Points-to Analysis as a System of Linear Equations

Rupesh Nasre, R. Govindarajan

Department of Computer Science and Automation,
Indian Institute of Science, Bangalore, India.
{nasre, govind}@csa.iisc.ernet.in

Abstract. We propose a novel formulation of the points-to analysis as a system of linear equations. With this, the efficiency of the points-to analysis can be significantly improved by leveraging the advances in solution procedures for solving the systems of linear equations. However, such a formulation is non-trivial and becomes challenging due to various facts, namely, multiple pointer indirections, address-of operators and multiple assignments to the same variable. Further, the problem is exacerbated by the need to keep the transformed equations linear. Despite this, we successfully model all the pointer operations. We propose a novel inclusion-based context-sensitive points-to analysis algorithm based on prime factorization, which can model all the pointer operations. Experimental evaluation on SPEC 2000 benchmarks and two large open source programs reveals that our approach is competitive to the state-of-the-art algorithms. With an average memory requirement of mere 21MB, our context-sensitive points-to analysis algorithm analyzes each benchmark in 55 seconds on an average.

1 Introduction

Points-to analysis enables several compiler optimizations and remains an important static analysis technique. Enormous growth of code bases in proprietary and open source software systems demands scalability of static analyses over billions of lines of code. Several points-to analysis algorithms have been proposed in literature that make this research area rich in content [1, 26, 5, 2, 18].

A points-to analysis is a method of statically determining whether two pointers may point to the same location at runtime. The two pointers are then said to be aliases of each other. For analyzing a general purpose C program, it is sufficient to consider all pointer statements of the following forms: address-of assignment ($\mathtt{p} = \&\mathtt{q}$), copy assignment ($\mathtt{p} = \mathtt{q}$), load assignment ($\mathtt{p} = *\mathtt{q}$) and store assignment ($*\mathtt{p} = \mathtt{q}$) [25].

A flow-insensitive analysis ignores the control flow in the program and, in turn, assumes that the statements could be executed in any order. A context-sensitive analysis takes into consideration the calling context of a statement while computing the points-to information. Storing complete context information can exponentially blow up the memory requirement and increase analysis time, making the analysis non-scalable for large programs.

It has been established that flow-sensitivity does not add a significant precision over a flow-insensitive analysis [16]. Therefore, we consider context-sensitive flow-insensitive points-to analysis in this paper.

A flow-insensitive points-to analysis iterates over a set of constraints obtained from points-to statements until it reaches a fixpoint. We observe that this phenomenon is similar in spirit to obtaining a solution to a system of linear equations. Each equation defines a constraint on the feasible solution and a linear solver progressively approaches the final solution in an iterative manner. Similarly, every points-to statement forms a constraint on the feasible points-to information and every iteration of a points-to analysis *refines* the points-to information obtained over the previous iteration. We exploit this similarity to *map* the input source program to a set of linear constraints, solve it using a standard linear equation solver and *unmap* the results to obtain the points-to information. As we show in the next section, a naive approach of converting points-to statements into a linear form faces several challenges due to (i) the distinction between ℓ-value and r-value in points-to statements, (ii) multiple dereferences of a pointer and (iii) the same variable defined in multiple statements. We address these challenges with novel mechanisms based on prime factorization of integers.

Major contributions of this paper are as below.

- A novel representation using prime factorization to store points-to facts.
- We transform points-to constraints into a system of linear equations without affecting precision.
- We show the effectiveness of our approach by comparing it with state-of-the-art context-sensitive algorithms using SPEC 2000 benchmarks and two large open source programs (*httpd* and *sendmail*). On an average, our method computes points-to information in 55 seconds using 21 MB memory proving competitive to other methods.

2 Points-to Analysis

In the following subsection, we describe a simple method to convert a set of points-to statements into a set of linear equations. Using it as a baseline, we discuss several challenges that such a method poses. Consequently, in Section 2.2, we introduce our novel transformation that addresses all the discussed issues and present our points-to analysis algorithm (Section 2.3). Later, we extend it for context-sensitivity using an invocation graph based approach (Section 2.4) and prove its soundness in the next section.

2.1 A First-cut Approach

Consider the set of C statements given in Figure 1(a). Let us define a transformation that translates &$\mathtt{x}$ into $\mathtt{x} - \mathtt{1}$ and $*^{\mathtt{k}}\mathtt{x}$ into $\mathtt{x} + \mathtt{k}$, where $\mathtt{k}$ is the number

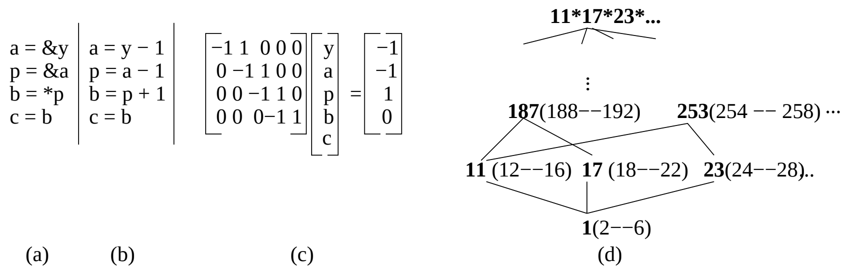

Fig. 1: (a, b, c) Example to illustrate points-to analysis as a system of linear equations. (d) Lattice over the compositions of primes guaranteeing five levels of dereferencing.

of '$*$'s used in dereferencing. Thus, $*\mathtt{x}$ is transformed as $\mathtt{x}+1$, $\mathtt{**x}$ as $\mathtt{x}+2$ and so on. A singleton variable without any operations is copied as it is, thus, $\mathtt{x}$ translates to $\mathtt{x}$. The transformed program now looks like Figure 1(b). This becomes a simple linear system of equations that can be written in matrix form $\mathtt{AX} = \mathtt{B}$ as shown in Figure 1(c).

This small example illustrates several interesting aspects. First, the choices of transformation functions for '$*$' and '&' are not independent, because '$*$' and '&' are complementary operations by language semantics, which should be carried to the linear transformation. Second, selecting $\mathtt{k} = 0$ is not a good choice because we lose information regarding the address-of and dereference operations, resulting in loss of precision in the analysis. Third, every row in matrix $\mathtt{A}$ has at most two $\mathtt{1}$s, i.e., every equation has at most two unknowns. All the entries in matrices $\mathtt{A}$ and $\mathtt{B}$ are $\mathtt{0}$, $\mathtt{1}$ or $-\mathtt{1}$.

Solving the above linear system using a solver yields the following parameterized result: $\mathtt{y} = \mathtt{r}, \mathtt{a} = \mathtt{r} - 1, \mathtt{p} = \mathtt{r} - 2, \mathtt{b} = \mathtt{r} - 1, \mathtt{c} = \mathtt{r} - 1$.

From the values of the variables, we can quickly conclude that $\mathtt{a}$, $\mathtt{b}$ and $\mathtt{c}$ are aliases. Further, since the value of $\mathtt{p}$ is one smaller than that of $\mathtt{a}$, we say that $\mathtt{p}$ points to $\mathtt{a}$, and in the same manner, $\mathtt{a}$ points to $\mathtt{y}$. Thus, our analysis computes all the points-to information obtained using Andersen's analysis, and is thus *sound*: $\mathtt{y} \rightarrow \{\}, \mathtt{a} \rightarrow \{\mathtt{y}\}, \mathtt{p} \rightarrow \{\mathtt{a}\}, \mathtt{b} \rightarrow \{\mathtt{y}\}, \mathtt{c} \rightarrow \{\mathtt{y}\}$.

Next, we discuss certain issues with this approach.

Imprecise analysis. Note that our solver also added a few spurious points-to pairs: $\mathtt{p} \rightarrow \{\mathtt{b}, \mathtt{c}\}$. Therefore, the first-cut approach described above gives an *imprecise* result.

Cyclic dependences. Note that each constraint in the above system of equations is of the form $\mathtt{a_i} - \mathtt{a_j} = \mathtt{b_{ij}}$ where $\mathtt{b_{ij}} \in \{0, 1, -1\}$. We can build a constraint graph $\mathtt{G} = (\mathtt{V}, \mathtt{E}, \mathtt{w})$ where

$\mathtt{V} = \{\mathtt{a_1}, ..., \mathtt{a_n}\} \cup \{\mathtt{a_0}\}$, $\mathtt{w}(\mathtt{a_i}, \mathtt{a_j}) = \mathtt{b_{ij}}a$, $\mathtt{w}(\mathtt{a_0}, \mathtt{a_i}) = 0$ and

$\mathtt{E} = \{(\mathtt{a_i}, \mathtt{a_j}) : \mathtt{a_i} - \mathtt{a_j} = \mathtt{b_{ij}}$ is a constraint$\} \cup \{(\mathtt{a_0}, \mathtt{a_1}), ..., (\mathtt{a_0}, \mathtt{a_n})\}$.

The above linear system has a feasible solution *iff* the corresponding constraint graph has no cycle with negative weight [3]. A linear solver would not output any solution for a system with a cycle. Our algorithm uses appropriate variable renaming that allows a standard linear solver to solve such equations.

Inconsistent equations. The above approach fails for multiple assignments to the same variable. For instance, $\mathtt{a} = \&\mathtt{x}, \mathtt{a} = \&\mathtt{y}$ is a valid program fragment. However, $\mathtt{a} = \mathtt{x} - \mathtt{1}, \mathtt{a} = \mathtt{y} - \mathtt{1}$ does not form a consistent equation system unless $\mathtt{x} = \mathtt{y}$. This issue is discussed in the context of *bug-finding* [12].

Nonlinear system of equations. One way to handle inconsistent equations is to multiply the constraints having the same unknown to generate a non-linear set of equations. Thus, $\mathtt{a} = \&\mathtt{x}$ and $\mathtt{a} = \&\mathtt{y}$ would generate a non-linear constraint $(\mathtt{a} - \mathtt{x} + \mathtt{1})(\mathtt{a} - \mathtt{y} + \mathtt{1}) = \mathtt{0}$. However, non-linear analysis is often more expensive than a linear analysis. Further, maintaining integral solutions across iterations using standard techniques is a difficult task.

Equations versus inequations. The inclusion-based analysis semantics for a points-to statement $\mathtt{a} = \mathtt{b}$ imply points-to-set(a) $\supseteq$ points-to-set(b). Transforming the statement into an equality $\mathtt{a} - \mathtt{b} = \mathtt{0}$ instead of an inequality can be imprecise, as equality in mathematics is bidirectional. It is easy to verify, however, that if a set of constraints contains each *ℓ-value* at most once *and* this holds across iterations of the analysis, the solution sets obtained using inequalities and equalities would be the same. We exploit this observation in our algorithm.

Dereferencing. As per our first-cut approach, transformations of points-to statements $\mathtt{a} = \&\mathtt{b}$ and $*\mathtt{a} = \mathtt{b}$ would be $\mathtt{a} = \mathtt{b} - \mathtt{1}$ and $\mathtt{a} + \mathtt{1} = \mathtt{b}$ respectively. According to the algebraic semantics, the above equations are equivalent, although the two points-to statements have different semantics. This necessitates one to take care of the *store* constraints separately.

2.2 The Modified Approach

We solve the issues with the above approach with a modified mechanism. Our approach is iterative, and in each iteration, it goes through four major steps, viz., preprocessing, solving the linear system of equations, post-processing and evaluating *special* constraints. We illustrate it using the following example.

$\mathtt{a} = \&\mathtt{x}; \mathtt{b} = \&\mathtt{y}; \mathtt{p} = \&\mathtt{a}; \mathtt{c} = *\mathtt{p}; *\mathtt{p} = \mathtt{b}; \mathtt{q} = \mathtt{p}; \mathtt{p} = *\mathtt{p}; \mathtt{a} = \mathtt{b}.$

Pre-processing. First, we move *store* constraints from the set of equations to a set of generative constraints (as they *generate* more linear equations) that are processed specially. We proceed with the remaining non-*store* constraints.

Second, all constraints of the form $\mathtt{v} = \mathtt{e}$ are converted to $\mathtt{v} = \mathtt{v_{i-1}} \oplus \mathtt{e}$. Here, $\mathtt{v_{i-1}}$ is the value of the variable $\mathtt{v}$ obtained in the last iteration. Initially, $\mathtt{v} = \mathtt{v_0} \oplus \mathtt{e}$. This transformation ensures monotonicity required for a flow-insensitive points-to analysis. The operator $\oplus$ would be concretized shortly. $\mathtt{v_0}$ is a constant, since it is already computed from the previous iteration.

Next, we assign unique prime numbers from a select set $\mathcal{P}$ to the right-hand side expression in each address-of constraint. We defer the definition of $\mathcal{P}$ to a later part of this subsection. Let $\&\mathtt{x}, \&\mathtt{y}$ and $\&\mathtt{a}$ be assigned arbitrary prime numbers, say $\&\mathtt{x} = \mathtt{17}; \&\mathtt{y} = \mathtt{29}$; and $\&\mathtt{a} = \mathtt{101}$. The addresses of the remaining variables $(\mathtt{b}, \mathtt{p}, \mathtt{q}, \mathtt{c})$ are assigned a special sentinel value χ. Further, all the variables of the form $\mathtt{v}$ and $\mathtt{v_i}$ are assigned an initial r-value of χ. Thus, $\mathtt{x}, \mathtt{y}, \mathtt{a}, \mathtt{b}, \mathtt{c}, \mathtt{p}, \mathtt{q}$ and $\mathtt{x_0}$, $\mathtt{y_0}$, $\mathtt{a_0}$, $\mathtt{b_0}$, $\mathtt{c_0}$, $\mathtt{p_0}, \mathtt{q_0}$ equal χ. We keep two-way maps of variables to their r-values and addresses. This step is performed only once in the analysis. In the rest of the paper, the term "address of a variable" refers to the prime number assigned to it by our static analysis.

Next, the dereference $*\mathtt{q}$ in every load statement $\mathtt{p} = *\mathtt{q}$ is replaced by expression $\mathtt{q_{i-1}} + \mathtt{1}$ where $\mathtt{i}$ is the current iteration. Therefore, $\mathtt{c} = *\mathtt{p}$ becomes $\mathtt{c} = \mathtt{c_0} \oplus (\mathtt{p_0}{+}1)$ and $\mathtt{p} = *\mathtt{p}$ becomes $\mathtt{p} = \mathtt{p_0} \oplus (\mathtt{p_0} + \mathtt{1})$. Note that by generating different versions of the same variable in this manner, we remove cyclic dependences altogether, since variables $\mathtt{v_i}$'s are not dependent on any other variable as they are never *defined* explicitly in the constraints. The renaming is only symbolic and appears only for exposition purposes. Since values from only the previous iteration are required, we simply make a copy $\mathtt{v_{copy}}$ for each variable $\mathtt{v}$ at the start of each iteration.

Last, we rename multiple occurrences of the same variable as an *ℓ-value* in different constraints to convert it to an SSA-like form. For each such renamed variable $\mathtt{v}'$, we store a constraint of the form $\mathtt{v} = \mathtt{v}'$ in a separate merging constraint set. Thus, assignments to $\mathtt{a}$ in $\mathtt{a} = \mathtt{x_0}$ and $\mathtt{a} = \mathtt{b_0}$ are replaced as $\mathtt{a} = \mathtt{x_0}$ and $\mathtt{a}' = \mathtt{b_0}$ and the constraint $\mathtt{a} = \mathtt{a}'$ is added to the merging constraints set. The constraints now look as follows.

Linear constraints: $\mathtt{a} = \mathtt{a_0} \oplus \&\mathtt{x}; \mathtt{b} = \mathtt{b_0} \oplus \&\mathtt{y}; \mathtt{p} = \mathtt{p_0} \oplus \&\mathtt{a};$
$\mathtt{c} = \mathtt{c_0} \oplus (\mathtt{p_0} + \mathtt{1}); \mathtt{q} = \mathtt{q_0} \oplus \mathtt{p}; \mathtt{p}' = \mathtt{p_0} \oplus (\mathtt{p_0} + \mathtt{1}); \mathtt{a}' = \mathtt{a_0} \oplus \mathtt{b}.$
Generative constraints: $*\mathtt{p} = \mathtt{b}$.
Merging constraints: $\mathtt{a} = \mathtt{a}'; \mathtt{p} = \mathtt{p}'$.

Substituting the r-values and the primes for the addresses of variables, we get
$\mathtt{a} = \chi \oplus \mathtt{17}, \mathtt{b} = \chi \oplus \mathtt{29}, \mathtt{p} = \chi \oplus \mathtt{101}, \mathtt{c} = \chi \oplus (\chi + \mathtt{1}),$
$\mathtt{q} = \chi \oplus \mathtt{p}, \mathtt{p}' = \chi \oplus (\chi + \mathtt{1}), \mathtt{a}' = \chi \oplus \mathtt{b}.$

χ and $\oplus$. We unfold the mystery behind the values of χ and $\oplus$ now. The rationale behind replacing the address of every address-taken variable with a prime number is to have a *non-decomposable* element defining the variable. We make use of *prime factorization* of integers to map a value to the corresponding points-to set. The first trivial but important observation towards this goal is that any pointee of any variable has to appear as address taken in at least one of the constraints. Therefore, the only pointees any pointer can have would exactly be the address-taken variables. Thus, a *composition* $\mathtt{v} = \mathtt{v_i} \oplus \mathtt{v_j} \oplus ...$ of the primes $\mathtt{v_i}, \mathtt{v_j}, ...$ representing address-taken variables defines the pointer $\mathtt{v}$ pointing to all these address-taken variables. The composition is defined by operator $\oplus$ and it defines a lattice over the finite set of all the pointers and pointees (Figure 1(d)). The top element $\top$ defines a composition of all address-taken variables ($\mathtt{v_0} \oplus \mathtt{v_1} \oplus ... \oplus \mathtt{v_n}$) and the bottom element $\bot$ defines the empty set. Since we use prime factorization, $\oplus$ becomes the multiplication operator $\times$ and χ is the identity element, i.e., $\mathtt{1}$. The reason behind using $\oplus$ and χ as placeholders is that it is possible to use an alternative lattice with different $\oplus$ and χ and achieve an equivalent transformation (as long as the equations remain linear). Since every positive integer has a unique prime factorization, we guarantee that the value of a pointer uniquely identifies its pointees. For instance, if $\mathtt{a} \rightarrow \{\mathtt{x}, \mathtt{y}\}$ and $\mathtt{b} \rightarrow \{\mathtt{y}, \mathtt{z}, \mathtt{w}\}$, then we can assign primes to $\&\mathtt{x}, \&\mathtt{y}, \&\mathtt{z}, \&\mathtt{w}$ arbitrarily as $\&\mathtt{x} = \mathtt{11}, \&\mathtt{y} = \mathtt{19}, \&\mathtt{z} = \mathtt{5}, \&\mathtt{w} = \mathtt{3}$ and the values of a and b would be calculated as $\mathtt{a} = \mathtt{11} \times \mathtt{19} = \mathtt{209}$ and $\mathtt{b} = \mathtt{19} \times \mathtt{5} \times \mathtt{3} = \mathtt{285}$. Since, 209 can only be factored as $\mathtt{11} \times \mathtt{19}$ and 285 does not have any other factorization than $\mathtt{19} \times \mathtt{5} \times \mathtt{3}$, we can obtain the points-to sets for pointers a and b from the factors.

Unfortunately, prime factorization is not known to be polynomial [19]. Therefore, for efficiency reasons, our implementation keeps track of the factors explicitly. We use a prime-factor-table for this purpose. The prime-factor-table stores all the prime factors of a value. We initially store all the primes p corresponding to the address-taken variables as $\mathtt{p} = \mathtt{p} \times \mathtt{1}$.

Thus, the system of equations now becomes

Linear constraints: $\mathtt{a} = \mathtt{17}; \mathtt{b} = \mathtt{29}; \mathtt{p} = \mathtt{101}; \mathtt{c} = \mathtt{2}; \mathtt{q} = \mathtt{p}; \mathtt{p}' = \mathtt{2}; \mathtt{a}' = \mathtt{b}$.

Generative constraints: $*\mathtt{p} = \mathtt{b}$. Merging constraints: $\mathtt{a} = \mathtt{a}'; \mathtt{p} = \mathtt{p}'$.

Solving the system. Solving the above system of equations using a standard linear solver gives us the following solution.

$\mathtt{a} = \mathtt{17}, \mathtt{b} = \mathtt{29}, \mathtt{p} = \mathtt{101}, \mathtt{c} = \mathtt{2}, \mathtt{q} = \mathtt{101}, \mathtt{p}' = \mathtt{2}, \mathtt{a}' = \mathtt{29}$.

Post-processing. Interpreting the values in the above solution obtained using a linear solver is straightforward except for those of c and $\mathtt{p}'$ (2 is not chosen to be one of the primes.). In the simple case, a value $\mathtt{v} + \mathtt{k}$ denotes $\mathtt{k}^{\mathtt{th}}$ dereference of v. To find v, our method checks each value ϑ in $(\mathtt{v} + \mathtt{k}), (\mathtt{v} + \mathtt{k} - \mathtt{1}), (\mathtt{v} + \mathtt{k} - \mathtt{2}), ...$ in the prime factor table. For the first ϑ that appears in the prime factor table, $\mathtt{v} = \vartheta$ and $\mathtt{k}' = \mathtt{v} + \mathtt{k} - \vartheta$ represents the level of dereferencing. We obtain the prime factors of ϑ from the table, which would correspond to the addresses of variables, reverse-map the addresses to their corresponding variables, then obtain the r-values of the variables from the map whose prime factors would denote the points-to set we want for expression $\mathtt{v} + \mathtt{k}$. Another level of reverse mapping-mapping would be required for $\mathtt{k} = \mathtt{2}$ and so on. We explain dereferencing method (Algorithm 3) later. Note that since our method can handle only a limited number of dereferences (k), the number of iterations required in the dereferencing step is also limited (and is typically small). Therefore, in the example, the value 2 of the variables c and $\mathtt{p}'$ is represented as $\mathtt{1} + \mathtt{1}$ where the second 1 denotes a dereference and the first 1 is the value of the variable being dereferenced. In this case, since $\mathtt{v} = \mathtt{1}$, which is the sentinel χ, its dereference results in an empty set and thus, both c and $\mathtt{p}'$ are assigned a value of 1.

Selection of primes. In general, a value ϑ may be interpreted as $\mathtt{v}_1 + \mathtt{k}_1$ as well as $\mathtt{v}_2 + \mathtt{k}_2$, if the values $\mathtt{v}_1$ and $\mathtt{v}_2$ happen to be close to each other. To avoid this ambiguity, the ranges $(\mathtt{v}_1 ... \mathtt{v}_1 + \mathtt{k})$ and $(\mathtt{v}_2 ... \mathtt{v}_2 + \mathtt{k})$ must be non-overlapping for a fixed k. This is accomplished by careful selection of the prime numbers representing the address-taken variables. Our analysis selects primes offline and guarantees that a certain $\mathtt{k}$ number of dereferences will never overlap with one another. In fact, we define our method for upto $\mathtt{k}$ levels of dereferencing. Our prime number set $\mathcal{P}$ is also defined for a specific $\mathtt{k}$. More specifically, for any prime numbers $\mathtt{p} \in \mathcal{P}$, the products of any one[1] or more $\mathtt{p}$ are distance more than $\mathtt{k}$ apart. Thus, $|\mathtt{p_i} - \mathtt{p_j}| > \mathtt{k}$ and $|\mathtt{p_i} * \mathtt{p_j} - \mathtt{p_l} * \mathtt{p_m}| > \mathtt{k}$ and $|\mathtt{p_i} * \mathtt{p_j} * \mathtt{p_l} - \mathtt{p_m} * \mathtt{p_n} * \mathtt{p_o}| > \mathtt{k}$ and so on. Note that $\mathcal{P}$ needs to be computed only once, offline. Also, typically, the number of dereferences in real-world programs is very small (< 5). The lattice for the prime number set $\mathcal{P}$ chosen for $k = 5$ is shown in Figure 1(d). Here, the bracketed values, e.g., (12,13,...,16) denote possible dereferencings of a variable which is assigned the value 11.

The next step is to merge the points-to sets of renamed variables, i.e., evaluating merging constraints. This changes $\mathtt{a}$ and $\mathtt{p}$ as $\mathtt{a} = 17 \times 29$ and $\mathtt{p} = 101 \times 1 = 101$.

After merging, we discard all the renamed variables.

Thus, at the end of the first iteration, the points-to set contained in the values is: $x \rightarrow \{\}, y \rightarrow \{\}, a \rightarrow \{x, y\}, b \rightarrow \{y\}, c \rightarrow \{\}, p \rightarrow \{a\}, q \rightarrow \{a\}$.

Evaluating special constraints: The final step is to evaluate the generative constraints and generate more linear constraints. In the first iteration, the store constraint $*\mathtt{p} = \mathtt{b}$ generates the copy constraint $\mathtt{a} = \mathtt{b}$ which already exists in the system. Thus, no new linear constraints are generated. Note that the generative constraints set is retained as more constraints may need to be added in further iterations. At the end of each iteration, our algorithm checks if any variable value is changed since the last iteration. If yes, then another iteration is required.

Subsequent iterations. The constraints, ready for iteration number two, are

Linear constraints: $\mathtt{a} = \mathtt{a_1} \times \&\mathtt{x}; \mathtt{b} = \mathtt{b_1} \times \&\mathtt{y}; \mathtt{p} = \mathtt{p_1} \times \&\mathtt{a};$
$\mathtt{c} = \mathtt{c_1} \times (\mathtt{p_1} + 1); \mathtt{q} = \mathtt{q_1} \times \mathtt{p}; \mathtt{p}' = \mathtt{p_1} \times (\mathtt{p_1} + 1); \mathtt{a}' = \mathtt{a_1} \times \mathtt{b}.$

Generative constraints: $*\mathtt{p} = \mathtt{b}$.

Merging constraints: $\mathtt{a} = \mathtt{a}'; \mathtt{p} = \mathtt{p}'$.

Here, $\mathtt{v_1}$ is the value of the variable $\mathtt{v}$ obtained in iteration 1. Thus the constraints to be solved by the linear solver are:

$\mathtt{a} = 17 \times 29 \times 17, \mathtt{b} = 29 \times 29, \mathtt{p} = 101 \times 101, \mathtt{c} = 101 + 1, \mathtt{q} = 101 \times \mathtt{p},$
$\mathtt{p}' = 101 \times (101 + 1), \mathtt{a}' = 17 \times 29 \times \mathtt{b}.$

The linear solver offers the following solution.

$\mathtt{a} = 17 \times 29 \times 17, \mathtt{b} = 29 \times 29, \mathtt{p} = 101 \times 101, \mathtt{c} = 102, \mathtt{q} = 101 \times 101 \times 101,$
$\mathtt{p}' = 101 \times 102, \mathtt{a}' = 17 \times 29 \times 29 \times 29.$

The solver returns each value as an integer (e.g., 8381) and not as factors (e.g., $17 \times 29 \times 17$). Our analysis finds the prime factors using the prime-factor-table.

Post-processing over the values starts with *pruning the powers* of the values containing repeated prime factors as they do not add any additional points-to information to the solution. Thus,

$\mathtt{a} = 17 \times 29, \mathtt{b} = 29, \mathtt{p} = 101, \mathtt{c} = 102, \mathtt{q} = 101, \mathtt{p}' = 101 \times 102, \mathtt{a}' = 17 \times 29.$

[1] product of one number is the number itself.

The next step is to dereference variables to obtain their points-to sets. Since, 17, 29, and 101 are directly available in prime factor table, the values of $\mathtt{a}, \mathtt{b}, \mathtt{p}, \mathtt{q}, \mathtt{a}'$ do not require a dereference. In case of c, 102 is not present in prime factor table, so the next value 101 is searched for, which indeed is present in the table. Thus, $(102 - 101)$ dereferences are done on 101. Further, 101 reverse-maps to &a and a forward-maps to the r-value 17×29. Hence $\mathtt{c} = 17 \times 29$, suggesting that c points to x and y.

The value of $\mathtt{p}'$ is an interesting case. The solution returned by the solver (10302) is neither a prime number, nor a short offset from the product of primes. Rather, it is a product of a prime and a short offset of the prime. We know that it is the value of variable $\mathtt{p}'$ whose original value was $\mathtt{p}_1 = 101$. This original value is used to find out the points-to set contained in value 10302. To achieve this, our method (always) divides the value obtained by the solver by the original value of the variable. Thus, we get $10302/101 = 102$. Our method then applies the dereferencing algorithm on 102 to get its points-to set, which, as explained above for c, computes the value 17×29 corresponding to the points-to set $\{\mathtt{x}, \mathtt{y}\}$. This updates $\mathtt{p}'$ to $101 \times 17 \times 29$.

It should be emphasized that our method never checks a number for primality. After prime-factor-table is initially populated with statically defined primes as a multiple of self and unity, a lookup in the table suffices for primality testing.

The next step is to evaluate the merging set to obtain the following.

$\mathtt{a} = 17 \times 29 \times 17 \times 29, \mathtt{p} = 101 \times (101 \times 17 \times 29)$,

which on pruning gives $\mathtt{a} = 17 \times 29, \mathtt{p} = 17 \times 29 \times 101$.

Thus, at the end of the second iteration, the points-to sets are

$\mathtt{x} \rightarrow \{\}, \mathtt{y} \rightarrow \{\}, \mathtt{a} \rightarrow \{\mathtt{x}, \mathtt{y}\}, \mathtt{b} \rightarrow \{\mathtt{y}\}, \mathtt{c} \rightarrow \{\mathtt{x}, \mathtt{y}\}, \mathtt{p} \rightarrow \{\mathtt{a}, \mathtt{x}, \mathtt{y}\}, \mathtt{q} \rightarrow \{\mathtt{a}\}$.

Executing the final step of evaluating the generative constraints, we obtain two additional linear constraints: $\mathtt{x} = \mathtt{b}, \mathtt{y} = \mathtt{b}$.

Following the same process, at the end of the third iteration we get $\mathtt{x} = 29, \mathtt{y} = 29$, $\mathtt{a} = 17 \times 29, \mathtt{b} = 29, \mathtt{c} = 17 \times 29, \mathtt{p} = 17 \times 29 \times 101, \mathtt{q} = 17 \times 29 \times 101$ which corresponds to the points-to set

$\mathtt{x} \rightarrow \{\mathtt{y}\}, \mathtt{y} \rightarrow \{\mathtt{y}\}, \mathtt{a} \rightarrow \{\mathtt{x}, \mathtt{y}\}, \mathtt{b} \rightarrow \{\mathtt{y}\}, \mathtt{c} \rightarrow \{\mathtt{x}, \mathtt{y}\}, \mathtt{p} \rightarrow \{\mathtt{a}, \mathtt{x}, \mathtt{y}\}, \mathtt{q} \rightarrow \{\mathtt{a}, \mathtt{x}, \mathtt{y}\}$

and no new linear constraints are added.

The fourth iteration makes no change to the values of the variables suggesting that a fixpoint solution is reached.

2.3 The Algorithm

Our points-to analysis is outlined in Algorithm 1. To avoid clutter, we have removed the details of pruning of powers, which is straightforward. The analysis assumes availability of the set of constraints C and the set of variables V used in C. An important data structure is the prime-factor-table which is implemented as a hash-table mapping a key to a set of prime numbers that form the factors of the key. Insertion of the tuple ($\mathtt{a} \times \mathtt{b}$, a, b) assumes existence of a and b in the table (our analysis guarantees that) if a or b is not unity, and is done by combining the prime factors for a and $\mathtt{b} \in \mathcal{P}$ from the table.

Algorithm 1 Points-to analysis as a system of equations.

```
Require: set C of points-to constraints, set V of variables
Ensure: each variable in V has a value indicating its points-to set
  for all v ∈ V do
    v = 1
  end for
  for each constraint c in C do
    if c is an address-of constraint a = &b then
      address-of(b) = nextprime()
      prime-factor-table.insert(a× address-of(b), a, address-of(b))
      a = a× address-of(b);
      C.remove(c)
    else if c is a store constraint *a = b then
      generative-constraints.add(c)
      C.remove(c)
    else if c is a load constraint a = *b then
      c = constraint(a = b + 1)
    end if
  end for

  repeat
    for all v ∈ V do
      v_copy = v
    end for
    for all c ∈ C of the form v = e do
      renamed = defined(v)
      if renamed == 0 then
        c = constraint(v = v_copy × e)
      else
        c = constraint(v^renamed = v_copy × e)
        merge-constraints.add(constraint(v = v^renamed))
      end if
      ++defined(v)
    end for
    V = linear-solve(C)
    for all v ∈ V do
      v = interpret(v, v_copy, V, prime-factor-table) {Algo. 3}
    end for
    for all c ∈ merging-constraints of the form v_1 = v_2 do
      prime-factor-table.insert(v_1 × v_2, v_1, v_2)
      v_1 = v_1 × v_2
    end for
    for all c ∈ generative-constraints of the form *a = b do
      S = get-points-to(a, prime-factor-table) {Algo. 2}
      for all s ∈ S do
        C.add(constraint(s = b))
      end for
    end for
  until V == set(v_copy)
```

Algorithm 2 Finding points-to set.

```
Require: Value v, prime-factor-table
  S = {}
  P = get-prime-factors(v, prime-factor-table)
  for all p ∈ P do
    S = S∪ reverse-lvalue(p)
  end for
  return S
```

Algorithm 3 Interpreting values.

```
Require: Value v, Value v_copy, set of variables V, prime-factor-table
  if v == 1 then
    return v
  else if v ∈ prime-factor-table then
    return v
  else if v/v_copy ∈ prime-factor-table then
    prime-factor-table.insert(v, v_copy, v/v_copy)
    return v
  else
    v = v/v_copy
    k = 1
    while (v − k) ∉ prime-factor-table do
      ++k
    end while
    v = (v − k)
    for i = 1 to k do
      S = get-points-to(v, prime-factor-table) {Algo. 2}
      prod = 1
      for all s ∈ S and s ≠ 1 do
        r = reverse-lvalue(s)
        prod = prod × r
        prime-factor-table.insert(prod, prod/r, r)
      end for
      v = prod
    end for
  end if
  return v × v_copy
```

The important step of interpreting the solution is done in Lines 33–35 using Algorithm 3. The algorithm checks for an entry of a variable's value in the prime-factor-table to see if it is a valid composition of primes. Both Algorithms 1 and 3 make use of Algorithm 2 for computing points-to set of a pointer. It finds the prime factors of the r-value of the pointer (Line 2) followed by an unmapping from the primes to the corresponding variables (Line 4).

At the end of Algorithm 1, the r-values of variables in `V` denote their computed points-to sets. `C` is no longer required.

Implementation issue. Similar to other works on finding linear relationships among program variables [4, 22], our analysis suffers from the issue of large values. Since we store points-to set as a multiplication of primes, the resulting values quickly go beyond the integer range of 64 bits. Hence we are required to use an integer library (GNU MP Bignum Library [13]) that emulates integer arithmetic over large unsigned integers.

2.4 Context-Sensitive Analysis

We extend Algorithm 1 for context-sensitivity using an invocation graph based approach [7]. It enables us to disallow non-realizable interprocedural execution paths. We handle recursion by iterating over the cyclic call-chain and computing a fixpoint of the points-to tuples. Our analysis is field-insensitive, i.e., we assume that any reference to a field inside a structure is to the whole structure.

3 Soundness and Precision

Soundness implies that our algorithm identifies every points-to fact identified by an inclusion-based analysis. Precision implies that our analysis does not compute a (proper) superset of the information compared to an inclusion-based analysis.

We first prove three properties of the solution to the system of linear equations.

Property 1: *Feasibility.*
Proof: By renaming the variable occurring in multiple assignments as $\mathtt{a}', \mathtt{a}'', ...$, we guarantee at most one definition per variable. Further, all constants involved in the equations are positive. Thus, there is no negative weight cycle in the constraint graph — in fact, there is neither a cycle nor a negative weight. This guarantees a feasible solution to the system.

Property 2: *Uniqueness.*
Proof: A variable attains a unique value if it is defined exactly once. Our initialization of all the variables to the value of $\chi = \mathtt{1}$ followed by the variable renaming assigns a unique value to each variable. For instance, let the system have only one constraint: $\mathtt{a} = \mathtt{b}$. In general, this system has infinite number of solutions because b is not restricted to any value. In our analysis, we initialize both (a and b) to 1.

Property 3: *Integrality.*
Proof: We are solving (and not optimizing) a system of equations that involves only addition, subtraction and multiplication over positive integers ($\mathtt{v_i}$ and constants). Further, each equation is of the form $\mathtt{v} = \mathtt{v_i} \times \mathtt{e}$ where both $\mathtt{v_i}$ and e are integral. Hence the system guarantees an integral solution.

We now prove soundness and precision of our analysis.

Lemma 1.1: *The analysis in Algorithm 1 is monotonic.*
Proof: Every address-taken variable is represented using a distinct prime number. Second, every positive integer has a unique prime factorization. Thus, our representation does not lead to a precision loss. Multiplication by an integer corresponds to including addresses represented by its prime factors. Division by an integer maps to the removal of the unique addresses represented by its prime factors. Multiplying the equations by $\mathtt{v_{copy}}$ in iteration i (Lines 25 and 27) thus ensures encompassing the points-to set computed in iteration $\mathtt{i} - \mathtt{1}$. The only division is done in Algorithm 3 (Line 9) (which is guaranteed to be without a remainder and hence no information loss), but the product is restored in Line 26. Thus, no points-to information is ever killed and we guarantee monotonicity.

Lemma 1.2: *Address-of statements are transformed safely.*
Proof: The effect of address-of statement is computed by assigning the prime number of the address-taken variable to the r-value of the destination variable

(Lines 5–9 of Algorithm 1).

Lemma 1.3: *Variable renaming is sound.*
Proof: Per constraint based semantics, statements $\mathtt{a} = \mathtt{e_1}, \mathtt{a} = \mathtt{e_2}, ..., \mathtt{a} = \mathtt{e_n}$ mean $\mathtt{a} \supseteq \mathtt{e_1}, \mathtt{a} \supseteq \mathtt{e_2}, ..., \mathtt{a} \supseteq \mathtt{e_n}$ which implies $\mathtt{a} \supseteq (\mathtt{e_1} \cup \mathtt{e_2} \cup ... \cup \mathtt{e_n})$. Renaming gives $\mathtt{a}' = \mathtt{e_1}, \mathtt{a}'' = \mathtt{e_2}, ..., \mathtt{a}'^{\mathtt{n}} = \mathtt{e_n}$ adds constraints $\mathtt{a}' \supseteq \mathtt{e_1}, \mathtt{a}'' \supseteq \mathtt{e_2}, ..., \mathtt{a}'^{\mathtt{n}} \supseteq \mathtt{e_n}$ which implies $(\mathtt{a}' \cup \mathtt{a}'' \cup ... \cup \mathtt{a}'^{\mathtt{n}}) \supseteq (\mathtt{e_1} \cup \mathtt{e_2} \cup ... \cup \mathtt{e_n})$. Merging the variables as $\mathtt{a} = \mathtt{a}', \mathtt{a} = \mathtt{a}'', ..., \mathtt{a} = \mathtt{a}'^{\mathtt{n}}$ adds constraint $\mathtt{a} \supseteq (\mathtt{a}' \cup \mathtt{a}'' \cup ... \cup \mathtt{a}'^{\mathtt{n}})$. By transitivity of $\supseteq$, $\mathtt{a} \supseteq (\mathtt{e_1} \cup \mathtt{e_2} \cup ... \cup \mathtt{e_n})$. Thus, variable renaming is sound.

Corollary 1.1: *Copy statements are transformed safely.*

Lemma 1.4: *Store statements are transformed safely.*
Proof: We define a points-to fact $\mathtt{f}$ to be *realizable* by a constraint $\mathtt{c}$ if evaluation of $\mathtt{c}$ *may* result in the computation of $\mathtt{f}$. $\mathtt{f}$ is *strictly-realizable* by $\mathtt{c}$ if for the computation of $\mathtt{f}$, evaluation of $\mathtt{c}$ is a *must*. For the sake of contradiction, assume that there is a valid points-to fact $\mathtt{a} \rightarrow \{\mathtt{x}\}$ that is strictly-realizable by the store constraint $\mathtt{a} = *\mathtt{p}$ and that does not get computed in our algorithm. Since the store statement, added to the generative constraint set, adds copy constraints $\mathtt{a} = \mathtt{b}, \mathtt{a} = \mathtt{c}, \mathtt{a} = \mathtt{d}, ...$ where $\mathtt{p} \rightarrow \{\mathtt{b}, \mathtt{c}, \mathtt{d}, ...\}$ at the end of an iteration after points-to information computation and interpretation is done, the contradiction means that $\mathtt{x} \notin (*\mathtt{b} \cup *\mathtt{c} \cup *\mathtt{d} \cup ...)$. This implies, $(\mathtt{x} \notin *\mathtt{b}) \wedge (\mathtt{x} \notin *\mathtt{c}) \wedge (\mathtt{x} \notin *\mathtt{d}) \wedge ....$ This suggests that the pointee $\mathtt{x}$ propagates to the pointer $\mathtt{a}$ via some other constraints, implying that the points-to fact $\mathtt{a} \rightarrow \{\mathtt{x}\}$ is not strictly-realizable by $\mathtt{a} = *\mathtt{p}$, contradicting our hypothesis.

Lemma 1.5: *Decomposing an r-value of* $\mathtt{p}$ *into its prime factors, unmapping the addresses as the primes to the corresponding variables, and mapping the variables to their r-values corresponds to a pointer dereference* $*\mathtt{p}$.

Lemma 1.6: *Load statements are transformed safely.*
Proof: For a $\mathtt{k}$-level dereference $*^{\mathtt{k}}\mathtt{v}$ in a load statement, every '*' adds 1 to $\mathtt{v}$'s r-value. Thus, for a unique $\mathtt{v} + \mathtt{k}$, the evaluation involves $\mathtt{k}$ dereferences. Lines 15–24 of Algorithm 3 do exactly this, and by Lemmas 1.3 and 1.5, load statements compute a safe superset.

Theorem 1: *The analysis is sound with respect to an inclusion-based analysis for a dereferencing level k.*
Proof: From Lemma 1.1—1.6 and Corollary 1.1.

Lemma 2.1: *Address-of statements are transformed precisely.*
Proof: Address of every address-taken variable is represented using a distinct prime value. Further, in Lines 5–9 of Algorithm 1, for every address-of statement $\mathtt{a} = \&\mathtt{b}$, the only primes that $\mathtt{a}$ is multiplied with are the addresses of $\mathtt{b}$s.

Lemma 2.2: *Variable renaming is precise.*
Proof: Since each variable is defined only once and by making use of Lemma 1.3 $\texttt{a} = (\texttt{e}_1 \cup \texttt{e}_2 \cup ... \cup \texttt{e}_\texttt{n})$.

Lemma 2.3: *Copy statements are transformed precisely.*
Proof: From Lemma 2.2 and since for a transformed copy statement $\texttt{a} = \texttt{a}_{\texttt{copy}} \times \texttt{b}$, only the primes computed as the points-to set of a so far (i.e., $\texttt{a}_{\texttt{copy}}$) and those of b are included. This inclusion is guaranteed to be unique due to the uniqueness of prime factorization. Thus, a does not point to any spurious variable address.

Lemma 2.4: *Store statements are transformed precisely.*
Proof: For the sake of contradiction, assume that a points-to fact $\texttt{a} \rightarrow \{\texttt{x}\}$ is computed spuriously by evaluating a store constraint $\texttt{a} = *\texttt{p}$ in Algorithm 1. This means at least one of the following copy constraints computed the fact: $\texttt{a} = \texttt{b}, \texttt{a} = \texttt{c}, \texttt{a} = \texttt{d}, ...$ where $\texttt{p} \rightarrow \{\texttt{b}, \texttt{c}, \texttt{d}, ...\}$. Thus, at least one of the copy constraints is imprecise. However, Lemma 2.3 falsifies the claim.

Lemma 2.5: *Load statements are transformed precisely.*
Proof: Number of dereferences denoted by $\texttt{v} + \texttt{k}$ is the same as that denoted by $*^{\texttt{k}}\texttt{v}$. By Lemma 1.5 and 2.3 and by the observation that Algorithm 3 does not include any extra pointee in the final dereference set.

Theorem 2: *The analysis is precise with respect to an inclusion-based analysis for a dereferencing level k.*
Proof: From Lemma 2.1–2.5.

Theorem 3: *Our analysis computes the same information as an inclusion-based points-to analysis.*
Proof: Immediate from Theorems 1 and 2.

4 Experimental Evaluation

We evaluate the effectiveness of our approach using 16 SPEC C/C++ benchmarks and two large open source programs, namely *httpd* and *sendmail*. For solving equations, we use C++ language extension of CPLEX® solver from IBM ILOG toolset [17]. We compare our approach, referred to as *linear*, with the following implementations.

- *anders*: This is the base Andersen's algorithm[1] that uses a simple iterative procedure over the points-to constraints to reach a fixpoint solution. The underlying data structure used is a sorted vector of pointees per pointer. We extend it for context-sensitivity.
- *bloom*: The bloom filter method uses an approximate representation for storing both the points-to facts and the context information using a bloom filter. As this representation results in false-positives, the method is approximate and introduces some loss in precision. For our experiments, we use the

medium configuration [23] which results in roughly 2% of precision loss for the chosen benchmarks.

– *bdd*: This is the *Lazy Cycle Detection*(LCD) algorithm implemented using Binary Decision Diagrams (BDD) from Hardekopf and Lin [14]. We extend it for context-sensitivity.

Analysis time. The analysis times in seconds required for each benchmark by different methods are given in Table 1. The analysis time is composed of reading an input points-to constraints file, applying the analysis over the constraints and computing the final points-to information as a fixpoint.

From Table 1, we observe that *anders* goes out of memory (*OOM*) for three benchmarks: *gcc*, *perlbmk* and *vortex*. For these three benchmarks *linear* obtains the points-to information in 1–3 minutes. Further comparing the analysis time of *anders*, *bdd* and *bloom* with those of *linear*, we find that *linear* takes considerably less time for most of the benchmarks. The average analysis time per benchmark is lower for *linear* by a factor of 20 when compared to *bloom* and by 30 when compared to *bdd*. Only in the case of *sendmail, mesa, twolf* and *ammp*, the analysis time of *bloom* is significantly better. It should be noted here that *bloom* has 2% precision loss in these applications [23] compared to *anders*, *bdd* and *linear*. Last, the analysis time of *linear* is 1–2 orders of magnitude smaller than *anders*, *bdd* or *bloom*, especially for large benchmarks (*gcc, perlbmk, vortex* and *eon*). We believe that the analysis time of *linear* can be further improved by taking advantage of sharing of tasks across iterations and by exploiting properties of simple linear equations in the linear solver.

Memory. Memory requirement in KB for the benchmarks is given in Table 1. *anders* goes out of memory for three benchmarks: *gcc*, *perlbmk* and *vortex*, suggesting a need for a scalable points-to analysis. *bloom*, *bdd* and *linear* successfully complete on all benchmarks. Similar to the analysis time, our approach *linear* outperforms *anders* and *bloom* in memory requirement especially for large benchmarks. The *bdd* method, which is known for its space efficiency, uses the minimum amount of memory. On an average, *linear* consumes 21MB requiring maximum 69MB for *gcc*. This is comparable to *bdd*'s average memory requirement of 12MB and maximum of 23MB for *gcc*. This small memory requirement is a key aspect that allows our method to scale better with program size.

Query time. We measured the amount of time required to answer an alias query *alias(p, q)*. The answer is a boolean value depending upon whether pointers p and q have any common pointee. *linear* uses a GCD-based algorithm to answer the query. If GCD(p, q) is 1, the pointers do not alias; otherwise, they alias. *anders* uses a sorted vector of pointees per pointer that needs to be traversed to find a common pointee. We used a set of ${}^{n}P_2$ queries over the set of all n pointers in the benchmark programs. Since it simply involves a small number of number-crunching operations, *linear* outperforms *anders* (offset by the cost of emulating large integer arithmetic). We found that the average query time for *linear* is 0.85ms compared to 1.496ms for *anders*.

Benchmark	KLOC	Time(sec)				Memory(KB)			
		anders	bloom	bdd	linear	anders	bloom	bdd	linear
gcc	222.185	OOM	10237.7	17411.208	196.62	OOM	113577	23776	68492
httpd	125.877	17.45	52.79	47.399	76.5	225513	48036	12656	27108
sendmail	113.264	5.96	25.35	117.528	84.76	197383	49455	14320	27940
perlbmk	81.442	OOM	2632.04	5879.913	101.69	OOM	54008	17628	29864
gap	71.367	144.18	152.1	330.233	89.53	97863	31786	11116	22784
vortex	67.216	OOM	1998.5	4725.745	68.32	OOM	23486	16248	18420
mesa	59.255	1.47	10.04	21.732	58.25	8261	20702	15900	18680
crafty	20.657	20.47	46.9	154.983	45.79	15986	4095	7620	16888
twolf	20.461	0.60	5.13	27.375	23.96	1594	12656	9280	15920
vpr	17.731	29.70	88.83	199.510	47.82	50210	8901	10252	10612
eon	17.679	231.17	1241.6	2391.831	106.47	385284	87814	26864	38908
ammp	13.486	1.12	15.19	54.648	19.59	5844	5746	9964	9976
parser	11.394	55.36	145.78	618.337	55.22	121588	16201	12888	14016
gzip	8.618	0.35	1.81	6.533	2.1	1447	1205	8232	11868
bzip2	4.650	0.15	1.35	4.703	1.62	519	878	7116	10244
mcf	2.414	0.11	5.04	32.049	3.4	220	1413	6192	8336
equake	1.515	0.22	1.1	4.054	0.92	161	1494	6288	12992
art	1.272	0.17	2.4	7.678	1.26	42	637	6144	9756
average	—	—	925.76	1779.75	54.66	—	26783	12360	20711

Table 1: Time(seconds) and memory(KB) required for context-sensitive analysis.

5 Related Work

The area of points-to analysis is rich in literature. See [16] for a survey. We mention only the most relevant related work in this section.

Points-to analysis. Most scalable algorithms proposed are based on unification [26][10]. Steensgaard[26] proposed an almost linear time algorithm that has been shown to scale to millions of lines of programs. However, precision of unification based approaches has always been an issue. Inclusion based approaches [1] that work on subsumption of points-to sets rather than a bidirectional similarity offer a better precision at the cost of theoretically cubic complexity. Several techniques [2][15][21][27] have been proposed to improve upon the original work by Andersen. [2] extracts similarity across points-to sets while [27] exploits similarity across contexts to make brilliant use of Binary Decision Diagrams to store information in a succinct manner. The idea of *bootstrapping* [18] first reduces the problem by partitioning the set of pointers into disjoint alias sets using a fast and less precise algorithm (e.g., [26]) and later running more precise analysis on each of the partitions. To address the analysis cost of a completely context-sensitive analysis, approximate representations were introduced to trade off precision for scalability. [5] proposed *one level flow*, [20] unified contexts, while [23] hashed contexts to alleviate the need to store complete context information. Various enhancements have also been made for the inclusion-based analyses: online cycle elimination [9] to break dependence cycles on the fly and offline variable substitution [24] to reduce the number of pointers tracked during the analysis.

Program analysis using linear algebra. An important use of linear algebra in program analysis has been to compute affine relations among program variables [22]. [4] applied abstract interpretation for discovering equality or inequality constraints among program variables. [11] proposed an SML based solver for computing a partial approximate solution for a general system of equations used in logic programs. Another area where analyses based on linear systems has been used is in finding security vulnerabilities. [12] proposed a context-sensitive light-weight analysis modeling string manipulations as a linear program to detect buffer overrun vulnerabilities. [6] presented a tool CSSV to find string manipulation errors. It converts a program written in a restricted subset of C into an integer program with assertions. A violation of an assertion signals a possible vulnerability. Recently, [8] proposed Newtonian Program Analysis as a generic method to solve iterative program analyses using Newton's method.

6 Conclusion

In this paper, we proposed a novel approach to transform a set of points-to constraints into a system of linear equations using prime factorization. We overcome the technical challenges by partitioning our inclusion-based analysis into a linear solver phase and a post-processing phase that interprets the resulting values and updates points-to information accordingly. The novel way of representing points-to information as a composition of primes allows us to keep the equations linear in every iteration. We show that our analysis is sound and precise with respect to an inclusion-based analysis for a fixed dereference level. Using a set of 16 SPEC 2000 benchmarks and two large open source programs, we show that our approach is not only feasible, but is also competitive to the state of the art solvers. More than the performance numbers reported here, the main contribution of this paper is the novel formulation of points-to analysis as a linear system based on prime factorization. In future, we would like to apply enhancements proposed for linear systems to our analysis and improve the analysis time.

References

1. Lars Ole Andersen. Program analysis and specialization for the C programming language. In PhD Thesis, DIKU, University of Copenhagen, 1994.
2. Marc Berndl, Ondrej Lhoták, Feng Qian, Laurie Hendren, and Navindra Umanee. Points-to analysis using BDDs. In PLDI, pages 103–114, 2003.
3. Thomas H. Cormen, Charles E. Leiserson, and Ronald L. Rivest. Introduction to algorithms. In McGraw Hill.
4. Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In POPL, pages 84–96, 1978.
5. Manuvir Das. Unification-based pointer analysis with directional assignments. In PLDI, pages 35–46, 2000.

6. Nurit Dor, Michael Rodeh, and Mooly Sagiv. Cssv: towards a realistic tool for statically detecting all buffer overflows in c. PLDI, 2003.
7. Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In PLDI, pages 242–256, 1994.
8. Javier Esparza, Stefan Kiefer, and Luttenberger Michael. Newtonian program analysis. In ICALP, 2008.
9. Manuel Fähndrich, Jeffrey S. Foster, Zhendong Su, and Alexander Aiken. Partial online cycle elimination in inclusion constraint graphs. In PLDI, 1998.
10. Manuel Fähndrich, Jakob Rehof, and Manuvir Das. Scalable context-sensitive flow analysis using instantiation constraints. In PLDI, 2000.
11. Christian Fecht and Helmut Seidl. An even faster solver for general systems of equations. In SAS, pages 189–204, 1996.
12. Vinod Ganapathy, Somesh Jha, David Chandler, David Melski, and David Vitek. Buffer overrun detection using linear programming and static analysis. In CCS, pages 345–354, 2003.
13. GNU-MP-Integer-Library. http://gmplib.org/.
14. Ben Hardekopf and Calvin Lin. The ant and the grasshopper: fast and accurate pointer analysis for millions of lines of code. In PLDI, pages 290–299, 2007.
15. Nevin Heintze and Olivier Tardieu. Ultra-fast aliasing analysis using CLA: a million lines of C code in a second. In PLDI, pages 254–263, 2001.
16. Michael Hind and Anthony Pioli. Which pointer analysis should i use? In ISSTA, pages 113–123, 2000.
17. ILOG-Toolkit. http://www.ilog.com/.
18. Vineet Kahlon. Bootstrapping: a technique for scalable flow and context-sensitive pointer alias analysis. In PLDI, pages 249–259, 2008.
19. Donald Knuth. The Art of Computer Programming, Volume 2: Seminumerical Algorithms. In Addison-Wesley, 1997.
20. Chris Lattner, Andrew Lenharth, and Vikram Adve. Making context-sensitive points-to analysis with heap cloning practical for the real world. In PLDI, pages 278–289, 2007.
21. O. Lhotak and L. Hendren. Scaling Java points-to analysis using spark. In CC, 2003.
22. Markus Müller-Olm and Helmut Seidl. Precise interprocedural analysis through linear algebra. In POPL, pages 330–341, 2004.
23. Rupesh Nasre, Kaushik Rajan, Govindarajan Ramaswamy, and Uday P. Khedker. Scalable context-sensitive points-to analysis using multi-dimensional bloom filters. In APLAS, 2009.
24. Atanas Rountev and Satish Chandra. Off-line variable substitution for scaling points-to analysis. In PLDI, pages 47–56, 2000.
25. Radu Rugina and Martin Rinard. Pointer analysis for multithreaded programs. In PLDI, pages 77–90, 1999.
26. Bjarne Steensgaard. Points-to analysis in almost linear time. In POPL, pages 32–41, 1996.
27. John Whaley and Monica S. Lam. An efficient inclusion-based points-to analysis for strictly-typed languages. In SAS, 2002.