Points-to and Side-effect Analyses for Programs Built with Precompiled Libraries

Atanas Rountev and Barbara G. Ryder

Department of Computer Science, Rutgers University, Piscataway, NJ 08854, USA
`{rountev,ryder}@cs.rutgers.edu`

Abstract. Large programs are typically built from separate modules. Traditional *whole-program analysis* cannot be used in the context of such modular development. In this paper we consider analysis for programs that combine client modules with precompiled library modules. We define *separate analyses* that allow library modules and client modules to be analyzed separately from each other. Our target analyses are Andersen's points-to analysis for C [1] and a side-effect analysis based on it. We perform separate points-to and side-effect analyses of a library module by using *worst-case assumptions* about the rest of the program. We also show how to construct *summary information* about a library module and how to use it for separate analysis of client modules. Our empirical results show that the separate points-to analyses are practical even for large modules, and that the cost of constructing and storing library summaries is low. This work is a step toward incorporating practical points-to and side-effect analyses in realistic compilers and software productivity tools.

1 Introduction

Large programs are typically built from separate modules. Such modular development allows better software management and provides a practical compilation model: instead of (re)compiling large programs from scratch, compilers can perform separate compilation of individual modules. This approach allows sharing of modules between programs; for example, an already compiled library module can be reused with no implementation or compilation cost. This development model also allows different modules to be developed by different teams, at different times and in separate locations.

Optimizing compilers and software productivity tools use static analyses to determine various properties of program behavior. Many of these analyses are performed by analyzing the whole program. However, such *whole-program analyses* cannot be used in the context of a modular development process. To make these analyses useful in real-world compilers and software tools, analysis techniques must be adapted to handle modular development.

This paper investigates one instance of this problem. We consider analysis for programs built with reusable *precompiled library modules*. Reusable modules are designed to be combined with many (yet unknown) clients, and are typically packaged as precompiled libraries that are subsequently linked with the client code. We consider programs with two modules: a *library module* that is developed and compiled independently of any particular client, and a *client module* that uses the functionality of the library module.[1] For such programs, compilers cannot use whole-program analyses because the two modules are compiled separately. We show how certain whole-program points-to and side-effect analyses can be extended to handle such applications. Our work is a step toward incorporating practical points-to and side-effect analyses in realistic compilers.

Points-to Analysis and Side-effect Analysis. Modification side-effect analysis (MOD) determines, for each statement, the variables whose values may be modified by that statement. The complementary USE analysis computes similar information for the uses of variable values. Such information plays a fundamental role in optimizing compilers and software tools: it enables a variety of other analyses (e.g., reaching definitions analysis, live variables analysis, etc.), which in turn are needed for code optimization and for program understanding, restructuring and testing. For brevity, we only discuss MOD analysis; all results trivially apply to USE analysis, because the two analysis problems are essentially identical.

Side-effect analysis for languages like C is difficult because of pointer usage; typically, a *points-to analysis* is needed to resolve pointer dereferences. Our work is focused on flow- and context-insensitive points-to analyses [1, 18, 21, 17, 5], which ignore the flow of control and the calling contexts of procedures. Such analyses are very efficient and can be used in production-strength compilers with little additional cost. Several combinations of a MOD analysis and a flow- and context-insensitive points-to analysis have been investigated [16, 15, 10]. Similarly to [16, 10], we consider a MOD analysis based on Andersen's points-to analysis for C [1]. Even though we investigate these specific analyses, our results also apply to similar flow- and context-insensitive points-to analyses (e.g., [18, 17, 5]) and related MOD analyses.

1.1 Separate Analysis of Modules

Optimizing compilers cannot use whole-program analyses for programs built with precompiled library modules. When compiling and optimizing a library module, the compiler has no available information about client modules. When compiling and optimizing a client module, only the library binary is available. Therefore, the compiler must use *separate analyses* that allow each module to be analyzed separately from the rest of the program. We define such separate analyses based on Andersen's analysis and the corresponding MOD analysis.

Worst-case Separate Analysis of Library Modules. The compilation and optimization of the library module is performed independently of any client modules. In this case, the separate analyses used by the optimizing compiler must make *worst-case assumptions* about the rest of the program. The resulting analysis solutions should represent all possible points-to and MOD relationships in the

[1] The work in [12] discusses programs built with more than one library module.

library module; these solutions can then be used for library compilation and optimization. In our approach, the worst-case separate analyses are implemented by adding *auxiliary statements* that model the effects of all possible statements in client modules. The library is combined with the auxiliary statements and the standard whole-program analyses are applied to it. This approach is easy to implement by reusing existing implementations of the whole-program analyses.

Summary-based Separate Analysis of Client Modules. During the compilation and optimization of a client module, the compiler can employ separate analyses that use precomputed *summary information* about the called library module. Such summary-based analyses of the client module can compute more precise points-to and MOD information, as opposed to making worst-case assumptions about the library in the case when only the library binary is available. This improvement is important because the precision of the analyses has significant impact on subsequent analyses and optimizations [16, 10].

The summary information should encode the effects of the library module on arbitrary client modules. Such information can be constructed at the time when the library module is compiled, and can be stored together with the library binary. The library summary can later be used during the separate analysis of any client module, and can be reused across different client modules. In our summary-based analyses, the client module is combined with the summary information, and the result is analyzed as if it were a complete program. This approach can reuse existing implementations of the corresponding whole-program analyses, which allows minimal implementation effort.

Summary Information. Our summaries contain a set of *summary elements*, where each element represents the points-to or MOD effects of one or more library statements. We extract an initial set of elements from the library code and then optimize it by merging equivalent elements and by eliminating irrelevant elements. The resulting summary is *precision-preserving*: with respect to client modules, the summary-based solutions are the same as the solutions that would have been computed if the standard whole-program analyses were possible.

Our approach for summary construction has several advantages. The summary can be generated completely independently of any callers and callees of the library; for example, unlike previous work, our approach can handle callbacks from the library to client modules. The summary construction algorithm is simple to implement, which makes it a good candidate for inclusion in realistic compilers. The summary optimizations reduce the cost of the summary-based analyses without sacrificing any precision. Finally, our experiments show that the cost of constructing and storing the summary is practical.

Contributions. The contributions of our work are the following:

– We propose an approach for separate points-to and MOD analyses of library modules, by using auxiliary statements to encode worst-case assumptions about the rest of the program. The approach can be easily implemented with existing implementations of the corresponding whole-program analyses.

– We present an approach for constructing summary information for a library module. The library summary is constructed independently of the rest of the program, and is optimized by merging equivalent summary elements and by eliminating irrelevant summary elements. With this summary, the summary-based analyses are as precise as the standard whole-program analyses.
– We show how to use the precomputed library summaries to perform separate points-to analysis and MOD analysis of client modules. The summary-based analyses can be implemented with minimal effort by reusing existing implementations of the corresponding whole-program analyses.
– We present empirical results showing that the worst-case points-to analysis and the summary construction algorithm are practical even for large library modules. The results also show that the summary optimizations can significantly reduce the size of the summary and the cost of the summary-based points-to analysis.

2 Whole-program Analyses

In this section we present a conceptual model of Andersen's points-to analysis for C [1] and a MOD analysis based on it. These whole-program analyses are the foundation for our worst-case and summary-based separate analyses.

We assume a simplified C-like program representation (defined formally in [12]), similar to the intermediate representations used in optimizing compilers. Similarly to [18, 17, 16, 6, 5, 10], structures and arrays are represented as monolithic objects without distinguishing their individual elements. Calls to `malloc` and similar functions are replaced by statements of the form "$x = \&heap_i$", where $heap_i$ is a variable unique to the allocation site. Because of the weak type system of C (due to typecasting and union types), we assume that type information is not used in the points-to and MOD analyses[2], and therefore the representation is untyped. Figure 1 shows an example of a program with two modules.

Let V be the set of variables in the program representation. We classify the elements of V as (i) *global variables*, (ii) *procedure variables*, which denote the names of procedures, (iii) *local variables*, including formals, and (iv) *heap variables* introduced at heap-allocation sites. Andersen's analysis constructs a *points-to graph* in which nodes correspond to variables from V. A directed edge (v_1, v_2) shows that v_1 may contain the address of v_2. Each statement defines a *transfer function* that adds edges to points-to graphs. For example, the function for "$*p = q$" is $f(G) = G \cup \{(x, y) \mid (p, x) \in G \wedge (q, y) \in G\}$. Conceptually, the analysis starts with an empty graph and applies transfer functions until a fixed point is reached. Figure 1 shows the points-to solution for the sample program.

We define a MOD algorithm which computes a set of modified variables $Mod(s) \subseteq V$ for each statement s. The algorithm is derived from similar MOD algorithms [16, 15, 10] by adding two *variable filters* that compensate for the some

[2] This analysis approach is the simplest to implement and therefore most likely to be the first one employed by realistic compilers. Our techniques can be easily adapted to approaches that use some form of type information.

```
global g
proc main() {
  local x,y,z,w
  1: x = 1
  2: y = 2
  3: z = &x
  4: g = &y
  5: w = &div
  6: exec(z,w)
}
proc div(a,b) {
  local c,d,e
  7: c = *a
  8: d = *b
  9: e = c / d
  10: *a = e
}
```

Client

```
proc exec(p,fp) {
  local s,u,q,t
  11: s = 3  12: u = 4
  13: t = p
  14: (*fp)(g,t)
  15: q = &s  16: neg(q)
  17: q = &u  18: neg(q)
  19: *t = u
  20: g = t
}
proc neg(r) {
  local i,j
  21: i = *r
  22: j = -i
  23: *r = j
}
```

Lib

Var	$Pt(v)$
z	x
g	x, y
w, fp	div
p, t, b	x
a	x, y
q, r	s, u

Stmt	$Mod(s)$
1	x
6	g, x, y
10	x, y
14	x, y
16,18	s, u
19	x
23	s, u

Fig. 1. Program with two modules $Client = \{main, div\}$ and $Lib = \{exec, neg\}$, and the points-to and MOD solutions computed by the whole-program analyses.

of the imprecision of the underlying points-to analysis. The first filter adds a variable v to $Mod(s)$ only if v represents memory locations whose lifetime is *active* immediately before and immediately after the execution of s. For example, if v is a non-static local variable in procedure P, it represents locations whose lifetime is active only for P and for procedures that are directly or transitively called by P. We define a relation $active(s, v)$ that holds only if (i) v is a global variable, a static local variable, or a heap variable, or (ii) v is a non-static local variable in procedure P_1, s belongs to procedure P_2, and either P_2 is reachable from P_1 in the program call graph, or P_1 and P_2 are the same procedure.

Our second filter is applied only to $Mod(s)$ of a call statement s. Suppose that s invokes procedure P. Among all locations whose lifetime is active with respect to s, only a subset is actually *accessible* by P and the procedures transitively called by P. An active location is accessible if it can be referenced either directly, or through a series of pointer dereferences. Only accessible variables can be modified by the call to P, and only they should be included in $Mod(s)$.

An active global variable is always directly accessible by P and the procedures transitively called by P. Similarly, an active static local could be directly accessible if its defining procedure is P or a callee of P. However, an active non-static local can only be accessed indirectly through pointer dereferences; any *direct* reference to this local in P and in the callees of P accesses a different run-time instance of the local. Similarly, heap variables can only be accessed through pointers. Based on these observations, for each call statement s we define a relation $accessible(s, v)$ that holds only if $active(s, v)$ holds and (i) v is a global variable or a static local variable, or (ii) $v \in Pt(a)$, where a is an actual

```
input     Stmt: set of statements          Proc: set of procedures
          SynMod: Stmt → V × {D, I}        Pt: V → P(V)
          Called: Stmt → P(Proc)
output    Mod: Stmt → P(V)
declare   ProcMod: Proc → P(V)
[1]   foreach s ∈ Stmt do
[2]       if SynMod(s) = (v, D) then
[3]           Mod(s) := {v}
[4]           if v is global or static local then
[5]               add {v} to ProcMod(EnclosingProc(s))
[6]       if SynMod(s) = (v, I) then
[7]           Mod(s) := {x | x ∈ Pt(v) ∧ active(s, x) }
[8]           add Mod(s) to ProcMod(EnclosingProc(s))
[9]   while changes occur in Mod or ProcMod do
[10]      foreach call statement s ∈ Stmt do
[11]          foreach P ∈ Called(s) do
[12]              Mod(s) := Mod(s) ∪ {x | x ∈ ProcMod(P) ∧ accessible(s, x)}
[13]          add Mod(s) to ProcMod(EnclosingProc(s))
```

Fig. 2. Whole-program MOD analysis. $\mathcal{P}(X)$ denotes the power set of X.

parameter of s, or (iii) $v \in Pt(w)$ and $accessible(s, w)$ holds for some $w \in V$. For example, variable u from Figure 1 is active for all statements in procedures *exec*, *div*, and *neg*. With respect to calls 16 and 18, u is accessible because it is pointed to by actual q; however, u is not accessible for call 14.

The MOD algorithm is shown in Figure 2. *SynMod* stores *syntactic modifications*, which are pairs (v, d) where v is a variable that occurs on the left-hand side of an assignment or a call, and $d \in \{D, I\}$ indicates whether the modification is direct or indirect. For example, in Figure 1, $SynMod(s)$ is (x, D) for statement 1 and (a, I) for statement 10. *Called* contains the procedures invoked by each call statement; indirect calls are resolved using the points-to solution. *ProcMod* stores the sets of variables modified by each procedure. Filters *active* and *accessible* are used whenever a variable is added to a *Mod* set (lines 7 and 12). In addition, we filter out *direct* modifications of non-static locals (lines 4–5), because the lifetime of the modified memory location terminates when the procedure returns. This filtering improves the precision of the analysis in the presence of recursion [12] and results in more compact summaries, as discussed in Section 4. Figure 1 shows some of the *Mod* sets for the sample program.

3 Worst-case Separate Analysis of Library Modules

During the compilation and optimization of a library module, a compiler must use separate analyses that make worst-case assumptions about possible client modules. In our approach, these assumptions are introduced by adding *auxiliary statements* to the library. The combination of the auxiliary statements and the library is treated as a complete program and the whole-program analyses

```
global v_ph
proc p_ph(f_1,..,f_n) returns p_ph_ret {
  v_ph = f_i  (1 ≤ i ≤ n)          v_ph = &v  (v ∈ V_exp)        v_ph = &v_ph
  v_ph = *v_ph                     *v_ph = v_ph                  v_ph = &p_ph
  v_ph = (*v_ph)(v_ph,..,v_ph)  (with m actuals)
  p_ph_ret = v_ph
}
```

Fig. 3. Placeholder procedure and auxiliary statements.

from Section 2 are applied to it. The resulting solutions are *safe abstractions* of all points-to and MOD relationships in all possible complete programs. These solutions can then be used for library compilation and optimization.

Given a library module *Lib*, consider a complete program p containing *Lib* and some client module. Let $PROG(Lib)$ be the (infinite) set of all such complete programs. We use V_p to denote the variable set of any such p. Let $V_L \subseteq V_p$ be the set of all variables that occur in statements in *Lib* (this set is independent of any particular p). Also, let $V_{exp} \subseteq V_L$ be the set of all variables that may be explicitly referenced by client modules; we refer to such variables as *exported*. Exported variables are either globals that could be directly accessed by library clients, or names of procedures that could be directly called by client code.

Example. We use module *Lib* from Figure 1 as our running example; for convenience, the module is shown again in Figure 4. For this module, $V_L = \{exec, p, fp, s, u, t, g, q, neg, r, i, j\}$. For the purpose of this example, we assume that $V_{exp} = \{g, exec\}$. Note that the complete program in Figure 1 is one of the (infinitely many) elements of $PROG(Lib)$.

Auxiliary Statements. The statements are located in a *placeholder procedure p_ph* which represents all procedures in all possible client modules. The statements use a *placeholder variable v_ph* which represents all global, local, and heap variables $v \in (V_p - V_L)$ for all $p \in PROG(Lib)$. The placeholder procedure and the auxiliary statements are shown in Figure 3. Each statement represents different kinds of statements that could occur in client modules; for example, "$v\_ph = *v\_ph$" represents statements of the form "$u = *w$", where $u, w \in (V_p - V_L)$.

The indirect call through *v_ph* represents all calls originating from client modules. In the worst-case analyses, the targets of this call could be (i) *p_ph*, (ii) the procedures from V_{exp}, or (iii) any library procedure whose address is taken somewhere in *Lib*. To model all possible formal-actual pairs, the number of actuals m should be equal to the maximum number of formals for all possible target procedures. Similarly, all callbacks from the library are represented by indirect calls to *p_ph*; thus, the number of formals n in *p_ph* should be equal to the maximum number of actuals used at indirect calls in the library.

Worst-case Analyses. The worst-case points-to and MOD analyses combine the library module with the auxiliary statements and apply the whole-program analyses from Section 2. An important advantage of this approach is its simple implementation by reusing already existing implementations of the whole-program

```
global g
proc exec(p,fp) {                                           proc neg(r) {
  local s,u,q,t     11: s = 3          12: u = 4              local i,j
  13: t = p         14: (*fp)(g,t)     15: q = &u             21: i = *r
  16: neg(q)        17: q = &s         18: neg(q)             22: j = -i
  19: *t = u        20: g = t                                 23: *r = j
}                                                           }
```

$Pt_{wc}(v) = \emptyset$ for $v \in \{s, u, i, j, neg\}$
$Pt_{wc}(v) = \{s, u\}$ for $v \in \{q, r\}$
$Pt_{wc}(v) = \{v\_ph, p\_ph, g, exec\}$ for every other $v \in V_L \cup \{v\_ph, p\_ph, f_i, p\_ph\_ret\}$

$Mod_{wc}(s) = \{v\_ph, g\}$ for $s \in \{14, 19\}$ and for the indirect call through v_ph
$Mod_{wc}(23) = \{s, u\}$
$Called_{wc}(s) = \{p\_ph, exec\}$ for $s = 14$ and for the indirect call through v_ph

Fig. 4. Module *Lib* ($V_{exp} = \{g, exec\}$) and the corresponding worst-case solutions.

analyses. It can be proven that the resulting worst-case solution is a safe abstraction of all points-to and MOD relationships in all $p \in PROG(Lib)$ [12].

Example. Consider module *Lib* from Figure 4. For the purpose of this example assume that $V_{exp} = \{g, exec\}$. The library has one indirect call with two actuals; thus, *p_ph* should have two formals ($n = 2$). The indirect call through *v_ph* has possible targets *exec* and *p_ph*, and should have two actuals ($m = 2$). The points-to solution shown in Figure 4 represents all possible points-to pairs in all complete programs. For example, pair (p, *v_ph*) shows that p can point to some unknown variable from some client module; similarly, (p, *p_ph*) indicates that p can point to an unknown procedure from some client module. The computed call graph represents all possible calls in all complete programs. For example, $Called_{wc}(14) = \{p\_ph, exec\}$ represents the possible callback from *exec* to some client module and the possible recursive call of *exec*. Similarly, the computed MOD solution represents all possible *Mod* sets in all complete programs.

4 Summary Construction and Summary-based Analysis

In this section we present our approach for constructing summary information for a library module, and show how to use the library summaries for separate summary-based analysis of client modules.

Some previous work on context-sensitive points-to analysis [9, 2, 3] uses *complete summary functions* to encode the cumulative effects of all statements in a procedure P and in all procedures transitively called by P. This approach can be used to produce summary information for a library module, by computing and storing the summary functions for all exported procedures. However, this technique makes the implicit assumption that a called procedure can be analyzed either before, or together with its callers. This assumption can be easily violated—for example, when a library module calls another library module, there are no guarantees that any summary information will be available for the callee [12]. In the presence of callbacks, the library module may call client modules that do not even exist at the time of summary construction. For example, for

1. Variable summary

$Procedures = \{exec, neg\}$ $\quad$ $Locals(exec) = \{p, fp, s, u, q, t\}$

$Globals = \{g\}$ $\quad$ $Locals(neg) = \{r, i, j\}$

2. Points-to summary

```
proc exec(p,fp)    q=&s      *t=u          i=*r
t=p                neg(q)    g=t           *r=j
(*fp)(g,t)         q=&u      proc neg(r)
```

3. Mod summary

$SynMod(exec) = \{(t, I), (g, D)\}$ $\quad$ $SynMod(neg) = \{(r, I)\}$

$SynCall(exec) = \{(fp, I), (neg, D)\}$ $\quad$ $SynCall(neg) = \emptyset$

Fig. 5. Basic summary for module *Lib*.

module *Lib* from Figure 4, the effects of *exec* cannot be expressed by a summary function because of the callback to some unknown client module.

We use a different summary construction approach that has several advantages. The summary can be constructed independently of any callers and callees of the library; therefore, our approach can handle callbacks. The summary construction algorithm is inexpensive and simple to implement, which makes it a good candidate for inclusion in realistic compilers. The summary is *precision-preserving*: for every statement in the client module, the MOD solution computed by the summary-based analyses is the same as the solution that would have been computed if the standard whole-program analyses were possible. This ensures the best possible cost and precision for the users of the MOD information.

The *basic summary* is the simplest summary information produced by our approach. Figure 5 shows the basic summary for module *Lib* from Figure 4. The summary has three parts. The *variable summary* contains all relevant library variables. The *points-to summary* contains all library statements that are relevant to points-to analysis. The `proc` declarations are used by the subsequent points-to analysis to model the formal-actual pairings at procedure calls. The *Mod summary* contains syntactic modifications and syntactic calls for each library procedure. A syntactic modification (defined in Section 2) is a pair (v, D) or (v, I) representing a direct or indirect modification. A syntactic call is a similar pair indicating a direct or indirect call through v. The Mod summary does not include direct modifications of non-static local variables—as discussed in Section 2, such modifications can be filtered out by the MOD analysis.

In the summary-based separate analysis, the program representation of a client module is combined with the library summary and the result is analyzed as if it were a complete program. Thus, already existing implementations of the whole-program analyses from Section 2 can be reused with only minor adjustments, which makes the approach simple to implement. Clearly, the computed points-to and MOD solutions are the same as the solutions that would have been produced by the standard whole-program analyses. However, the cost of the summary-based analyses is essentially the same as the cost of the whole-program analyses. The next section presents summary optimizations that reduce this cost.

1. Variable summary

$Procedures = \{exec, neg\}$ $Locals(exec) = \{fp, s, u\}$ $Reps = \{rep_1, rep_2\}$
$Globals = \{g\}$ $Locals(neg) = \{i, j\}$

2. Points-to summary

```
proc exec(rep1,fp)    rep2=&s    *rep1=u    i=*rep2
(*fp)(g,rep1)         rep2=&u    g=rep1     *rep2=j
```

3. Mod summary

$SynMod(exec) = \{(rep_1, I), (g, D)\}$ $SynMod(neg) = \{(rep_2, I)\}$
$SynCall(exec) = \{(fp, I), (neg, D)\}$ $SynCall(neg) = \emptyset$

Fig. 6. Optimization through variable substitution.

5 Summary Optimizations

In this section we describe three techniques for optimizing the basic summary. The resulting *optimized summary* has two important features. First, the summary is precision-preserving: for each statement in a client module, the summary-based MOD solution computed with the optimized summary is the same as the solution computed with the basic summary. Second, as shown by our experiments in Section 6, the cost of the summary-based analyses is significantly reduced when using the optimized summary, compared to using the basic summary.

Variable Substitution. Variable substitution is a technique for reducing the cost of points-to analysis by replacing a set of variables with a single representative variable. We use a specific precision-preserving substitution proposed in [11, 13]. With this technique we produce a more compact summary, which in turn reduces the cost of the subsequent summary-based analyses without any loss of precision.

Two variables are *equivalent* if they have the same points-to sets. The substitution is based on mutually disjoint sets of variables $V_1, \ldots, V_k$ such that for each set V_i (i) all elements of V_i are equivalent, (ii) no element of V_i has its address taken (i.e., no element is pointed to by other variables), and (iii) no element of V_i is exported. For example, for module *Lib* in Figure 4, possible sets are $V_1 = \{p, t\}$ and $V_2 = \{q, r\}$. Each V_i has associated a *representative variable* rep_i. We optimize the points-to summary by replacing all occurrences of a variable $v \in V_i$ with rep_i. In addition, in the Mod summary, every pair (v, I) is replaced with (rep_i, I). It can be proven that this optimization is precision-preserving—given the modified summary, the subsequent MOD analysis computes the same solution as the solution computed with the basic summary.

We identify equivalent variables using a linear-time algorithm presented in [13], which extends a similar algorithm from [11]. The algorithm constructs a *subset graph* in which nodes represent expressions and edges represent subset relationships between the points-to solutions for the nodes. For example, edge (q, p) shows that $Pt(q) \subseteq Pt(p)$. The graph represents all subset relationships that can be directly inferred from individual library statements. A strongly-connected component (SCC) in the graph corresponds to expressions with equal points-to sets. The algorithm constructs the SCC-DAG condensation of the graph and traverses it in topological sort order; this traversal identifies SCCs with equal points-to sets. Due to space limitations, the details of the algorithm are not presented in this paper; we refer the interested reader to [13] for more details.

Example. Consider module *Lib* from Figure 4. Since t is assigned the value of p and there are no indirect assignments to t (because its address is not taken), t has exactly the same points-to set as p. In this case, the algorithm can identify set $V_1 = \{p, t\}$. Similarly, the algorithm can detect set $V_2 = \{q, r\}$. After the substitution, the summary can be simplified by eliminating trivial statements. For example, "`t=p`" is transformed into "`rep1=rep1`", which can be eliminated. Call "`neg(rep2)`" can also be eliminated. Since this is the only call to *neg* (and *neg* is not exported), declaration "`proc neg(rep2)`" can be removed as well. Figure 6 shows the resulting summary, derived from the summary in Figure 5.

In addition to producing a more compact summary, we use the substitution to reduce the cost of the worst-case points-to analysis. During the analysis, every occurrence of $v \in V_i$ is treated as an occurrence of rep_i; in the final solution, the points-to set of v is defined to be the same as the points-to set computed for rep_i. This technique produces the same points-to sets as the original analysis.

Statement Elimination. After variable substitution, the points-to summary is simplified by removing statements that have no effect on client modules. A variable $v \in V_L$ is *client-inactive* if v is a non-static local in procedure P and there is no path from P to procedure p_ph in the call graph used for the worst-case MOD analysis of the library module. The worst-case call graph represents all possible call graphs for all complete programs; thus, for any such v, $active(s, v)$ (defined in Section 2) is false for all statements s in all client modules.

Let $Reach_{wc}(u)$ be the set of all variables reachable from u in the points-to graph computed by the worst-case points-to analysis of the library module.[3] A variable $v \in V_L$ is *client-inaccessible* if (i) v is not a global, static local, or procedure variable, and (ii) v does not belong to $Reach_{wc}(u)$ for any global or static local variable u, including v_ph. It is easy to show that in this case $accessible(s, v)$ is false for all call statements s in all client modules.

In the summary-based MOD analysis, any variable that is client-inactive or client-inaccessible will not be included in any *Mod* set for any statement in a client module. We refer to such variables as *removable*. The optimization eliminates from the points-to summary certain statements that are only relevant with respect to removable variables. As a result, the summary-based points-to analysis computes a solution in which some points-to pairs (p, v) are missing. For any such pair, v is a removable variable; thus, the optimization is precision-preserving (i.e., *Mod* sets for statements in client modules do not change).

The optimization is based on the fact that certain variables can only be used to access removable variables. Variable v is *irrelevant* if $Reach_{wc}(v)$ contains only removable variables. Certain statements involving irrelevant variables can be safely eliminated from the points-to summary: (i) "$p = \&q$", if q is removable and irrelevant, (ii) "$p = q$", "$p = *q$", and "$*p = q$", if p or q is irrelevant, and

[3] w is reachable from u if there exists a path from u to w containing at least one edge.

1. Variable summary

$\mathit{Procedures} = \{\mathit{exec}, \mathit{neg}\}$ $\quad$ $\mathit{Locals}(\mathit{exec}) = \{\mathit{fp}\}$ $\quad$ $\mathit{Reps} = \{\mathit{rep}_1\}$

$\mathit{Globals} = \{g\}$ $\quad$ $\mathit{Locals}(\mathit{neg}) = \emptyset$

2. Points-to summary

```
proc exec(rep1,fp)   (*fp)(g,rep1)    g=rep1
```

3. Mod summary

$\mathit{SynMod}(\mathit{exec}) = \{(\mathit{rep}_1, I), (g, D)\}$ $\quad$ $\mathit{SynMod}(\mathit{neg}) = \emptyset$

$\mathit{SynCall}(\mathit{exec}) = \{(\mathit{fp}, I), (\mathit{neg}, D)\}$ $\quad$ $\mathit{SynCall}(\mathit{neg}) = \emptyset$

Fig. 7. Final optimized summary.

(iii) calls "$p = f(q_1, \ldots, q_n)$" and "$p = (*\mathit{fp})(q_1, \ldots, q_n)$", if all of $p, q_1, \ldots, q_n$ are irrelevant. Intuitively, the removal of such statements does not "break" points-to chains that end at non-removable variables. It can be proven that the points-to solution computed after this elimination differs from the original solution only by points-to pairs (p, v) in which v is a removable variable [12].[4]

Example. Consider module *Lib* and the worst-case solutions from Figure 4. Variables $\{r, i, j\}$ are client-inactive because the worst-case call graph does not contain a path from *neg* to *p_ph*. Variables $\{p, \mathit{fp}, s, u, q, t, r, i, j\}$ are client-inaccessible because they are not reachable from g or *v_ph* in the worst-case points-to graph. Variables $\{s, u, i, j\}$ have empty points-to sets and are irrelevant. Variables $\{q, r, \mathit{rep}_2\}$ can only reach s and u and are also irrelevant. Therefore, the following statements can be removed from the points-to summary in Figure 6: `rep2=&s`, `rep2=&u`, `*rep1=u`, `i=*rep2`, and `*rep2=j`.

Modification Elimination. This optimization removes from the Mod summary each syntactic modification (v, I) for which $Pt_{wc}(v)$ contains only removable variables. Clearly, this does not affect the *Mod* sets of statements in client modules. In Figure 6, (rep_2, I) can be removed because rep_2 points only to removable variables s and u. The final optimized summary for our running example is shown in Figure 7.

6 Empirical Results

For our initial experiments, we implemented the worst-case and summary-based points-to analyses, as well as the summary construction techniques from Section 5. Our implementation of Andersen's analysis 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. We measured (i) the cost of the worst-case points-to analysis of the library, (ii) the cost of constructing the optimized summary, (iii) the size of the optimized summary, and (iv) the cost of

[4] Identifying client-inaccessible and irrelevant variables requires reachability computations in the worst-case points-to graph; the cost of these traversals can be reduced by merging *v_ph* with all of its successor nodes, without any loss of precision [12].

Table 1. Data programs and libraries. Last two columns show absolute and relative library size in lines of code and in number of pointer-related statements.

Program	Library	LOC	Statements
gnuplot-3.7.1	libgd-1.3	22.2K (34%)	2965 (6%)
gasp-1.2	libiberty	11.2K (43%)	6259 (50%)
bzip2-0.9.0c	libbz2	4.5K (71%)	7263 (86%)
unzip-5.40	zlib-1.1.3	8.0K (29%)	9005 (33%)
fudgit-2.41	readline-2.0	14.8K (50%)	10788 (39%)
cjpeg-5b	libjpeg-5b	19.1K (84%)	17343 (84%)
tiff2ps-3.4	libtiff-3.4	19.6K (94%)	20688 (84%)
povray-3.1	libpng-1.0.3	25.7K (19%)	25322 (23%)

the summary-based points-to analysis of the client module. At present, our implementation does not perform MOD analysis. Nevertheless, these preliminary results are important because the total cost of the two analyses is typically dominated by the cost of the points-to analysis [16, 15].

Table 1 describes our C data programs. Each program contains a well-defined library module, 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`. We added to each library module a set of stubs representing the effects of standard library functions (e.g., `strcpy`, `cos`, `rand`); the stubs are summaries produced by hand from the specifications of the library functions.[5]

Table 1 shows the number of lines of source code for each library, as an absolute value and as percentage of the number for the whole program. For example, for `povray` 19% (25.7K) of the source code lines are in the library module, and the remaining 81% (108.2K) are in the client module. The table also shows the number of pointer-related statements in the intermediate representation of the library, as an absolute value and as percentage of the whole-program number. These numbers represent the size of the input for the points-to analysis.

For our first set of experiments, we measured the cost of the worst-case points-to analysis and the cost of constructing the optimized summary. The results are shown in Figure 8(a).[6] Column T_{wc} shows the running time of the worst-case points-to analysis of the library module. Column T_{sub} contains the time to compute the variable substitution. Column T_{elim} shows the time to identify removable and irrelevant variables in order to perform statement elimination and modification elimination. Finally, column S_{max} contains the maximum amount of memory needed during the points-to analysis and the summary construction.

[5] We plan to investigate 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.

[6] All experiments were performed on a 360MHz *Sun Ultra-60* with 512Mb physical memory. The reported times are the median values out of five runs.

(a) Library Module

Library	T_{wc} (s)	T_{sub} (s)	T_{elim} (s)	S_{max} (Mb)
libgd	3.9	0.4	0.1	10.1
libiberty	5.8	0.5	0.1	10.4
libbz2	4.1	0.8	0.1	8.5
zlib	6.5	1.0	0.1	11.4
readline	10.2	1.3	0.1	12.3
libjpeg	15.1	2.5	0.1	19.7
libtiff	23.4	3.6	0.2	25.2
libpng	19.8	4.0	0.2	21.8

(b) Client Module

Program	T_b (s)	Δ_T	S_b (Mb)	Δ_S
gnuplot	47.3	4%	79.6	5%
povray	111.6	17%	146.7	16%
unzip	21.0	25%	39.5	16%
fudgit	39.8	21%	59.5	17%
gasp	9.7	32%	18.6	26%
cjpeg	15.7	59%	32.1	56%
tiff2ps	22.5	61%	37.5	59%
bzip2	5.4	69%	13.0	54%

Fig. 8. (a) Cost of the library analyses: running time (in seconds) of the worst-case points-to analysis (T_{wc}) and time for constructing the optimized summary (T_{sub} and T_{elim}). S_{max} is the maximum memory usage. (b) Cost of the points-to analyses of the client. T_b is the analysis time with the basic summary, and Δ_T is the reduction in analysis time when using the optimized summary. S_b and Δ_S are the corresponding measurements for analysis memory.

Clearly, the cost of the worst-case points-to analysis and the cost of the summary construction are low. Even for the larger libraries (around 20K LOC), the running time and memory usage are practical. These results indicate that both the worst-case points-to analysis and the summary construction algorithm are realistic candidates for inclusion in optimizing compilers.

Our second set of experiments investigated the difference between the basic summary from Section 4 and the optimized summary from Section 5. We first compared the sizes of the two summaries. For brevity, in this paper we summarize these measurements without explicitly showing the summary sizes. The size of the optimized summary was between 21% and 53% (31% on average) of the size of the basic summary. Clearly, the optimizations described in Section 5 result in significant compaction in the library summaries. In addition, we measured the size of the optimized summary as percentage of the size of the library binary. This percentage was between 14% and 76% (43% on average), which shows that the space overhead of storing the summary is practical.[7]

Next, we measured the differences between the basic summary and the optimized summary with respect to the cost of the summary-based points-to analysis of the client module. The results are shown in Figure 8(b). The order of the programs in the table is based on the relative size of the library, as shown by the percentages in the last column of Table 1. Column T_b shows the analysis time when using the basic summary. Column Δ_T shows the reduction in analysis time when using the optimized summary. The reduction is proportional to the relative size of the library. For example, in `bzip2` the majority of the program is in the library, and the cost reduction is significant. In `gnuplot` only 6% of the pointer-related statements are in the library, and the analysis cost is reduced

[7] The sizes depend on the file format used to store the summaries. We use a simple text-based format; more optimized formats could further reduce summary size.

accordingly. Similarly, the reduction Δ_S in the memory usage of the analysis is proportional to the relative size of the library.

The results from these experiments clearly show that the optimizations from Section 5 can have significant beneficial impact on the size of the summary and the cost of the subsequent summary-based points-to analysis.

7 Related Work

The work in [16, 10] investigates various points-to analyses and their applications, including Andersen's analysis and MOD analyses based on it. Our conceptual whole-program MOD analysis is based on this work. At call sites, [10] filters out certain variables whose lifetime has terminated; our use of the *active* filter is similar to this approach.

A general approach for analyzing program fragments is presented in [14]. The interactions of the fragment with the rest of the program are modeled by summary values and summary functions. The worst-case separate analyses are examples of fragment analyses, in which auxiliary statements play the same role as summary values and summary functions. Unlike [14], we use abstractions not only for lattice elements (i.e., points-to graphs and *Mod* sets), but also for statements, procedures, and call graphs.

The summary-based separate analyses are also examples of fragment analyses, in which the summary information is used similarly to the summary functions from [14]. However, instead of using a complete summary function for each exported library procedure, we use a set of elementary transfer functions. This approach is different from summary construction techniques based on context-sensitive analysis [9, 2, 3]. In these techniques, a called procedure is analyzed before or together with its callers. In contrast, our summaries can be constructed completely independently from the rest of the program. This allows handling of callbacks to unknown client modules and calls to unanalyzed library modules. In addition, for compilers that already incorporate a flow- and context-insensitive points-to analysis, our summary construction approach is significantly easier to implement than the context-sensitive techniques for summary construction. This minimal implementation effort is an important advantage for realistic compilers.

Escape analysis for Java determines if an object can escape the method or thread that created it. Our notion of accessible variables is similar to the idea of escaping objects. Some escape analyses [20, 4] calculate specialized points-to information as part of the analysis. In this context, escape summary information for a module can be computed in isolation from the rest of the program. The techniques used in this manner for escape analysis cannot be used for general-purpose points-to analysis.

Flanagan and Felleisen [7] present an approach for componential set-based analysis of functional languages. They derive a simplified constraint system for each program module and store it in a constraint file; these systems are later combined and information is propagated between them. The constraint files used in this work are similar in concept to our use of library summaries.

Guyer and Lin [8] propose annotations for describing libraries in the domain of high-performance computing. The annotations encode high-level semantic information and are produced by a library expert. Some annotations describe the points-to and MOD/USE effects of library procedures, in a manner resembling complete summary functions. The annotations are used for source-to-source optimizations of the library and application code. This approach allows domain experts to produce high-level information that cannot be obtained through static analysis. However, such approaches cannot be used in general-purpose compilers.

Sweeney and Tip [19] describe analyses and optimizations for the removal of unused functionality in Java modules. Worst-case assumptions are used for code located outside of the optimized modules. This work also presents techniques that allow a library creator to specify certain library properties (e.g., usage of reflection) that are later used when optimizing the library and its clients.

8 Conclusions and Future Work

Traditional whole-program analyses cannot be used in the context of a modular development process. This problem presents a serious challenge for the designers of program analyses. In this paper we show how Andersen's points-to analysis and the corresponding MOD analysis can be used for programs built with precompiled libraries. Our approach can be trivially extended to USE analysis. In addition, our techniques can be applied to other flow- and context-insensitive points-to analyses (e.g., [18, 17, 5]) and to MOD/USE analyses based on them.

We show how to perform worst-case analysis of library modules and summary-based analysis of client modules. These separate analyses can reuse already existing implementations of the corresponding whole-program analyses. We also present an approach for constructing summary information for library modules. The summaries can be constructed completely independently from the rest of the program; unlike previous work, this approach can handle callbacks to unknown clients and calls to unanalyzed libraries. Summary construction is inexpensive and simple to implement, which makes it a practical candidate for inclusion in optimizing compilers. We present summary optimizations that can significantly reduce the cost of the summary-based analyses without sacrificing any precision; these savings occur every time a new client module is analyzed.

An interesting direction of future research is to investigate separate analyses derived from flow-insensitive, context-sensitive points-to analyses. In particular, it is interesting to consider what kind of precision-preserving summary information is appropriate for such analyses. Another open problem is to investigate separate analyses and summary construction in the context of standard analyses that need MOD/USE information (e.g., live variables analysis and reaching definitions analysis), especially when the target analyses are flow-sensitive.

Acknowledgments. We thank Matthew Arnold for his comments on an earlier version of this paper. We also thank the reviewers for their helpful suggestions for improving the paper. This research was supported, in part, by NSF grants CCR-9804065 and CCR-9900988, and by Siemens Corporate Research.

References

1. L. Andersen. *Program Analysis and Specialization for the C Programming Language*. PhD thesis, DIKU, University of Copenhagen, 1994.
2. R. Chatterjee, B. G. Ryder, and W. Landi. Relevant context inference. In *Symposium on Principles of Programming Languages*, pages 133–146, 1999.
3. B. Cheng and W. Hwu. Modular interprocedural pointer analysis using access paths. In *Conference on Programming Language Design and Implementation*, pages 57–69, 2000.
4. J. Choi, M. Gupta, M. Serrano, V. Sreedhar, and S. Midkiff. Escape analysis for Java. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 1–19, 1999.
5. M. Das. Unification-based pointer analysis with directional assignments. In *Conference on Programming Language Design and Implementation*, pages 35–46, 2000.
6. M. Fähndrich, J. Foster, Z. Su, and A. Aiken. Partial online cycle elimination in inclusion constraint graphs. In *Conference on Programming Language Design and Implementation*, pages 85–96, 1998.
7. C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Trans. Programming Languages and Systems*, 21(2):370–416, Mar. 1999.
8. S. Guyer and C. Lin. Optimizing the use of high performance software libraries. In *Workshop on Languages and Compilers for Parallel Computing*, 2000.
9. M. J. Harrold and G. Rothermel. Separate computation of alias information for reuse. *IEEE Trans. Software Engineering*, 22(7):442–460, July 1996.
10. M. Hind and A. Pioli. Which pointer analysis should I use? In *International Symposium on Software Testing and Analysis*, pages 113–123, 2000.
11. A. Rountev and S. Chandra. Off-line variable substitution for scaling points-to analysis. In *Conference on Programming Language Design and Implementation*, pages 47–56, 2000.
12. A. Rountev and B. G. Ryder. Points-to and side-effect analyses for programs built with precompiled libraries. Technical Report 423, Rutgers University, Oct. 2000.
13. A. Rountev and B. G. Ryder. Practical points-to analysis for programs built with libraries. Technical Report 410, Rutgers University, Feb. 2000.
14. A. Rountev, B. G. Ryder, and W. Landi. Data-flow analysis of program fragments. In *Symposium on the Foundations of Software Engineering*, LNCS 1687, 1999.
15. B. G. Ryder, W. Landi, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural side effect analysis with pointer aliasing. Technical Report 336, Rutgers University, May 1998. To appear in ACM TOPLAS.
16. M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Static Analysis Symposium*, LNCS 1302, pages 16–34, 1997.
17. M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Symposium on Principles of Programming Languages*, pages 1–14, 1997.
18. B. Steensgaard. Points-to analysis in almost linear time. In *Symposium on Principles of Programming Languages*, pages 32–41, 1996.
19. P. Sweeney and F. Tip. Extracting library-based object-oriented applications. In *Symposium on the Foundations of Software Engineering*, pages 98–107, 2000.
20. J. Whaley and M. Rinard. Compositional pointer and escape analysis for Java programs. In *Conference on Object-Oriented Programming Systems, Languages, and Applications*, pages 187–206, 1999.
21. S. Zhang, B. G. Ryder, and W. Landi. Program decomposition for pointer aliasing: A step towards practical analyses. In *Symposium on the Foundations of Software Engineering*, pages 81–92, 1996.