Practical Points-to Analysis for Programs Built with Libraries*

Atanas Rountev
Department of Computer Science
Rutgers University
Piscataway, NJ 08854, USA
+1 732 445 4070
rountev@cs.rutgers.edu

Barbara G. Ryder
Department of Computer Science
Rutgers University
Piscataway, NJ 08854, USA
+1 732 445 3699
ryder@cs.rutgers.edu

Abstract

Traditional *whole-program analysis* cannot be directly applied to programs that include precompiled libraries. Such programs could be analyzed separately from the included libraries by using precomputed *summary information* about each library. This paper describes one such *separate analysis* derived from Andersen's whole-program points-to analysis [2]. The analysis uses a summary which is a compact representation of the points-to effects of all statements in the library. The summary is generated by substituting some of the library variables with placeholder variables. By replacing many variables with the same placeholder, we can reduce the size of the summary and the cost of the separate analysis. We use a substitution which summarizes the library effects without losing precision or exposing the internals of the library. Our experiments show that the cost of computing and storing the summary is practical, and that the substitution technique significantly reduces the cost of the separate analysis.
Keywords: Data-flow analysis, pointer analysis

1 Introduction

The process of software development, validation and evolution requires information about various properties of complex programs. *Data-flow analysis* extracts semantic information which can be used for code optimization, program understanding and modification, code verification, and data-flow-based testing.

Traditionally, data-flow analysis is designed to analyze whole programs. However, such *whole-program analysis* cannot be directly applied to programs that include precompiled libraries. In some cases, the source code for the library may not be available—for example, when the library is obtained as a binary from an external developer. In this case, whole-program analysis is not possible. Even if the library source is available, the users of the data-flow analysis need to know details about the library configuration (e.g., they need to analyze the makefiles, track the compilation flags, etc.). However, this could be inconvenient (or even unrealistic) in the cases when the library is outside the scope of user's expertise and control. Last but not least, the library could be included by many client programs; for each of them, the library would have to be reanalyzed from scratch. The analysis cost could be reduced if some of the analysis work for the library is done in advance and the results are later reused for all client programs.

Addressing these issues is a serious challenge for the designers of data-flow analyses. The importance of the problem is likely to increase due to the strong trend toward reuse of software components. Many organizations use reusable components to reduce the development costs and shorten the development cycles of large industrial software systems [24]. Components can come from in-house component repositories, or can be commercial off-the-shelf components purchased from an external vendor. In many cases, the users of a component either have no source code available, or have no access to the personnel and expertise needed to understand and use the component source code.

There are various theoretical and practical issues that need to be resolved in order to adapt existing analyses to development environments which contain precompiled libraries. In the simplest technique, a whole-program analysis can be modified to make worst-case assumptions about the library. This approach is simple to implement, but may result in significant loss of precision: for example, in a language like C, the analysis must assume that each library call modifies all variables whose address is taken somewhere in the program. In the extreme case, it is possible that a very imprecise approximation renders the analysis results useless.

A more precise approach is to preanalyze the library,

*This research was partially supported by NSF grants CCR-9804065 and CCR-9900988.

compute some form of *summary information*, and later use it when analyzing library clients. The summary information encodes the effects of the library and can be computed when the library is compiled; at this time, the library source code and configuration details are known. The summary can be made available to the clients of the library by storing it together with the library binary or header files.

To be able to compute and use library summary information, the designers of a data-flow analysis need to answer several questions: What kinds of summary information are useful for the particular analysis? How should summary safety and precision be defined? How is the summary information represented? How can it be integrated with the analysis? How to compute the summary information safely, precisely, and efficiently?

1.1 Whole-program Points-to Analysis

The goal of this paper is to answer the above questions for Andersen's points-to analysis for C [2]. For each pointer p in the program, *points-to analysis* computes the set of memory locations whose address may be stored in p during program execution (i.e., the *points-to set* of p). This information is needed to determine the memory locations accessed through the pointer dereference $*p$, as well as the targets of indirect calls (i.e., calls through function pointers in C). Without such information, it is impossible to perform a wide variety of other analyses (e.g., live variables analysis, side-effect analysis, dependence analysis) that need information about the memory accessed by program statements and the flow of control at indirect calls. In turn, such analyses are necessary for various compiler optimizations (e.g., constant propagation, dead code elimination), program understanding applications (e.g., interprocedural slicing, semantic browsing), and data-flow-based testing.

Flow-sensitive points-to analyses take into account the flow of control between program statements [10, 9, 5, 17, 25, 3]. Most such analyses are also *context-sensitive*—they try to consider only valid call/return sequences of procedures. In general, these analyses are precise but expensive. *Flow-insensitive* points-to analyses have better scalability because they ignore the flow of control between program statements. Some such analyses only take into account the control-flow between procedures [9, 11]; in some cases, this approach allows certain degree of context-sensitivity. Other analyses ignore the flow of control completely [2, 21, 26, 19]; such analyses are both flow- and context-insensitive.

Andersen's analysis [2] is one of the more precise flow- and context-insensitive points-to analyses. It has been shown to scale for large programs [6, 23, 15] and is a natural candidate for use in industrial-strength software tools. We only consider this analysis; however, our techniques can be directly applied to other flow- and context-insensitive points-to analyses—for example, the analyses by Steensgaard [21] and Shapiro and Horwitz [19].

1.2 Separate Points-to Analysis

We consider programs consisting of two components: a *main component* and a *library component*. We assume that the library is compiled independently and can be subsequently included by a variety of client programs. The summary information for the library is computed at library compilation time. The summary is essentially a compact representation of the points-to effects of all statements in the library. Using this summary, we can analyze the main component of a client program separately from the actual library. Such *separate analysis* is closely related to the whole-program Andersen's analysis and can be formally derived from it. Our separate analysis is designed to achieve the following goals:

Precision. The separate analysis is designed to be as precise as the whole-program analysis. This is important because the precision of the points-to information can greatly affect the cost and precision of subsequent analyses and applications [18, 22, 11].

Practicality. The cost of computing and storing the summary should be practical. In particular, the time to compute the summary should not be excessive (e.g., no more than a few minutes for a medium-sized library) and the disk space needed to store the summary should be reasonable (e.g., comparable to the disk space for the library binary). The experiments from Section 7 show that our summary computation takes a few seconds, and the summary size is (on average) 36% of the size of the library binary.

Cost reduction. The cost of the separate analysis should be reduced by performing some of the analysis work at summary generation time and encoding the results in the summary. The experiments from Section 7 show that our approach achieves significant cost reduction.

Information hiding. The summary should not expose the internals of the library. In some cases, the summary may be generated by a library developer who only provides the library binary, but is unwilling to provide the library source code. In such cases, the names of private library variables and procedures (e.g., a name such as `compute_huffman_code`) may convey sensitive information. Our approach ensures that only publicly exported library names occur in the summary. Furthermore, as described in Section 4, the summary does not contain any control-flow information. This approach should provide adequate protection against attempts to use the summary for reverse engineering of the library.

The design of our separate points-to analysis is based on a general approach for designing separate flow- and context-insensitive analyses. This approach (described in Section 2) is a modification of an earlier model for analyzing program fragments [16]; the relationship with this previous work is discussed in Section 8. The approach shows how to derive a separate analysis from a given whole-program analysis, and how to generate summaries for the resulting separate analysis. The general idea behind the derivation is to abstract away some information about the library; the particular form of abstraction depends on the intended clients of the analysis.

In the case of Andersen's analysis, the approach of Section 2 can be implemented using *variable substitution* [15]. The idea behind this technique is to replace a set of program variables with a single representative variable. In [15] this approach is used to reduce the cost of Andersen's whole-program analysis. The particular substitution used is designed to preserve the precision of the analysis. In this paper we use a similar kind of substitution; as a result, we can significantly reduce the size of the summary and the cost of the separate analysis, while ensuring that the separate analysis is as precise as the whole-program analysis.

The contributions of this work are the following:

- We propose a model for flow- and context-insensitive analysis of programs built with libraries. We show how to derive a separate analysis from a given whole-program analysis, and how to generate and use library summary information.
- We apply the model to Andersen's whole-program analysis. The resulting separate analysis has the same precision as the original analysis, and uses a summary which encodes partial analysis results computed at summary generation time.
- We show empirical evidence that our approach is effective and practical. For our data programs, the summaries are generated in a few seconds, and summary size is (on average) 36% of the size of the library binary. The experiments show that our summary generation technique significantly reduces the cost of the subsequent separate analysis.

The rest of the paper is organized as follows. Section 2 discusses the theoretical model for separate analysis. Section 3 describes Andersen's analysis. Section 4 shows how to use variable substitution to derive the separate analysis, and defines a particular substitution based on equivalence sets. Section 5 describes the computation of equivalence sets. Section 6 shows how to handle multiple libraries. The experimental results are presented in Section 7. Section 8 discusses related work, and Section 9 presents conclusions and future work.

2 Analysis in the Presence of Libraries

This section presents a theoretical approach for designing separate flow- and context-insensitive analyses and generating library summaries for them.

2.1 Whole-Program Analysis

Given a complete program, a *whole-program analysis* extracts a lattice L with partial order $\leq$, meet operation $\wedge$, and greatest element $\top$. The elements of the lattice encode the analyzed program properties; the partial order represents the relative precision of lattice elements. The analysis also extracts a finite set F of *transfer functions*; each $f : L \to L$ is a monotone[1] function which encodes the data-flow effects of a program statement. A *flow- and context-insensitive* analysis ignores the flow of control in the program and computes one solution $S \in L$ for the whole program. The solution is *safe* if and only if for every finite sequence $f_0, f_1, \ldots, f_n$ of transfer functions, we have

$$S \leq (f_0 \circ f_1 \circ \ldots \circ f_n)(\top)$$

A safe solution is typically obtained by an iterative fixed-point computation.[2] The final solution S is a fixed point of each transfer function f (e.g., $S = f(S)$).

2.2 Separate Analysis

Our goal is to design a *separate analysis* which analyzes the main component of a program separately from the library component. In particular, we are interested in *deriving* a separate analysis from a given whole-program analysis.

Consider a whole-program analysis $\mathcal{A}$ which maps a program to a lattice L and a set of transfer functions F. A separate analysis $\mathcal{A}'$, derived from $\mathcal{A}$, takes as input the main component of the program and some summary information about the library component. $\mathcal{A}'$ constructs a lattice L' and a set of transfer functions F', and uses F' to compute a fixed-point solution.

The New Lattice $\mathcal{A}'$ constructs a lattice L' with partial order $\leq$, meet operation $\wedge$, and greatest element $\top'$. The new lattice is a modification of the whole-program lattice L in which some information about the library is abstracted away. The particular form of abstraction

[1] That is, $x \leq y$ implies $f(x) \leq f(y)$.

[2] As a technical detail, we have to assume that the transfer functions do not perform kills—that is, $f(x) \leq x$ for any f and x. This restriction ensures the convergence of the iteration.

is determined by the designers of $\mathcal{A}'$; it is based on the needs of the analysis clients, the desired analysis performance (e.g., a smaller L' may result in faster analysis), and the requirements for information hiding. For example, the abstraction described in Section 4 is designed to meet the goals from Section 1.

The abstraction can be formalized by using a standard technique from the field of abstract interpretation [4]. The designers of $\mathcal{A}'$ have to define an *abstraction function* $\alpha : L \to L'$ and a *concretization function* $\gamma : L' \to L$. For any $x \in L$, the value $\alpha(x)$ is the element of L' that best describes (represents) x. The concretization function γ can be thought of as the inverse of α. We assume that the following standard properties hold [4]:

- α and γ are monotone
- $\forall x \in L : \ \gamma(\alpha(x)) \leq x$
- $\forall x' \in L' : \ \alpha(\gamma(x')) = x'$

The monotonicity ensures that α and γ preserve the relative precision of lattice elements. The second property guarantees that abstraction produces a safe approximation. The last property requires that concretization introduces no loss of information.

Definition 1 *A solution $S' \in L'$ computed by $\mathcal{A}'$ is safe if and only if for every finite sequence $f_0, f_1, \ldots, f_n$ of transfer functions from F, we have*

$$S' \leq \alpha((f_0 \circ f_1 \circ \ldots \circ f_n)(\top))$$

The New Transfer Functions Our goal is to ensure that the fixed-point solution computed by $\mathcal{A}'$ is safe according to Definition 1. Safety can be guaranteed by a standard approach [4] which ensures that each function from F is abstracted by some function from F'. More formally, for every $f \in F$, there should exist $f' \in F'$ such that

$$\forall x \in L : \ f'(\alpha(x)) \leq \alpha(f(x))$$

This property means that an application of f' in L' safely represents the effects of f in L. It can be proven that any definition of F' which satisfies this requirement produces a safe separate analysis.

We are particularly interested in one such definition:

$$F' = \{ \gamma \circ f \circ \alpha \mid f \in F \}$$

Since F' can be easily constructed from F, this definition has certain advantages over alternative approaches for constructing F'; this issue is discussed further in Section 8. For a separate analysis based on this definition, a library summary can be generated as follows: the functions from F which correspond to statements in the library are modified using α and γ. The resulting subset of F' is stored as library summary. Later, during the separate analysis, the rest of F' is constructed from the statements in the main component. The separate points-to analysis from Section 4 is designed using this approach.

3 Whole-Program Andersen's Analysis

Andersen's analysis computes a safe approximation of the points-to relationships between memory locations. The analysis defines a finite set V of *variables*; each variable represents one or more memory locations. Points-to relationships are represented by 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 locations represented by v_1 may contain the address of one of the memory locations represented by v_2.

The analysis lattice is $L = 2^{V \times V}$ and each lattice element is a points-to graph. The lattice is inverted—the partial order is the $\supseteq$ relation, and $\top$ is the empty graph. The transfer functions represent the points-to effects of program statements. For the purposes of this paper, we assume that the program is represented by a set of *basic statements*, as described by the grammar in Figure 1; some preprocessing may be needed to generate this canonical form.

Each transfer function corresponds to a basic statement. For example, given a points-to graph G, the function for "$p = q$" is $f(G) = G \cup \{(p,x) \mid (q,x) \in G\}$.[3] The transfer function for a direct call is equivalent to a set of assignments from actual to formal parameters, plus an optional assignment for the return value of the called procedure. The transfer function for an indirect call through a pointer p is equivalent to a set of direct calls, one for each procedure that p points to. The transfer functions are monotone and can be used to compute a safe fixed-point solution.

4 Separate Andersen's Analysis

For Andersen's analysis, the approach from Section 2 can be implemented using *variable substitution* [15]. The substitution technique is proposed in [15] as a general approach for reducing the cost of Andersen's whole-program analysis. We show that the same technique can be used to design a separate points-to analysis and to generate the corresponding library summaries. We use a particular substitution which reduces the size of the summary and the cost of the separate analysis, while ensuring that the separate analysis is as precise as the whole-program analysis.

[3] Since the analysis is flow-insensitive, points-to pairs cannot be killed—for example, f cannot remove edges $(p, y) \in G$.

$$
\begin{array}{lll}
\textit{Var} & \rightarrow & \text{identifier} \\
\textit{Assign} & \rightarrow & \textit{Var} = \&\textit{Var} \\
 & \mid & \textit{Var} = \textit{Var} \\
 & \mid & \textit{Var} = *\textit{Var} \\
 & \mid & *\textit{Var} = \textit{Var} \\
\textit{ProcDef} & \rightarrow & \textit{Var}(\textit{Var}, \ldots, \textit{Var}) \Rightarrow \textit{Var} \\
\textit{Call} & \rightarrow & \textit{Var} = \textit{Var}(\textit{Var}, \ldots, \textit{Var}) \\
\textit{ProcPtrCall} & \rightarrow & \textit{Var} = (*\textit{Var})(\textit{Var}, \ldots, \textit{Var}) \\
\textit{BasicStmt} & \rightarrow & \textit{Assign} \mid \textit{ProcDef} \mid \textit{Call} \mid \textit{ProcPtrCall}
\end{array}
$$

Figure 1: Grammar for basic statements. *ProcDef* contains the procedure name, the formals, and a unique variable representing the return value of the procedure. A procedure call (direct or through a pointer) contains a variable to which the return value is assigned.

4.1 Variable Substitution

Variable substitution is a technique in which some elements of V are replaced (substituted) by auxiliary representative variables; this produces a new variable set V'. The representative variables are "fresh"—that is, they are not elements of V. The transformation is defined by a *substitution function* $\sigma : V \rightarrow V'$. If a variable v is replaced by the representative variable r_i, we define $\sigma(v) = r_i$. If v is preserved in V', we have $\sigma(v) = v$.

We can use this technique to implement the derivation schema from Section 2. The new analysis lattice is $L' = 2^{V' \times V'}$. The abstraction and concretization functions are

$$
\alpha(G) = \{(\sigma(u), \sigma(v)) \mid (u, v) \in G\}
$$
$$
\gamma(G') = \{(u, v) \mid (\sigma(u), \sigma(v)) \in G'\}
$$

According to Definition 1, a *safe solution* for the separate analysis contains the points-to pair $(\sigma(u), \sigma(v))$ for any points-to pair (u, v) computed by Andersen's whole-program analysis.

The new set of transfer function F' can be derived from F by replacing (substituting) every occurrence of each variable v with $\sigma(v)$. For example, if F contains the transfer function for the assignment "$p = q$", F' will contain the transfer function for the assignment "$\sigma(p) = \sigma(q)$". This transformation can be thought of as rewriting the original program.[4] It can be proven that variable substitution applied to $f \in F$ results in $f' \in F'$ such that $f' = \gamma \circ f \circ \alpha$. As shown in Section 2, this property ensures the safety of the separate analysis.

[4] As a technical detail, we must restrict σ to map different procedure names from the original program to different variables from V'. If two procedure names are mapped to the same element of V', direct procedure calls may become ambiguous.

This formalism presents a uniform approach for generating and using library summaries. Summary generation begins by extracting all variables and all basic statements (as defined in Figure 1) from the library source code. Then some of the library variables are replaced by representative variables. The resulting set of variables and modified basic statements is stored as the library summary. Later, at the beginning of the separate analysis, the summary is unioned with the variables and basic statements from the main component. Then a fixed-point solution is computed as in the whole-program Andersen's analysis; in fact, an already existing implementation of the whole-program analysis can be directly used to implement the separate analysis.

This approach is based on a substitution function which replaces only variables that do not occur outside the library. Applying the substitution to the variables from the library produces a subset of V'. Basic statements are essentially compact representations of transfer functions. Therefore, applying the substitution to the basic statements from the library produces a subset of F'. These subsets are stored as library summary; the rest of V' and F' is constructed from the main component at the beginning of the separate analysis.

The substitution function should preserve the names of all globals and procedures exported by the library because of possible references to them in the main component. All other variables from the library should be replaced by representative variables. As discussed in Section 1, one of our goals is to hide the internals of the library from the users of the summary. For example, a variable with a name such as `union_find_init` may convey sensitive information about the library; if the variable is not exported, it should be replaced by a representative variable with a non-descriptive name such as `rep_563`. This name obfuscation, together with the complete lack of control-flow information in the summary, should provide adequate information hiding.

4.2 Cost Reduction

The simplest summary can be constructed by a substitution function which maps each non-exported variable to a *unique* representative. In this case, there is a one-to-one correspondence between V' and V, and the separate analysis is essentially equivalent to the whole-program Andersen's analysis. We refer to this kind of summary as *baseline summary*.

The disadvantage of this approach is that no analysis work is done at summary generation time. Since the summary could be used by many client programs, it is desirable to precompute some analysis information about the library and to encode it in the summary. This can be achieved by a substitution function which maps many elements of V to the same representative. This reduces the number of variables; as a result, the cost of the following separate analysis is reduced because the constructed points-to graph is smaller. The experiments from Section 7 show that significant cost reduction can be achieved with this approach.

Reducing the number of variables also reduces the number of transfer functions. For example, if variables x and y are mapped to the same representative, the two statements "$x = z$" and "$y = z$" are replaced by a single statement in F'. The reduction in the number of variables and statements results in smaller summaries and less disk space needed to store them.

4.3 Precision

We assume that the points-to solution is used to determine the memory locations accessed through pointer dereferences in the main component. Thus, the analysis outputs the points-to sets of all variables that occur in the main component. For any such variable v, let $Pt(v)$ and $Pt'(v)$ be the points-to sets computed by the whole-program and the separate analysis, respectively. Because the separate analysis is safe, it is guaranteed that for every $x \in Pt(v)$ we have $\sigma(x) \in Pt'(v)$.

We want to ensure that the points-to sets computed by the separate analysis are *as precise* as the points-to sets computed by the whole-program analysis. We consider $Pt(v)$ and $Pt'(v)$ to have the same precision if there exists a *one-to-one correspondence* between them. Based on the standard definition of such a correspondence, we require the following properties to be true for any variable v that may occur in the main component:

- *Injective Property*: For any $x, y \in Pt(v)$ such that $x \neq y$, we have $\sigma(x) \neq \sigma(y)$
- *Surjective Property*: For any $x' \in Pt'(v)$, there exists $x \in Pt(v)$ such that $\sigma(x) = x'$

The only variables that could be elements of points-to sets are the ones whose address is taken somewhere in the program. The injective property can be guaranteed if for any such v we ensure that $\sigma(v)$ is unique—that is, no other element of V is mapped to $\sigma(v)$.

The surjective property states that the separate analysis does not create spurious points-to relationships. In general, this property does not hold. For example, consider two library variables p and q such that $Pt(p) = \{x\}$, $Pt(q) = \{y\}$, and the substitution preserves x and y. If both p and q are replaced by the same representative r, we have $Pt'(r) \supseteq \{x, y\}$. Suppose t is preserved by the substitution and its only value comes from the assignment "$t = p$" in the original program; then the separate analysis will create the spurious points-to pair (t, y).

To satisfy the surjective property, we use a substitution function based on *equivalence sets* [15]. An equivalence set contains variables that have the same points-to set in the solution computed by Andersen's whole-program analysis. The substitution uses a collection $S_1, \ldots, S_k$ of disjoint equivalence sets computed by the algorithm in Section 5. Each set S_i is assigned a unique representative variable r_i. If a non-exported variable v belongs to the equivalence set S_i (and its address is not taken), we define $\sigma(v) = r_i$; otherwise, v is mapped to a unique representative variable. The surjective property of this substitution is a corollary of a result from [15].

4.4 Further Simplifications

The size of the summary can be reduced by two techniques used in [15] to reduce the cost of the whole-program analysis. After the summary is generated, it can be simplified through non-pointer elimination. A *non-pointer* is a variable with an empty points-to set. Along with computing equivalence sets, the algorithm from Section 5 detects some of the non-pointers in the library.[5] This information can be used to reduce the number of statements in the summary. For example, if x is a non-pointer, we can safely eliminate every assignment whose left-hand side or right-hand side is $\sigma(x)$.

The algorithm from Section 5 is also capable of detecting some variables that have singleton points-to sets. If we know that $Pt(x) = \{y\}$, each occurrence of $*\sigma(x)$ can be replaced by $\sigma(y)$. This *known-pointer elimination* may reduce the size of the summary by creating duplicate statements which can be eliminated; it also reduces the amount of work for the subsequent separate points-to analysis.

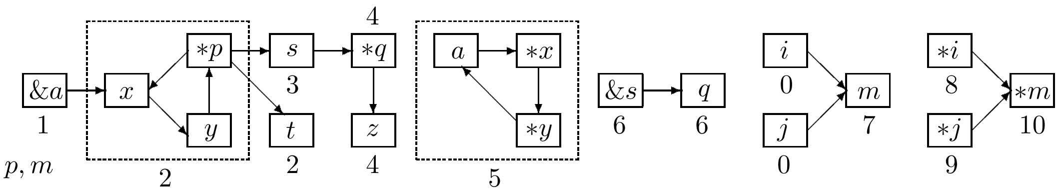
5 Computation of Equivalence Sets

This section briefly describes our approach for computing equivalence sets. The algorithm we use is a modification of a similar algorithm from [15]; the differences are due to the lack of information about the callers of the library. The algorithm has linear complexity and produces high-quality equivalence sets.

Subset Graph The algorithm traverses all basic statements in the library and constructs a *subset graph* G_s. The nodes in G_s represent points-to sets. For each variable v, G_s contains nodes $n(v)$, $n(*v)$, and (optionally) $n(\&v)$. Node $n(v)$ represents Andersen's points-to set $Pt(v)$. Node $n(*v)$ represents the union of $Pt(x)$ for all $x \in Pt(v)$. Node $n(\&v)$ is created only if the address

[5] Declared types cannot be used to determine non-pointers because of the weak type system of C.

$x = \&a$ $\quad a = *y$
$x = *p$ $\quad q = \&s$
$y = x$ $\quad z = *q$
$*p = y$ $\quad m = i$
$s = *p$ $\quad m = j$
$t = *p$ $\quad$ *exported* p, m



Substitution function		After substitution		Final summary	
$\sigma(x) = r2$	$\sigma(a) = r5$	$r2 = \&r5$	$r5 = *r2$	$r2 = \&r5$	$r5 = *r2$
$\sigma(y) = r2$	$\sigma(q) = r6$	$r2 = *p$	$r6 = \&r3$	$r2 = *p$	$r6 = \&r3$
$\sigma(p) = p$	$\sigma(i) = r0$	$r2 = r2$	$r4 = *r6$	$*p = r2$	$r4 = r3$
$\sigma(s) = r3$	$\sigma(j) = r0$	$*p = r2$	$m = r0$	$r3 = *p$	
$\sigma(t) = r2$	$\sigma(m) = m$	$r3 = *p$	$m = r0$		
$\sigma(z) = r4$		$r2 = *p$			

Figure 2: Subset graph and substitution function. Variables p and m are exported (i.e., they are visible in the main component). The numbers show SCC labels assigned by the algorithm. The equivalence sets with more than one element are $\{x, y, t\}$ and $\{i, j\}$. The final summary is obtained after non-pointer and known-pointer elimination.

of v is taken somewhere in the library; it represents the points-to set $\{v\}$.

Edges in G_s represent subset relationships between points-to sets: for every edge (n_1, n_2), we have $Pt(n_1) \subseteq Pt(n_2)$. The edge set encodes all subset relationships that can be directly inferred from the basic statements in the library. For example,

- for each assignment "$p = \&x$", the graph contains edges $(n(\&x), n(p))$ and $(n(x), n(*p))$
- for each assignment "$p = q$", the graph contains edges $(n(q), n(p))$ and $(n(*q), n(*p))$

The two other kinds of assignments from Figure 1 are represented in similar fashion. A direct call generates edges corresponding to a set of assignments from actuals to formals. The target procedures of an indirect call are unknown; thus, no edges are generated by such calls. Figure 2 shows a set of basic statements and the corresponding subset graph (isolated nodes are not shown).

Direct Nodes A *direct node* is a variable node $n(v)$ whose points-to set is equal to the union of the points-to sets of its predecessor nodes in the subset graph. Any value that may be assigned to v should be explicitly represented in the graph. For example, if v is an exported global or a formal parameter of an exported procedure, it may get its value from unknown statements in the main component; thus, node $n(v)$ is *not* direct. Similarly, if the address of v is taken somewhere in the library, v can be assigned indirectly through a pointer dereference, and $n(v)$ is not a direct node.[6] In the example from Figure 2, the variable nodes that are not direct are $n(p)$, $n(m)$, $n(a)$ and $n(s)$.

[6] There are additional cases related to variables that get their values because of indirect calls [15].

Computation of Equivalence Sets Because edges in the graph represent subset relationships, all nodes in the same strongly connected component (SCC) have the same points-to set; thus, the variables in a SCC form an equivalence set. The algorithm computes the SCCs of the subset graph and creates the corresponding SCC-DAG G'_s; each node in G'_s represents a SCC of G_s.

A direct node $n' \in G'_s$ corresponds to a SCC that contains only direct nodes from the original subset graph. The points-to set of n' is equal to the union of the points-to sets of its predecessors in G'_s. Therefore, if all predecessors of n' belong to the same equivalence set, n' itself belongs to that same equivalence set. This property is used by the algorithm to create equivalence sets that contain more that one SCC.

The equivalence sets are computed by traversing G'_s in topological sort order and assigning an integer label to each node. If a node n' is direct and all of its predecessors have the same label, n' is assigned that label as well; otherwise, n' is assigned a "fresh" label. At the end, the variables that belong to SCCs with the same label form an equivalence set.

The algorithm pays special attention to direct nodes n' that have no predecessors. Such n' contain variables that are non-pointers. For example, if node $n(v)$ belongs to n', in the original program v gets all of its values from non-pointer assignments (e.g., "$v = 0$"). All such n' are assigned label 0 to identify them as non-pointer SCCs. This information is later used to perform non-pointer elimination.

If a node $n(\&v)$ is assigned label l, we know that the points-to set corresponding to l is exactly $\{v\}$. Any variable that is assigned label l has a known points-to set; the pair (l, v) is remembered and later used to perform known-pointer elimination.

The example in Figure 2 shows one possible assignment of SCC labels. A label can be inherited from the predecessor nodes—for example, SCC $\{n(t)\}$ is direct and gets the label of its predecessor, which results in adding t to the equivalence set $\{x, y\}$. Label 0 is used for the non-pointer SCCs $\{n(i)\}$ and $\{n(j)\}$. Even though both predecessors of $\{n(m)\}$ have the same label, it cannot be used for $\{n(m)\}$ because m is exported and may be modified in the main component. Finally, note that $\{n(q)\}$ gets label 6, which corresponds to the known points-to set $\{s\}$.

The substitution function can be defined based on the computed labels; an example is given in Figure 2. Variables $r2$, $r3$, etc. are "fresh" representative variables. The final summary is obtained after non-pointer and known-pointer elimination.

6 Handling Multiple Libraries

The separate points-to analysis from Section 4 was presented under the assumption that the program contains only one library component. We can use the same approach for programs containing multiple libraries. In this case, each library component is preanalyzed in order to generate a summary for it. The separate analysis of the main component uses the summaries for all libraries to determine the lattice and the transfer functions, and to compute a fixed-point solution.

Each summary can be generated using variable substitution, similarly to Section 4. However, if the library references globals and procedures exported by another library, their names should be preserved by the substitution function. This restriction ensures that names which occur in both libraries are preserved in the two summaries, and will be properly matched by the separate analysis of the main component.

The algorithm for computing equivalence sets must be modified to take into account possible interactions between libraries. Consider a library L_1 which references globals and invokes procedures exported by another library L_2. In the subset graph for L_1 we have to label as *non-direct* any node corresponding to a global exported by L_2—some of the values of this global may come from unknown statements inside L_2. The same should be done for any variable which is assigned the return value of a call to a procedure exported by L_2. Essentially, this means that the equivalence sets for L_1 are computed under worst-case assumptions about L_2. This approach allows the summary for each library to be computed completely independently of the other libraries.

If the summary for L_2 has already been computed, it can be used instead of worst-case assumptions. In this more precise approach, the equivalence sets for L_1 can be computed by using a subset graph which represents the statements in L_1 *and* the statements in the summary for L_2. This may result in larger equivalence sets for the variables from L_1. For example, if a global exported by L_2 is only read (but not modified) inside L_2, the corresponding node can be classified as direct; this may help detect more variables from L_1 that are in the same equivalence set. As another example, the summary for L_2 may show that a procedure exported by L_2 returns a non-pointer value; this may allow better non-pointer elimination in the summary for L_1.

Program	LOC	Vars	Stmts
bzip2-0.9.0c	6.3K	8646	8453
tiff2ps-3.4	20.9K	24446	24618
cjpeg-5b	22.7K	20289	20610
gasp-1.2	26.0K	12999	12611
unzip-5.40	27.6K	26743	27001
fudgit-2.41	29.8K	28356	27903
gnuplot-3.7.1	66.2K	53174	50522
povray-3.1	133.9K	114736	112369

Table 1: Data programs.

Library	LOC	Vars	Stmts
libbz2	4.5K (71%)	73%	81%
libtiff-3.4	19.6K (94%)	81%	82%
libjpeg-5b	19.1K (84%)	79%	82%
libiberty	11.2K (43%)	45%	46%
zlib-1.1.3	8.0K (29%)	29%	32%
readline-2.0	14.8K (50%)	33%	37%
libgd-1.3	22.2K (34%)	5%	5%
libpng-1.0.3	25.7K (19%)	21%	22%

Table 2: Libraries.

7 Experiments and Results

This section presents our experiments in computing and using two kinds of library summaries for Andersen's analysis. The first kind—the *baseline summary*—is constructed by replacing each non-exported library variable by a unique representative. As described in Section 4.2, this summary does not encode any precomputed analysis information. The second kind of summary—the *reduced summary*—is computed using substitution of equivalence sets, as described in Section 4; the equivalence sets are computed with the algorithm from Section 5. Our experiments have two goals: first, to deter-

Library	T_B (sec)	T_R (sec)	S_B (Kb)	S_R (Kb)	*Bin* (Kb)
libbz2	0.7	4.2	161.2	23.9	53.6
libtiff	2.6	13.5	566.6	125.4	765.7
libjpeg	2.0	11.0	460.2	99.2	132.6
libiberty	0.6	3.9	148.3	42.5	195.4
zlib	0.9	5.1	204.6	26.3	70.6
readline	0.9	6.3	263.4	59.4	193.5
libgd	0.8	2.4	69.3	17.2	159.2
libpng	2.8	16.2	696.1	105.9	221.9

Table 3: Summary cost. T_B and T_R are the times to generate the baseline and the reduced summary. S_B and S_R are the summary sizes. The last column shows the size of the library binary.

Program	T_B (sec)	Δ_T	S_B (Mb)	Δ_S
bzip2	5.4	63%	13.0	49%
tiff2ps	22.5	54%	37.5	53%
cjpeg	15.7	54%	32.1	50%
gasp	9.7	27%	18.6	23%
unzip	21.0	22%	39.5	13%
fudgit	39.8	17%	59.5	14%
gnuplot	47.3	3%	79.6	5%
povray	111.6	14%	146.7	14%

Table 4: Cost of the separate analysis. T_B is the analysis time with the baseline summary. Δ_T is the time reduction when using the reduced summary. S_B and Δ_S are the corresponding measurements for analysis memory.

mine the cost of computing and storing the two kinds of summaries, and second, to estimate the impact of the summaries on the cost of the following points-to analysis of the main component.

Our implementation of Andersen's analysis is written in ML and is based on the *Bane* toolkit for constraint-based analysis [6]. The analysis is performed by generating and solving a system of set-inclusion constraints. Structures and array are treated as monolithic objects and their elements are not distinguished. Calls to standard library functions (e.g., `strcpy`, `cos`, `printf`) are handled by stubs which simulate the effects of the called functions. The stubs are summaries produced by hand from the specifications of the library functions.[7]

All experiments were performed on a 360MHz *Sun Ultra-60* machine with 512Mb physical memory. The reported times are the best out of three runs. The memory consumption was measured by instrumenting the ML garbage collector to report the amount of live data after each garbage collection.

Our data programs are described in Table 1. The table shows the size of each program in lines of source code. It also shows the number of variables and basic statements, as defined by the grammar in Figure 1; these numbers represent the size of the input for the points-to analysis. All sizes are measured including the libraries described below.

Each data program contains a well-defined library component, which is designed as a general-purpose library and is developed independently of any client applications. For example, `unzip` is an extraction utility for compressed archives which uses the general-purpose data compression library `zlib`. As another example, `fudgit` is a fitting program; it uses the GNU Readline Library to provide command line interface.

The libraries are described in Table 2; each row shows the library used by the program in the corresponding row of Table 1. The size of the library source is given as an absolute value and as a percentage of the size of the program. The last two columns show the number of variables and basic statements as a percentage of the whole-program values; the numbers indicate what portion of the points-to analysis input is related to the library.

Our first set of experiments measured the cost of computing and storing the library summaries. First, we measured the time needed to generate the summaries; the results are shown in columns T_B and T_R of Table 3. For the baseline summary, T_B is the time to traverse the intermediate representation and write the summary to disk. For the reduced summary, T_R also includes the time to compute the equivalence sets and to perform substitution and elimination. The results clearly show that in both cases summary computation is inexpensive.

We also measured the amount of disk space needed to store the summaries; the results are shown in columns S_B and S_R of Table 3. The last column shows the size of the binary of the compiled library. The size of the reduced summary is between 11% and 75% of the binary size (36% on average), and therefore is practical.

Our second set of experiments measured the impact of the summaries on the cost of the separate analysis. Table 4 shows the change in analysis cost when the reduced summary is used instead of the baseline summary. Column T_B shows the analysis time with the baseline summary, including the time to read the summary from disk. Column Δ_T shows the reduction in analysis time when using the reduced summary. Column S_B shows the amount of memory needed by the separate analysis with the baseline summary; the last column shows the reduction when using the reduced summary.

[7] We are currently investigating how our approach can be used to produce summaries for the standard libraries. This problem presents interesting challenges, because many of the standard libraries operate in the domain of the operating system.

The baseline summary does not encode any precomputed summary information. On the other hand, some analysis work is done when generating the reduced summary; as a result, the cost of the separate analysis is reduced. The change is proportional to the relative size of the library. For example, for `bzip2` the majority of the program is in the library, and the cost reduction is significant. In `gnuplot` only 5% of the variables and basic statements come from the library, and the analysis cost is reduced accordingly.

8 Related Work

The work in [16] presents an approach for analyzing program fragments. Fragment analysis uses summary values representing the effects of calls to the fragment, and summary functions representing the effects of calls from the fragment to outside procedures. The relationship between fragment analysis and whole-program analysis is expressed by an abstraction relation used to define requirements which ensure the safety of the fragment analysis. Although a set of the properties that a fragment analysis should satisfy is defined, no constructive way of designing such analyses is presented.

Our separate analysis is an instance of the above approach when the analyzed fragment is the main component. However, the formalism is more restrictive than that in [16] because we use an abstraction function; this allows construction of the separate analysis by replacing each transfer function f by $\gamma \circ f \circ \alpha$. Unlike [16], we do not explicitly use a single summary function to represent the effects of calls to a procedure exported by the library component. Instead, we use a set of simple functions derived from the original transfer functions in the library; this set is essentially a compact representation of the library data-flow effects. This approach is simple to implement and has the advantage that the summary is easy to integrate with the rest of the analysis; in fact, an already existing implementation of a whole-program analysis can be used *without any modification*.

The idea of using a single summary function to represent the effects of a procedure is used by some context-sensitive interprocedural analyses [20, 10, 14, 25, 3] and is generalized to program components in [8]. In all such analyses, the summary function is computed through function composition and meet.[8] This approach allows a single function to represent the effects of all execution paths that are possible during the lifetime of the procedure. However, function composition and meet are expensive operations; even though various techniques have been employed to make this approach more efficient, analyses based on this idea tend to be complex and expensive. Our technique for summarizing the effects of a library is a simple and practical alternative to this approach.

Harrold and Rothermel [8] present a separate flow- and context-sensitive pointer analysis in which a software module is analyzed separately and later linked with other modules. They compute and store the module solution for all possible calling contexts; this solution is essentially a tabular representation of a summary function for the module. Chatterjee et al. [3] use the strongly connected component decomposition of the call graph of an object-oriented program to define a modular points-to analysis. The data-flow effects of each method are summarized by a function used subsequently in the analysis of calls to this method. Flanagan and Felleisen [7] show how to perform componential constraint-based analysis for Scheme. They derive a simplified constraint system for each program component and store it in a constraint file; these systems are then combined and information is propagated between them. The separate information used by these approaches for describing a component or a method is similar in concept to our use of library summaries.

Various analyses use representative variables as placeholders for sets of related variables [13, 10, 5, 25, 8, 6, 3, 16, 12, 15]. In particular, in [15] variable substitution is introduced as a general technique for reducing the cost of whole-program points-to analysis, possibly at the expense of some precision loss. This work also shows that substitution based on equivalence sets preserves the precision of Andersen's analysis; here these ideas are instantiated to generation of precise and practical library summaries.

Alpern et al. [1] present an approach for detecting variables that have the same numerical value at a program point, an idea similar to the equivalence sets of pointers from Section 4.3. Their algorithm builds a value graph which represents a symbolic execution of the program; equality of variables is detected by finding graph nodes which represent the same value.

9 Conclusions and Future Work

We have shown a theoretical approach for deriving a separate analysis from a flow- and context-insensitive whole-program analysis. The approach allows construction of library summaries through modification of each transfer function in the library. We apply this approach to Andersen's analysis by using variable substitution based on equivalence sets. The resulting separate analysis preserves the precision of the whole-program analysis without exposing the internals of the library. The generated summaries encode partial analysis information which reduces the cost of the separate analysis.

[8] $h = f \wedge g$ iff $h(x) = f(x) \wedge g(x)$ for all x.

The main advantage of our approach is its practicality. As our experiments show, the summaries are inexpensive to compute and store. Furthermore, the summary generation algorithm is relatively simple and can be implemented without significant effort. Finally, the summary information can be easily integrated with an already existing implementation of the whole-program analysis.

One direction of future work is to consider analyses that are clients of the points-to information—for example, side-effect analysis, live variables analysis, or dependence analysis. It is a an open question how to generate and use summary information for these analyses, and how they should interact with the separate points-to analysis. Since most of these analyses have some form of flow or context sensitivity, it might be necessary to generalize the theoretical approach from Section 2 for flow- or context-sensitive analyses.

10 Acknowledgments

We would like to thank Satish Chandra for his comments on an earlier version of this paper.

References

[1] B. Alpern, M. Wegman, and K. Zadeck. Detecting equality of variables in programs. In *Proc. Symp. Principles of Programming Languages*, pages 1–11, 1988.

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

[3] R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *Proc. Symp. Principles of Programming Languages*, pages 133–146, 1999.

[4] P. Cousot and R. Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixed points. In *Proc. Symp. Principles of Programming Languages*, pages 238–252, 1977.

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

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

[7] C. Flanagan and M. Felleisen. Componential set-based analysis. In *Proc. Conf. Programming Language Design and Implementation*, pages 235–248, 1997.

[8] M. J. Harrold and G. Rothermel. Separate computation of alias information for reuse. *IEEE Trans. Software Engineering*, 22(7):442–460, July 1996.

[9] M. Hind, M. Burke, P. Carini, and J. Choi. Interprocedural pointer alias analysis. *ACM Trans. Programming Languages and Systems*, 21(4):848–894, May 1999.

[10] W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proc. Conf. Programming Language Design and Implementation*, pages 235–248, 1992.

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

[12] 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.

[13] T. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Proc. Symp. Principles of Programming Languages*, pages 184–196, 1990.

[14] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Proc. Symp. Principles of Programming Languages*, pages 49–61, 1995.

[15] A. Rountev and S. Chandra. Off-line variable substituition for scaling points-to analysis. In *Proc. Conf. Programming Language Design and Implementation*, 2000.

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

[17] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proc. Conf. Programming Language Design and Implementation*, pages 13–22, 1995.

[18] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Proc. Static Analysis Symposium*, LNCS 1302, pages 16–34, 1997.

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

[20] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 189–234. Prentice Hall, 1981.

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

[22] P. Stocks, B. G. Ryder, W. Landi, and S. Zhang. Comparing flow- and context-sensitivity on the modification side-effects problem. In *Proc. International Symposium on Software Testing and Analysis*, pages 21–31, 1998.

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

[24] E. Weyuker. Testing component-based software: A cautionary tale. *IEEE Software*, 15(5):54–59, Sept. 1998.

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

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