron and Jeanne Ferrante. Efficiently computing ϕ-nodes on-the-fly. *ACM Trans. Program. Lang. Syst.*, 17:487–506, May 1995.

[19] Dhananjay M. Dhamdhere, Barry K. Rosen, and F. Kenneth Zadeck. How to analyze large programs efficiently and informatively. In *Proceedings of the ACM SIGPLAN conference on Programming language design and implementation*, PLDI '92, pages 212–223, New York, NY, USA, 1992. ACM.

[20] Isil Dillig, Thomas Dillig, and Alex Aiken. Sound, complete and scalable path-sensitive analysis. In *Proceedings of the 2008 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '08, pages 270–280, New York, NY, USA, 2008. ACM.

[21] Isil Dillig, Thomas Dillig, and Alex Aiken. Precise reasoning for programs using containers. In *Proceedings of the 38th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '11, pages 187–200, New York, NY, USA, 2011. ACM.

[22] Alexey Gotsman, Josh Berdine, and Byron Cook. Interprocedural shape analysis with separated heap abstractions. In *Proceedings of the International Symposium on Static Analysis*, pages 240–260, 2006.

[23] Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In *Proceedings of the 2007 ACM SIGPLAN conference on Programming language design and implementation*, PLDI '07, pages 290–299, New York, NY, USA, 2007. ACM.

[24] Ben Hardekopf and Calvin Lin. Semi-sparse flow-sensitive pointer analysis. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL '09, pages 226–238, New York, NY, USA, 2009. ACM.

[25] Ben Hardekopf and Calvin Lin. Flow-sensitive pointer analysis for millions of lines of code. In *Proceedings of the Symposium on Code Generation and Optimization*, 2011.

[26] Williams L. Harrison III. *The Interprocedural Analysis and Automatic Parallelization of Scheme Programs.* PhD thesis, Center for Supercomputing Research and Development, University of Illinois at Urabana-Champaign, February 1989.

[27] Michael Hind and Anthony Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Proceedings of the International Symposium on Static Analysis*, pages 57–81. Springer-Verlag, 1998.

[28] Suresh Jagannathan, Peter Thiemann, Stephen Weeks, and Andrew Wright. Single and loving it: must-alias analysis for higher-order languages. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 329–341, 1998.

[29] Yongin Jhee, Minsik Jin, Yungbum Jung, Deokhwan Kim, Soonho Kong, Heejong Lee, Hakjoo Oh, Daejun Park, and Kwangkeun Yi. Abstract interpretation + impure catalysts: Our Sparrow experience. Presentation at

the Workshop of the 30 Years of Abstract Interpretation, San Francisco, `ropas.snu.ac.kr/~kwang/paper/30yai-08.pdf`, January 2008.

[30] Richard Johnson and Keshav Pingali. Dependence-based program analysis. In *Proceedings of the ACM SIGPLAN 1993 conference on Programming language design and implementation*, PLDI '93, pages 78–89, New York, NY, USA, 1993. ACM.

[31] Yungbum Jung, Jaehwang Kim, Jaeho Shin, and Kwangkeun Yi. Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In *Proceedings of the International Symposium on Static Analysis*, pages 203–217, 2005.

[32] Yungbum Jung and Kwangkeun Yi. Practical memory leak detector based on parameterized procedural summaries. In *Proceedings of the International Symposium on Memory Management*, pages 131–140, 2008.

[33] Bageshri Karkare and Uday P. Khedker. An improved bound for call strings based interprocedural analysis of bit vector frameworks. *ACM Trans on Programming Languages and Systems*, 29(6):38, 2007.

[34] Uday Khedker, Amitabha Sanyal, and Bageshri Karkare. *Data Flow Analysis: Theory and Practice*. CRC Press, 2009.

[35] Uday P. Khedker and Bageshri Karkare. Efficiency, precision, simplicity, and generality in interprocedural data flow analysis: Resurrecting the classical call strings method. In *Proceedings of the International Conference on Compiler Construction*, pages 213–228, 2008.

[36] Chris Lattner, Andrew Lenharth, and Vikram Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *Proceedings of the 2007 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'07)*, San Diego, California, June 2007.

[37] Lian Li, Cristina Cifuentes, and Nathan Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, ESEC/FSE '11, pages 343–353, New York, NY, USA, 2011. ACM.

[38] J. Lind-Nielson. BuDDy, a binary decision diagram package.

[39] Mark Marron, Manuel Hermenegildo, Deepak Kapur, and Darko Stefanovic. Efficient context-sensitive shape analysis with graph based heap models. In *Proceedings of the International Conference on Compiler Construction*, pages 245–259, 2008.

[40] Florian Martin. PAG - an efficient program analyzer generator. *International Journal on Software Tools for Technology Transfer*, 2(1):46–67, 1998.

[41] Florian Martin. Experimental comparison of call string and functional approaches to interprocedural analysis. In *Proceedings of the International Conference on Compiler Construction*, pages 63–75, 1999.

[42] MathWorks. Polyspace embedded software verification. `http://www.mathworks.com/products/polyspace/index.html`.

[43] Matthew Might, Benjamin Chambers, and Olin Shivers. Model checking via ΓCFA. In *Proceedings of the 8th International Conference on Verification, Model Checking and Abstract Interpretation (VMCAI 2007)*, pages 59–73, Nice, France, January 2007.

[44] Matthew Might and Olin Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *Proceedings of the ACM SIGPLAN-SIGACT International Conference on Functional Programming*, pages 13–25, 2006.

[45] Ana Milanova, Atanas Rountev, and Barbara G. Ryder. Precise and efficient call graph construction for c programs with function pointers. *Journal of Automated Software Engineering*, 2004.

[46] A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, 2006.

[47] Eugene M. Myers. A precise inter-procedural data flow algorithm. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 219–230, 1981.

[48] Hakjoo Oh. Large spurious cycle in global static analyses and its algorithmic mitigation. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, volume 5904 of *Lecture Notes in Computer Science*, pages 14–29. Springer-Verlag, December 2009.

[49] Hakjoo Oh, Lucas Brutschy, and Kwangkeun Yi. Access analysis-based tight localization of abstract memories. In *VMCAI 2011: 12th International Conference on Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes in Computer Science*, pages 356–370. Springer, 2011.

[50] Hakjoo Oh and Kwangkeun Yi. An algorithmic mitigation of large spurious interprocedural cycles in static analysis. *Software: Practice and Experience*, 40(8):585–603, 2010.

[51] Hakjoo Oh and Kwangkeun Yi. Access-based localization with bypassing. In *Proceedings of the Asian Symposium on Programming Languages and Systems*, volume 7078 of *Lecture Notes in Computer Science*, pages 50–65. Springer-Verlag, December 2011.

[52] Peter W. O'Hearn, John C. Reynolds, and Hongseok Yang. Local reasoning about programs that alter data structures. In *Proceedings of the 15th International Workshop on Computer Science Logic*, CSL '01, pages 1–19, London, UK, 2001. Springer-Verlag.

[53] Keshav Pingali and Gianfranco Bilardi. Optimal control dependence computation and the roman chariots problem. *ACM Trans. Program. Lang. Syst.*, 19:462–491, May 1997.

[54] G. Ramalingam. On sparse evaluation representations. *Theoretical Computer Science*, 277(1-2):119–147, 2002.

[55] John H. Reif and Harry R. Lewis. Symbolic evaluation and the global value graph. In *Proceedings of the 4th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 104–118, 1977.

[56] Thomas Reps. Program analysis via graph reachability. *Information and Software Technology*, 40:5–19, 1998.

[57] Thomas Reps, Susan Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 49–61, 1995.

[58] John C. Reynolds. Separation logic: A logic for shared mutable data structures. In *Proceedings of the IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002.

[59] Noam Rinetzky, Jörg Bauer, Thomas Reps, Mooly Sagiv, and Reinhard Wilhelm. A semantics for procedure local heaps and its abstractions. In *Proceedings of The ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 296–309, 2005.

[60] Noam Rinetzky, Mooly Sagiv, and Eran Yahav. Interprocedural shape analysis for cutpoint-free programs. In *Proceedings of the International Symposium on Static Analysis*, pages 284–302, 2005.

[61] Xavier Rival and Laurent Mauborgne. The trace partitioning abstract domain. *ACM Trans on Programming Languages and System*, 29(5):26–51, 2007.

[62] Mooly Sagiv, Thomas Reps, and Susan Horwitz. Precise interprocedural dataflow analysis with applications to constant propagation. *Theoretical Computer Sicence*, 167(1-2):131–170, 1996.

[63] Marc Shapiro and Susan Horwitz. The effects of the precision of pointer analysis. In *Proceedings of the International Symposium on Static Analysis*, pages 16–34, 1997.

[64] Micha Sharir and Amir Pnueli. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, chapter 7. Prentice-Hall, 1981.

[65] Manu Sridharan and Rastislav Bodík. Refinement-based context-sensitive points-to analysis for java. In *Proceedings of the ACM SIGPLAN-SIGACT Conference on Programming Language Design and Implementation*, pages 387–400, 2006.

[66] Teck Bok Tok, Samuel Z. Guyer, and Calvin Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *Proceedings of the International Conference on Compiler Construction*, pages 17–31, 2006.

[67] Arnaud Venet and Guillaume Brat. Precise and efficient static array bound checking for large embedded C programs. In *Proceedings of the ACM SIGPLAN 2004 conference on Programming language design and implementation*, PLDI '04, pages 231–242, New York, NY, USA, 2004. ACM.

[68] Mark N. Wegman and F. Kenneth Zadeck. Constant propagation with conditional branches. *ACM Trans. Program. Lang. Syst.*, 13:181–210, April 1991.

[69] John Whaley and Monica S. Lam. Cloning-based context-sensitive pointer alias analysis using binary decision diagrams. In *Proceedings of the ACM SIGPLAN-SIGACT Conference on Programming Language Design and Implementation*, pages 131–144, 2004.

[70] Guoqing Xu, Atanas Rountev, and Manu Sridharan. Scaling CFL-reachability-based points-to analysis using context-sensitive must-not-alias analysis. In *Proceedings of the European Conference on Object-Oriented Programming*, pages 98–122, 2009.

[71] Hongseok Yang, Oukseh Lee, Josh Berdine, Cristiano Calcagno, Byron Cook, Dino Distefano, and Peter O'Hearn. Scalable shape analysis for systems code. In *Proceedings of the International Conference on Computer Aided Verification*, pages 385–398, 2008.

[72] Hongseok Yang, Oukseh Lee, Cristiano Calcagno, Dino Distefano, and Peter O'Hearn. On scalable shape analysis. Technical Memorandum RR-07-10, Queen Mary University of London, Department of Computer Science, November 2007.

[73] Hongtao Yu, Jingling Xue, Wei Huo, Xiaobing Feng, and Zhaoqing Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *Proceedings of the 8th annual IEEE/ACM international symposium on Code generation and optimization*, CGO '10, pages 218–229, New York, NY, USA, 2010. ACM.

[74] Misha Zitser, D. E. Shaw Group, and Tim Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *In SIGSOFT 4/FSE-12: Proceedings of the 12th ACM SIGSOFT twelfth international symposium on Foundations of software engineering*, pages 97–106. ACM Press, 2004.

초 록

정확하게, 실행상황을 모두 포섭하면서, 초대형의 프로그램을 한꺼번에 분석하기 위한 정적분석 기술을 제시한다. 먼저 바탕이 되는 정적분석기를 요약 해석의 틀에서 디자인한다. 이 단계의 분석기는 실행상황을 모두 포섭하고, 우리가 염두에 둔 만큼 정확하다는 것이 보장되지만 큰 프로그램을 분석할 수 있을 정도의 성능을 갖추진 못한 상태이다. 그 다음 단계로, 필요한 메모리 부위, 시점, 상황 선별을 통해서 분석기의 성능을 극대화시킨다. 분석기의 안전성과 정확도는 첫번째 단계에서 디자인된 그대로 유지된다. 이 논문은 위의 기술들을 엄밀히 정의하고 그 기술들이 백만라인 이상의 매우 큰 C 프로그램을 분석할 수 있을정도로 정적분석기의 성능을 향상시킴을 실험적으로 보인다.

주요어 : 프로그래밍 언어, 정적 프로그램 분석, 요약 해석, 전체 분석, 선별 및 집중하기

학 번 : 2007-30226