Off-line Variable Substitution for Scaling Points-to Analysis

Atanas Rountev
Department of Computer Science
Rutgers University
rountev@cs.rutgers.edu

Satish Chandra
Bell Laboratories
Lucent Technologies
chandra@research.bell-labs.com

Abstract

Most compiler optimizations and software productivity tools rely on information about the effects of pointer dereferences in a program. The purpose of points-to analysis is to compute this information safely, and as accurately as is practical. Unfortunately, accurate points-to information is difficult to obtain for large programs, because the time and space requirements of the analysis become prohibitive.

We consider the problem of scaling flow- and context-insensitive points-to analysis to large programs, perhaps containing hundreds of thousands of lines of code. Our approach is based on a *variable substitution* transformation, which is performed off-line, i.e., before a standard points-to analysis is performed. The general idea of variable substitution is that a set of variables in a program can be replaced by a single representative variable, thereby reducing the input size of the problem. Our main contribution is a linear-time algorithm which finds a particular variable substitution that maintains the precision of the standard analysis, and is also very effective in reducing the size of the problem.

We report our experience in performing points-to analysis on large C programs, including some industrial-sized ones. Experiments show that our algorithm can reduce the cost of Andersen's points-to analysis [2] substantially: on average, it reduced the running time by 53% and the memory cost by 59%, relative to an efficient baseline implementation of the analysis.

1 Introduction

The goal of points-to analysis is to compute, for a given pointer p in a program, the set of variables whose address p may contain during the execution of the program. Points-to computation provides a conservative estimate of the set of variables read or written indirectly through the pointer dereference $*p$. This information is important in implementing a number of compiler optimizations (e.g., common subexpression elimination) as well as software productivity tools (e.g., program slicing). Without such information, overly pessimistic assumptions must be made about pointer dereferences.

Precision and efficiency of points-to analysis are important issues. Without good precision—small points-to sets—the analysis may be of limited use for its intended purpose.[1] Without good efficiency, it is infeasible to scale the analysis to large programs. Unfortunately, precision and efficiency usually come at each other's expense.

In this paper, we consider the problem of performing points-to analysis on rather large programs—perhaps, up to a million lines of code—while maintaining a reasonable level of precision. The first points-to analysis algorithm that could handle programs of this size is due to Steensgaard [15]. Steensgaard's analysis runs in time almost linear in the size of the program. Unfortunately, the results obtained by this analysis are relatively imprecise. An alternative algorithm due to Andersen [2] can produce much more precise results. However, Andersen's analysis has cubic time complexity, which limits its scalability. (These two algorithms are described further in Section 2.1.) Straightforward implementations of this analysis are ill-suited for handling large programs, because the time and space requirements become very high and often prohibitive. The key to scaling Andersen's analysis to larger programs, therefore, is in reducing the space and time requirements of the implementation. An example of such an approach is the implementation of Andersen's analysis in [5], which uses cycle elimination to reduce significantly the resource requirements of the analysis. (Cycle elimination is discussed further in Section 4.3.)

We have designed and implemented a new algorithm that improves the scalability of Andersen's analysis by reducing the input size of the problem. Using this algorithm, we have been able to perform Andersen's analysis on a program containing nearly 500,000 lines of code in about 7 minutes. Our experiments show that in practice, our algorithm can reduce time requirements by as much as 65% (53% on average), and space requirements by 68% (59% on average), relative to the time and space requirements of an efficient implementation based on the techniques in [5] and [16], which include cycle elimination and other optimizations.

We find it useful to describe our algorithm in the framework of *variable substitution* for points-to analysis. The general

PLDI 2000, Vancouver, British Columbia, Canada.

[1] Admittedly, the question whether the results are precise enough for an intended application is a difficult one to answer without actually putting the results to use. We will talk of precision only in a relative sense.

idea behind variable substitution is to reduce the cost of points-to analysis by replacing a set of program variables with a single representative variable. Depending on how the substitution is performed, this may result in less precise points-to results. For example, one can consider Steensgaard's analysis as performing a specific kind of variable substitution that results in loss of precision relative to Andersen's analysis. The approach that we present in this paper is essentially a new kind of variable substitution, with two key differences: one, it maintains the precision of Andersen's analysis, and two, it is performed off-line, i.e., before the actual analysis. An off-line technique decouples the resource optimization from the core analysis, making it independent of any particular implementation of the core analysis.

The off-line variable substitution used by our algorithm exploits a commonly found program behavior, that many pointer variables in a program have the same points-to set. The reason for this behavior is that once an address is taken, it is often passed around the program; for example, the address value may be used as an actual argument in a chain of procedure calls. If we could precompute a set of pointer variables that must have the same points-to "answer", we only need to perform points-to analysis for one representative variable from that set. This lets us reduce the size of the problem that we hand over to the analysis. While we have concentrated on reducing the cost of Andersen's analysis, we conjecture that such precomputation can be helpful in other points-to analyses as well.

The contributions of our work are the following:

- We propose off-line variable substitution as a technique for reducing the cost of points-to analysis, possibly at the expense of some loss of precision.
- We present a linear-time algorithm for computing a particular substitution which substantially reduces the cost of Andersen's analysis without any loss of precision.
- We present an experimental study on a set of large C programs, up to nearly 750,000 lines of code, to show the advantages of our substitution, and describe why these advantages are achieved.

The rest of the paper is organized as follows. Section 2 provides background material on points-to analysis algorithms and describes the notion of variable substitution. Section 3 defines the formal model for off-line variable substitution, and presents a particular substitution based on an equivalence relation on program variables; Section 4 presents an algorithm for computing this equivalence relation. Section 5 describes the design and implementation of points-to analysis based on our algorithm. Section 6 presents experimental results on a set of large programs. Section 7 discusses related work, and Section 8 presents conclusions and future work.

2 Points-to Analysis and Variable Substitution

Points-to analysis starts by defining a finite set V of *variables*; each variable represents one or more memory locations. Points-to relationships are represented using a *points-to graph* in which nodes correspond to variables. A directed edge from node v_1 to node v_2 shows that one of the memory

$a = \&b$
$b = \&c$
$a = \&d$
$d = \&e$

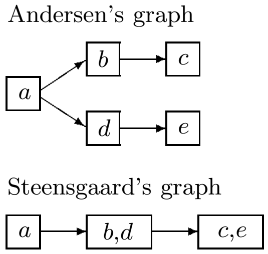

Figure 1: Andersen's vs Steensgaard's analysis.

locations represented by v_1 may contain the address of one of the memory locations represented by v_2. *Flow-sensitive* points-to analysis takes into account the flow of control between different program points; in contrast, *flow-insensitive* points-to analysis ignores control flow.[2] *Context-sensitive* points-to analysis considers (sometimes approximately) only program paths along which calls and returns are properly matched, while *context-insensitive* points-to analysis does not make this distinction.

2.1 Flow- and Context-Insensitive Points-to Analysis

Flow- and context-insensitive points-to analysis computes a single points-to graph for the whole program. Andersen's analysis [2] is a relatively precise analysis from this category; it has $O(n^3)$ worst-case complexity, where n is the size of the program. The analysis can be modeled as an iterative computation in which each iteration starts with a partial points-to graph and then adds new edges according to the semantics of program statements. Thus, for the assignment "$p = q$", new edges are added from p to all current successors of q; for the statement "$p = *q$", new edges are added from p to the successors of all successors of q; and so on for other statements. This model can be formalized by defining a lattice of points-to graphs $L = 2^{V \times V}$ and a transfer function $f : L \rightarrow L$, which encodes the cumulative points-to effect of all program statements. The final solution is the limit of the finite sequence $\emptyset, f(\emptyset), f^2(\emptyset)$, etc.

A relatively inexpensive flow- and context-insensitive points-to analysis is Steensgaard's analysis [15]. It has $O(n\alpha(n,n))$ complexity (where α is the inverse of Ackermann's function), but is much less precise than Andersen's analysis. In this approach, the nodes in the points-to graph represent sets of variables, and each node has at most one outgoing edge. After new edges are added in the manner described above, a node may have more than one outgoing edge and all of its successors need to be merged into a single node. An example of Andersen's and Steensgaard's analyses, due to [14], is shown in Figure 1; points-to pairs (b, e) and (d, c) in Steensgaard's graph are spurious.

The nodes of Steensgaard's graph define a partition of V into disjoint equivalence classes; the variables at each node form a separate equivalence class. The partition is stored in a fast union/find data structure [17], which allows efficient node merges. One of the variables in each class is used as an equivalence class representative (ECR) [15]. During the analysis, every reference to a variable v is translated into a reference to the ECR for the equivalence class to which v

[2] Some analyses (e.g., [7, 9]) only ignore intraprocedural flow of control.

currently belongs.

Shapiro and Horwitz [14] propose a family of points-to analyses whose cost and precision range from those of Steensgaard's analysis to those of Andersen's analysis. Similarly to Steensgaard's analysis, the nodes in the points-to graph represent sets of variables. Each node has at most k outgoing edges, where k is one of the parameters of the analysis. This is achieved by assigning each variable to one of k categories and merging nodes only if they are in the same category. The boundary cases $k = 1$ and $k = |V|$ correspond to Steensgaard's and Andersen's analysis, respectively.

2.2 Variable Substitution

A common feature of Steensgaard's and Shapiro-Horwitz's analyses is the use of an ECR as a placeholder for all variables in the equivalence class. The analyses can be thought of as performing "on-the-fly" translation from variables to representatives. Alternatively, this approach can be considered as modifying the program by replacing (substituting) each occurrence of a variable with the current representative for that variable. For example, the assignment "$p = q$" can be thought of as "$e(p) = e(q)$", where $e(p)$ and $e(q)$ are the current ECRs for p and q.

As seen from Steensgaard's and Shapiro-Horwitz's analyses, replacing a set of variables with a single placeholder variable can be used to reduce the cost of the analysis; of course, some loss of precision may occur. We call this general cost-saving technique *variable substitution*. In our specific variable substitution, rather than performing the substitution *during* the analysis (as in Steensgaard's and Shapiro-Horwitz's analyses), we perform it *off-line*, or *before* the analysis. In this approach, the program is first modified by replacing every occurrence of each variable with the representative for that variable. Points-to analysis is then applied to the modified program and the resulting solution is used to recover a solution for the original program. The next section describes this approach in the context of Andersen's analysis.

3 Off-line Variable Substitution

In this section, we first define a model for off-line variable substitution and show that it is safe (i.e., the substitution does not cause any points-to relation to be missed from the final answer). We then describe a substitution based on a specific partitioning of program variables into equivalence sets. This substitution preserves the precision of Andersen's analysis, and in practice, also reduces substantially the cost of the remaining points-to computation. We also discuss simplifications made possible by discovering the precise points-to solutions of some variables during our computation of equivalence sets; the computation itself is described in Section 4.

3.1 Definition and Safety

To simplify the presentation, we assume that the program is represented by a set of *basic statements*, as described by the grammar in Figure 2; some preprocessing may be needed to transform the program to this standard form.

$$
\begin{array}{lll}
\mathit{Var} & \rightarrow & \text{identifier} \\
\mathit{Assign} & \rightarrow & \mathit{Var} = \&\mathit{Var} \\
 & | & \mathit{Var} = \mathit{Var} \\
 & | & \mathit{Var} = *\mathit{Var} \\
 & | & *\mathit{Var} = \mathit{Var} \\
\mathit{FunDef} & \rightarrow & \mathit{Var}(\mathit{Var}, \ldots, \mathit{Var}) \Rightarrow \mathit{Var} \\
\mathit{Call} & \rightarrow & \mathit{Var} = \mathit{Var}(\mathit{Var}, \ldots, \mathit{Var}) \\
\mathit{FunPtrCall} & \rightarrow & \mathit{Var} = (*\mathit{Var})(\mathit{Var}, \ldots, \mathit{Var}) \\
\mathit{Stmt} & \rightarrow & \mathit{Assign} \mid \mathit{FunDef} \mid \mathit{Call} \mid \mathit{FunPtrCall}
\end{array}
$$

Figure 2: Grammar for basic statements. *FunDef* contains the function name, the formals, and a unique variable representing the return value of the function. A function call contains a variable to which the return value is assigned.

Off-line variable substitution is based on a substitution function $\sigma : V \rightarrow V'$ such that each variable from V is either mapped to itself, or to a "fresh" variable that is not in V. Thus, some variables from V are preserved by σ, while others are replaced by representative variables. In general, many variables could be mapped to the same representative. Based on σ, the original program is modified by replacing each occurrence of a variable v with $\sigma(v)$.

Let L and L' denote the lattices of points-to graphs for the original and the modified program, respectively. The modified program corresponds to a new transfer function $f' : L' \rightarrow L'$. Applying Andersen's analysis to this modified program yields a solution G'_S which is the limit of the sequence $\emptyset, f'(\emptyset), f'(f'(\emptyset))$, etc. A solution G_S for the original program can be obtained as $G_S = \{(v_i, v_j) | (\sigma(v_i), \sigma(v_j)) \in G'_S\}$.

This approach is illustrated by the example in Figure 3. Note that trivial assignments such as "$x = x$" can be eliminated after the substitution. In general, G'_S could be significantly smaller than Andersen's points-to graph G_A for the original program. Thus, the running time and memory cost of the analysis could be significantly reduced. It is also clear that some loss of precision may occur—for example, edges (a, q) and (p, q) in G_S are spurious.

The safety of off-line variable substitution is guaranteed by the following claim:

Theorem 1 *$G_A \subseteq G_S$, where G_A is Andersen's points-to graph for the original program.*

For each $G \in L$, let $\sigma_L(G) = \{(\sigma(u), \sigma(v)) | (u, v) \in G\}$. We can prove by induction on i that $\sigma_L(G_i) \subseteq G'_i$, where G_i is the partial points-to graph for the original program after iteration i, and G'_i is the corresponding graph for the modified program. Thus, $\sigma_L(G_A) \subseteq G'_S$, which implies $G_A \subseteq G_S$.

Off-line variable substitution can be used for constructing approximate points-to analyses—different substitution functions result in different tradeoffs between cost and precision. Even though we present and use off-line variable substitution in the context of Andersen's analysis, the technique is applicable to other kinds of points-to analysis. For example, it could be useful in investigating tradeoffs between cost and precision for pointer analyses with some degree of flow or context sensitivity (e.g., [8, 7, 4, 18, 9]).

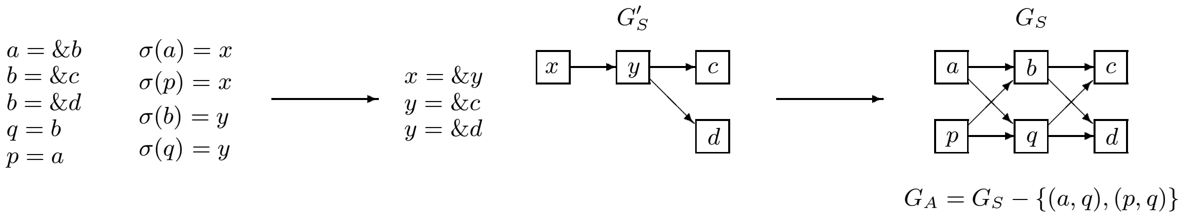

Figure 3: Off-line variable substitution example. Graph G'_S is the points-to graph after the substitution. Graph G_S is the corresponding points-to graph in terms of variables from the original program. Edges (a, q) and (p, q) are spurious with respect to Andersen's points-to graph G_A; this substitution does not preserve precision.

3.2 Substitution of Equivalence Sets

In this subsection we propose a particular off-line variable substitution that reduces the cost of Andersen's analysis *without* any loss of precision. The substitution is based on a collection of *equivalence sets*. An equivalence set is a set of variables that have the same points-to solution in G_A. Any two variables that belong to the same equivalence set are *equivalent variables*. In Section 4 we show how to compute some equivalence sets in linear time. We would like to keep the cost of this computation low, because it must be included in the total cost of the analysis.

The substitution is based on a precomputed collection of disjoint equivalence sets $S_0, S_1, \ldots, S_k$. (This collection need not cover the entire set of variables.) We use a single "fresh" representative variable r_i for each equivalence set S_i. For each $v \in V$, $\sigma(v)$ is defined as follows:

- If v is a function name, let $\sigma(v) = v$
- Otherwise, if v's address is taken somewhere in the program, let $\sigma(v) = v$
- Otherwise, if $v \in S_i$, let $\sigma(v) = r_i$
- Otherwise, let $\sigma(v) = v$

The precision of this substitution is guaranteed by the following theorem:

Theorem 2 *For the above definition of σ, $G_A = G_S$.*

We can prove by induction on i that $G'_i \subseteq \sigma_L(G_A)$, where G'_i and σ_L are the same as in the proof of Theorem 1; thus, $G_S \subseteq G_A$ and therefore $G_S = G_A$. The proof is based on three observations. First, equivalent variables behave similarly with respect to Andersen's analysis and can be replaced by a single representative without any loss of precision. Second, it is necessary to preserve all variables that are function names; this is needed to ensure that actual/formal pairs in the modified program are properly matched. Third, it is necessary to preserve all targets of points-to edges; this is achieved by preserving all variables whose address is taken somewhere in the program. Consider again the example in Figure 3. Variables b and q are equivalent. However, since the address of b is taken, mapping both b and q to the same representative variable results in the spurious edges (a, q) and (p, q).

If there is a large number of equivalent variables, G'_S is significantly smaller than G_A, resulting in reduction in the running time and the memory cost of Andersen's analysis.

3.3 Non-pointer Elimination

After the substitution has been performed, the cost of the analysis can be further reduced by non-pointer elimination. A *non-pointer* is a variable with an empty points-to set in G_A. Our computation of equivalence sets is capable of identifying some of the non-pointers in the program.[3] Suppose that all variables in equivalence sets $S_i, S_j, \ldots, S_m$ are non-pointers. The following statements involving the representatives $r_i, r_j, \ldots, r_m$ are irrelevant to the propagation of pointer values and can be safely eliminated:

- Assignments "$p = r_i$" and "$r_i = q$"
- Assignments "$*p = r_i$" and "$r_i = *q$"
- Direct calls "$r_i = f(r_j, \ldots, r_m)$"
- Indirect calls "$r_i = (*fp)(r_j, \ldots, r_m)$"

Our computation of equivalence sets can sometimes also identify variables that point to exactly one variable in G_A; we refer to such variables as *known pointers*. If all variables in some equivalence set S_i point only to the variable x, each occurrence of $*r_i$ in the modified program can be replaced by x. This, in turn, may reduce the cost of the subsequent points-to analysis. We refer to this transformation as *known-pointer instantiation*.

4 Equivalence Sets Computation

In this section we present our linear-time algorithm for computing equivalence sets. During the computation, the algorithm also identifies some of the non-pointers and known pointers in the program. In order to keep the cost of the computation low, the algorithm only computes equivalence sets that are "easy" to detect; nevertheless, our experiments show that this approach discovers a significant number of equivalent variables.

The algorithm starts by constructing a *subset graph* $G_\subseteq$. Intuitively, the edges in $G_\subseteq$ represent subset relationships between the points-to solutions for the nodes. For each program variable v, the subset graph contains node $n(v)$ and node $n(*v)$. If the address of v is taken anywhere in the program, the subset graph also contains node $n(\&v)$. The edge set of $G_\subseteq$ represents all subset relationships that can be directly inferred from the program; it is defined as follows:

[3] Due to the weak type system of C, declared types cannot be used to identify non-pointers.

$r = \&p$ $\quad s = *r$
$p = \&x$ $\quad t = *r$
$q = \&y$ $\quad y = *p$
$p = q$ $\quad i = j$
$q = *r$ $\quad i = k$
$s = \&x$

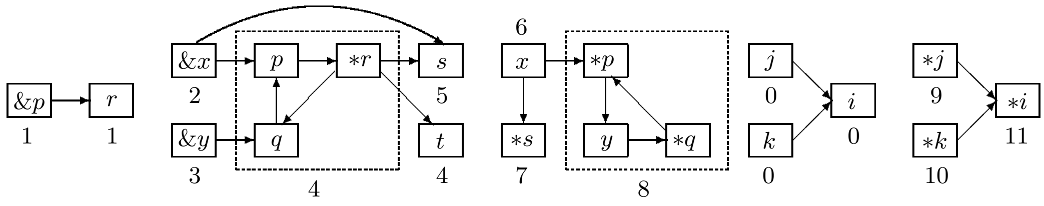

Figure 4: Subset graph example. The numbers denote the labels assigned by the algorithm from Figure 6. The direct SCCs are $\{n(r)\}, \{n(s)\}, \{n(t)\}, \{n(j)\}, \{n(k)\}$, and $\{n(i)\}$. The non-trivial equivalence sets are $\{p, q, t\}$ and $\{j, k, i\}$; the latter is a non-pointer set.

- For each assignment "$p = \&x$", $G_\subseteq$ contains the edges $(n(\&x), n(p))$ and $(n(x), n(*p))$.
- For each assignment "$p = q$", $G_\subseteq$ contains the edges $(n(q), n(p))$ and $(n(*q), n(*p))$.
- For each assignment "$p = *q$", $G_\subseteq$ contains the edge $(n(*q), n(p))$
- For each assignment "$*p = q$", $G_\subseteq$ contains the edge $(n(q), n(*p))$
- For each pair of an actual a and a corresponding formal f in a direct call, $G_\subseteq$ contains the edges $(n(a), n(f))$ and $(n(*a), n(*f))$
- For each direct call where r is the return variable of the called function and p is assigned the return value at the call, $G_\subseteq$ contains the edges $(n(r), n(p))$ and $(n(*r), n(*p))$

Figure 4 shows a set of basic statements and the corresponding subset graph; isolated nodes in the graph are not shown. Andersen's points-to graph for this program is given in Figure 5.

Let $Pt(v)$ be the points-to set for variable v in the final Andersen's points-to solution. We can associate a points-to set with each node in $G_\subseteq$ as follows: the points-to set of $n(v)$ is $Pt(v)$; the points-to set of $n(*v)$ is the union of $Pt(x)$ for all variables $x \in Pt(v)$; the points-to set of $n(\&v)$ is $\{v\}$. It is straightforward to prove the following claim:

Theorem 3 *For every edge (n_1, n_2) in the subset graph, $Pt(n_1) \subseteq Pt(n_2)$.*

As a corollary, all nodes in a strongly connected component (SCC) of the subset graph have the same points-to solution.

4.1 Direct Nodes

Intuitively, a *direct node* $n(v)$ corresponds to a variable v for which we can track directly all values assigned to v. The points-to set of a direct node depends only on the points-to sets of its predecessor nodes in the subset graph. We do not consider dereference nodes $n(*v)$ and address-of nodes $n(\&v)$ to be direct. A variable node $n(v)$ is direct only if all of the following conditions hold:

- Variable v never has its address taken, because in this case we cannot track indirect assignments to v through statements of the form "$*p = q$".
- Variable v is not a formal parameter for a function whose address is taken, because we may not know all the corresponding actuals from which the formal gets its values.
- Variable v is not assigned the return value of a call through a function pointer, because, again, we do not know all the corresponding return values.

The last two conditions are needed because we do not know function pointer targets *a priori*; recall that we ignore indirect calls when constructing the subset graph. In the example from Figure 4, all variable nodes except $n(p)$, $n(x)$ and $n(y)$ are direct. It is easy to prove the following theorem:

Theorem 4 *If n is a direct node, $Pt(n)$ is equal to the union of $Pt(m)$, where m is a predecessor of n in the subset graph.*

As a corollary, if all predecessors of n have the same points-to set, n also has that same points-to set.

4.2 Computation of Equivalence Sets

After constructing the subset graph, the analysis computes its strongly connected components and builds its SCC condensation. The resulting graph $G'_\subseteq$ is a directed acyclic graph in which each node corresponds to a SCC in $G_\subseteq$. A direct node n' in $G'_\subseteq$ represents a SCC that contains *only* direct nodes from $G_\subseteq$. Using Theorem 4, it is easy to prove that the points-to set of such n' is the union of the points-to sets of its predecessors.

The equivalence sets can be computed by traversing $G'_\subseteq$ in topological sort order, as shown in Figure 6. The algorithm assigns an integer label to each node. If two nodes are assigned the same label, their points-to sets are the same. Special consideration is given to nodes that can be shown to have empty points-to sets—all such nodes are assigned label 0. Labels that correspond to known points-to sets are stored in the partial map *Solved*.

The algorithm outputs two maps. *VarLabel* is a complete map from program variables to integer labels. It encodes the equivalence sets—if two variables have the same label, they belong to the same equivalence set. Furthermore, any variable with label 0 is a non-pointer. *Solved* is a partial map from integer labels to program variables; if $(l, x) \in$ *Solved*, the points-to set of any variable with label l is exactly $\{x\}$. The algorithm runs in time $O(max(|N'_\subseteq|, |E'_\subseteq|))$, which, in turn, is linear in the number of normalized statements in the input program.

$r = \&p$	$s = *r$
$p = \&x$	$t = *r$
$q = \&y$	$y = *p$
$p = q$	$i = j$
$q = *r$	$i = k$
$s = \&x$	

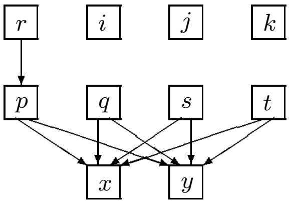

Figure 5: Andersen's points-to graph for Figure 4. Variables p, q, s, and t have the same points-to set. Variables i, j, and k are non-pointers.

The example in Figure 4 shows one possible assignment of labels to the SCCs of the subset graph. Suppose $\{n(\&p)\}$ is the first SCC in the topological sort order. Since it is not direct, it is assigned label 1 and $(1, p)$ is added to *Solved*. The SCC $\{n(r)\}$ is direct and gets label 1; furthermore, the points-to solution for r can be looked up in *Solved* as $\{p\}$.

The SCC $\{n(q), n(p), n(*r)\}$ is not direct and is assigned the fresh label 4. Since $\{n(t)\}$ is direct, its gets the same label; as a result, the equivalence set $\{p, q, t\}$ is detected. The SCC $\{n(j)\}$ is direct and has no predecessors; thus, in the original C program j gets all of its values from non-pointer assignments such as "$j = 1$". Since j and k are non-pointers, $\{n(j)\}$ and $\{n(k)\}$ are assigned label 0. The SCC $\{n(i)\}$ is direct and gets label 0 as well; as a result, the non-pointer equivalence set $\{j, k, i\}$ is discovered. Note that the analysis does not detect the non-pointers x and y. Also, even though s has the same points-to solution as p, q, and t, the algorithm cannot add s to that equivalence set because the points-to sets of the two predecessors of $n(s)$ are different. We could potentially maintain additional data structures to be able to track inclusion properties among labels, which would allow us to infer that $n(s)$ should also get the label 4.

One might wonder why we keep $n(*v)$ nodes in $G_{\subseteq}$ at all, since these are not direct nodes, and any SCC that contains a $n(*v)$ node is not direct as well. The reasons for keeping such nodes are, one, that they may lead to forming larger SCC components, and two, a successor of a non-direct node in $G'_{\subseteq}$ could be a direct node which inherits the label from its predecessor, thus adding members to the equivalence set. Both reasons are illustrated by node $n(*r)$ in Figure 4.

In the example, the only non-singleton equivalence sets are $S_4 = \{p, q, t\}$ and $S_0 = \{i, j, k\}$. They result in the following substitutions: $\sigma(q) = r_4$, $\sigma(t) = r_4$, $\sigma(i) = r_0$, $\sigma(j) = r_0$, and $\sigma(k) = r_0$. Variable p is not substituted by r_4 because its address is taken in the program. The rest of the variables belong to singleton equivalence sets and are not substituted. Variable r is a known pointer, as the computation reveals that r only points to p. Consequently, we instantiate each occurrence of $*r$ by p. We present the following input to Andersen's analysis: $r = \&p, p = \&x, r_4 = \&y, p = r_4, r_4 = p, s = \&x, s = p, r_4 = p, y = *p$. By Theorem 2, the points-to solution obtained by Andersen's analysis on this input program, after reverse mapping, is the same as the points-to solution obtained by applying Andersen's analysis on the original program.

4.3 Comparison with Cycle Elimination

Our algorithm, in particular the subset graph construction, bears some connection with the idea of cycle elimination in

```
input     G'⊆ = (N'⊆, E'⊆)
output    VarLabel : array[V] of integer
          Solved : map integer → V
declare   SccLabel : array[N'⊆] of integer
          counter: integer

counter := 1;
foreach n' ∈ N'⊆ in topological sort order do
    if n' is not a direct node then
        SccLabel[n'] := counter;
        counter := counter + 1;
        if n' is an address-of node n(&v) then
            add (SccLabel[n'],v) to Solved;
    else if n' has no predecessors then
        SccLabel[n'] := 0;
    else
        if all predecessors of n'
          have the same label l then
            SccLabel[n'] := l;
        else
            SccLabel[n'] := counter;
            counter := counter + 1;
foreach v ∈ V do
    VarLabel[v] := SccLabel[ContainingScc(n(v))];
```

Figure 6: Computation of equivalence sets.

set-inclusion constraint graphs [5]. In both graphs, nodes represent points-to sets and edges represent subset relationships.[4] A cycle in the constraint graph corresponds to variables $v_0, \ldots, v_k$ such that

$$Pt(v_0) \subseteq Pt(v_1) \subseteq \ldots \subseteq Pt(v_k) \subseteq Pt(v_0)$$

where $Pt(v_i)$ is the points-to set of v_i. All variables on a cycle have equal points-to sets and can be replaced by a single representative variable. In [5], cycle detection and elimination is used to improve the performance of Andersen's analysis.

The strongly connected components in the subset graph also find equivalence relationships due to cycles, but only in the initial set of inclusion relationships apparent from the program. Most of the benefit of cycle elimination in [5], however, comes from finding and eliminating cycles online—as fresh ones appear during constraint solving, rather than only in the initial graph. Our algorithm does not discover such cycles, because unlike [5], no new edges are added during the analysis. For example, consider the assignments "$p = \&x$" and "$*p = y$". In [5], the analysis will use the points-to pair (p, x) to determine that the value of y is assigned to x; this results in the new constraint $Pt(y) \subseteq Pt(x)$, which may result in new cycles being found. In our subset graph, there will be no edge from $n(y)$ to $n(x)$.

Instead, our algorithm creates equivalence sets that cross cycle boundaries. By propagating a label from a node to some of its successors, our approach is capable of discovering equivalent variables that do not belong to any cycle. For example, in Figure 4, variables p, q, and t are found to be in the same equivalence set, even though they do not form a cycle in the sense of constraint graphs.

[4]We refer the reader to [5] to see how set constraints can be used to model Andersen's analysis.

5 Design and Implementation

We investigated the impact of off-line variable substitution on the cost of Andersen's analysis by comparing two versions of the analysis. The *standard version* implements the traditional Andersen's analysis; the *substitution version* uses the algorithm from Section 4 to compute equivalence sets and then performs off-line variable substitution, non-pointer elimination and known-pointer instantiation.

The standard version is implemented in ML and uses a front end for C also implemented in ML. The front end builds a simplified program representation in which the statements are normalized to consist of only a few simple forms; for example, each statement contains at most one pointer dereference. The analysis traverses the simplified representation and extracts a set of basic statements, as described in Figure 2. Similarly to [15, 14, 5], structures and arrays are treated as monolithic objects and their individual elements are not distinguished. Library functions are handled by providing stubs that simulate their points-to effects. Dynamic memory allocation is handled by considering each call to *malloc* and other heap-allocating functions as equivalent to taking the address of a new variable unique to the site of the *malloc* call.

As a practical matter, we phase our implementation by writing intermediate results to disk. Thus the front end extracts the basic statements and stores them on disk. The analysis works on a list of basic statements available in a file. This approach allows clean separation between the front end, equivalence set computation (if any), and the main analysis.

The substitution version of Andersen's analysis is organized as follows. The equivalence set computation reads the basic statements from disk, computes variable labels and known pointers, and terminates by writing this information back to disk. The next phase, invoked separately, reads the information from disk, performs off-line variable substitution, non-pointer elimination and known-pointer instantiation, and computes Andersen's solution similarly to the standard version.

The standard version and the last phase of the substitution version are both based on the implementation of Andersen's analysis in BANE (Berkeley ANalysis Engine) [1]. BANE is a toolkit for constructing constraint-based program analyses. The public distribution of BANE[5] contains an efficient constraint-solving engine, and an implementation of Andersen's analysis that uses this engine [5, 16]; to the best of our knowledge, this is the fastest publicly available implementation of Andersen's analysis. Since our front end and program representation are different, we did not use BANE's implementation of the analysis directly. Instead, we created a version that traverses our intermediate representation, generates exactly the same kind of constraints as BANE's implementation would have generated, and then uses the constraint-solving engine provided by BANE to compute the points-to solution.

6 Experiments and Results

For our experiments, we used a set of C programs ranging in size from 30K to around 750K LOC. Some characteristics of the programs are given in Table 1. The third column in Table 1 gives the number of basic statements (of the form shown in Figure 2) and the last column shows the number of variables in the basic statements. Most of the programs are publicly available, except `switch1` and `switch2`, which are proprietary programs used in Lucent's products.

[5]http://bane.cs.berkeley.edu

Program	LOC [K]	NumStmt [K]	NumVar [K]
nethack	30.7	50.2	52.9
burlap	49.6	67.5	66.6
vortex	67.2	55.9	71.8
emacs	99.4	108.7	124.7
povray	133.9	116.8	119.1
gcc	217.7	255.3	254.5
switch1	~ 500	366.5	419.9
switch2	~ 750	765.7	794.8

Table 1: Data programs.

Program	T_A [sec]	S_A [MB]	T_{IO} [sec]
nethack	114.4	76.5	3.6
burlap	143.6	94.4	4.8
vortex	154.8	91.9	4.9
emacs	235.2	145.2	8.8
povray	382.4	193.0	8.7
gcc	942.3	347.4	18.9
switch1	1054.7	485.3	28.4
switch2	—	—	57.3

Table 2: Cost of Andersen's analysis. T_A is the running time and S_A is the memory used by the analysis. T_{IO} is the time for disk write/read of basic statements.

All experiments were performed using a single 195 MHz processor on a multi-processor SGI Origin machine with 1.5 GB physical memory. The reported time measurements are the best values out of three runs. The space was measured by instrumenting the ML garbage collector to report the amount of live data after each garbage collection (therefore, the numbers are somewhat approximate).

Table 2 shows the cost of the standard version of Andersen's analysis. Column T_A gives the time to perform Andersen's analysis after the basic statements are read from disk. Column S_A shows the space needed by the analysis. Column T_{IO} gives the total time for disk write/read of basic statements; the average increase in the running time due to disk IO is 3%.

The results show that the running time of the standard version itself is quite reasonable, given the cubic worst-case complexity of Andersen's analysis. The results also show that for large programs the memory cost of the analysis can be significant. In fact, our largest program could not be analyzed because the analysis ran out of memory. Clearly, memory could be a bottleneck when analyzing large programs.

Table 3 shows the overall performance of the substitution version. Columns T_E and S_E show the time and space needed to compute the equivalence sets. This computation currently has a simple implementation with no performance tuning; we expect to reduce both the running time and the consumed space by using a more mature implementation. Columns T'_A and S'_A show the cost of Andersen's analysis after substitution, non-pointer elimination, and known-pointer instantiation. The reduction in analy-

Program	T_E [sec]	S_E [MB]	T'_A [sec]	S'_A [MB]	ΔT [%]	ΔS [%]
nethack	16.9	20.3	46.0	37.1	45.0%	51.5%
burlap	21.5	25.2	46.5	41.4	52.6%	56.1%
vortex	22.3	27.6	57.9	41.7	48.2%	54.6%
emacs	35.0	46.3	63.3	45.2	58.2%	68.1%
povray	45.1	44.6	165.8	90.1	44.8%	53.3%
gcc	113.3	95.7	214.7	121.4	65.2%	65.1%
switch1	223.4	157.4	211.3	137.6	58.8%	67.6%
switch2	465.2	297.0	—	—	—	—

Table 3: Overall performance of the substitution version. T_E and S_E are the time and space needed to compute the equivalence sets. T'_A and S'_A are the time and space for Andersen's analysis after substitution, non-pointer elimination, and known-pointer instantiation. The last two columns show the reduction in analysis cost.

Program	Overall [%]	SCC [%]	Nptr [%]
nethack	66.0%	3.4%	35.0%
burlap	74.5%	4.7%	50.2%
vortex	71.4%	1.5%	46.7%
emacs	79.1%	5.0%	54.4%
povray	76.1%	4.7%	38.3%
gcc	76.8%	4.4%	31.1%
switch1	81.9%	2.0%	56.6%
switch2	78.4%	3.4%	38.8%

Table 4: Reduction in the number of variables. The columns show the overall reduction, the reduction due to SCCs in the subset graph, and the reduction due to non-pointers.

Program	Size 10^0-10^1	Size 10^1-10^2	Size 10^2-10^3	Size $> 10^3$
nethack	4116	274	12	0
burlap	3596	230	12	0
vortex	4492	271	5	1
emacs	4563	334	18	2
povray	6834	1021	32	0
gcc	16009	1624	78	4
switch1	25667	1417	97	4
switch2	52541	3334	426	13

Table 5: Distribution of equivalence set sizes.

sis time ΔT is computed as $1 - (T_E + T'_A)/T_A$, where T_A is the time from Table 2. Since the equivalence sets are computed separately from Andersen's analysis, the overall space cost is $max(S_E, S'_A)$ and the reduction in space is $\Delta S = 1 - max(S_E, S'_A)/S_A$. It is clear that the reduction in the number of variables results in significant reduction of the analysis cost—on average, 53% for running time and 59% for space.

Even with significant reduction in the number of variables, the analysis of `switch2` ran out of memory. However, it would be misleading to conclude that either the original `switch2`, or the program after variable substitution, definitely needed more than 1.5 GB of space. The garbage collector in SML/NJ has peak virtual-memory requirements that are considerably higher than the size of the live data.[6] In fact, at the time the substitution version ran out of memory on a 1.5 GB machine, the live data was only about 500 MB. But based on the experience with other programs, we extrapolate that `switch2` could run in much less memory using the substitution version than using the standard version.

Table 4 shows the effectiveness of substitution based on equivalence sets. The second column gives the fraction of variables eliminated due to substitution. On average, the number of variables is reduced by 76%. We performed further experiments to estimate the impact of different sources of this reduction. First, we estimated the reduction that can be obtained by only computing the SCCs in the subset graph. In this scenario, two variables are equivalent only if they belong to the same SCC. The reduction in the number of variables is shown in the third column of Table 4. Clearly, little can be gained from SCC computation alone; this is consistent with the observations in [5].

Next, we measured the reduction obtained by performing substitution only on non-pointer variables. The reduction in the number of variables is shown in the last column of Table 4. Clearly, the approach from Section 4 detects a significant number of non-pointers, resulting in average reduction of 44%. We also computed the number of all variables that have empty points-to sets in the final Andersen's solution (the numbers are not shown), and discovered that, on average, 82% of them are detected by the approach from Section 4. The large number of non-pointers is due to the fact that in C programs many variables are not intended to be used as pointers and have no effect on points-to analysis. Because of the weak type system of C (due to type casting), these variables cannot be eliminated solely on the basis of their declared types.

The remainder of the reduction is obtained from equivalence sets that span multiple SCC nodes. To obtain some insight into these equivalence sets, we computed the distribution of their sizes; the results are presented in Table 5. Singleton equivalence sets are trivial for the purposes of substitution and are not counted. The single non-pointer equivalence set is not counted either. In each case, we found that most of the equivalence sets are small, and in fact, the number of larger sets tends to decrease polynomially with increasing set size. We also examined some of the programs manually to see why variables form large equivalence sets. We found this was mostly due to passing around pointers to global data structures. Since such pointers are copied many times, including copies made from actual to formal parameters, a large number of pointers end up having the same points-to set. To our knowledge, this empirical behavior has not been noticed or exploited in program analysis work before.

As described in Section 4, the computation of equivalence sets detects some of the known pointers in the program. Af-

[6]The garbage collector [11] is optimized for speed and aimed at medium-sized data sets; our data sizes are unusually large for it.

ter the substitution, dereferences of such variables (or dereferences of their representatives) are instantiated by their known targets, resulting in a somewhat simpler program. Our algorithm finds between 7% and 20% (12% on average) of all variables as having a known single target. Further experiments show that known-pointer instantiation improves space and time reductions only marginally (about 2%).

7 Related Work

There is a large body of work on pointer analysis. Some of the analyses concentrate on pointers that point to heap-allocated memory (e.g., [6, 13]). Others analyze pointers that point to stack-based memory; among them, some are flow-sensitive (e.g., [8, 7, 4, 18]), while others are flow-insensitive (e.g., [2, 7, 15, 19, 14, 9]).

Several analyses use placeholder variables to represent sets of related variables. The use of equivalence class representatives in Steensgaard's and Shapiro-Horwitz's analyses was already discussed in Section 2. Other examples are the non-visible names from [8], the invisible names from [4], the extended parameters from [18], the cycle witness variables from [5], the placeholder variables from [12], and the equivalence class representatives from [10]. In general, the placeholders need not be used throughout the program; for example, in [8, 7, 4], a placeholder is used when analyzing a called procedure, and the original variable is used in the calling procedure.

The flow-sensitive pointer analysis from [7] uses a subset of a procedure's control-flow graph which only contains the program points at which the solution could change. This reduces the size of the problem because a set of program points that share the same solution can be represented by a single program point. This technique resembles our use of a representative variable for a set of variables that have the same points-to solution.

The most closely related work that focuses on speeding up Andersen's analysis is cycle elimination in the Bane analysis system [1], which was already discussed in Section 4.3. The Bane engine was recently enhanced with another important optimization, called projection merging [16]. The version of Bane that we used incorporates projection merging as well.

The problem of scaling class analysis of object-oriented programs also lends itself to solutions somewhat analogous to those used for point-to analysis. While a relatively precise $O(n^3)$ algorithm exists, researchers have proposed different levels of approximation, usually derived by merging nodes in a graph representation of the problem [3]. It would be interesting to see if our techniques for points-to analysis are also useful for class analysis.

8 Conclusions and Future Work

We have shown that a particular form of off-line variable substitution based on equivalence sets can be used to reduce the cost of Andersen's analysis without any loss of precision. The computation of equivalence sets has linear-time complexity and significantly reduces the number of variables (by 76% on average). This reduction translates into significant reduction in the analysis running time (53% on average) and memory cost (59% on average). We believe that this technique can be widely adapted, as it is simple, and independent of any particular implementation of Andersen's analysis.

Off-line variable substitution can be used to develop approximate versions of Andersen's analysis. It would be interesting to investigate substitutions that further reduce the cost of the analysis, possibly at the expense of some precision. Off-line variable substitution could also be useful for developing approximate versions of other pointer analyses, including analyses with some degree of flow or context sensitivity. Finally, it would be interesting to consider uses of the substitution technique for analyses similar to pointer analysis—for example, class analysis and escape analysis for object-oriented languages.

9 Acknowledgments

We would like to thank the Bane team at Berkeley for distributing their code and answering questions about it. Darren Atkinson provided the preprocessed source code for `burlap`, `emacs` and `gcc`. John Reppy provided assistance with instrumenting the garbage collector. Nevin Heintze and Dino Oliva made several improvements to the front end used in this work. Barbara Ryder, Matthew Arnold, and Peter Mataga provided helpful comments on earlier versions of this paper. The first author was supported by Bell Laboratories summer internship and by NSF grant CCR-9900988.

References

[1] A. Aiken, M. Fähndrich, J. Foster, and Z. Su. A toolkit for constructing type- and constraint-based program analyses. In *Proc. Workshop on Types in Compilation*, LNCS 1473, pages 78–96, 1998.

[2] L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, May 1994.

[3] G. DeFouw, D. Grove, and C. Chambers. Fast interprocedural class analysis. In *Proc. Symposium on Principles of Programming Languages*, pages 222–236, 1998.

[4] M. Emami, R. Ghiya, and L. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proc. Conference on Programming Language Design and Implementation*, pages 242–257, 1994.

[5] M. Fähndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Proc. Conference on Programming Language Design and Implementation*, pages 85–96, 1998.

[6] R. Ghiya and L. Hendren. Is it a tree, a DAG or a cyclic graph? In *Proc. Symposium on Principles of Programming Languages*, pages 1–15, 1996.

[7] M. Hind, M. Burke, P. Carini, and J. D. Choi. Interprocedural pointer alias analysis. *ACM Transactions on Programming Languages and Systems*, 21(4):848–894, July 1999.

[8] W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proc.*

Conference on Programming Language Design and Implementation, pages 235–248, 1992.

[9] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *Proc. Symposium on the Foundations of Software Engineering*, LNCS 1687, pages 199–215, 1999.

[10] D. Liang and M. J. Harrold. Equivalence analysis: A general technique to improve the efficiency of data-flow analyses in the presence of pointers. In *Proc. Workshop on Program Analysis for Software Tools and Engineering*, pages 39–46, 1999.

[11] J. Reppy. A high-performance garbage collector for Standard ML. Technical memorandum, AT&T Bell Laboratories, Dec. 1993.

[12] A. Rountev, B. G. Ryder, and W. Landi. Data-flow analysis of program fragments. In *Proc. Symposium on the Foundations of Software Engineering*, LNCS 1687, pages 235–252, 1999.

[13] M. Sagiv, T. Reps, and R. Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, Jan. 1998.

[14] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Proc. Symposium on Principles of Programming Languages*, pages 1–14, 1997.

[15] B. Steensgaard. Points-to analysis in almost linear time. In *Proc. Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[16] Z. Su, M. Fähndrich, and A. Aiken. Projection merging: Reducing redundancies in inclusion constraint graphs. In *Proc. Symposium on Principles of Programming Languages*, pages 81–95, 2000.

[17] R. E. Tarjan. *Data Structures and Network Algorithms*. Society for Industrial and Applied Mathematics, 1983.

[18] R. Wilson and M. Lam. Efficient context-sensitive pointer analysis for C programs. In *Proc. Conference on Programming Language Design and Implementation*, pages 1–12, 1995.

[19] S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step towards practical analyses. In *Proc. Symposium on the Foundations of Software Engineering*, pages 81–92, 1996.