Design and Implementation of Sparse Global Analyses for C-like Languages

Hakjoo Oh Kihong Heo Wonchan Lee Woosuk Lee Kwangkeun Yi

Seoul National University

{pronto,khheo,wclee,wslee,kwang}@ropas.snu.ac.kr

Abstract

In this article we present a general method for achieving global static analyzers that are precise, sound, yet also scalable. Our method generalizes the sparse analysis techniques on top of the abstract interpretation framework to support relational as well as non-relational semantics properties for C-like languages. We first use the abstract interpretation framework to have a global static analyzer whose scalability is unattended. 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 non-relational and relational static analyses; we present their implementation results that scales to analyze upto 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.

Categories and Subject Descriptors F.3.2 [*Semantics of Programming Languages*]: Program Analysis

Keywords Static analysis, abstract interpretation, sparse analysis

1. Introduction

Precise, sound, scalable yet global static analyzers has been unachievable in general. Other than almost syntactic properties, once the target property become 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 article we present a general method for achieving global static analyzers that are precise, sound, yet also scalable. Our approach generalize the sparse analysis ideas on top of the abstract interpretation framework. Since the abstract interpretation framework [9, 11] guides us to design sound yet arbitrarily precise static analyzers for any target language, we first use the framework to have a global static analyzer whose scalability is unattended. 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.

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 [7, 8, 14, 15, 18, 19, 22, 34–36, 38, 40] 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 non-relational and relational static analyses; we present their implementation results that scales to analyze upto one million lines of C programs.

Contributions Our contributions are as follows.

- We propose a general framework for designing 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.
- We design sparse non-relational and relational analysis which are still general as themselves. We can instantiate these designs with a particular non-relational and relational abstract domain, respectively.
- We prove the practicality of our framework by experimentally demonstrating the achieved speedup of sparse analysis versions. Our analysis can analyze programs up to 1 million lines of C code with interval domain and up to 100K lines of C code with octagon domain.

Outline The rest of this paper is organized as follows. Section 2 explains our sparse analysis framework. Section 3 and 4 design sparse non-relational and relational analysis, respectively, based on our framework. Section 5 discusses several issues involved in the implementations. Section 6 presents the experimental studies. Section 7 discusses related work.

PLDI'12, June 11–16, Beijing, China.

Copyright © 2012 ACM 978-1-4503-1205-9/12/04…$10.00

2. Sparse Analysis Framework

2.1 Notation

We write R^+ and $R^\star$ for the transitive and reflexive-transitive closure of binary relation R. Given function $f \in A \to B$, we write $f|_C$ for the restriction of function f to the domain $\mathsf{dom}(f) \cap C$. We write $f\backslash_C$ for the restriction of f to the domain $\mathsf{dom}(f) - C$. 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 $f[a \mapsto b]$ for the function got from function f by changing the value for a to b. 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).

2.2 Program

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,

$$F(X) = \lambda c \in \mathbb{C}. f_c\Big(\bigcup_{c_p \hookrightarrow c} X(c_p)\Big). \tag{1}$$

where $f_c \in 2^{\mathbb{S}} \to 2^{\mathbb{S}}$ is a semantic function at control point c. We leave out the definition of the concrete semantic function f_c that depends on target languages. Our framework is independent from target languages.

2.3 Baseline Abstraction

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

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

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 locations, 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 [4, 13, 31, 39], 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 [31] and polyhedrons [12].

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

$$\hat{F}(\hat{X}) = \lambda c \in \mathbb{C}. \hat{f}_c\Big(\bigsqcup_{c' \hookrightarrow c} \hat{X}(c')\Big). \tag{3}$$

where $\hat{f}_c \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ is an abstract semantic function at control point c. The soundness of abstract semantics is followed by fixpoint transfer theorem [11].

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

2.4 Sparse Analysis by Eliminating Unnecessary Propagation

The abstract semantic function given in (3) propagates some abstract values unnecessarily. For example, suppose that we analyze statement $x := y$ using a non-relational domain, like interval domain [9]. 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 (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; that is, we make the semantic function propagate only the abstract values newly computed at one control point to the other where they are actually used. In the rest of this section, we explain how to make abstract semantic function (3) sparse with the guarantees of soundness and precision.

2.5 Definition and Use Set

To be correct, we first need to precisely define what are "definitions" and "uses". They are defined in terms of abstract semantics, i.e., abstract semantic function $\hat{f}_c$.

Definition 1 (Definition set)**.** *Let $\mathcal{S}$ be the least fixpoint* $\mathbf{lfp}\hat{F}$ *of the original semantic function $\hat{F}$. Definition set $\mathsf{D}(c)$ at control point c is a set of abstract locations that are assigned a new abstract value by abstract semantic function $\hat{f}_c$, i.e.*

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

Definition 2 (Use set)**.** *Let $\mathcal{S}$ be the least fixpoint* $\mathbf{lfp}\hat{F}$ *of the original semantic function $\hat{F}$. Use set $\mathsf{U}(c)$ at control point c is a set of abstract locations that are used by abstract semantic function $\hat{f}_c$, i.e.*

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

Example 1. *Consider the following simple subset of C:*

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

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.*

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

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

2.6 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 two control points. 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. Formal definition of data dependency is given below: (For simplicity, in this definition, we only consider straight-line programs.)

Definition 3 (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:*

$$\begin{array}{rcl} c_d \overset{l}{\rightsquigarrow} c_u & \triangleq & c_d \hookrightarrow^+ c_u \\ & \wedge & l \in \mathsf{D}(c_d) \cap \mathsf{U}(c_u) \\ & \wedge & \forall c_i \in \mathbb{C}. c_d \hookrightarrow^+ c_i \hookrightarrow^+ c_u \implies l \notin \mathsf{D}(c_i). \end{array}$$

The definition means that if control point c_u is reachable from control point c_d, a value of abstract location l can be defined at c_u and used at c_d, and there is no intermediate control point c_i that can change the value of l, then we can directly propagate the value of l from c_d to c_u.

Example 2. *In the program presented Example 1, we can find two data dependencies,* $10 \overset{x}{\rightsquigarrow} 11$ *and* $11 \overset{x}{\rightsquigarrow} 12$.

Comparison with Def-use Chains Note that our notion of data dependency is different from the conventional notion of def-use chains. If we want to conservatively collect all the possible def-use chains of the given definition set and use set, we should exclude only the paths from definition points to use points when there exists a point that always kills the definition. However, data dependency in Definition 3 excludes a path even when there exists a point that might, but not always, kill the definition. We can slightly modify Definition 3 to express def-use chain relation $\rightsquigarrow_{\mathsf{du}}$ as follows:

$$\begin{array}{rcl} c_d \overset{l}{\rightsquigarrow}_{\mathsf{du}} c_u & \triangleq & c_d \hookrightarrow^+ c_u \\ & \wedge & l \in \mathsf{D}(c_d) \cap \mathsf{U}(c_u) \\ & \wedge & \forall c_i \in \mathbb{C}. c_d \hookrightarrow^+ c_i \hookrightarrow^+ c_u \implies l \notin \mathsf{D}_{\mathsf{must}}(c_i) \end{array}$$

where $\mathsf{D}_{\mathsf{must}}(c) \triangleq \{l \in \mathbb{L} \mid \forall \hat{s} \sqsubseteq \bigsqcup_{c' \hookrightarrow c} (\mathbf{lfp}\hat{F})(c'). \hat{f}_c(\hat{s})(l) \neq \hat{s}(l)\}$. The relation contains the comprehensive set of def-use chains that appear during the analysis. For example, 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 1.

The reason why we use our notion of data dependencies instead of def-use chains becomes evident in Section 2.8, where we discuss the approximations of them.

2.7 Sparse Abstract Semantic Function

Using data dependency, we can make abstract semantic function sparse, which propagates between control points only the abstract values that participate in the fixpoint computation. Sparse abstract function $\hat{F}_s$, whose definition is given below, is the same as the original except that it propagates abstract values along to the data dependency, not to control dependency:

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

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.

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

Lemma 2 (Correctness)**.** *Let $\mathcal{S}$ and $\mathcal{S}_s$ be $\mathbf{lfp}\hat{F}$ and $\mathbf{lfp}\hat{F}_s$. Then,*

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

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 domain of $\mathcal{S}_s(c)$ for each $c \in \mathbb{C}$. Let $c_1, \cdots, c_n$ be control points and x and y be abstract locations such that $c_1 \hookrightarrow \cdots \hookrightarrow c_n$, $c_1 \overset{x}{\rightsquigarrow} c_n$. For brevity, we only consider the case with the following assumptions: $\mathsf{D}(c_n) = \mathsf{U}(c_n) = \{x\}$, $c_1 \overset{x}{\rightsquigarrow} c_n$ is the only data dependency on c_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). Then, the fixpoint equations of $\hat{F}$ are as follows:

$$\mathcal{S}(c_2) = \hat{f}_{c_2}(\mathcal{S}(c_1)) \quad \cdots \quad \mathcal{S}(c_n) = \hat{f}_{c_n}(\mathcal{S}(c_{n-1})). \tag{4}$$

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

$$\begin{array}{rll} \mathcal{S}(c_n)(x) & = \hat{f}_{c_n}(\mathcal{S}(c_{n-1}))(x) & (\because (4)) \\ & = \hat{f}_{c_n}(\mathcal{S}(c_{n-1})|_x)(x) & (\because \text{ Def. of } \mathsf{U} \text{ and } \mathsf{U}(c_n) = \{x\}) \\ & = \hat{f}_{c_n}(\mathcal{S}(c_1)|_x)(x) & (\because \text{ Def. of } \rightsquigarrow \text{ and } c_1 \overset{x}{\rightsquigarrow} c_n) \end{array}$$

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.

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

Note that $\mathsf{dom}(\mathcal{S}_s(c_n)) = \mathsf{D}(c_n) \cup \mathsf{U}(c_n) = \{x\}$.

□

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.

2.8 Sparse Analysis with Approximated Data Dependency

Sparse analysis designed until Section 2.7 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 (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:*

$$\begin{array}{rcl} c_d \overset{l}{\rightsquigarrow}_a c_u & \triangleq & c_d \hookrightarrow^+ c_u \\ & \wedge & l \in \hat{\mathsf{D}}(c_d) \cap \hat{\mathsf{U}}(c_u) \\ & \wedge & \forall c_i. c_d \hookrightarrow^+ c_i \hookrightarrow^+ c_u \implies l \notin \hat{\mathsf{D}}(c_i) \end{array}$$

The definition is the same 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 $\hat{\mathsf{D}}$ 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 3. *Consider the same program presented in Example 1. 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:

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

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

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

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* $10 \overset{x}{\not\rightsquigarrow}_a 12$*). However, if* $\{x\} \subseteq \hat{\mathsf{U}}(11)$*, then 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 5. *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 things 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 3 (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,*

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

Proof. (Sketch) We can prove the lemma by showing the equivalence of the fixpoint equations up to the domain of $\mathcal{S}_a(c)$ for each $c \in \mathbb{C}$, as we did for Lemma 2. We make the same assumptions as in the proof of Lemma 2 (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:

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

The fixpoint equations of $\hat{F}$ we derived in the proof of Lemma 2 are as follows:

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

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 4. *Consider the same setting of Example 3, assuming the points-to set of* p *being* $\{x\}$*. . 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 5). 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.*

2.9 Designing Sparse Analysis Steps in 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 [9]. Note that the abstract domain should be a member of the family explained in Section 2.3.

(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 5).

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).

3. Designing Sparse Non-Relational Analysis

As a concrete example, we show how to design sparse non-relational analyses within our framework. Following Section 2.9, we proceed in two steps: (1) We design a conventional non-relational analysis based on abstract interpretation. Relying on the abstract interpretation framework [9, 10], 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 5).

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. $\mathbb{V} = \mathbb{Z} + \mathbb{L}$):

$$\begin{array}{c} x := e \mid *x := e \mid \{\!\!\{ x < n \}\!\!\} \\ \text{where } \; e \to n \mid x \mid \&x \mid *x \mid e{+}e \end{array}$$

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 $\{\!\!\{ x < n \}\!\!\}$ makes the program continues only when the condition evaluates to true.

3.1 Step 1: Designing Non-sparse Analysis

Abstract Domain From the baseline abstraction (in Section 2.3), 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 $\alpha_{\mathbb{S}}$.)

$$\hat{\mathbb{S}} = \hat{\mathbb{L}} \to \hat{\mathbb{V}} \qquad \hat{\mathbb{L}} = \mathit{Var} \qquad \hat{\mathbb{V}} = \hat{\mathbb{Z}} \times \hat{\mathbb{P}} \qquad \hat{\mathbb{P}} = 2^{\hat{\mathbb{L}}}$$

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 (3), $\hat{F}$, where the abstract semantic function $\hat{f}_c \in \hat{\mathbb{S}} \to \hat{\mathbb{S}}$ is defined as follows:

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

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. $\{\!\{x < n\}\!\}$ 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:

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

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

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 $\hat{\mathbb{D}}_{pre}$ and $\hat{F}_{pre}$ be the domain and semantic function of such a pre-analysis, which satisfies the following two conditions.

$$\mathbb{C} \to \hat{\mathbb{S}} \underset{\alpha_{pre}}{\overset{\gamma_{pre}}{\leftrightarrows}} \hat{\mathbb{D}}_{pre} \qquad \alpha_{pre} \circ \hat{F} \sqsubseteq \hat{F}_{pre} \circ \alpha_{pre}$$

By abstract interpretation framework [9, 10], such a pre-analysis is guaranteed to be conservative, i.e., $\alpha_{pre}(\mathbf{lfp}\hat{F}) \sqsubseteq \mathbf{lfp}\hat{F}_{pre}$. As an example, in experiments (Section 6), we use a simple abstraction as follows:

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

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

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

$\hat{\mathsf{D}}$ is defined to include locations whose values are potentially defined (changed). In the definition of $\hat{f}_c$ for $x := e$ and $\{\!\{x < n\}\!\}$, 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 $\hat{\mathcal{T}}_{pre}(c)$. Such $\hat{\mathsf{D}}$ satisfies the safe approximation condition (Definition 5), 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 $\hat{\mathcal{E}}$.

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

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 $\hat{s}_c = \hat{\mathcal{T}}_{pre}(c)$)

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

Using $\hat{\mathcal{T}}_{pre}$ 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 $\{\!\{x < n\}\!\}$, $\mathcal{U}$ additionally includes x.

Lemma 5. $\hat{\mathsf{D}}$ *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 [18, 19].

Semi-sparse analysis [18] 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 $\hat{\mathcal{T}}_{pre}$ such that $\hat{\mathcal{T}}_{pre}(c)(x).\hat{\mathbb{P}} = \hat{\mathbb{L}}$ for all xs that are not top-level variables.

Staged Flow-Sensitive Analysis [19] 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 [19], although the framework cannot guarantee the correctness anymore.

4. Designing 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, *Packs* $\subseteq 2^{Var}$ such that $\bigcup$*Packs* $=$ *Var*, are given by users or a pre-analysis [13, 31]. In a packed relational analysis, abstract states ($\hat{\mathbb{S}}$) map variable packs (*Packs*) to a relational domain ($\hat{\mathbb{R}}$), i.e., $\hat{\mathbb{S}} =$ *Packs* $\to \hat{\mathbb{R}}$, which is a member of the abstract domains that our framework deals with (Section 2.3). 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 [13]. Thus, realistic relational analyzers usually exploit small packs of variables [4, 13, 39].

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

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 $\mathit{Packs} = \{\langle\!\langle x,y\rangle\!\rangle, \langle\!\langle x,z\rangle\!\rangle, \langle\!\langle y,z\rangle\!\rangle\}$, definitions are $\{\langle\!\langle x,y\rangle\!\rangle, \langle\!\langle x,z\rangle\!\rangle\}$.

In Section 4.1, we define relational analysis with variable packing. In Section 4.2, we define safe $\hat{\mathsf{D}}$ and $\hat{\mathsf{U}}$.

4.1 Step 1: Designing Non-sparse Analysis

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

$$\mathit{cmd} \to x := e \mid \{\!\!\{x < n\}\!\!\} \quad \text{where } e \to n \mid x \mid e{+}e$$

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 2.3), 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 $2^{\mathbb{S}} \underset{\alpha_{\hat{\mathbb{R}}}}{\overset{\gamma_{\hat{\mathbb{R}}}}{\leftrightarrows}} \hat{\mathbb{R}}$.)

$$\hat{\mathbb{S}} = \hat{\mathbb{L}} \to \hat{\mathbb{V}} \qquad \hat{\mathbb{L}} = \mathit{Packs} \qquad \hat{\mathbb{V}} = \hat{\mathbb{R}}$$

Note that the abstraction is generic so we can choose any relational domain for $\hat{\mathbb{R}}$, such as octagon [31].

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 $\mathit{Packs} = \{\langle\!\langle a,c\rangle\!\rangle, \langle\!\langle b,c\rangle\!\rangle\}$. 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 $\hat{\mathcal{R}} \in \mathit{cmd}^{rel} \to \hat{\mathbb{R}} \to \hat{\mathbb{R}}$ for relational domain $\hat{\mathbb{R}}$ is defined over the following internal language:

$$\mathit{cmd}^{rel} \to x := e^{rel} \mid \{\!\!\{x < \hat{\mathbb{Z}}\}\!\!\} \text{ where } e^{rel} \to \hat{\mathbb{Z}} \mid x \mid e^{rel}{+}e^{rel}$$

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 (3), where the abstract semantic function $\hat{f}_c$ is defined as follows:

$$\hat{f}_c(\hat{s}) = \hat{s}[p_1 \mapsto \hat{\mathcal{R}}(\mathit{cmd}_1)(\hat{s}(p_1)), \ldots, p_k \mapsto \hat{\mathcal{R}}(\mathit{cmd}_k)(\hat{s}(p_k))]$$

where

$$\begin{array}{rcl} \{p_1,\ldots,p_k\} & = & \left\{ \begin{array}{ll} \mathsf{pack}(x) & \mathsf{cmd}(c) = x := e \\ \mathsf{pack}(x) & \mathsf{cmd}(c) = \{\!\!\{x < n\}\!\!\} \end{array} \right. \\ \mathit{cmd}_i & = & \mathscr{T}(p_i)(\hat{s})(\mathsf{cmd}(c)) \end{array}$$

For variable x, $\mathsf{pack}(x)$ returns the set of packs that contains x, i.e., $\mathsf{pack}(x) = \{p \in \mathit{Packs} \mid x \in p\}$. For both $x := e$ and $\{\!\!\{x < n\}\!\!\}$, we update only the packs that include x. $\mathscr{T}$ is the function that transforms cmd into cmd^{rel}. Given a variable pack p, state $\hat{s}$, and command cmd, $\mathscr{T}(p)(\hat{s}) \in \mathit{cmd} \to \mathit{cmd}^{rel}$ returns transformed command for a given command.

$$\begin{array}{rcl} \mathscr{T}(p)(\hat{s})(x := e) & = & x := \mathscr{T}_e(p)(\hat{s})(e) \\ \mathscr{T}(p)(\hat{s})(\{\!\!\{x < n\}\!\!\}) & = & \{\!\!\{x < \mathscr{T}_e(p)(\hat{s})(n)\}\!\!\} \end{array}$$

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

$$\begin{array}{rcl} \mathscr{T}_e(p)(\hat{s})(n) & = & \alpha_{\mathbb{Z}}(\{n\}) \\ \mathscr{T}_e(p)(\hat{s})(x) & = & \left\{ \begin{array}{ll} x & \text{if } x \in p \\ \pi_x(\hat{s}) & \text{otherwise} \end{array} \right. \\ \mathscr{T}_e(p)(\hat{s})(e_1{+}e_2) & = & \mathscr{T}_e(p)(\hat{s})(e_1){+}\mathscr{T}_e(p)(\hat{s})(e_2) \end{array}$$

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

4.2 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:

$$\hat{\mathsf{D}}(c) = \left\{ \begin{array}{ll} \mathsf{pack}(x) & \mathsf{cmd}(c) = x := e \\ \mathsf{pack}(x) & \mathsf{cmd}(c) = \{\!\!\{x < n\}\!\!\} \end{array} \right.$$

$\hat{\mathsf{U}}$ 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 [31]. In section 6, we implement our analyzer based on this definition of π_x.

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

$$\hat{\mathsf{U}}(c) = \left\{ \begin{array}{ll} \mathsf{pack}(x) \cup \{\{l\} \mid l \in \mathcal{U}(e)\} & \mathsf{cmd}(c) = x := e \\ \mathsf{pack}(x) & \mathsf{cmd}(c) = \{\!\!\{x < n\}\!\!\} \end{array} \right.$$

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

Lemma 7. $\hat{\mathsf{D}}$ *and* $\hat{\mathsf{U}}$ *are safe approximations.*

5. Implementation Techniques

Implementing sparse analysis presents unique challenges regarding construction and management of data dependencies. 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 6).

Generation of Data Dependencies We use the standard SSA algorithm to generate data dependencies. Because our notion of data dependencies ($\rightsquigarrow$, Definition 3) 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 [40].

Interprocedural Extension 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 $1 \overset{x}{\leadsto} 2$ and $3 \overset{x}{\leadsto} 4$ but also include spurious dependencies $1 \overset{x}{\leadsto} 4$ and $3 \overset{x}{\leadsto} 2$, 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 call-sites, which generates overwhelming number of spurious dependencies. In our experiments, such spurious dependencies made this approach hardly scalable. Staged pointer analysis algorithm [19] 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 [32], 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 3.2) to resolve function pointers. Because the pre-analysis is fairly precise, the precision loss caused by this approximation of the callgraph would be reasonably small in practice [30]. 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 $\texttt{f} \to \texttt{g} \to \texttt{h}$ 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 ($\leadsto$) until convergence. Suppose $a \overset{l}{\leadsto} b$, $b \overset{l}{\leadsto} c$, and that l is not defined nor used in b, then we remove those two dependencies and add $a \overset{l}{\leadsto} c$. 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 1), the data dependencies consist of 201 K abstract locations spanning over 2.8 M statements. Thus, storing dependency relation $\leadsto$ 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, $\langle c_1, c_3, l\rangle \in (\leadsto)$ 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.

6. Experiments

In this section, we evaluate sparse non-relational and relational static analyses designed in Section 3 and Section 4.

For the non-relational analysis, we use the interval domain [9], a representative non-relational domain that is widely used in practice [1, 2, 4, 13, 23]. For the relational analysis, we use the octagon domain [31], a representative relational domain whose effectiveness is well-known in practice [4, 13, 25, 39].

We have analyzed 18 software packages. Table 1 shows characteristics of our benchmark programs. The benchmarks are various open-source applications, and most of them are from GNU open-source projects. Standard library calls are summarized using hand-crafted 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.

6.1 Interval Domain-based Sparse Analysis

Setting The baseline analyzer, $\mathsf{Interval}_{\mathsf{base}}$, 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 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 [9] 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 [5] and selective memory operators [4]. In particular, the analysis aggressively exploits the technique of localization [33, 37, 41]. We use access-based localization [33]: 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 [33].

From the baseline, we made two analyzers: $\mathsf{Interval}_{\mathsf{vanilla}}$ and $\mathsf{Interval}_{\mathsf{sparse}}$. $\mathsf{Interval}_{\mathsf{vanilla}}$ is identical to $\mathsf{Interval}_{\mathsf{base}}$ except that $\mathsf{Interval}_{\mathsf{vanilla}}$ does not perform the access-based localization. We compare the performance between $\mathsf{Interval}_{\mathsf{vanilla}}$ and $\mathsf{Interval}_{\mathsf{base}}$ just to check that our baseline analyzer is not a straw-man. $\mathsf{Interval}_{\mathsf{sparse}}$ 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 3, we use a flow-insensitive pre-analysis. Data dependencies are generated as described in Section 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 [27].

Result Table 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 (analy-

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 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.

sis). For $\mathsf{Interval}_{\mathsf{base}}$, the time includes auxiliary analysis time for access-based localization [33] as well as actual analysis time. For $\mathsf{Interval}_{\mathsf{sparse}}$, **Dep** includes times for pre-analysis and generation of data dependencies. **Fix** represent the time for fixpoint computation.

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

$\mathsf{Interval}_{\mathsf{sparse}}$ is faster than $\mathsf{Interval}_{\mathsf{base}}$ by 5–110x and saves memory by 3–92%. In particular, the analysis' scalability has been remarkably improved: $\mathsf{Interval}_{\mathsf{sparse}}$ scales to 1.4M LOC, which is an order of magnitude larger than that of $\mathsf{Interval}_{\mathsf{base}}$.

There are some counterintuitive results. First, the analysis time for $\mathsf{Interval}_{\mathsf{sparse}}$ 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 [24, 42]. Column **maxSCC** in Table 1 reports the sizes of the largest recursive cycle (precisely speaking, strongly connected component) in programs. 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 ($\mathsf{Interval}_{\mathsf{base}}$). Instead, the pre-analysis finds an approximation with small cost (compared to $\mathsf{Interval}_{\mathsf{base}}$). 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.

6.2 Octagon Domain-based Sparse Analysis

Setting We implemented octagon domain-based static analyzers $\mathsf{Octagon}_{\mathsf{vanilla}}$, $\mathsf{Octagon}_{\mathsf{base}}$, and $\mathsf{Octagon}_{\mathsf{sparse}}$ on top of the interval domain-based analysis engine explained in Section 6.1. We replaced interval-based abstract domain by octagon-based domain with variable packings. Non-numerical values (such as pointers, array, and structures) are handled in the same way as the interval analysis. Semantic functions are appropriately changed. Besides abstract domain and semantics, exactly the same engineering efforts have been also put into octagon-based analyzers. $\mathsf{Octagon}_{\mathsf{base}}$ performs the access-based localization [33] in terms of variable packs. $\mathsf{Octagon}_{\mathsf{vanilla}}$ is same as $\mathsf{Octagon}_{\mathsf{base}}$ but does not perform the localization and $\mathsf{Octagon}_{\mathsf{sparse}}$ is sparse version of $\mathsf{Octagon}_{\mathsf{base}}$. To represent octagon domain, we use Apron library [21].

In all experiments, we used a syntax-directed packing strategy. Given a program, we first run a flow-insensitive interval domain-based analysis (proposed in Section 3.2) to find the set of abstract locations. Then, by using a syntactic pre-analysis, we collect groups of abstract locations that are likely to be logically related. *Packs* are the set of all such groups. Then, relational analysis for the program uses the *Packs*. Our packing heuristic is similar to Miné's approach [13, 31], which groups abstract locations that have syntactic locality. For examples, abstract locations involved in the linear expressions or loops are grouped together. Scope of the locality is limited within each of syntactic C blocks. We also group abstract locations involved in actual and formal parameters, which is necessary to capture relations across procedure boundaries. In our packing, some large packs whose sizes exceed a threshold (10 abstract locations) are split down into smaller ones. The three analyzers use the same packing heuristic.

Result We also compared main analysis time and peak memory consumption of $\mathsf{Octagon}_{\mathsf{vanilla}}$, $\mathsf{Octagon}_{\mathsf{base}}$, and $\mathsf{Octagon}_{\mathsf{sparse}}$ in the same way as interval analysis. The performance numbers are described in Table 3.

Programs	$\mathsf{Interval}_{\mathsf{vanilla}}$		$\mathsf{Interval}_{\mathsf{base}}$		**Spd**$\uparrow_1$	**Mem**$\downarrow_1$	$\mathsf{Interval}_{\mathsf{sparse}}$						**Spd**$\uparrow_2$	**Mem**$\downarrow_2$
	Time	**Mem**	**Time**	**Mem**			**Dep**	**Fix**	**Total**	**Mem**	$\hat{\mathsf{D}}(c)$	$\hat{\mathsf{U}}(c)$		
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 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**$\uparrow_1$ is the speed-up of $\mathsf{Interval}_{\mathsf{base}}$ over $\mathsf{Interval}_{\mathsf{vanilla}}$. **Mem**$\downarrow_1$ shows the memory savings of $\mathsf{Interval}_{\mathsf{base}}$ over $\mathsf{Interval}_{\mathsf{vanilla}}$. **Spd**$\uparrow_2$ is the speed-up of $\mathsf{Interval}_{\mathsf{sparse}}$ over $\mathsf{Interval}_{\mathsf{base}}$. **Mem**$\downarrow_2$ shows the memory savings of $\mathsf{Interval}_{\mathsf{sparse}}$ over $\mathsf{Interval}_{\mathsf{base}}$. $\hat{\mathsf{D}}(c)$ and $\hat{\mathsf{U}}(c)$ show the average size of $\hat{\mathsf{D}}(c)$ and $\hat{\mathsf{U}}(c)$, respectively.

While $\mathsf{Octagon}_{\mathsf{vanilla}}$ requires extremely large amount of time and memory space but $\mathsf{Octagon}_{\mathsf{base}}$ makes the analysis realistic by leveraging the access-based localization. $\mathsf{Octagon}_{\mathsf{base}}$ is able to analyze 20 KLOC within 6 hours and 588MB of memory. With the localization, analysis speed of $\mathsf{Octagon}_{\mathsf{base}}$ increases by 10x–20x and memory consumption decreases by 50%–76%. Though $\mathsf{Octagon}_{\mathsf{base}}$ saves a lot of memory, the analysis is still not scalable at all. For example, bc-1.06 requires 5 times more memory than gzip-1.2.4a. This memory consumption is not reasonable considering program size and interval analysis result.

Thanks to sparse analysis technique, $\mathsf{Octagon}_{\mathsf{sparse}}$ becomes more practical and scales to 130 KLOC within 25 mins and 9.8 GB of memory consumption. $\mathsf{Octagon}_{\mathsf{sparse}}$ is 30–377x faster than $\mathsf{Octagon}_{\mathsf{base}}$ and saves memory consumption by 84%–95%. Note that the performance gap between sparse and non-sparse versions is more remarkable than those in interval analysis. It is because relational analysis has much more computational cost and memory consumption for each abstract value than non-relational analysis.

6.3 Discussion

Sparsity We discuss the relation between performance and sparsity. Column $\hat{\mathsf{D}}(c)$ and $\hat{\mathsf{U}}(c)$ in Table 2 and Table 3 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.[2] 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.

[2] The average sizes of $\hat{\mathsf{D}}(c)$ and $\hat{\mathsf{U}}(c)$ are quite similar. Because our abstract semantics considers weak update.

Variable Packing For maximal precision, packing strategy should be more carefully devised for each target program. However, note that our purpose of experiments is to show relative performance of $\mathsf{Octagon}_{\mathsf{sparse}}$ over $\mathsf{Octagon}_{\mathsf{base}}$, and we applied the same packing strategy for all analyzers. Though our general-purpose packing strategy is not specialized to each program, the packing strategy reasonably groups logically related variables. The average size of packs is 5–7 for our benchmarks. Domain-specific packing strategies, such as ones used in Astrée [31] or CGS [39], reports the similar results: 3–4 [31] or 5 [39].

7. Related Work

Scalable Sparse Pointer Analysis Recently, scalability of flow-sensitive pointer analysis has been greatly improved using sparse analysis; in 2009, Hardekopf et al. [18] presented a 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 [19, 26].

We already showed that our framework subsumes two scalable sparse pointer analysis presented in [18, 19].

Scalable Global Analyzers Our interval and octagon domain-based analyzers achieve higher scalability (up to 1 MLOC and 130 KLOC, respectively) than the previous general-purpose global analyzers. Zitser et al. [43] report that PolySpace C Verifier [28], a commercial tool for detection of runtime errors, cannot analyze sendmail because of scalability problem. Both our interval and octagon domain-based analyzers can analyze sendmail. Airac [23, 32], a general-purpose interval domain-based global static analyzer, scales only to 30 KLOC in global analysis. Recently, a significant progress has been reported by Oh et al. [33], but it still does not scale over 120 KLOC. Other similar (interval domain-based) analyzers are also not scalable to large code bases [1, 2]. Nevertheless, there have been scalable domain-specific static analyzers, like Astrée [4, 13] and CGS [39], which scale to hundreds of thousands lines of code. However, Astrée targets on programs that do

Programs	$\mathsf{Octagon}_{\mathsf{vanilla}}$		$\mathsf{Octagon}_{\mathsf{base}}$		Spd$\uparrow_1$	Mem$\downarrow_1$	$\mathsf{Octagon}_{\mathsf{sparse}}$						Spd$\uparrow_2$	Mem$\downarrow_2$
	Time	Mem	Time	Mem			Dep	Fix	Total	Mem	$\hat{\mathsf{D}}(c)$	$\hat{\mathsf{U}}(c)$		
gzip-1.2.4a	9,649	5,744	483	1,355	20 x	76 %	2	13	15	211	2.3	2.5	30 x	84 %
bc-1.06	15,027	10,090	1,454	5,065	10 x	50 %	4	17	21	381	3.1	3.4	55 x	92 %
tar-1.13	∞	N/A	21,125	13,810	N/A	N/A	6	46	52	588	2.5	2.5	377 x	95 %
less-382	∞	N/A	∞	N/A	N/A	N/A	5	126	131	405	7.0	7.1	N/A	N/A
make-3.76.1	∞	N/A	∞	N/A	N/A	N/A	16	23	39	554	4.0	4.1	N/A	N/A
wget-1.9	∞	N/A	∞	N/A	N/A	N/A	8	181	189	1,098	2.0	2.1	N/A	N/A
screen-4.0.2	∞	N/A	∞	N/A	N/A	N/A	724	6,445	7,169	19,143	38.8	39.7	N/A	N/A
a2ps-4.14	∞	N/A	∞	N/A	N/A	N/A	31	763	794	1,229	2.4	2.6	N/A	N/A
bash-2.05a	∞	N/A	∞	N/A	N/A	N/A	45	362	407	1,875	2.5	2.6	N/A	N/A
lsh-2.0.4	∞	N/A	∞	N/A	N/A	N/A	391	1,162	1,553	4,449	5.4	5.5	N/A	N/A
sendmail-8.13.6	∞	N/A	∞	N/A	N/A	N/A	517	946	1,463	9,881	15.9	16.0	N/A	N/A

Table 3. Performance of octagon analysis: all columns are same as those in Table 2

not have recursion and backward gotos, which enables a very efficient interpretation-based analysis [13], and CGS is not fully flow-sensitive [39]. There are other summary-based approaches [16, 17] for scalable global analysis, which are independent of our abstract interpretation-based approach.

Sparse Analysis Techniques for Dataflow Analysis 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 [36] developed a sparse analysis algorithm for constant propagation and Wegman et al. [40] extended it to conditional constant propagation. Dhamdhere et al. [15] 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 [8, 14, 20, 35], 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 computation 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.

Localization Our framework allows us to design a correct sparse static analysis that exploits more sparsity than localization techniques [29, 33, 37, 41]. 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 [37, 41], higher-order flow analysis [29], and numeric abstract interpretation [33].

BDDs In this paper, we propose another use of Binary Decision Diagram (BDD) [6] in representation of data dependencies of analysis. Most of the previous uses are limited to compact representations of points-to sets in pointer analysis [3, 18, 19].

Acknowledgments

We thank Lucas Brutschy and Yoonseok Ko for their contribution to the early implementation of our analyzer. We thank Deokhwan Kim for useful comments and suggestions. We would also like to thank the anonymous PLDI reviewers for their constructive feedback on this paper. This work was supported by the Engineering Research Center of Excellence Program of Korea Ministry of Education, Science and Technology(MEST) / National Research Foundation of Korea(NRF) (Grant 2011-0000971) and the Brain Korea 21 Project, School of Electrical Engineering and Computer Science, Seoul National University in 2011.

References

[1] X. Allamigeon, W. Godard, and C. Hymans. Static analysis of string manipulations in critical embedded C programs. In *SAS*, 2006.

[2] G. Balakrishnan and T. Reps. Analyzing memory accesses in x86 binary executables. In *CC*, 2004.

[3] M. Berndl, O. Lhoták, F. Qian, L. Hendren, and N. Umanee. Points-to analysis using bdds. In *PLDI*, 2003.

[4] 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 *PLDI*, 2003.

[5] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *Proceedings of the International Conference on Formal Methods in Programming and their Applications*, 1993.

[6] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEETC*, 1986.

[7] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *PLDI*, 1990.

[8] J.-D. Choi, R. Cytron, and J. Ferrante. Automatic construction of sparse data flow evaluation graphs. In *POPL*, 1991.

[9] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, 1977.

[10] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *POPL*, 1979.

[11] P. Cousot and R. Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 1992.

[12] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, 1978.

[13] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, and X. Rival. Why does astrée scale up? *Formal Methods in System Design*, 2009.

[14] R. K. Cytron and J. Ferrante. Efficiently computing ϕ-nodes on-the-fly. *TOPLAS*, 1995.

[15] D. M. Dhamdhere, B. K. Rosen, and F. K. Zadeck. How to analyze large programs efficiently and informatively. In *PLDI*, 1992.

[16] I. Dillig, T. Dillig, and A. Aiken. Sound, complete and scalable path-sensitive analysis. In *PLDI*, 2008.

[17] I. Dillig, T. Dillig, and A. Aiken. Precise reasoning for programs using containers. In *POPL*, 2011.

[18] B. Hardekopf and C. Lin. Semi-sparse flow-sensitive pointer analysis. In *POPL*, 2009.

[19] B. Hardekopf and C. Lin. Flow-sensitive pointer analysis for millions of lines of code. In *CGO*, 2011.

[20] M. Hind and A. Pioli. Assessing the effects of flow-sensitivity on pointer alias analyses. In *SAS*, 1998.

[21] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, 2009.

[22] R. Johnson and K. Pingali. Dependence-based program analysis. In *PLDI*, 1993.

[23] Y. Jung, J. Kim, J. Shin, and K. Yi. Taming false alarms from a domain-unaware C analyzer by a bayesian statistical post analysis. In *SAS*, 2005.

[24] C. Lattner, A. Lenharth, and V. Adve. Making Context-Sensitive Points-to Analysis with Heap Cloning Practical For The Real World. In *PLDI*, 2007.

[25] W. Lee, W. Lee, and K. Yi. Sound non-statistical clustering of static analysis alarms. In *VMCAI*, 2012. To Appear.

[26] L. Li, C. Cifuentes, and N. Keynes. Boosting the performance of flow-sensitive points-to analysis using value flow. In *FSE*, 2011.

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

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

[29] M. Might and O. Shivers. Improving flow analyses via ΓCFA: Abstract garbage collection and counting. In *ICFP*, 2006.

[30] A. Milanova, A. Rountev, and B. G. Ryder. Precise and efficient call graph construction for c programs with function pointers. *Journal of Automated Software Engineering*, 2004.

[31] A. Miné. The Octagon Abstract Domain. *HOSC*, 2006.

[32] H. Oh. Large spurious cycle in global static analyses and its algorithmic mitigation. In *APLAS*, 2009.

[33] H. Oh, L. Brutschy, and K. Yi. Access analysis-based tight localization of abstract memories. In *VMCAI*, 2011.

[34] K. Pingali and G. Bilardi. Optimal control dependence computation and the roman chariots problem. *TOPLAS*, 1997.

[35] G. Ramalingam. On sparse evaluation representations. *Theoretical Computer Science*, 2002.

[36] J. H. Reif and H. R. Lewis. Symbolic evaluation and the global value graph. In *POPL*, 1977.

[37] N. Rinetzky, J. Bauer, T. Reps, M. Sagiv, and R. Wilhelm. A semantics for procedure local heaps and its abstractions. In *POPL*, 2005.

[38] T. B. Tok, S. Z. Guyer, and C. Lin. Efficient flow-sensitive interprocedural data-flow analysis in the presence of pointers. In *CC*, 2006.

[39] A. Venet and G. Brat. Precise and efficient static array bound checking for large embedded c programs. In *PLDI*, 2004.

[40] M. N. Wegman and F. K. Zadeck. Constant propagation with conditional branches. *TOPLAS*, 1991.

[41] H. Yang, O. Lee, J. Berdine, C. Calcagno, B. Cook, D. Distefano, and P. O'Hearn. Scalable shape analysis for systems code. In *CAV*, 2008.

[42] H. Yu, J. Xue, W. Huo, X. Feng, and Z. Zhang. Level by level: making flow- and context-sensitive pointer analysis scalable for millions of lines of code. In *CGO*, 2010.

[43] M. Zitser, D. E. S. Group, and T. Leek. Testing static analysis tools using exploitable buffer overflows from open source code. In *FSE*, 2004.