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

Atanas Rountev 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 [2] 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. We present empirical results showing that the separate points-to analyses are practical even for large modules, and that the cost of constructing and storing library summaries is low. Our 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 smaller separate modules. Such modular development allows humans to understand and manage complex software systems. This development model also enables practical compilation: instead of (re)compiling large programs from scratch, compilers can perform separate compilation of individual modules. Modular development allows sharing of modules between programs: an already compiled library module can be reused with no development effort or compilation cost. Modularity also enables better distribution of the implementation effort—different modules can 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 the runtime behavior of programs. Some of these analyses can be performed locally (e.g., within a single procedure), while others require the whole program. Such *whole-program analyses* can detect complex interactions between different parts of the program and can compute information about "nonlocalized" program properties. However, whole-program analyses cannot be used in the context of a modular development process. To make these analyses useful in realistic compilers and software tools, analysis techniques must be adapted to handle such modular development. This problem is one of the most serious challenges for the transfer of technology from program analysis research to real-world compilers and tools.

In this paper we investigate one instance of this problem. Our work considers points-to analysis and side-effect analysis for programs built with *precompiled library modules*. Reusable modules are routinely employed in order to reduce development costs. Such modules are designed to be used by a variety of (yet unknown) clients, and are typically packaged as precompiled libraries that are statically or dynamically linked with the client code. To simplify the discussion, for most of the paper we consider programs containing two modules: a *library module* that is developed and compiled independently of any particular client, and a *client module* that uses the functionality provided by the library module. (Section 6 discusses analysis of programs built with multiple library modules.)

For programs built with precompiled library modules, compilers cannot use whole-program analyses because the library module and the client module are compiled separately. In this paper we show how a particular whole-program points-to analysis and the corresponding whole-program side-effect analysis can be extended to handle applications built in this manner. Our work is a step toward incorporating practical points-to and side-effect analyses in realistic compilers.

1.1 Points-To Analysis and Side-effect Analysis

The goal of *modification side-effect analysis* (MOD) is to determine, for each program statement, the set of variables whose values may be modified by executing that statement. The complementary USE analysis computes similar information with respect to the uses of variables by program statements. Such information plays a fundamental role in optimizing compilers and software productivity tools. For example, MOD/USE information is needed to perform live variables analysis, reaching definitions analysis, and constant propagation [1], which in turn are used for code motion, common subexpression elimination, removal of dead assignments, register allocation, and detection of unreachable code. Tools for program understanding, restructuring and testing need MOD/USE information to perform a variety of analyses (e.g., dependence analysis for program slicing). To simplify the discussion, in this paper 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 *pointers* allow indirect memory accesses and indirect procedure calls. Typically, MOD analysis uses the output of

a *points-to analysis* to resolve indirect memory accesses and to determine the targets of indirect calls. Various whole-program points-to analyses have been developed [15, 13, 7, 2, 27, 24, 28, 22, 16, 11, 6, 9, 4]. Our work is focused on the category of flow- and context-insensitive analyses [2, 24, 28, 22, 6]. Such analyses ignore the flow of control between program points and do not distinguish between different calling contexts of procedures. As a result, analyses from this category 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 theoretically and empirically [21, 25, 20, 14]. Similarly to [21, 14], we consider a whole-program MOD analysis based on Andersen's whole-program flow- and context-insensitive points-to analysis [2]. Even though we investigate these particular analyses, our results also apply to similar flow- and context-insensitive points-to analyses (e.g., [24, 22, 6]) and to MOD analyses based on them.

1.2 Separate Analysis of Modules

Compilers cannot use whole-program points-to and MOD analyses for programs built with precompiled library modules. When compiling a library module, there is no available information about client modules. When compiling 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. In this paper we define such separate analyses by extending Andersen's analysis and the corresponding MOD analysis. Our work enables compilers to use two fundamental program analyses—points-to analysis and side-effect analysis—for programs built with precompiled library modules.

Worst-case Separate Analysis of Library Modules We first address the problem of performing separate points-to analysis and MOD analysis for a library module during the compilation of the module. Since the compilation is performed independently of any client modules, the analyses must make *worst-case assumptions* about the interactions between the library module and the rest of the program.

In our approach, the worst-case separate analyses are implemented by adding *auxiliary statements* to the library module. The auxiliary statements model the effects of all possible statements from client modules (including calls to the library), as well as callbacks from the library to client modules. The library, combined with the auxiliary statements, is treated *as a complete program* and the standard whole-program analyses are applied to it. The resulting solutions represent all possible points-to and MOD relationships for all possible client modules. An important advantage of this approach is its simple implementation by reusing already existing implementations of the whole-program analyses.

Summary-based Separate Analysis of Client Modules The separate analyses of a client module can use precomputed *summary information* about the called library module. Such summary-based analyses can compute more precise points-to and MOD information for the client module, compared to worst-case analyses. This improvement is important because the precision of the analysis solution has significant impact on subsequent analyses and optimizations [21, 20, 25, 14].

The summary information encodes the effects of the library module on the rest of the program. 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 a 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 whole-program points-to and MOD analyses. Thus, both the summary-based and the worst-case separate analyses can be implemented with minimal additional effort.

Summary Information When designing a summary-based separate analysis, the key issue is to determine what summary information is appropriate for the target analysis, and how to extract this information during the compilation of the library. In our approach, the library summary contains a set of *summary elements*. Each element represents the points-to effects or MOD effects of one or more library statements. For example, an element of the form (p, D) shows that the library *directly modifies* variable p. Similarly, (p, I) indicates modification of all variables *indirectly accessible* through a pointer dereference of p. Element "$p = q$" shows that all values of q are assigned to p, which implies that the points-to set of q is a subset of the points-to set of p.

We extract an initial set of elements from the library source code, and then use two techniques to optimize the summary. First, we use variable substitution [17] to replace a set of library variables with a single placeholder variable. This optimization produces a more compact summary by merging several summary elements into one representative element. Second, we identify library variables that do not have any effect on client modules; this allows us to eliminate the corresponding summary elements. Both optimizations are *precision-preserving*: with respect to client modules, the summary-based analyses are as precise as the standard whole-program analyses.

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 (described in Section 5) 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.

1.3 Contributions

The contributions of our work are the following:

- We propose an approach for separate points-to analysis and MOD analysis of library modules, by using auxiliary statements to encode worst-case assumptions about the rest of the program. The approach is easy to implement by reusing 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 respect to client modules, performing summary-based analyses with this summary is as precise as using the standard whole-program analyses.

Program → GlobalDecl* Procedure*
GlobalDecl → **global** VarList
Procedure → **proc** Var(Formals) [**returns** Var] **{** Body **}**
Formals → ϵ | VarList
Body → LocalDecl* StaticLocalDecl* Stmt*
LocalDecl → **local** VarList
StaticLocalDecl → **static local** VarList
Stmt → Assignment | Call | IndirectCall
Assignment → Var = Var | Var = **&**Var | Var = *****Var |
*****Var = Var | Var = NptrExpr
NptrExpr → **const** | **unop** Var | Var **binop** Var
Call → [Var =] Var(Actuals)
IndirectCall → [Var =] (*****Var)(Actuals)
Actuals → ϵ | VarList
VarList → Var | Var , VarList
Var → **id**

Figure 1: Grammar for the program representation. Terminals are shown in boldface. A procedure may have a special return variable (defined by **returns**) which is assigned the return value of the procedure. NptrExpr denotes an expression with non-pointer values.

- 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 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.

Overview The rest of the paper is organized as follows. Section 2 describes Andersen's analysis and the corresponding MOD analysis. Section 3 presents the worst-case analyses of the library module. Section 4 discusses the structure of our summary information and its use for summary-based analyses of client modules. Section 5 describes the techniques for optimizing the library summary. Section 6 discusses analysis of programs with multiple library modules. The experimental results are presented in Section 7. Section 8 discusses related work, and Section 9 presents conclusions and future work.

2 Whole-program Analyses

In this section we present a conceptual model of Andersen's points-to analysis [2] and a MOD analysis based on it. These whole-program analyses are the foundation for our worst-case and summary-based separate analyses. The model of Andersen's analysis highlights the key analysis features. The model of the MOD analysis is derived from similar MOD analyses [21, 14] by introducing extensions that enable certain summary optimizations.

2.1 Program Representation

For the purpose of this paper, we assume that the program representation is defined by the grammar in Figure 1. The statements in the representation have a simplified form similar to the simplified intermediate representations often used in optimizing compilers. Control-flow statements (e.g., `if`, `while`, etc.) are irrelevant with respect to points-to and MOD analyses, and are not included in the representation. 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[1], and therefore the program representation is untyped. Similarly to [24, 22, 21, 8, 11, 6, 9, 14], we assume that the analyses treat structures and arrays as monolithic objects and do not distinguish between their individual elements. Finally, we assume that calls to *malloc* and other heap-allocating functions are replaced by statements of the form "$x = \&heap_i$", where $heap_i$ is a variable unique to the allocation site.

Figure 2 shows a complete program with two modules: module *Client* containing procedures *main* and *div*, and module *Lib* containing procedures *exec* and *neg*.

```
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	z	g	w	p, t	fp	a	b	q, r
$Pt(v)$	x	x, y	div	x	div	x, y	x	s, u
Stmt	1	6	10	14	16	18	19	23
$Mod(s)$	x	g, x, y	x, y	x, y	s, u	s, u	x	s, u

Figure 2: Program with two modules $Client = \{main, div\}$ and $Lib = \{exec, neg\}$, and the corresponding points-to and MOD solutions computed by the whole-program analyses.

2.2 Points-to Analysis

Let V be the set of variables in the program representation, as defined by the last production in Figure 1. We classify the elements of V as (i) *global variables*, (ii) *procedure variables*, which occur immediately after the **proc** terminal and denote the names of procedures, (iii) *local variables*, including formals and return variables, and (iv) *heap variables* introduced at heap-allocation sites.

Points-to relationships are encoded by a *points-to graph* in which nodes correspond to variables from V. 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. We can define a lattice of points-to graphs $L = \mathcal{P}(V \times V)$, where $\mathcal{P}(X)$ denotes the power set of X.

[1] 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 alternative approaches—for example, for analyses that make unsafe assumptions about typecasting and unions, or perform some form of type analysis.

The statements in the program define a set F of *transfer functions* $f : L \to L$. Conceptually, the analysis starts with an empty graph and applies functions from F until a fixed-point solution is obtained. Each transfer function represents the points-to effects of a program statement. For example, the transfer function for "$*p = q$" is $f(G) = G \cup \{(x, y) \mid (p, x) \in G \wedge (q, y) \in G\}$. 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. An indirect call through a pointer p is equivalent to a set of direct calls for all procedures in the points-to set of p.

Figure 2 shows the points-to solution computed for the sample program; $Pt(v)$ denotes the points-to set of v.

2.3 Side-effect Analysis

In this section we present a conceptual algorithm for whole-program MOD analysis. The algorithm computes a set of modified variables $Mod(s) \subseteq V$ for each statement s. The elements of $Mod(s)$ represent memory locations whose values after the execution of s may be different from the values they had before the execution of s. The algorithm is derived from similar MOD algorithms [21, 14] by adding filters for active and accessible variables.

2.3.1 Active and Accessible Variables

A variable v should be included in $Mod(s)$ 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 memory locations whose lifetime is active only for procedures that are directly or transitively called by P. We define a relation $active(s, v)$ that holds only if

- v is a global variable, a static local variable, or a heap variable, or
- 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

Consider a call statement s that invokes a 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. Intuitively, an active location is accessible if it can be referenced either directly, or through a series of pointer dereferences. Globals and static locals can be directly accessible. Any location indirectly accessible through an actual parameter of the call is also accessible by P. More formally, we define a relation $accessible(s, v)$ that holds only if

- v is a global variable or a static local variable, or
- $v \in Pt(a)$, where a is an actual parameter of s, or
- $v \in Pt(w)$ and $accessible(s, w)$ holds for some $w \in V$

If $active(s, v)$ holds but $accessible(s, v)$ does not hold, the active locations represented by v cannot be modified by the invocation of P; thus, v should not occur in $Mod(s)$.

For example, variable u from Figure 2 is active for all statements in procedures *exec*, *div*, and *neg*. With respect to calls 16 and 18, u is accessible because it belongs to the points-to set of actual q; however, u is not accessible with respect to call 14.

```
input   Stmt: set of statements
        Proc: set of procedures
        EnclosingProc: Stmt → Proc
        SynMod: Stmt → V × {D, I}
        Pt: V → P(V)
output  Mod: Stmt → P(V)
declare Called: Stmt → P(Proc)
        ProcMod: Proc → P(V)

[1]  foreach s ∈ Stmt do
[2]    if s is a direct call to procedure P then
[3]      Called(s) := {P}
[4]    if s is an indirect call through pointer q then
[5]      Called(s) := { P | procedure P ∈ Pt(q)}

[6]  foreach s ∈ Stmt do
[7]    if SynMod(s) = (v, D) then
[8]      Mod(s) := {v}
[9]      if v is global or static local then
[10]       add {v} to ProcMod(EnclosingProc(s))
[11]   if SynMod(s) = (v, I) then
[12]     Mod(s) := {x | x ∈ Pt(v) ∧ active(s, x) }
[13]     add Mod(s) to ProcMod(EnclosingProc(s))

[14] while changes occur in Mod or ProcMod do
[15]   foreach call statement s ∈ Stmt do
[16]     foreach P ∈ Called(s) do
[17]       Mod(s) := Mod(s) ∪ {x | x ∈ ProcMod(P) ∧
[18]                 active(s, x) ∧ accessible(s, x)}
[19]     add Mod(s) to ProcMod(EnclosingProc(s))
```

Figure 3: Whole-program MOD analysis.

In our conceptual MOD algorithm, a variable v is added to $Mod(s)$ only if $active(s, v)$ holds.[2] In addition, v is included in $Mod(s)$ for a call statement s only if $accessible(s, v)$ also holds. The filters *active* and *accessible* can improve the precision of the MOD analysis by compensating for some of the imprecision introduced by the flow- and context-insensitivity of the points-to analysis. Whether these filters will be used in an actual implementation of the MOD analysis is an engineering decision that should be based on the tradeoffs between the cost of the filter computation and the impact of the improved precision on the MOD analysis and the subsequent users of the MOD solution. We use the filters in our algorithm formulation in order to define a more precise semantics for the MOD analysis. This allows us to identify parts of the library that are irrelevant with respect to *Mod* sets in client modules, and therefore can be excluded from the library summary.

2.3.2 Analysis Algorithm

The conceptual MOD algorithm is described in Figure 3. Input map *SynMod* stores the syntactic modifications that occur in program statements. A *syntactic modification* is a pair (v, d) of a variable v that occurs on the left-hand side of an assignment or call, and a flag $d \in \{D, I\}$ that indicates whether the modification is direct or indirect. For example, in Figure 2, $SynMod(s)$ is (x, D) for statement 1 and (a, I) for statement 10.

The first analysis phase (lines 1–5 in Figure 3) uses the

[2] Previous work by Hind and Pioli [14] uses a similar approach.

```
global v_ph
proc p_ph(f1,..,fn) returns p_ph_ret {
  v_ph = fi   (1 ≤ i ≤ n)
  v_ph = &v   (v ∈ Vexp)
  v_ph = &p_ph
  v_ph = &v_ph
  v_ph = *v_ph
  *v_ph = v_ph
  v_ph = (*v_ph)(v_ph,..,v_ph)   (m actuals)
  p_ph_ret = v_ph
}
```

Figure 4: Placeholder procedure and auxiliary statements.

points-to solution to resolve indirect calls and to construct a safe approximation of the program call graph, encoded in map *Called*. The second phase (lines 6–13) uses the points-to solution to compute initial *Mod* sets for all statements, without taking into account the effects of procedure calls. This phase also initializes map *ProcMod*, which stores the sets of variables modified by each procedure. The third phase (lines 14–19) is a fixed-point computation based on the program call graph. This phase uses $ProcMod(P)$ to update the *Mod* sets of calls to P and the *ProcMod* sets for the calling procedures.

We use filters *active* and *accessible* whenever we add a variable to the *Mod* set of a statement (lines 12 and 18). In addition, we filter out direct modifications of non-static local variables (lines 9–10). This filtering is based on the following observation: a direct modification of a non-static local variable v in procedure P cannot be observed by the callers of P because the lifetime of the modified memory location terminates when P returns. In this case, v should not be included in $ProcMod(P)$. This filtering not only improves the precision of the analysis, but also results in more compact summaries, as discussed in Section 4.

In the absence of recursion, the above filtering is subsumed by the use of *active* at line 18. For programs with recursion, not all cases can be filtered out by *active* and *accessible*. Consider the following example:

```
global g;  proc P() { local x; x=1; g=&x; P();}
```

The *Mod* set of the recursive call should not include x, even though x is active and accessible according to the definitions from Section 2.3.1; thus, the direct modification of x should be filtered out from $ProcMod(P)$.

Figure 2 shows the computed *Mod* sets for some of the statements in the sample program.

3 Worst-case Separate Analysis of Library Modules

In this section we show how to perform worst-case points-to and MOD analyses of a library module. These analyses are used during the compilation of the library, and must make worst-case assumptions about the interactions with rest of the program. The worst-case assumptions are introduced by adding *auxiliary statements* to the library module. The combination of the auxiliary statements and the library is treated *as a complete program* and the whole-program analyses from Section 2 are applied to it. The resulting solutions are *safe abstractions* of all possible points-to and MOD relationships for all complete programs containing the library module.

Given a library module *Lib*, consider a complete program p containing module *Lib* and a client module $Client_p$.

```
global g
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
}
```

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

Stmt	14	16	19	call in p_ph
$Mod_{wc}(s)$	v_ph, g	s, u	v_ph, g	v_ph, g
$Called_{wc}(s)$	$p\_ph, exec$	neg	—	$p\_ph, exec$

Figure 5: Worst-case solutions for *Lib*.

Let $PROG(Lib)$ be the (infinite) set of all such complete programs. For any such p, we use V_p to denote the set of variables in the program, as defined in Section 2.2.

Let $V_L \subseteq V_p$ be the set of all variables that occur in statements in *Lib*. This set is independent of p, and can be constructed even when the library in analyzed in isolation. We also define set $V_{exp} \subseteq V_L$ which contains all variables that may be explicitly referenced by client modules; we refer to such variables as *exported*. Exported variables are either global variables that could be directly accessed by library clients, or names of library procedures that could be directly called by client code. In C programs, the exported variables are typically declared in one or more header files, which are subsequently included by the code of the client modules.

Example. We use module *Lib* from Figure 2 as our running example; for convenience, the module is shown again in Figure 5. 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 procedures *main* and *div* from Figure 2 define one of the (infinitely) many possible client modules of *Lib*.

3.1 Auxiliary Statements

The auxiliary statements model the effects of all possible client modules. The statements are located inside a *placeholder procedure* p_ph which represents all procedures in all possible client modules. The auxiliary 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 4. Each auxiliary statement represents different kinds of actual statements that could occur in client modules. For example, "$v\_ph = \&v$" for $v \in V_{exp}$ models statements of the form "$u = \&v$", where $u \in (V_p - V_L)$ for some complete program p. Similarly, "$v\_ph = *v\_ph$" 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 subsequent 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, m should be equal to the maximum

number of formals for all possible target procedures.[3] Similarly, all indirect calls from the library to a client module are represented by indirect calls to *p_ph*; thus, the number of formals in *p_ph* should be equal to the maximum number of actuals used at indirect calls in the library.

3.2 Worst-case Analyses

The worst-case points-to and MOD analyses treat the library module plus the auxiliary statements as a complete program, 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 analyses.

Example. Consider module *Lib* from Figure 5. 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 the number of actuals should be $m = 2$.

The worst-case points-to solution for *Lib* is shown in Figure 5. The solution 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 declared in some client module; similarly, (p, p_ph) indicates that p can point to an unknown procedure from some client module.[4]

The first phase of the MOD analysis (lines 1–5 in Figure 3) computes a call graph which 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, as well as the possible recursive call of *exec*. The remaining phases of the MOD algorithm compute a solution which represents all possible *Mod* sets in all complete programs. Some of the Mod_{wc} sets are shown in Figure 5. Note that s and u are not included in Mod_{wc} for the indirect call inside *p_ph* or for call 14, because they are not accessible (even though they are active according to the definition from Section 2.3.1).

3.3 Analysis Safety

Let $V'_L = V_L \cup \{v\_ph, p\_ph\}$. For each $p \in PROG(Lib)$, we define the following *abstraction function* $\alpha_p : V_p \to V'_L$

$$\alpha_p(v) = \begin{cases} v & \text{for } v \in V_L \\ p\_ph & \text{for } v \notin V_L, \text{ if } v \text{ is a procedure variable} \\ v\_ph & \text{for all other } v \notin V_L \end{cases}$$

We also define an abstraction function for sets of variables $\beta_p(S) = \{\alpha_p(x) | x \in S\}$ and for sets of variable pairs $\gamma_p(S) = \{(\alpha_p(x), \alpha_p(y)) \mid (x, y) \in S\}$.

We use the abstraction functions to define the notion of safety for the worst-case analyses. Let $Pt_p \subseteq V_p \times V_p$ be the whole-program points-to solution for program p, and $Pt_{wc} \subseteq V'_L \times V'_L$ be the worst-case points-to solution for *Lib*. The worst-case solution is *safe* with respect to p only if $\gamma_p(Pt_p) \subseteq Pt_{wc}$. In other words, every points-to pair from the whole-program solution should be represented in the worst-case solution. In this representation, all global, local, and heap variables that are not part of *Lib* are represented by the *v_ph* placeholder; similarly, all procedures declared in client modules are represented by the *p_ph* placeholder.

[3] We assume that the subsequent points-to analysis can handle indirect calls at which the number of actuals is different from the number of formals. Typecasting allows this behavior in valid C programs.

[4] Note that some procedure variables (e.g., *exec* and *p_ph*) have non-empty points-to sets. These spurious sets have no effect on the precision of the subsequent clients of the points-to information.

Theorem 1 *For any $p \in PROG(Lib)$, $\gamma_p(Pt_p) \subseteq Pt_{wc}$.*

To prove this claim, it is enough to show that for each transfer function f in the whole-program analysis, there exist transfer functions $f'_1, \ldots, f'_n$ in the worst-case analysis such that $\gamma_p(S) \subseteq S'$ implies $\gamma_p(f(S)) \subseteq (f'_1 \circ \ldots \circ f'_n)(S')$. This property can be proven through case analysis based on the different kinds of transfer functions.

As a corollary, the call graph computed by the worst-case MOD analysis represents all call graphs computed by the whole-program MOD analyses. For every call statement $s \in Lib$, we have $\beta_p(Called_p(s)) \subseteq Called_{wc}(s)$. Similarly, for every call statement $s \in Client_p$, we have $\beta_p(Called_p(s)) \subseteq Called_{wc}(s')$, where s' is the indirect call inside *p_ph*.

The worst-case MOD analysis is *safe* with respect to program p only if all elements of $Mod_p(s)$ are represented by $Mod_{wc}(s)$ for all library statements s.

Theorem 2 *For any $p \in PROG(Lib)$ and any statement $s \in Lib$, $\beta_p(Mod_p(s)) \subseteq Mod_{wc}(s)$.*

This claim can be proven by induction on the number of iterations in the fixed-point computation of *Mod* sets. The proof is based on Theorem 1 and the call graph relationships described above.

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.

4.1 Summary Construction

Previous work on context-sensitive points-to analysis uses summary information computed for each program procedure [15, 27, 12, 3, 4]). Conceptually, these techniques can be considered instances of the functional approach to context sensitivity [23], in which the effects of a procedure P are represented by a *summary function* that encodes the cumulative effects of the transfer functions for all statements in P and in all procedures transitively called by P.

Some analyses construct *partial* summary functions that are defined only for some input values [15, 27]. Other analyses use *complete* summary functions defined for all possible inputs [12, 3, 4]. The latter approach can be used to produce summary information for a library module, by computing and storing the complete summary functions for all procedures exported by the module [12].

The above approaches have one major disadvantage: 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. (This situation is discussed in Section 6.) In the presence of callbacks, the library module may call client modules that do not even exist at the time when the summary is being constructed. For example, for *Lib* from Figure 5, the effects of *exec* cannot be expressed by a summary function because of the indirect call to some unknown client module.

In practice, callbacks are often used to increase the flexibility of reusable libraries by allowing client code to define, extend, or modify the behavior of the library. A classic example is a generic sorting procedure that takes as input a pointer to a comparison function defined in the client code.

1. Variable summary
   $Procedures = \{exec, neg\}$
   $Globals = \{g\}$
   $Locals(exec) = \{p, fp, s, u, q, t\}$
   $Locals(neg) = \{r, i, j\}$
2. Points-to summary

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

3. Mod summary
   $SynMod(exec) = \{(t, I), (g, D)\}$
   $SynCall(exec) = \{(fp, I), (neg, D)\}$
   $SynMod(neg) = \{(r, I)\}$
   $SynCall(neg) = \emptyset$

Figure 6: Basic summary for *Lib*.

Several of the libraries used in our experiments employ the callback mechanism. For example, the GNU Readline Library provides command line interface with features such as name completion and line editing. Several globals exported by the library can point to client-supplied procedures and can be used for callbacks. A typical example is global variable `rl_directory_completion_hook` which can be used to call client procedures that handle the completion of directory names.

Our approach for summary construction is conceptually different from approaches based on complete summary functions. We use summary information that is close in form to the program representation defined in Figure 1. For example, the summary contains a set of statements similar to the statements from the program representation; each summary statement represents the points-to effects of one or more library statements. Essentially, in the summary we use elementary transfer functions instead of cumulative summary functions.

This approach has two advantages. First, the summary can be constructed completely independently of any callers and callees of the library; therefore, unlike previous work, our approach can handle callbacks. Second, the summary construction algorithm (described in Section 5) is inexpensive and simple to implement, which makes it a good candidate for inclusion in realistic compilers.

The summary information is designed to be *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 applied to the client module and the original library. This ensures the best possible cost and precision for the subsequent users of the MOD information.

Basic Summary The simplest summary information produced by our approach is the *basic summary*. This summary is the starting point for constructing the *optimized summary* described in Section 5.

Figure 6 shows the basic summary for module *Lib* from Figure 5. The summary has three parts. The *variable summary* contains information about relevant library variables. The *points-to summary* contains all library statements that are relevant to points-to analysis; statements of the form "Var = NptrExpr" (see Figure 1) are not included. Duplicate statements are included only once. 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 2.3.2) 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 2.3.2, such modifications should be filtered out by the MOD analysis.

4.2 Summary-based Separate Analysis of Client Modules

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 a complete program*. Thus, already existing implementations of whole-program points-to and MOD analyses can be reused with only minor adjustments, which makes the approach simple to implement. For example, the MOD analysis can treat each library procedure P as containing a single placeholder statement with *Called* set constructed from $SynCall(P)$ and with multiple syntactic definitions determined by $SynMod(P)$. Clearly, for each statement in the client module, the separate MOD solution computed with the basic summary is the same as the solution that would have been produced by the standard whole-program MOD analysis.

The basic summary is easy to construct and use. However, for this summary, the cost of the summary-based analyses could be as high as the cost of the standard whole-program analyses. This cost can be reduced by performing some analysis work during summary construction and encoding the results in the summary. The next section describes several summary optimizations that achieve this goal.

5 Summary Optimizations

In this section we describe three techniques for optimizing the basic summary. We first use *variable substitution* to simplify the statements in the points-to summary. Next, we perform *statement elimination* for statements that have no effect on client modules. Finally, we use *modification elimination* for syntactic modifications that are irrelevant with respect to client modules. The result of these three steps is an optimized summary with 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 demonstrated by our experiments in Section 7, the cost of the summary-based analyses is significantly reduced when using the optimized summary, compared to using the basic summary.

5.1 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 [17, 18]. 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 disjoint sets of variables

1. Variable summary
   $Procedures = \{exec, neg\}$
   $Globals = \{g\}$
   $Locals(exec) = \{fp, s, u\}$
   $Locals(neg) = \{i, j\}$
   $Representatives = \{rep_1, rep_2\}$
2. Points-to summary

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

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

Figure 7: Optimization through variable substitution.

$V_i \subseteq V_L$ with the following restrictions. First, all elements of V_i are equivalent. Second, no element of V_i has its address taken (i.e., no element is pointed-to by other variables). Finally, no element of V_i is exported. For example, for module *Lib* in Figure 5, possible sets are $V_1 = \{p, t\}$ and $V_2 = \{q, r\}$ (the construction of these sets is described later).

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). This ensures that the subsequent MOD analysis will use the appropriate points-to sets to resolve indirect modifications and indirect calls.[5] It can be shown 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 [18], which extends a similar algorithm from [17]. The algorithm constructs a *subset graph* in which nodes represent expressions of the form "&v", "v" and "$*v$". Graph edges represent subset relationships between the points-to solutions for the nodes, and are constructed from the statements in the library. For example, statement "$p = q$" generates edge (q, p), which shows that $Pt(q) \subseteq Pt(p)$, as well as edge $(*q, *p)$, which shows that any variable pointed-to by some $u \in Pt(q)$ is also pointed-to by some $w \in Pt(p)$. Other kinds of library statements generate edges in a similar fashion [18].

The subset graph represents all subset relationships that can be *directly* inferred from individual library statements. Relationships created indirectly through pointer dereferences are not represented. For example, if the address of a variable v is taken, there could be indirect assignments to v; the subset relationships created by such assignments are not represented by any of v's incoming edges.

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, based on the following observation: if all predecessor nodes of variable v have the same points-to set, *and* if all values assigned to v are represented by incoming edges (x, v) in the graph, then v has that same points-to set. The second restriction on v can be ensured by several conditions: (i) v is not an exported global or a formal of an exported procedure, (ii) the address of v is not taken, (iii) v is not a formal of a procedure whose address is taken, and (iv) v is not assigned the return value of an indirect call [18].

Example. Consider module *Lib* from Figure 5. 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. Thus, 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. Similarly, we have a call "`neg(rep2)`" and a procedure declaration "`proc neg(rep2)`". The call can be removed because it is equivalent to "`rep2=rep2`"; since this is the only call to *neg*, the `proc` declaration can also be removed. Figure 7 shows the result of applying the substitution to the basic summary from Figure 6.

In addition to producing a more compact summary, we use the computed substitution to reduce the cost of the worst-case points-to analysis. During the analysis, every occurrence of $v \in V_i$ in library statements 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. It is easy to show that this technique produces the same points-to sets as the original analysis.

5.2 Statement Elimination

The idea of the next optimization is to simplify the points-to summary by removing certain statements that have no effect on the summary-based analyses of 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 placeholder procedure *p_ph* in the call graph computed by the worst-case MOD analysis of the library module. As discussed in Section 3.3, this call graph represents all possible call graphs for all complete programs containing the library module. Thus, if v is client-inactive, $active(s, v)$ (defined in Section 2.3.1) 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.[6] 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*. Based on Theorem 1, it is easy to show that for any such v, $accessible(s, v)$ 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 a subset of statements that are only relevant with respect to removable variables. As a result, the subsequent summary-based points-to analysis computes a solution in which certain points-to pairs (p, v) are missing. If all such missing pairs have a second element v which is a removable variable, the optimization is precision-preserving (i.e., *Mod* sets computed for statements in client

[5] Note that filters *active* and *accessible* are not affected by this optimization because variables rep_i do not have their addresses taken and are never elements of points-to sets or *Mod* sets.

[6] w is reachable from u if there exists a path from u to w that contains at least one edge.

1. Variable summary
   $Procedures = \{exec, neg\}$
   $Globals = \{g\}$
   $Locals(exec) = \{fp\}$
   $Locals(neg) = \emptyset$
   $Representatives = \{rep_1\}$
2. Points-to summary
   `proc exec(rep1,fp)  (*fp)(g,rep1)  g=rep1`
3. Mod summary
   $SynMod(exec) = \{(rep_1, I), (g, D)\}$
   $SynCall(exec) = \{(fp, I), (neg, D)\}$
   $SynMod(neg) = \emptyset$
   $SynCall(neg) = \emptyset$

Figure 8: Final optimized summary.

modules do not change). Thus, our goal is to ensure that only removable variables are missing from the final points-to sets.

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:

- "$p = \&q$", if q is removable and irrelevant
- "$p = q$", "$p = *q$", and "$*p = q$", if p or q is irrelevant
- calls "$p = f(q_1, \ldots, q_n)$" and "$p = (*fp)(q_1, \ldots, q_n)$", 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. This guarantees the safety of the optimization:

Theorem 3 *The points-to solution computed after statement elimination differs from the solution computed without statement elimination only by points-to pairs (p, v) in which v is a removable variable.*

It can be proven by induction that the above claim holds for all intermediate partial solutions computed by the points-to analysis. As a corollary, the property holds for the final points-to solution.

Example. Consider module *Lib* and the worst-case solutions from Figure 5. Variables $\{r, i, j\}$ are client-inactive because the worst-case call graph does not contain a path from *neg* to *p_ph*. Variables $\{p, fp, s, u, q, t, r, i, j\}$ 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 $\{q, r, rep_2\}$ can only reach s and u and are also irrelevant. Therefore, the following statements can be removed from the points-to summary in Figure 7:

```
rep2=&s  rep2=&u  *rep1=u  i=*rep2  *rep2=j
```

The resulting points-to summary is shown in Figure 8.

Identifying client-inaccessible and irrelevant variables requires various reachability computations in the worst-case points-to graph. We reduce the cost of these traversals by merging *v_ph* with all of its successor nodes in the graph. During the reachability computations, all successors of *v_ph* are considered merged with *v_ph*, and any edges incident to these nodes are redirected to *v_ph*. It can be proven that with this optimization, the traversals of the points-to graph identify exactly the same client-inaccessible and irrelevant variables as with the original points-to graph. The proof is based on two observations. First, the successors of *v_ph* form a complete subgraph, due to auxiliary statement "$*v\_ph = v\_ph$". Second, any node reachable from a successor of *v_ph* is also a successor of *v_ph*, due to "$v\_ph = *v\_ph$".

5.3 Modification Elimination

The final optimization is the removal of syntactic modifications that are irrelevant with respect to client modules. Syntactic modification (v, I) can be removed from the Mod summary if $Pt_{wc}(v)$ contains only removable variables. Clearly, this optimization does not affect the *Mod* sets of statements in client modules. In Figure 7, syntactic modification (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 8.

6 Programs with Multiple Libraries

Consider a complete program containing one client module and multiple library modules $L_1, \ldots, L_n$. In the simplest case, the library modules are independent: no module references variables exported by other modules. The separate analyses and the summary optimizations presented earlier can be trivially extended to handle this case.

Suppose that module L_1 references variables exported by another module L_2—either by accessing exported globals, or by calling exported procedures. We say that such variables are *imported* by L_1. The worst-case analyses and the summary construction for L_1 must be extended to handle the additional interactions with L_2.

Two approaches can be used in this situation: (i) the analyses of L_1 can be performed completely independently of L_2, or (ii) L_1 can be analyzed by using an existing library summary for L_2. The first approach is needed if no summary information about L_2 is available—for example, if no summary construction capabilities existed at the time when L_2 was compiled. The second approach can be applied if a library summary had been constructed during the compilation of L_2, and had been stored together with L_2's binary.

If a precomputed summary for L_2 is available, it can be combined with L_1 and the result can be treated as a single library to which the standard worst-case analyses are applied. In addition, the techniques from Section 5 can be applied to this "combined library" to identify variables from L_1 that are equivalent, removable, or irrelevant. This information can be used to optimize the basic summary for L_1.

In the case when no summary for L_2 is available, the worst-case analyses of L_1 are obtained by three adjustments of the worst-case analyses from Section 3. First, an auxiliary statement "$v\_ph = \&v$" should be introduced for any imported global v. Second, all direct calls to imported procedures should be replaced by calls to placeholder procedure *p_ph*. Finally, statements "$v = \&P$", where P is an imported procedure, should be replaced with "$v = \&p\_ph$". The last two adjustments ensure that direct and indirect calls to imported procedures are modeled by calls to *p_ph*.

When no summary for L_2 is available, the summary optimizations from Section 5 need two adjustments in order to construct an optimized summary for L_1. First, variable substitution should not be performed for imported variables; this ensures that variables that occur in both libraries will be properly matched by the separate analyses of the client

Program	Library	LOC	Stmt
gnuplot3.7.1	libgd1.3	22.2K (34%)	2965 (6%)
gasp1.2	libiberty	11.2K (43%)	6259 (50%)
bzip2-0.9.0c	libbz2	4.5K (71%)	7263 (86%)
unzip5.40	zlib1.1.3	8.0K (29%)	9005 (33%)
fudgit2.41	readline2.0	14.8K (50%)	10788 (39%)
cjpeg5b	libjpeg5b	19.1K (84%)	17343 (84%)
tiff2ps3.4	libtiff3.4	19.6K (94%)	20688 (84%)
povray3.1	libpng1.0.3	25.7K (19%)	25322 (23%)

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.

module. Second, the algorithm for detecting equivalent variables (described in Section 5.1) should take into account the fact that certain variables may be assigned values that are not represented by incoming edges in the subset graph; such variables are either imported globals, or variables assigned the values returned by calls to imported procedures. No adjustments are needed for the rest of the summary optimizations.

7 Empirical Results

For our initial experiments, we implemented the worst-case and summary-based Andersen's points-to analysis, 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 [8]; 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 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 [21, 20].

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`. Similarly, `fudgit` is a fitting program which uses the GNU Readline Library to provide command line interface.

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.[7] During the separate analyses and the summary construction, the stubs were treated as part of the library module.

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.

[7] 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.

Library	T_{wc} (sec)	T_{sub} (sec)	T_{elim} (sec)	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

Table 2: Running time of the worst-case points-to analysis (T_{wc}) and time for constructing the optimized summary (T_{sub} and T_{elim}). Last column shows the maximum memory usage.

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 Table 2.[8] Column T_{wc} shows the running time of the worst-case points-to analysis of the library module. Column T_{sub} contains the time needed by the algorithm from Section 5.1 to compute the variable substitution. Column T_{elim} shows the time needed to identify removable and irrelevant variables (as described in Section 5.2) 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.

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 good candidates for inclusion in realistic compilers.

Our second set of experiments investigated the difference between the basic summary described in Section 4 and the optimized summary described in 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.[9]

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 Table 3. 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.

[8] 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.

[9] 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.

Program	T_b (sec)	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%

Table 3: Cost of the summary-based points-to analysis of the client module. T_b is the analysis time with the basic summary. 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.

Column T_b in Table 3 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 accordingly. Similarly, the reduction Δ_S in the memory usage of the analysis is proportional to the relative size of the library.

These results clearly show that the optimizations presented in Section 5 can have significant impact on the size of the summary and the cost of the subsequent summary-based points-to analysis.

8 Related Work

The work in [19] presents a general approach for analyzing program fragments by using *summary values* to model the effects of calls to the fragment, and *summary functions* to model the effects of calls from the fragment to outside procedures. The worst-case separate analyses are examples of fragment analyses. However, we do not use worst-case summary values or summary functions at fragment boundaries; their role is played by the worst-case auxiliary statements. Unlike [19], we use abstractions not only for lattice elements (i.e., points-to graphs and *Mod* sets), but also for statements, procedures, and call graphs; for example, calls to *p_ph* are abstractions of callbacks from the library to client modules.

The summary-based separate analyses are also examples of fragment analyses, in which the summary information plays the same role as the summary functions from [19]. 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 previous summary construction techniques based on context-sensitive points-to analysis [12, 3, 4]. In these techniques, a called procedure is analyzed before or together with its callers. In contrast, in our approach the library summary 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 [26, 5] calculate specialized points-to information as part of the analysis. In this context, calls to unknown methods can be handled; this allows escape summary information for a module to be computed in isolation from the rest of the program. However, the techniques which are used in this manner for escape analysis cannot be used for general-purpose points-to analysis.

Flanagan and Felleisen [10] 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.

The work in [17] uses variable substitution to reduce the cost of whole-program points-to analysis. Our algorithm for computing equivalent library variables is an extension of a similar algorithm from [17]; the differences are due to the lack of information about the callers and callees of the library.

Various analyses use representative variables as placeholders for sets of related variables [15, 7, 27, 12, 8, 3, 19, 17]. In our separate analyses, we use placeholders *v_ph* and *p_ph* to represent sets of unknown variables from client modules, and placeholders rep_i to represent sets of equivalent variables.

9 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 library modules. 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., [24, 22, 6]) and to MOD/USE analyses based on them. The work in this paper is a step toward incorporating practical points-to and MOD/USE analyses in realistic compilers and software productivity tools.

Our work shows how to perform practical worst-case analysis of library modules and summary-based analysis of client modules. These separate analyses can reuse already existing implementations of 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 good candidate for inclusion in realistic compilers. We present summary optimizations that can significantly reduce the cost of the summary-based analyses without sacrificing any precision; these reductions 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 (e.g., [13, 16, 11, 9, 4]). 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 anal-

yses and summary construction in the context of standard analyses that need MOD/USE information—for example, live variables analysis and reaching definitions analysis. This problem presents interesting challenges, especially when the target analyses are flow-sensitive.

10 Acknowledgments

We would like to thank Matthew Arnold for his comments on an earlier version of this paper. This research was supported, in part, by NSF grants CCR-9804065 and CCR-9900988, and by Siemens Corporate Research.

References

[1] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[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 *Symposium on Principles of Programming Languages*, pages 133–146, 1999.

[4] B. Cheng and W. Hwu. Modular interprocedural pointer analysis using access paths. In *Conference on Programming Language Design and Implementation*, pages 57–69, 2000.

[5] 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.

[6] M. Das. Unification-based pointer analysis with directional assignments. In *Conference on Programming Language Design and Implementation*, pages 35–46, 2000.

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

[8] 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.

[9] M. Fähndrich, J. Rehof, and M. Das. Scalable context-sensitive flow analysis using instantiation constraints. In *Conference on Programming Language Design and Implementation*, pages 253–263, 2000.

[10] C. Flanagan and M. Felleisen. Componential set-based analysis. *ACM Trans. Programming Languages and Systems*, 21(2):370–416, Mar. 1999.

[11] J. Foster, M. Fähndrich, and A. Aiken. Polymorphic versus monomorphic flow-insensitive points-to analysis for C. In *Static Analysis Symposium*, LNCS 1824, pages 175–198, 2000.

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

[13] 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.

[14] M. Hind and A. Pioli. Which pointer analysis should I use? In *International Symposium on Software Testing and Analysis*, pages 113–123, 2000.

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

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

[17] 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.

[18] A. Rountev and B. G. Ryder. Practical points-to analysis for programs built with libraries. Technical Report DCS-TR-410, Rutgers University, Feb. 2000.

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

[20] B. G. Ryder, W. Landi, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural side effect analysis with pointer aliasing. Technical Report DCS-TR-336, Rutgers University, May 1998. Under revision for journal publication.

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

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

[23] 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.

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

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

[26] 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.

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

[28] 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.