8-1996

Separate Computation of Alias Information fior Reuse

Mary Jean Harrold
Ohio State University, 395 Dresse Lab, 2015 Neil Ave., Columbus, OH

Gregg Rothermel
University of Nebraska-Lincoln, grothermel2@unl.edu

Separate Computation
of Alias Information for Reuse

Mary Jean Harrold, *Member, IEEE Computer Society,*
and Gregg Rothermel, *Member, IEEE Computer Society*

Abstract—Interprocedural data flow information is useful for many software testing and analysis techniques, including data flow testing, regression testing, program slicing, and impact analysis. For programs with aliases, these testing and analysis techniques can yield invalid results, unless the data flow information accounts for aliasing effects. Recent research provides algorithms for performing interprocedural data flow analysis in the presence of aliases; however, these algorithms are expensive, and achieve precise results only on complete programs. This paper presents an algorithm for performing alias analysis on incomplete programs that lets individual software components such as library routines, subroutines, or subsystems be independently analyzed. The paper also presents an algorithm for reusing the results of this separate analysis when the individual software components are linked with calling modules. Our algorithms let us analyze frequently used software components, such as library routines or classes, independently, and reuse the results of that analysis when analyzing calling programs, without incurring the expense of completely reanalyzing each calling program. Our algorithms also provide a way to analyze large systems incrementally.

Index Terms—Aliasing, data flow analysis, pointers, reuse, separate analysis, static analysis.

——————————————— ✦ ———————————————

1 INTRODUCTION

M ANY software testing and analysis techniques, including data flow testing, regression testing, program slicing, and impact analysis, require interprocedural data flow information. These techniques can be ineffective unless the data flow information accounts for the effects of aliases (an *alias* occurs at some program point when two or more names exist for the same object) caused by reference parameters and pointer variables. Some techniques for interprocedural analysis [4] represent all invocation paths in a program, causing them to be potentially exponential in time and space. Other techniques [2], [3], [10], [11] use some type of summary information to avoid potential exponential growth, but with some loss of precision. However, these techniques require a complete program on which to perform analysis; for large systems this may be prohibitive in both time and space.

Software engineering practices encourage modular development of software, in which individual software components are separately compiled and later linked with other components. A similar process, wherein a software component is analyzed separately and later linked with other components, can provide savings in time and space. Separate analysis can save time by eliminating the need to reanalyze the component in each of its calling contexts;

• *M.J. Harrold and G. Rothermel are with the Department of Computer and Information Science, Ohio State University, 395 Dresse Lab, 2015 Neil Ave., Columbus, OH 43210-1277.*
E-mail: {harrold, grother}@cis.ohio-state.edu.

separate analysis can save space by reducing the amount of memory required to perform the analysis. To provide such savings, a separate analysis technique must compute as much information as possible about a software component, and store it for later use. The technique must provide a link algorithm, that reuses previously computed results when a piece of software that incorporates the component is analyzed.

This paper presents a technique for separate analysis of modules that addresses the interprocedural may alias problem. By *module*, we mean a single procedure, or a group of interacting procedures that has a single entry point. By *interprocedural may alias problem*, we mean the problem of determining the set of all $[N, (a, b)]$ in a program P, where N is a statement, and a and b are names in P, such that there exists a realizable path from the entry of P to the point that follows N on which a and b may reference the same object. (A *realizable path* represents a legal call and return sequence in the program such that whenever control returns from a procedure in the program, it returns to the call site that invoked it.) Our technique consists of two algorithms. The first algorithm performs may alias analysis on a separate module M, simulating the effects of calling contexts to produce may alias link information. The second algorithm performs may alias analysis on programs that use M, reusing may alias link information to avoid reanalyzing M. Our separate analysis and link algorithms can be used when the calling module is a program or another module. We first describe the way in which our separate analysis and link algorithms can be used for modules that are separately analyzed and then linked with a complete program. Then, we discuss the application of our algorithms to modules that are analyzed and then linked with other modules, enabling incremental analysis of a large system. Our algorithms are based on the interprocedural

```
        algorithm    ComputeMayAlias
        input        P : a complete program
        output       MayAlias : set of [N, PA], where PA may be aliased after execution of N
        declare      G : an interprocedural control flow graph (ICFG)
                     CondMayAlias : set of [(N, AA), PA], where PA may be aliased at the end of N
                                    if AA is aliased at the entry to the procedure that contains N
                     Worklist : list of [(N, AA), PA]; initially empty
        begin
[1]     construct ICFG G
[2]     foreach N in G do / * compute conditional may alias introductions */
[3]         if N is a call statement or an assignment to a pointer then
[4]             add conditional may aliases introduced by N to Worklist and CondMayAlias
[5]         while Worklist is not empty  do /* compute conditional may aliases */
[6]             remove [(N, AA), PA] from Worklist
[7]             propagate through successors of N; update Worklist and CondMayAlias
[8]         foreach [(N, AA), PA] in CondMayAlias do /* compute may aliases */
[9]             add [N, PA] to MayAlias
        end
```

Fig. 1. Landi and Ryder's algorithm for computing may alias information.

may alias algorithms of Landi [8] and Landi and Ryder [9], [10]; thus, our algorithms handle aliasing due to reference parameters and single and multiple level pointers, and handle recursive procedures.

One advantage of our algorithms is that they let us analyze frequently used software modules, such as library routines or classes, independently, and reuse the results of that analysis when we analyze calling programs, without incurring the expense of completely reanalyzing each calling program. With this approach, the cost of interprocedural may alias analysis for a module can be amortized over all programs that use the module. A second advantage of our algorithms is that they provide a way to analyze large systems incrementally.

In the next section, we present an overview of the algorithm on which our technique is based. Section 3 presents our separate analysis and link algorithms, discusses the precision of our results, describes versions of the algorithms that handle incomplete programs, discusses the complexity of our technique, and reviews related work. Section 4 presents our conclusions and discusses future work.

2 INTERPROCEDURAL MAY ALIAS ANALYSIS

Landi and Ryder [8], [9], [10] present an algorithm that computes interprocedural may alias information for complete programs. ComputeMayAlias, shown in Fig. 1, is a version of their algorithm.

ComputeMayAlias takes a program P as input, and outputs a set, MayAlias, of ordered pairs of form $[N, PA]$, where N is a program statement and PA represents a pair of names that may refer to the same memory location after the execution of N. The algorithm uses a worklist, Worklist, to compute a set of conditional may aliases, CondMayAlias. Both CondMayAlias and Worklist consist of tuples, $[(N, AA), PA]$, where N is a program statement, AA is a set of assumed aliases,[1] and PA is an alias pair. A tuple $[(N, AA),$

1. Although there may be an exponential number of possible sets of assumed aliases, Landi and Ryder [10] show that it is sufficient to consider sets of assumed aliases of cardinality less than or equal to one.

$PA]$, is a predicate that is true if and only if AA holding on entry to the procedure that contains N implies that PA holds after N is executed.

To compute interprocedural may alias information for P, ComputeMayAlias constructs G, an *interprocedural control flow graph* (ICFG) for P. An ICFG contains control flow graphs for each procedure in P; a *control flow graph* consists of nodes that represent statements in the procedure and edges that represent flow of control between statements [1]. Control flow graphs are augmented with *entry* and *exit* nodes. Call sites in P are rendered as *call* and *return* nodes. Call nodes are connected to entry nodes of called procedures, and exit nodes are connected to return nodes of calling procedures. Fig. 2 shows a program and its ICFG.

After ComputeMayAlias builds G, it considers each node N in G to identify conditional may aliases introduced in P. If N is an assignment to a pointer,[2] then N creates an alias pair regardless of aliases that hold prior to N; the condition, or assumed alias, responsible for such an alias pair is ϕ. For example, in Fig. 2, statement main2 is an alias introduction site in which the address of z is assigned to s. After execution of statement main2, *s and z are aliased regardless of aliases that exist before execution of main2. Thus, ComputeMayAlias adds [(main2, ϕ), (*s, z)] to CondMayAlias and Worklist. Similarly, statement main3 is an alias introduction site in which x is assigned to r; thus, ComputeMayAlias adds [(main3, ϕ), (*r, *x)] to CondMayAlias and Worklist. Statements C2, C6, and C7 also contain assignments to pointers; ComputeMayAlias performs similar actions at these statements.

Alias pairs may also be introduced at call sites where parameter bindings are present. For example, at the call site in statement C3 of Fig. 2, p is passed to a, and q is passed to b. Because q is global to D, *q and *b are aliased at D's entry. Thus, ComputeMayAlias adds [(D1, (*b, *q)), (*b, *q)] to CondMayAlias and Worklist. The assumed alias in this case is (*b, *q) rather than ϕ because the existence of (*b, *q) fol-

2. We use N to refer to both a node in G and the program statement that the node represents.

```
int *q, *r, *s, *x;

main() {          C() {               D(a, b) {
    int z;            int *p, y;          int *a, *b;

    s = &z;           p = x;              *a = *a+*b+*q
    r = x;            D(p, q);        }
    C();              if (*q > 0)
    C();                  q = r;
}                         s = &y;
                  }
```


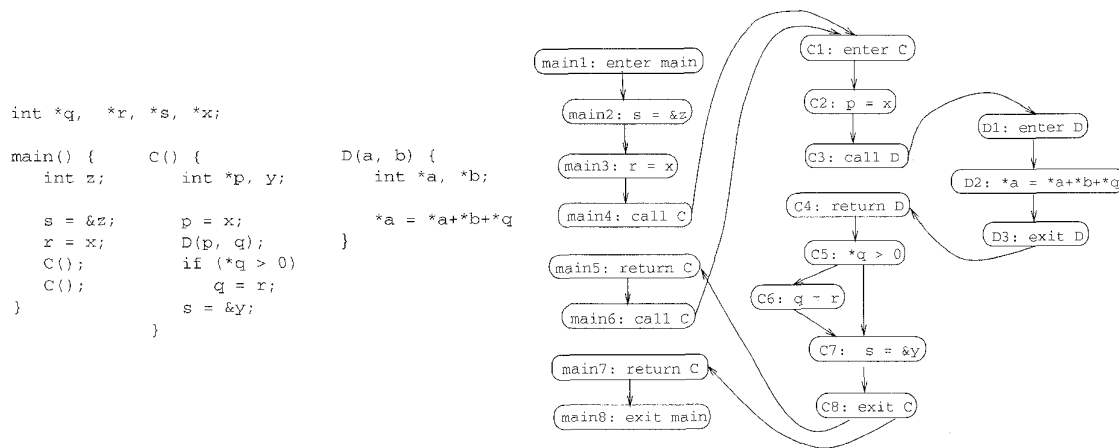
Fig. 2. An example program and its ICFG.

lowing D1 is conditional on *b and *q being aliased on entry to D. Assumed alias (*b, *q) facilitates computation of may alias information that respects calling context.

Table 1 shows the conditional may aliases that ComputeMayAlias introduces for the program of Fig. 2. (The notation "NV" that appears in the table is explained later in this section.)

TABLE 1
CONDMAYALIAS AFTER MAY ALIAS INTRODUCTIONS
FOR THE PROGRAM OF FIG. 2

$CondMayAlias$ $[(N, AA), PA]$	explanation
(main2, ϕ), (*s, z)	pointer assignment
(main3, ϕ), (*r, *x)	pointer assignment
(C2, ϕ), (*p, *x)	pointer assignment
(C6, ϕ), (*q, *r)	pointer assignment
(C7, ϕ), (*s, y)	pointer assignment
(D1, (*a, NV)), (*a, NV)	parameter binding
(D1, (*b, *q)), (*b, *q)	parameter binding

After ComputeMayAlias identifies alias introductions, it uses Worklist to compute CondMayAlias. The **while** loop at line 5 iterates until Worklist is empty. On each iteration of the loop, ComputeMayAlias removes a conditional may alias [(N, AA), PA] from Worklist, and examines each successor of N; subsequent actions depend on the type of statement associated with each successor. Table 2 shows CondMayAlias after this step is complete. (The notation "NV" that appears in the table is explained later in this section.) The first column, and its subcolumns, list CondMayAlias for each node in the example. For procedures C and D, the left subcolumn lists the conditional may aliases added to CondMayAlias for nodes in the procedures for the first call to C from main, and the right subcolumn lists the conditional may aliases added to CondMayAlias for nodes in the procedures for the second call to C from main. The rightmost column in the table lists the reason for including the associated conditional may aliases in CondMayAlias:

- *introduction* indicates that the conditional may alias is added to CondMayAlias during initial may alias introduction (line 2 of ComputeMayAlias);
- *propagation* indicates that the conditional may alias is added to CondMayAlias during propagation be-

cause it "flows through" the node being considered (lines 5 through 7 of ComputeMayAlias); and

- *generation* indicates that the conditional may alias is added to CondMayAlias during the propagation because of the interaction of the statement being considered with a propagated conditional may alias (also at lines 5 through 7 of ComputeMayAlias).

At call statements, ComputeMayAlias computes the effects of conditional may aliases that reach the call on conditional may aliases that hold following the entry node of the called procedure. For example, at some point during analysis of the program of Fig. 2, ComputeMayAlias adds [(main4, ϕ), (*r, *x)] to Worklist, indicating that alias pair (*r, *x), introduced in statement main3, may hold immediately after the call to C in statement main4. When ComputeMayAlias examines this conditional may alias, it adds [(C1, (*r, *x)), (*r, *x)] to CondMayAlias and Worklist. Another alias pair, (*s, z), holds immediately after main4, but z is *nonvisible* in (not in the scope of) C. However, the fact that a variable v is nonvisible in a procedure P does not prevent P from creating or destroying aliases that involve v by manipulating other variables that are visible in P and are aliased to v in P. Thus, an algorithm that computes may alias information must account for aliases that involve nonvisible variables. Landi and Ryder show that their algorithm needs only one place holder, NV, for nonvisible variables. Thus, ComputeMayAlias adds [(C1, (*s, NV), (*s, NV)], where NV represents nonvisible variables that may be aliased to *s, to CondMayAlias, and to Worklist.

ComputeMayAlias processes exit nodes by propagating conditional may alias information to appropriate return nodes. Suppose R is a *return* node, X is the *exit* node associated with R, E is the *entry* node associated with X, and C is the *call* node associated with R. ComputeMayAlias creates [(R, AA), PA] if and only if one of the following conditions holds:

1) [(X, ϕ), PA] holds (in which case $AA = \phi$), or
2) PA holds at X conditional on assumed alias AA' holding at E, and AA' holds at C conditional on assumed alias AA.

For example (case (1)), [(C6, ϕ), (*q, *r)] is introduced at statement C6, and after propagation, [(C8, ϕ), (*q, *r)]

TABLE 2
CONDMAYALIAS AFTER PROPAGATION FOR THE PROGRAM OF FIG. 2

$CondMayAlias$ $[(N, AA), PA]$		explanation
$[(main2, \phi), (*s, z)]$		introduction
$[(main3, \phi), (*s, z)]$		propagation
$[(main3, \phi), (*r, *x)]$		introduction
$[(main4, \phi), (*s, z)], [(main4, \phi), (*r, *x)]$		propagation
$[(main5, \phi), (*r, *x)], [(main5, \phi), (*q, *x)], [(main5, \phi), (*q, *r)]$		propagation
$[(main6, \phi), (*r, *x)], [(main6, \phi), (*q, *x)], [(main6, \phi), (*q, *r)]$		propagation
$[(main7, \phi), (*r, *x)], [(main7, \phi), (*q, *x)], [(main7, \phi), (*q, *r)]$		propagation
Tuples obtained for first call to C	Tuples obtained for second call to C	
$[(C1, (*s, NV)), (*s, NV)], [(C1, (*r, *x)), (*r, *x)]$	$[(C1, (*q, *x)), (*q, *x)], [(C1, (*q, *r)), (*q, *r)]$	propagation
$[(C2, (*s, NV)), (*s, NV)], [(C2, (*r, *x)), (*r, *x)]$	$[(C2, (*q, *x)), (*q, *x)], [(C2, (*q, *r)), (*q, *r)]$	propagation
$[(C2, \phi), (*p, *x)]$		introduction
$[(C2, (*r, *x)), (*p, *r)]$	$[(C2, (*q, *x)), (*p, *q)]$	generation
(to get tuples for C3, replace C2 in the tuples for C2 with C3		propagation
$[(C4, (*s, NV)), (*s, NV)], [(C4, (*r, *x)), (*r, *x)],$	$[(C4, (*q, *x)), (*q, *x)], [(C4, (*q, *r)), (*q, *r)]$	propagation
$[(C4, \phi), (*p, *x)], [(C4, (*r, *x)), (*p, *r)]$	$[(C4, (*q, *x)), (*p, *q)]$	propagation
(to get tuples for C5, replace C4 in the tuples for C4 with C5)		propagation
$[(C6, (*s, NV)), (*s, NV)], [(C6, (*r, *x)), (*r, *x)],$		propagation
$[(C6, (*r, *x)), (*p, *x)], [(C6, \phi), (*p, *r)]$		propagation
$[(C6, \phi), (*q, *r)]$		introduction
$[(C6, (*r, *x)), (*q, *x)], [(C6, (*r, *x)), (*p, *q)]$		generation
$[(C7, (*r, *x)), (*r, *x)], [(C7, \phi), (*p, *x)],$	$[(C7, (*q, x)), (*q, *x)], [(C7, (*q, x)), (*p, *q)]$	propagation
$[(C7, (*r, *x)), (*p, *r)], [(C7, \phi), (*q, *r)],$		propagation
$[(C7, (*r, *x)), (*q, *x)], [C7, (*r, *x), (*p, *q)]$		propagation
$[(C7, \phi), (*s, y)]$		introduction
(to get tuples for C8, replace C7 in the tuples for C7 with C8)		propagation
Tuples obtained for first call to D	Tuples obtained for second call to D	
$[(D1, (*s, NV)), (*s, NV)], [(D1, (*r, *x)), (*r, *x)],$	$[(D1, (*q, *x)), (*q, *x)], [(D1, (*q, *r)), (*q, *r)],$	propagation
$[(D1, (NV, *x)), (NV, *x)], [(D1, (NV, *r)), (NV, *r)]$	$[(D1, (NV, *q)), (NV, *q)]$	propagation
$[(D1, (*a, NV)), (*a, NV)], [(D1, (*b, *q)), (*b, *q)]$		introduction
$[(D1, (*a, *x)), (*a, *x)], [(D1, (*a, *r)), (*a, *r)]$	$[(D1, (*b, *r)), (*b, *r)], [(D1, (*b*x)), (*b, *x)],$	generation
	$[(D1, (*b, NV)), (*b, NV)],$	generation
	$[(D1, (*a, *q)), (*a, *q)], [(D1, (*a, *b)), (*a, *b)]$	generation
(to get tuples for D_i i = 2, 3, replace D1 in the tuples for D1 with D_i i = 2, 3)		propagation

holds; thus, ComputeMayAlias creates conditional may alias [(main5, ϕ), (*q, *r)]. As a further example (case (2)), in the program of Fig. 2, (*q, *x) is aliased at C8 conditional on assumed alias (*r, *x) holding at C1, and (*r, *x) holds at main4 conditional on assumed alias ϕ, so ComputeMayAlias creates conditional may alias [(main5, ϕ), (*q, *x)]. Because ComputeMayAlias considers associated call nodes when it propagates conditional may aliases forward from an exit node, it preserves the calling context of called procedures; this restricts propagation to realizable paths in the ICFG.

To see how ComputeMayAlias handles conditional may aliases that contain nonvisible variables, consider statement C7. At statement C7, s is reassigned; this assignment kills all aliases of *s because the reassignment to s changes their bindings. Thus, when ComputeMayAlias examines [(C5, (*s, NV)), (*s, NV)] and [(C6, (*s, NV)), (*s, NV)], it does not create conditional may alias [(C7, (*s, NV)), (*s, NV)]. In the statements in main after main5, (*s, z) is no longer an alias pair.

A pointer assignment statement can affect alias information in many ways. Landi and Ryder give rules for each possible situation; we discuss a few of these rules. Consider the effect of the pointer assignment in statement C2 on [(C1, (*r, *x)), (*r, *x)]. When [(C1, (*r, *x)), (*r, *x)] is propagated through statement C2, conditional may alias [(C2, (*r, *x)), (*p, *r)] is created: If *r and *x may be aliased, and x is assigned to p, then *r and *p may be aliased. Thus, ComputeMayAlias creates [(C2, (*r, *x)), (*p, *r)]. Similarly, [(C1, (*r, *x)), (*r, *x)] propagates through C5, causing ComputeMayAlias to create [(C6, (*r, *x)), (*q, *x)].

At any other type of statement, the conditional may aliases that hold before the statement is executed also hold after the statement is executed, because alias information just "flows through" these statements. Thus, ComputeMayAlias just propagates conditional may aliases through such statements.

Finally, when multiple conditional may aliases exist at some program point, these aliases may combine to induce further aliases. Landi and Ryder show that the cost of precisely calculating aliases created in this fashion is prohibitive; however, they show that their algorithm computes safe, conservative results with respect to these aliases. In the example of Fig. 2, two such may aliases created by multiple conditions are [(D1, (*a, *q)), (*a, *q)] and [(D1, (*a, *b)), (*a, *b)]. We postpone further discussion of this issue and of the method for addressing it to Section 3.3.

To compute may aliases, ComputeMayAlias converts each [(N, AA), PA] in CondMayAlias to [N, PA], and adds it to MayAlias. Table 3 shows the complete may alias solution for the program of Fig. 2.

3 SEPARATE ANALYSIS OF MODULES

Separate analysis considers a module M in isolation. This analysis provides information about M that can be stored with M, and reused when programs that use M are analyzed, to obtain a complete solution without completely reanalyzing M. Fig. 3 gives an overview of our separate analysis and link algorithms. ComputeMayAlias-Module takes a module M as input, calculates may alias link

TABLE 3
MAYALIAS AFTER EACH STATEMENT FOR THE PROGRAM IN FIG. 2

$MayAlias[N, PA]$		
main2, (*s,z)		
main3, (*s,z) , main3, (*r,*x)		
main4, (*s,z) , main4, (*r,*x)		
main5, (*r,*x) , main5, (*q,*x) , main5, (*q,*r)		
main6, (*r,*x) , main6, (*q,*x) , main6, (*q,*r)		
main7, (*r,*x) , main6, (*q,*x) , main6, (*q,*r)		
Pairs computed for first call to C		**Pairs computed for second call to C**
C1, (*r,*x)		C1, (*q,*x) , C1, (*q,*r)
C2, (*r,*x) , C2, (*p,*x) , C2, (*p,*r)		C2, (*q,*x) , C2, (*q,*r) , C2, (*p,*q)
C3, (*r,*x) , C3, (*p,*x) , C3, (*p,*r)		C3, (*q,*x) , C3, (*q,*r) , C3, (*p,*q)
C4, (*r,*x) , C4, (*p,*x) , C4, (*p,*r)		C4, (*q,*x) , C4, (*q,*r) , C4, (*p,*q)
C5, (*r,*x) , C5, (*p,*x) , C5, (*p,*r)		C5, (*q,*x) , C5, (*q,*r) , C5, (*p,*q)
C6, (*r,*x) , C6, (*p,*x) , C6, (*p,*r) , C6, (*q,*r) , C6, (*q,*x) , C6, (*p,*q)		
C7, (*r,*x) , C7, (*p,*x) , C7, (*p,*r) , C7, (*q,*r) , C7, (*q,*x) , C7, (*p,*q) , C7, (*s,y)		
C8, (*r,*x) , C8, (*p,*x) , C8, (*p,*r) , C8, (*q,*r) , C8, (*q,*x) , C8, (*p,*q) , C8, (*s,y)		
Pairs computed for first call to D		**Pairs computed for second call to D**
D1, (*r,*x) , D1, (*b,*q) , D1, (*a,*x) , D1, (*a,*r)		D1, (*q,*x) , D1, (*q,*r) , D1, (*b,*r) , D1, (*b,*x) , D1, (*a,*b) , D1, (*a,*q)

information for M, and stores that information. To obtain may alias link information for M that is sufficient for use in the contexts of applications programs that call M, ComputeMayAlias-Module simulates the aliasing effects that are possible in all calling contexts. To do this, it analyzes M under the assumption that all possible aliases reach the call to M, and tracks the effects of these aliases. When AnalyzeApplication analyzes an applications program P that uses M, it uses the may alias link information for M to obtain may alias information for P, instead of completely reanalyzing M.

Section 3.1 presents ComputeMayAlias-Module. Section 3.2 presents AnalyzeApplication. The remainder of Section 3 discusses additional issues.
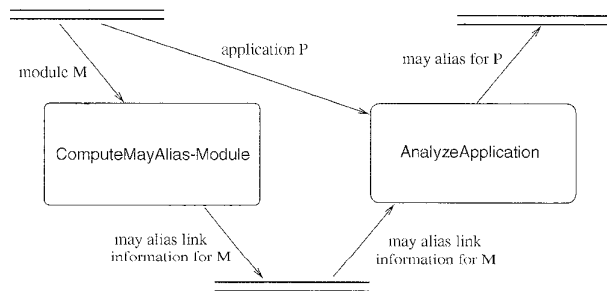

Fig. 3. Overview of separate analysis and link algorithms.

3.1 Computation of Link Information for Separately Analyzed Modules

Fig. 4 presents ComputeMayAlias-Module, our algorithm for obtaining may alias link information for a module M. ComputeMayAlias-Module, like ComputeMayAlias, propagates conditional may alias information throughout the module using a graph, and calculates may alias information from the conditional may aliases. The algorithms differ, however, in two significant ways. First,

when ComputeMayAlias-Module analyzes a module, it induces conditional may aliases at the entry to the module and propagates them throughout the module, to track the effects of external alias information on the may alias solution for the module. Second, ComputeMayAlias-Module computes and outputs may alias link information that facilitates reuse of module-specific alias information when the module is analyzed in the context of a calling program.

To enable ComputeMayAlias-Module to track the effects of external aliases that reach a call to a module, we use *inducement conditions*. An inducement condition is a may alias (a, b) that can reach a call to M. By adding inducement conditions to conditional may alias information, we distinguish two classes of aliases: those whose existence depends on may aliases that reach a call to M, and those whose existence does not depend on external may aliases. We specify this distinction more precisely as follows:

- If $[(N, AA), PA]$ is a conditional may alias at node N, and $[(N, AA), PA]$ exists at N if and only if some may alias IC exists on entry to M, then $[(N, AA)_{IC}, PA]$ is true. In this case, IC is the inducement condition for $[(N, AA)_{IC}, PA]$, and $[(N, AA)_{IC}, PA]$ is an *induced conditional may alias*.
- If $[(N, AA), PA]$ is a conditional may alias at node N, and $[(N, AA), PA]$ exists at N independent of may aliases that exist on entry to M, then $[(N, AA)_{\phi}, PA]$ is true. In this case, $[(N, AA)_{\phi}, PA]$ is a *noninduced conditional may alias*.

Similarly, to allow ComputeMayAlias-Module to create may alias link information, we augment may alias information, which ComputeMayAlias keeps in tuples of the form $[N, PA]$, to include inducement conditions, by rendering it in the form $[N_{IC}, PA]$, where IC may be an alias pair or ϕ. This form distinguishes *induced may aliases* from *noninduced may aliases*, and tracks inducement conditions for may aliases in a module M.

```
algorithm    ComputeMayAlias-Module
input        M : a module
output       CondMayAlias-LinkInfo: subset of CondMayAlias-Module
             MayAliasInfo-Link: subset of MayAlias-Module
             ICFG-Module: reduced ICFG for M
declare      G : an interprocedural control flow graph (ICFG) for M, with entry node E and exit node X
             Worklist : list of [(N, AA)_{IC}, PA], initially empty
             CondMayAlias-Module : set of [(N, (AA))_{IC}, (PA)]
             MayAlias-Module : set of [N_{IC}, (PA)]
             PASet : set of names potentially aliased in M
begin
[1]     construct G, an ICFG for M /* construct the ICFG for M */
[2]     compute PASet for M /* compute the PASet for M */
[3]     foreach PA in PASet do /* compute conditional may alias introductions for M */
[4]         add [(E, PA)_{PA}, PA] to Worklist and to CondMayAlias-Module
[5]     foreach N in G do
[6]         if N is an assignment to a pointer or a call statement then
[7]             add conditional may aliases introduced by N to Worklist and to CondMayAlias-Module
[8]     while Worklist is not empty do /* compute conditional may alias information for M */
[9]         remove [(N, AA)_{IC}, PA] from Worklist
[10]        propagate at N and update Worklist and CondMayAlias-Module
[11]    foreach [(N, AA)_{IC}, PA] in CondMayAlias-Module do /* compute may alias information for M */
[12]        add [N_{IC}, PA] to MayAlias-Module
[13]    ICFG-Module = node set {E, X} and edge set {(E, X)}
[14]    foreach [(X, AA)_{IC}, PA] in CondMayAlias-Module do /* output may alias link information for M */
[15]        add [(X, AA), PA] to CondMayAlias-LinkInfo
[16]    foreach may alias [N_{IC}, PA] in MayAlias-Module do
[17]        add [N_{IC}, PA] to MayAlias-LinkInfo
[18]    output CondMayAlias-LinkInfo, MayAlias-LinkInfo, and ICFG-Module
end
```

Fig. 4. Algorithm for computing may alias link information for a module.

```
C(){                 D(a, b) {
    int *p, y;           int *a, *b;

    p = x;               *a = *a+*b+*q
    D(p, q);         }
    if (*q > 0)
        q = r;
    s = &y;
}
```
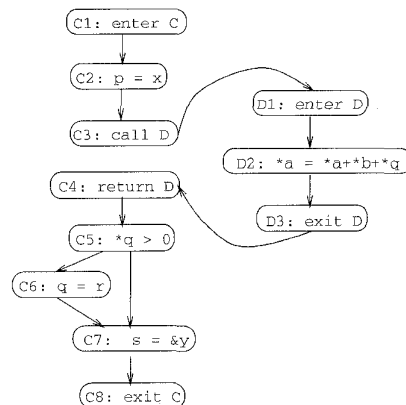

Fig. 5. A module and its ICFG.

In the remainder of this section, we discuss ComputeMayAlias-Module, and illustrate its operation on the partial program of Fig. 5. This partial program is a component of the program of Fig. 1.

3.1.1 Construct the ICFG for Module M

Suppose ComputeMayAlias-Module is called with mod-ule M. To compute may alias information for M,

ComputeMayAlias-Module first constructs G, the ICFG for M.[3] Fig. 5 shows the ICFG for our example module.

3. The constraints placed on call graphs (of which the ICFG is a variant), and thus, their construction, depend on the intended application for the call graph [14]. We place constraints that are suitable for our applications, as follows. When programs do not contain pointers to functions, the ICFG is trivial to compute; in our experience, function pointers are rarely used in C programs. When programs do contain pointers to functions, we make conservative assumptions about the targets of function calls; we can increase the precision of these assumptions, at some cost, by using algorithms that perform points-to analysis. In situations where we cannot determine precisely which function is invoked at a call site, the call and return nodes for that site are attached to each function that may be a recipient of that call. In this case, our algorithm may identify may aliases that do not hold in the program in practice; however, the algorithm will produce conservative, safe results. Such results are sufficient for many applications.

3.1.2 Compute the PASet for Module M

Given a module M, our separate analysis computes analysis information for M without knowledge of any calling environment, while accounting for the effects of calling environments. For the may alias problem, the analysis must account for the potential effects on M of aliases introduced by a calling environment. We refer to the set of aliases that may be introduced by calling environments as the *potential alias set* (*PASet*). Alias pairs in the *PASet* may involve three types of objects:

1) global variables that are defined or used in, and thus, known during the analysis of, M (*global*),
2) parameters to M (*parameter*), and
3) variables that do not appear in M but can appear in a calling program (*nonaccessed*).

There are nine combinations of these three types of objects, as Table 4 shows.

TABLE 4
POSSIBLE COMBINATIONS OF PAIRS OF OBJECTS

Pairs of Types of Objects
(global, global)
(global, parameter), (parameter, global)
(global, nonaccessed), (nonaccessed, global)
(parameter, parameter)
(parameter, nonaccessed), (nonaccessed, parameter)
(nonaccessed, nonaccessed)

For variables a and b, alias pairs (a, b) and (b, a) both represent the fact that a and b may be aliased. Because these pairs are symmetric, we consider only one of them. Thus, to consider aliasing effects caused by calling environments, we consider the following types of pairs of objects: (global, global), (global, parameter), (global, nonaccessed), (parameter, parameter), (parameter, nonaccessed), and (nonaccessed, nonaccessed). We discuss these types of *PASet* elements, and illustrate them using our example. Tables 5 and 6 give *PASets* for modules C and D, respectively. In the tables, "NA" stands for nonaccessed. We show the *PASet* for D only to illustrate elements that would belong in the *PASet* for D if D were analyzed separately. The remainder of our treatment of the partial program of Fig. 5 views the program as one module with C as its entry point.

Alias pairs of the form (global, global) may be created in a calling module and propagated to M. For example, in module C of Fig. 5, global variables q, r, s, and x are accessed in C. If they are aliased in some application that calls C, these aliases could propagate to C. Thus, `ComputeMayAlias-Module` adds (*q, *r), (*q, *s), (*q, *x), (*r, *s), (*r, *x), and (*s, *x) to the *PASet* for C.

Alias pairs of the form (parameter, parameter) may be created in a calling module and propagated to M. For example, a and b are parameters to module D of Fig. 5, which has two pointer variable parameters. If D were called with the same actual parameter for both a and b, then *a and *b would be aliased on entry to D. Thus, `ComputeMayAlias-Module` adds (*a, *b) to the *PASet* for D.

Alias pairs of the form (global, parameter) may be created in a calling environment and propagated to M. For example, in module D of Fig. 5, q may be bound to the actual parameter in a calling module and propagated to D. Thus, `ComputeMayAlias-Module` adds (*q, *a) and (*q, *b) to the *PASet* for D.

TABLE 5
PASET FOR MODULE C

Type	Alias Pairs in $PASet$
(global, global)	(*q, *r), (*q, *s), (*q, *x), (*r, *s), (*r, *x), (*s, *x)
(global, nonaccessed)	(*q, NA), (*r, NA), (*s, NA), (*x, NA)
all other types	none

TABLE 6
PASET FOR MODULE D

Type	Alias Pairs in $PASet$
(parameter, parameter)	(*a, *b)
(global, parameter)	(*q, *a), (*q, *b)
(parameter, nonaccessed)	(*a, NA), (*b, NA)
(global, nonaccessed)	(*q, NA)
(global, global)	none

Alias pairs of the form (global, nonaccessed) and (parameter, nonaccessed) can also affect the may alias information in M. `ComputeMayAlias-Module` uses Landi and Ryder's method, discussed in Section 2, of summarizing nonaccessed variables using a placeholder; we choose NA as this placeholder. `ComputeMayAlias-Module` creates an alias pair of the form (v, NA) for each parameter or global variable v accessed in M. Thus, for the module of Fig. 5, the algorithm adds (global, nonaccessed) pairs (*q, NA), (*r, NA), (*s, NA), and (*x, NA) to the *PASet* for C. If the algorithm were run on module D separately, it would add (*a, NA), (*b, NA), and (*q, NA) to the *PASet* for D.

Aliasing effects that occur when nonaccessed variables are paired with globals or parameters bear further discussion. Nonaccessed variables may be visible (in scope) in module M or nonvisible in M. Landi and Ryder's approach handles only complete programs, in which all variables visible in a procedure are known. In contrast, if we analyze a module independently of a calling program, there may be variables in particular calling programs that are visible in the module, but are not explicitly referenced in the module (nonaccessed); when we analyze the module, we do not know the names of these variables. Where aliases that involve a nonaccessed, nonvisible variable v are concerned, a module M can create or destroy aliases of v that hold *outside* M. However, M cannot affect aliases of v that hold *inside* M. In contrast, where aliases that involve nonaccessed, visible variable v are concerned, M can create or destroy aliases *both inside and outside* M. Thus, we must treat nonaccessed, nonvisible variables and nonaccessed, visible variables differently. However, this different treatment is confined to the link algorithm, and does not affect the algorithm for partial analysis; during the partial analysis, one placeholder suffices for both types of nonaccessed variables.

Finally, alias pairs of the form (nonaccessed, nonaccessed) may be created in a calling module and propagated to M. For example, in an applications program that uses module C, global variables *k and *l may be aliased. Neither k nor l appear in module C; nevertheless, alias pair

TABLE 7
CondMayAlias-Module After May Alias Introductions for the Module of Fig. 5

$CondMayAlias\text{-}Module$ $[(N, AA)_{IC}, PA]$	explanation
$[(C1,(*q,*r))_{(*q,*r)},(*q,*r)]$, $[(C1,(*q,*s))_{(*q,*s)},(*q,*s)]$, $[(C1,(*q,*x))_{(*q,*x)},(*q,*x)]$	induced
$[(C1,(*r,*s))_{(*r,*s)},(*r,*s)]$, $[(C1,(*r,*x))_{(*r,*x)},(*r,*x)]$, $[(C1,(*s,*x))_{(*s,*x)},(*s,*x)]$	induced
$[(C1,(*q,NA))_{(*q,NA)},(*q,NA)]$, $[(C1,(*r,NA))_{(*r,NA)},(*r,NA)]$,	induced
$[(C1,(*s,NA))_{(*s,NA)},(*s,NA)]$, $[(C1,(*x,NA))_{(*x,NA)},(*x,NA)]$	induced
$[(C2,\phi)_{\phi},(*p,*x)]$	pointer assignment
$[(C6,\phi)_{\phi},(*q,*r)]$	pointer assignment
$[(C7,\phi)_{\phi},(*s,y)]$	pointer assignment
$[(D1,(*a,NA))_{\phi},(*a,NA)]$, $[(D1,(*b,*q))_{\phi},(*b,*q)]$	parameter binding

(*k, *1) holds at every statement in C. We could handle these aliasing effects by introducing alias pair (NA, NA) into PASet, but this promotes unnecessary work. Instead, we account for the effects of these aliases during the link algorithm.

3.1.3 Compute Conditional May Alias Introductions for Module M

After ComputeMayAlias-Module computes PASet, it computes conditional may aliases that may be introduced at particular nodes in G. First, at the entry node to M, the algorithm forces introductions of all conditional may aliases that could reach M from an applications program, by creating a conditional may alias for each element in PASet. Each of these conditional may aliases has itself as inducement condition; that is, for each conditional may alias $[(E, PA)_{IC}, PA]$ created at this step, $IC = PA$. This inducement condition indicates that the existence of the conditional may alias depends on a conditional may alias of IC holding at the call to M. For example, for the module of Fig. 5, ComputeMayAlias-Module creates conditional may alias $[(C1, (*q, *r))_{(*q,*r)}, (*q, *r)]$, and the nine other induced conditional may aliases listed in Table 7.

Next, ComputeMayAlias-Module computes conditional may alias introductions at pointer assignment nodes, and entry nodes other than the entry to M, using the same rules used by ComputeMayAlias. However, these conditional may aliases have null inducement conditions because their existence does not depend on particular aliases reaching a call to M. Table 7 lists these conditional may alias introductions.

ComputeMayAlias-Module adds all conditional may aliases, whether induced or not, to Worklist and to CondMayAlias-Module.

3.1.4 Compute Conditional May Alias Information for Module M

Like Landi and Ryder's ComputeMayAlias algorithm, ComputeMayAlias-Module next propagates conditional may aliases introduced in the previous step throughout M using G. However, to support the subsequent step of calculating may alias information that can be reused during analysis of a calling program, the algorithm preserves inducement conditions during propagation. This means that at each node N, ComputeMayAlias propagates $[(N, AA)_{IC}, PA]$ through N: if ComputeMayAlias propagation rules state that $[(N, AA), PA]$ holding at N causes

$[(N', AA'), PA']$ to hold at successor node N' of N, then ComputeMayAlias-Module generates $[(N', AA')_{IC}, PA']$. For example, in the module of Fig. 5, $[(C1, (*r, *x))_{(*r,*x)}, (*r, *x)]$ holds, and by ComputeMayAlias rules, if $[(C1, (*r, *x)), (*r, *x)]$ holds then $[(C2, (*r, *x)), (*p, *r)]$ holds. Thus, ComputeMayAlias-Module generates conditional may alias $[(C2, (*r, *x))_{(*r,*x)}, (*p, *r)]$, and adds it to Worklist and to CondMayAlias-Module. During this propagation, aliases created by multiple conditions may also be created. We postpone discussion of this issue until Section 3.3. Table 8 shows the results of the conditional may alias propagation for the example module.

3.1.5 Compute May Alias Information for Module M

By forcing all potential conditional may aliases at entry to M, and propagating them throughout G, ComputeMayAlias-Module collects conditional may alias information that accounts for all possible calling contexts. ComputeMayAlias-Module uses this information to calculate may alias information for M that also accounts for all calling contexts. To calculate may aliases, ComputeMayAlias-Module uses ComputeMayAlias rules for obtaining may alias information from conditional may alias information, but retains inducement conditions with may aliases. For example, for the program of Fig. 5, because $[(C6, (*r, *x))_{(*r,*x)}, (*q, *x)]$ is a conditional may alias for C, ComputeMayAlias-Module creates may alias $[(C6_{(*r,*x)}, (*q, *x)]$, and places it in MayAlias-Module. Table 9 shows the may alias information computed by ComputeMayAlias-Module for the example module.

3.1.6 Output May Alias Link Information for Module M

ComputeMayAlias-Module packages the results of conditional may alias and may alias analyses of module M into a form that supports reuse of that information when a program that calls M is analyzed. ComputeMayAlias-Module then outputs that information. Three types of information must be saved:

1) a reduced ICFG for M, ICFG-Module,
2) conditional may alias link information, CondMayAlias-LinkInfo, and
3) may alias link information, MayAlias-LinkInfo.

ICFG-Module is a reduced ICFG for M; this graph is used to incorporate module analysis results for M into the analysis of an applications program. The reduced graph contains

TABLE 8
CONDMAYALIAS-MODULE AFTER CONDITIONAL MAY ALIAS COMPUTATION FOR THE MODULE OF FIG. 5

$CondMayAlias\text{-}Module\ [(N, AA)_{IC}, PA]$	explanation
$[(C1, (*q, *r))_{(*q,*r)}, (*q, *r)], [(C1, (*q, *s))_{(*q,*s)}, (*q, *s)], [(C1, (*q, *x))_{(*q,*x)}(*q, *x)]$	introduction
$[(C1, (*r, *s))_{(*r,*s)}, (*r, *s)], [(C1, (*r, *x))_{(*r,*x)}, (*r, *x)], [(C1, (*s, *x))_{(*s,*x)}(*s, *x)]$	introduction
$[(C1, (*q, NA))_{(*q,NA)}, (*q, NA)], [(C1, (*r, NA))_{(*r,NA)}, (*r, NA)]$	introduction
$[(C1, (*s, NA))_{(*s,NA)}, (*s, NA)]\ [(C1, (*x, NA))_{(*x,NA)}, (*x, NA)]$	introduction
$[(C2, \phi)_\phi, (*p, *x)]$	introduction
$[(C2, (*q, *r))_{(*q,*r)}, (*q, *r)], [(C2, (*q, *s))_{(*q,*s)}, (*q, *s)], [(C2, (*q, *x))_{(*q,*x)}(*q, *x)]$	propagation
$[(C2, (*r, *s))_{(*r,*s)}, (*r, *s)], [(C2, (*r, *x))_{(*r,*x)}, (*r, *x)], [(C2, (*s, *x))_{(*s,*x)}(*s, *x)]$	propagation
$[(C2, (*q, NA))_{(*q,NA)}, (*q, NA)], [(C2, (*r, NA))_{(*r,NA)}, (*r, NA)]$	propagation
$[(C2, (*s, NA))_{(*s,NA)}, (*s, NA)]\ [(C2, (*x, NA))_{(*x,NA)}, (*x, NA)]$	propagation
$[(C2, (*q, *x))_{(*q,*x)}, (*p, *q)], [(C2, (*s, *x))_{(*s,*x)}, (*p, *s)]$	generation
$[(C2, (*r, *x))_{(*r,*x)}, (*p, *r)], [(C2, (*x, NA))_{(*x,NA)}, (*p, NA)]$	generation
(to get tuples for Ci, i = 3, 4, 5 replace C2 in the tuples for C2 with Ci, i = 3, 4, 5)	propagation
$[(C6, \phi)_\phi, (*q, *r)]$	introduction
$[(C6, (*r, *s))_{(*r,*s)}, (*r, *s)], [(C6, (*r, *x))_{(*r,*x)}, (*r, *x)], [(C6, (*s, *x))_{(*s,*x)}(*s, *x)]$	propagation
$[(C6, (*r, NV))_{(*r,NV)}, (*r, NV)], [(C6, (*s, NV))_{(*s,NV)}, (*s, NV)], [(C6, (*x, NV))_{(*x,NV)}, (*x, NV)]$	propagation
$[(C6, (*r, NA))_{(*r,NA)}, (*r, NA)], [(C6, (*s, NA))_{(*s,NA)}, (*s, NA)], [(C6, (*x, NA))_{(*x,NA)}, (*x, NA)]$	propagation
$[(C6, (*x, NV))_{(*x,NV)}, (*p, NV)], [(C6, (*x, NA))_{(*x,NA)}, (*p, NA)], [(C6, (*s, *x))_{(*s,*x)}, (*p, *s)]$	propagation
$[(C6, \phi)_\phi, (*p, *x)], [(C6, (*r, *x))_{(*r,*x)}, (*p, *r)]$	propagation
$[(C6, (*r, *s))_{(*r,*s)}, (*q, *s)], [(C6, (*r, *x))_{(*r,*x)}, (*q, *x)], [(C6, (*r, *x))_{(*r,*x)}, (*p, *q)],$	generation
$[(C6, (*r, NV))_{(*r,NV)}, (*q, NV)], [(C6, (*r, NA))_{(*r,NA)}, (*q, NA)]$	generation
$[(C7, \phi)_\phi, (*s, y)]$	introduction
$[(C7, \phi)_{(\phi)}, (*q, *r)], [(C7, \phi)_{(\phi)}, (*p, *x)], [(C7, (*q, *x))_{(*q,*x)}(*q, *x)], [(C7, (*r, *s))_{(*r,*s)}, (*r, *s)]$	propagation
$[(C7, (*r, *x))_{(*r,*x)}, (*r, *x)], [(C7, (*q, NA))_{(*q,NA)}, (*q, NA)], [(C7, (*r, NA))_{(*r,NA)}, (*r, NA)]$	propagation
$[(C7, (*x, NA))_{(*x,NA)}, (*x, NA)], [(C7, (*q, *x))_{(*q,*x)}, (*p, *q)], [(C7, (*r, *x))_{(*r,*x)}, (*p, *r)]$	propagation
$[(C7, (*x, NA))_{(*x,NA)}, (*p, NA)], [(C7, (*r, *x))_{(*r,*x)}, (*q, *x)], [(C7, (*r, *x))_{(*r,*x)}, (*p, *q)]$	propagation
$[(C7, (*r, NA))_{(*r,NA)}, (*q, NA)]$	propagation
(to get tuples for C8 replace C7 in the tuples for C7 with C8)	propagation
$[(D1, (*a, NA))_\phi, (*a, NA)], [(D1, (*b, *q))_\phi, (*b, *q)]$	introduction
$[(D1, (*q, *r))_{(*q,*r)}, (*q, *r)], [(D1, (*q, *s))_{(*q,*s)}, (*q, *s)], [(D1, (*q, *x))_{(*q,*x)}(*q, *x)]$	propagation
$[(D1, (*r, *s))_{(*r,*s)}, (*r, *s)], [(D1, (*r, *x))_{(*r,*x)}, (*r, *x)], [(D1, (*s, *x))_{(*s,*x)}(*s, *x)]$	propagation
$[(D1, (*q, NA))_{(*q,NA)}, (*q, NA)], [(D1, (*r, NA))_{(*r,NA)}, (*r, NA)]$	propagation
$[(D1, (*s, NA))_{(*s,NA)}, (*s, NA)]\ [(D1, (*x, NA))_{(*x,NA)}, (*x, NA)]$	propagation
$[(D1, (*a, *x))_{(*r,*x)}, (*a, *x)], [(D1, (*a, *r))_{(*r,*x)}, (*a, *r)], [(D1, (*a, *s))_{(*s,*x)}, (*a, *s)]$	generation
$[(D1, (*b, *q))_{(*q,*r)}, (*b, *q)], [(D1, (*b, *s))_{(*q,*s)}, (*b, *s)], [(D1, (*b, *x))_{(*q,*x)}, (*b, *x)]$	generation
$[(D1, (*a, NA))_{(*x,NA)}, (*a, NA)], [(D1, (*b, NA))_{(*q,NA)}, (*b, NA)]$	generation
(to get tuples for Di, i = 2, 3, replace D1 in the tuples for D1 with Di, i = 2, 3)	propagation

TABLE 9
MAYALIAS-MODULE AFTER MAY ALIAS CALCULATION FOR MODULE OF FIG. 5

$MayAlias\text{-}Module\ [N_{IC}, PA]$
$[C1_{(*q,*r)}, (*q, *r)], [C1_{(*q,*s)}, (*q, *s)], [C1_{(*q,*x)}(*q, *x)], [C1_{(*r,*s)}, (*r, *s)], [C1_{(*r,*x)}, (*r, *x)]$
$[C1_{(*s,*x)}(*s, *x)], [C1_{(*q,NA)}, (*q, NA)], [C1_{(*r,NA)}, (*r, NA)], [C1_{(*s,NA)}, (*s, NA)], [C1_{(*x,NA)}, (*x, NA)]$
$[C2_\phi, (*p, *x)], [C2_{(*q,*r)}, (*q, *r)], [C2_{(*q,*s)}, (*q, *s)], [C2_{(*q,*x)}(*q, *x)], [C2_{(*r,*s)}, (*r, *s)]$
$[C2_{(*r,*x)}, (*r, *x)], [C2_{(*s,*x)}(*s, *x)], [C2_{(*q,NA)}, (*q, NA)], [C2_{(*r,NA)}, (*r, NA)], [C2_{(*s,NA)}, (*s, NA)]$
$[C2_{(*x,NA)}, (*x, NA)], [C2_{(*q,*x)}, (*p, *q)], [C2_{(*s,*x)}, (*p, *s)], [C2_{(*r,*x)}, (*p, *r)], [C2_{(*x,NA)}, (*p, NA)]$
(to get tuples for Ci, i = 3, 4, 5 replace C2 in the tuples for C2 with Ci, i = 3, 4, 5)
$[C6_\phi, (*q, *r)], [C6_{(*r,*s)}, (*r, *s)], [C6_{(*r,*x)}, (*r, *x)], [C6_{(*s,*x)}(*s, *x)], [C6_{(*r,NA)}, (*r, NA)], [C6_{(*s,NA)}, (*s, NA)]$
$[C6_{(*x,NA)}, (*x, NA)], [C6_{(*x,NA)}, (*p, NA)], [C6_{(*x,NA)}, (*p, NA)], [C6_\phi, (*p, *x)]$
$[C6_{(*r,*x)}, (*p, *r)], [C6_{(*r,*s)}, (*q, *s)], [C6_{(*r,*x)}, (*q, *x)], [C6_{(*r,*x)}, (*p, *q)], [C6_{(*r,NA)}, (*q, NA)]$
$[C7_\phi, (*s, y)], [C7_{(\phi)}, (*q, *r)], [C7_{(\phi)}, (*p, *x)], [C7_{(*q,*x)}(*q, *x)], [C7_{(*r,*s)}, (*r, *s)]$
$[C7_{(*r,*x)}, (*r, *x)], [C7_{(*q,NA)}, (*q, NA)], [C7_{(*r,NA)}, (*r, NA)], [C7_{(*x,NA)}, (*x, NA)], [C7_{(*q,*x)}, (*p, *q)]$
$[C7_{(*r,*x)}, (*p, *r)], [C7_{(*x,NA)}, (*p, NA)], [C7_{(*r,*x)}, (*q, *x)], [C7_{(*r,*x)}, (*p, *q)], [C7_{(*r,NA)}, (*q, NA)]$
(to get tuples for C8 replace C7 in the tuples for C7 with C8)
$[D1_\phi, (*a, NA)], [D1_\phi, (*b, *q)], [D1_{(*q,*r)}, (*q, *r)], [D1_{(*q,*s)}, (*q, *s)], [D1_{(*q,*x)}(*q, *x)]$
$[D1_{(*r,*s)}, (*r, *s)], [D1_{(*r,*x)}, (*r, *x)], [D1_{(*s,*x)}(*s, *x)], [D1_{(*q,NA)}, (*q, NA)], [D1_{(*r,NA)}, (*r, NA)]$
$[D1_{(*s,NA)}, (*s, NA)], [D1_{(*x,NA)}, (*x, NA)], [D1_\phi, (*a, *x)], [D1_{(*r,*x)}, (*a, *r)], [D1_{(*s,*x)}, (*a, *s)]$
$[D1_{(*q,*s)}, (*b, *s)], [D1_{(*q,*x)}, (*b, *x)], [D1_{(*q,*r)}, (*b, *r)]\ [D1_{(*x,NA)}, (*a, NA)], [D1_{(*q,NA)}, (*b, NA)]$
(to get tuples for Di, i = 2, 3, replace D1 in the tuples for D1 with Di, i = 2, 3)

only the entry and exit nodes of G, with a single edge from the entry node to the exit node.

CondMayAlias-LinkInfo is the conditional may alias information required to incorporate module analysis results for M into the analysis of an applications program. It suf-

fices to output the conditional may aliases that reach the exit node in G and do not involve local variables. Table 10 reports this information for the example program.

MayAlias-LinkInfo is the may alias information required to incorporate module analysis results into the analysis of

TABLE 10
CondMayAlias-LinkInfo for the Module of Fig. 5

$CondMayAlias\text{-}LinkInfo$ $[(N, AA)_{IC}, PA]$
$[(C8, \phi)_\phi, (*q, *r)], \ [(C8, (*q, *x))_{(*q, *x)}(*q, *x)], \ [(C8, (*r, *s))_{(*r, *s)}, (*r, *s)]$ $[(C8, (*r, *x))_{(*r, *x)}, (*r, *x)], \ [(C8, (*q, NA))_{(*q, NA)}, (*q, NA)], \ [(C8, (*r, NA))_{(*r, NA)}, (*r, NA)]$ $[(C8, (*x, NA))_{(*x, NA)}, (*x, NA)], \ [(C8, (*r, *x))_{(*r, *x)}, (*q, *x)], \ [(C8, (*r, NA))_{(*r, NA)}, (*q, NA)]$

```
algorithm    AnalyzeApplication
input        P : a program
output       MayAlias : set of [N, PA], where PA may be aliased after N
declare      G : an interprocedural control flow graph (ICFG) for P
             Worklist : list of [(N, AA), PA]; initially empty
             CondMayAlias : set of [(N, (AA)), (PA)] for P
             CondMayAlias-LinkInfo: conditional may aliases for modules
             MayAlias-LinkInfo: alias link information for modules
begin
[1]     construct G, an ICFG for P
[2]     foreach N in G do /* compute conditional may alias introductions for P */
[3]         if N is an assignment to a pointer or a call node then
[4]             add conditional may aliases introduced by N to Worklist and CondMayAlias
[5]         if N is an exit node of a separately analyzed module M then
[6]             foreach [(N, φ)_φ, PA] in CondMayAlias-LinkInfo for M do
[7]                 add [(N, φ), PA] to Worklist and CondMayAlias
[8]     while Worklist is not empty do /* compute conditional may alias for P */
[9]         remove [(N, AA), PA] from Worklist
[10]        if N is a call node then
[11]            propagate at N; add to Worklist and CondMayAlias
[12]        elseif N is an entry node of a separately analyzed module M (with exit node X) then
[13]            foreach [(X, AA')_IC, PA'] in CondMayAlias-LinkInfo such that IC = PA do
[14]                add [(X, AA'), PA'] to CondMayAlias and Worklist
[15]            else propagate at N and add to Worklist and CondMayAlias
[16]    foreach [(N, AA), PA] in CondMayAlias where AA ≠ (V, NA) do /* compute may alias for P */
[17]        add [N, PA] to MayAlias
[18]    foreach may alias [N_IC, PA] in MayAlias-LinkInfo do
[19]        /* let E be the entry node of the module M that includes N */
[20]        if IC = φ then add [N, PA] to MayAlias
[21]        elseif PA = (V, NA)
[22]            foreach [E, (V, U)] ∈ MayAlias where U is visible in M do add [N, (V, U)] to MayAlias
[23]        elseif [(E, IC), IC] ∈ CondMayAlias add [N, PA] to MayAlias
[24]    create aliases for (NA,NA) pairs where necessary
end
```

Fig. 6. Algorithm that uses separate analysis results to compute may alias information.

an applications program. We require knowledge of both induced and noninduced may aliases, including inducement conditions for the latter. *MayAlias-LinkInfo* output by ComputeMayAlias-Module for the example program is the same as the *MayAlias-Module* information shown in Table 9.

3.2 Analysis of Applications Using Separate Analysis Results

When ComputeMayAlias-Module is used to analyze modules, we can incorporate the results of that analysis into the analysis of applications programs that call those modules using algorithm AnalyzeApplication. AnalyzeApplication, shown in Fig. 6, takes an applications program P, and returns *MayAlias*, the set of may aliases for P. After computing the ICFG for P, AnalyzeApplication proceeds like ComputeMayAlias, introducing conditional may aliases, propagating those conditional may aliases, and using the results of that

propagation to calculate may alias information for the program. However, the procedures for performing these tasks are modified to make use of the results of the separate analyses of called modules.

3.2.1 Construct the ICFG for Program P

AnalyzeApplication first constructs an ICFG for P, using reduced ICFGs that were previously computed by ComputeMayAlias-Module wherever possible. Fig. 7 depicts the ICFG that AnalyzeApplication builds for an example program that calls previously analyzed module C (initially presented in Fig. 5). We refer to this example throughout this section.

3.2.2 Compute Conditional May Alias Introductions for Program P

When AnalyzeApplication introduces conditional may aliases, it follows the same procedures at call and assignment statements as ComputeMayAlias (lines 3 and 4). For example, for the program of Fig. 7, the algorithm introduces
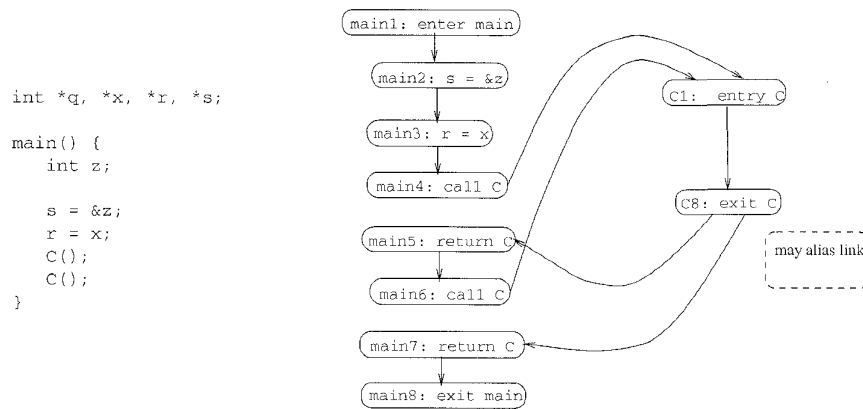
```
int *q, *x, *r, *s;

main() {
    int z;

    s = &z;
    r = x;
    C();
    C();
}
```


Fig. 7. An example program and its reduced ICFG.

conditional may alias [(main3, ϕ), (*r, *x)]. By introducing these conditional may aliases, AnalyzeApplication accounts for aliases introduced in the portion of the applications program that was not previously analyzed.

AnalyzeApplication must also account, however, for aliases that are introduced in separately analyzed modules and propagate out of those modules. Because of the separate analysis, these aliases are present, with assumed alias ϕ, in the CondMayAlias-LinkInfo for the separately analyzed module. To account for such aliases, AnalyzeApplication (lines 5-7) introduces conditional may alias [(N, ϕ), PA] for each [(N, ϕ)$_\phi$, PA] in CondMayAlias-LinkInfo. For example, in separately analyzed module C, (*q, *r) was introduced at C6 and found, when that module was analyzed, to reach the exit of the module at C8. Thus, [(C8, ϕ)$_\phi$, (*q, *r)] is in CondMayAlias-LinkInfo, and thus, AnalyzeApplication introduces conditional may alias [(C8, ϕ), (*q, *r)]. Table 11 shows the results of the conditional may alias introduction step for the example program.

TABLE 11
CONDMAYALIAS AFTER MAY ALIAS INTRODUCTIONS
FOR THE PROGRAM OF FIG. 7

CondMayAlias [(N, AA), PA]	explanation
[(main2, ϕ), (*s, z)], [(main3, ϕ), (*r, *x)]	pointer assignment
[(C8, ϕ), (*q, *r)]	exit node

3.2.3 Compute Conditional May Alias Information for Program P

When AnalyzeApplication propagates conditional may alias information, it follows the same procedures as ComputeMayAlias at all nodes other than call nodes to, and entry nodes of, separately analyzed modules. At call nodes to separately analyzed modules, AnalyzeApplication behaves exactly like ComputeMayAlias except for aliases that involve two variables that are nonaccessed in the module: ComputeMayAlias propagates these aliases directly to the associated return nodes because these aliases necessarily survive the call. At the entry node E of separately analyzed module M, for each conditional may alias [(E, AA), PA], AnalyzeApplication considers each conditional may alias [(X, AA')$_{IC}$, PA'] in CondMayAlias-LinkInfo (where X is

the exit node of M) that has inducement condition IC = PA. For each such conditional may alias AnalyzeApplication adds [(X, AA'), PA'] to Worklist and to CondMayAlias. In this way, the algorithm uses link information to account for the effects of aliases that reach the module, without reanalyzing the module. Table 12 shows the results of the conditional may alias propagation step for the example program.

3.2.4 Compute May Alias Information for Program P

To calculate may aliases, AnalyzeApplication, like ComputeMayAlias, uses the conditional may aliases present at nodes in a program P, and drops the conditional portion of each conditional may alias to obtain the associated may alias. For example, at node main3 of the example program of Fig. 7, AnalyzeApplication finds conditional may alias [(main3, ϕ), (*r, *x)], and from this, computes may alias [main3, (*r, *x)].

After calculating the aforementioned may aliases, AnalyzeApplication uses MayAlias-LinkInfo for each separately analyzed module M to determine the may aliases that hold in those modules in the context of the application. For each may alias [N_{IC}, PA] in MayAlias-LinkInfo for separately analyzed module M with entry node E, there are three cases to consider:

- If IC is ϕ, then [N_{IC}, PA] is a may alias in M independent [N, PA] to MayAlias (line 20). For example, in the program of Fig. 7, when AnalyzeApplication considers may alias [C2$_\phi$, (*p, *x)], it adds [C2, (*p, *x)] to MayAlias.
- If PA = (V, NA), and if V is in the scope of M, then every nonaccessed object U that may be aliased to V at E may be aliased to V at N. In this case (lines 21 and 22), the algorithm adds [N, (V, U)] to MayAlias for each such U.
- If neither of the preceding cases holds, [N_{IC}, PA] holds conditional on alias IC holding on entry to M, and [N_{IC}, PA] does not involve a nonaccessed object. For each such [N_{IC}, PA], AnalyzeApplication (line 23) adds it to MayAlias only if conditional may alias [(E, IC), IC] holds. For example, in the program of Fig. 7, when AnalyzeApplication considers may alias [C6$_{(*r, *x)}$, (*p, *q)], it notes that [(C1, (*r, *x)), (*r, *x)] holds, and thus, adds [C6, (*p, *q)] to MayAlias.

TABLE 12
CondMayAlias After Conditional May Alias Computation for the Program of Fig. 7

CondMayAlias [(N, AA), PA]		explanation
[(main2, φ), (*s, z)]		introduction
[(main3, φ), (*s, z)]		propagation
[(main3, φ), (*r, *x)]		introduction
[(main4, φ), (*s, z)], [(main4, φ), (*r, *x)]		propagation
[(main5, φ), (*r, *x)], [(main5, φ), (*q, *x)], [(main5, φ), (*q, *r)]		propagation
[(main6, φ), (*r, *x)], [(main6, φ), (*q, *x)], [(main6, φ), (*q, *r)]		propagation
[(main7, φ), (*r, *x)], [(main7, φ), (*q, *x)], [(main7, φ), (*q, *r)]		propagation
Tuples obtained for first call to C	Tuples obtained for second call to C	
[(C1, (*s, NV)), (*s, NV)], [(C1, (*r, *x)), (*r, *x)]	[(C1, (*q, *x)), (*q, *x)], [(C1, (*q, *r)), (*q, *r)]	propagation
[(C8, (*r, x)), (*r, *x)], [(C8, φ), (*q, *r)], [(C8, (*r, *x)), (*q, *x)]	[(C8, (*q, x)), (*q, *x)]	propagation

TABLE 13
MayAlias for the Program of Fig. 7

MayAlias[N, PA]	
main2, (*s, z)	
main3, (*s, z), main3, (*r, *x)	
main4, (*s, z), main4, (*r, *x)	
main5, (*r, *x), main5, (*q, *x), main5, (*q, *r)	
main6, (*r, *x), main6, (*q, *x), main6, (*q, *r)	
main7, (*r, *x), main7, (*q, *x), main7, (*q, *r)	
Pairs computed during first call to C	Pairs computed during second call to C
C1, (*r, *x)	C1, (*q, *x), C1, (*q, *r)
C2, (*r, *x), C2, (*p, *x), C2, (*p, *r)	C2, (*q, *x), C2, (*q, *r), C2, (*p, *q)
C3, (*r, *x), C3, (*p, *x), C3, (*p, *r)	C3, (*q, *x), C3, (*q, *r), C3, (*p, *q)
C4, (*r, *x), C4, (*p, *x), C4, (*p, *r)	C4, (*q, *x), C4, (*q, *r), C4, (*p, *q)
C5, (*r, *x), C5, (*p, *x), C5, (*p, *r)	C5, (*q, *x), C5, (*q, *r), C5, (*p, *q)
C6, (*r, *x), C6, (*p, *x), C6, (*p, *r), C6, (*q, *r), C6, (*q, *x), C6, (*p, *q)	
C7, (*r, *x), C7, (*p, *x), C7, (*p, *r), C7, (*q, *r), C7, (*q, *x), C7, (*p, *q), C7, (*s, y)	
C8, (*r, *x), C8, (*p, *x), C8, (*p, *r), C8, (*q, *r), C8, (*q, *x), C8, (*p, *q), C8, (*s, y)	
Pairs computed during first call to D	Pairs computed during second call to D
D1, (*r, *x), D1, (*b, *q), D1, (*a, *x), D1, (*a, *r)	D1, (*q, *x), D1, (*q, *r), D1, (*b, *r), D1, (*b, *x)

One final action is required, to handle aliases of the form (nonaccessed, nonaccessed). For each separately analyzed module, for each conditional may alias [(N, AA), PA] that reaches a call to that module such that PA contains two variables nonaccessed but visible in the module, AnalyzeApplication attaches may alias [N', PA] to each node in the module.

Table 13 gives the MayAlias set computed by AnalyzeApplication for the program of Fig. 7.

3.3 Computation of Aliases Introduced by Multiple Conditions

During may alias analysis, aliases may be introduced when multiple conditional may aliases reach a program point along some path. We call these aliases aliases introduced by multiple conditions. For example, we saw in the program of Fig. 2 that in the second call to C, where (*q, *x) and (*a, *x) reach the entry to D, ComputeMayAlias generates may alias pair (*a, *q) at D1. In this case, AA represents both (*q, *x) and (*a, *x). Because the cost of tracking multiple conditions is prohibitive, Landi and Ryder show that it is sufficient to use just one of these two assumed aliases. Thus, ComputeMayAlias can create either [(D1, (*q, *x)), (*a, *q)] or [(D1, (*a, *x)), (*a, *q)]; in the illustration of ComputeMayAlias in Sec-

tion 2, the algorithm generated the second conditional may alias.

Whereas ComputeMayAlias uses conditional may aliases that actually occur in a program, ComputeMayAlias-Module uses a set of assumed conditional may aliases that is a superset of those that may actually occur in a particular calling context. If ComputeMayAlias-Module handles aliases introduced by multiple conditions in the same manner as ComputeMayAlias, it may produce results that are less precise than those produced by ComputeMayAlias. For example, when ComputeMayAlias-Module analyzes the module of Fig. 5, it generates conditional may aliases $[(D1, (*a, *x))_{(*r, *x)} (*a, *x)]$ and $[(D1, (*q, *x))_{(*q, *x)}, (*q, *x)]$. In applications programs where both (*r, *x) and (*q, *x) reach C, (*a, *q) holds at D1; however, if either (*r, *x) or (*q, *x) does not reach C, then (*a, *q) does not hold at D1. If ComputeMayAlias-Module either creates, or does not create, link information that lists (*a, *q), there may be applications programs for which this information is incorrect. Thus, aliases introduced by multiple conditions require special handling.

ComputeMayAlias-Module calculates aliases introduced by multiple conditions that are triggered by pairs of conditional may aliases in which neither conditional may alias is induced, or in which inducement conditions

```
algorithm   ComputeMayAlias-Module(PrecisionLevel)
input       M : a module
output      CondMayAlias-LinkInfo: subset of CondMayAlias-Module
            MayAliasInfo-Link: subset of MayAlias-Module
            ICFG-Module: reduced ICFG for M
declare     G : an interprocedural control flow graph (ICFG) for M, with entry node E and exit node X
            Worklist : list of [(N, AA)_IC, PA], initially empty
            CondMayAlias-Module : set of [(N, (AA))_IC, (PA)]
            MayAlias-Module : set of [N_IC, (PA)]
            PASet : set of names potentially aliased in M
```

$\qquad$ **begin**
[1]   construct G, an ICFG for M /* construct the ICFG for M */
[2]   compute $PASet$ for M /* compute the PASet for M */
[3]   **foreach** PA in $PASet$ **do** /* compute conditional may alias introductions for M */
[4]   add $[(E, PA)_{PA}, PA]$ to $Worklist$ and to $CondMayAlias\text{-}Module$
[5]   **foreach** N in G **do**
[6]   **if** N is an assignment to a pointer or a call statement **then**
[7]   add conditional may aliases introduced by N to $Worklist$ and to $CondMayAlias\text{-}Module$
[8]   **while** $Worklist$ is not empty **do** /* compute conditional may alias information for M */
[9]   remove $[(N, AA)_{IC}, PA]$ from $Worklist$
[10]   propagate at N and update $Worklist$ and $CondMayAlias\text{-}Module$
[10.1*]   **if** $PrecisionLevel$ is "CMA-overestimate" **then** /* generate aliases introduced by multiple conditions */
[10.2*]   add aliases introduced by multiple conditions to $Worklist$ and $CondMayAlias - Module$ */
[10.3*]   **endif**
[11]   **foreach** $[(N, AA)_{IC}, PA]$ in $CondMayAlias\text{-}Module$ **do** /* compute may alias information for M */
[12]   add $[N_{IC}, PA]$ to $MayAlias\text{-}Module$
[13*]   **if** $PrecisionLevel$ is "CMA-precise" $ICFG\text{-}Module = G$
[13.1*]   **else** $ICFG\text{-}Module =$ node set $\{E, X\}$ and edge set $\{(E, X)\}$
[14*]   **foreach** $[(N, AA)_{IC}, PA]$ in $CondMayAlias\text{-}Module$, /* output may alias link information for M */
$\qquad\qquad$ such that $N = X$, $N = E$, or N is a pointer variable assignment node **do**
[15]   add $[(X, AA), PA]$ to $CondMayAlias\text{-}LinkInfo$
[16]   **foreach** may alias $[N_{IC}, PA]$ in $MayAlias\text{-}Module$ **do**
[17]   add $[N_{IC}, PA]$ to $MayAlias\text{-}LinkInfo$
[18]   **output** $CondMayAlias\text{-}LinkInfo$, $MayAlias\text{-}LinkInfo$, and $ICFG\text{-}Module$
$\qquad$ **end**

Fig. 8. Algorithm for computing may alias link information for a module, with three possible levels of precision. This algorithm is a modification of algorithm `ComputeMayAlias-Module` of Fig. 4; new or modified lines are marked with asterisks.

are identical, during the propagation step in lines 8 through 10; these conditional may aliases do not present a problem. However, there may be additional pairs of conditional may aliases introduced by multiple conditions in particular calling contexts that depend on induced conditional may aliases. Our technique provides three levels of analysis with respect to these aliases. These levels differ in terms of the precision of the alias information that they produce, relative to the precision of the alias information that `ComputeMayAlias` produces. In the following discussion, we differentiate these three levels of analysis by referring to them as *CMA-underestimate* analysis, *CMA-overestimate* analysis, and *CMA-precise* analysis. Using CMA-underestimate analysis, `ComputeMayAlias-Module` may omit some aliases that `ComputeMayAlias` identifies. Using CMA-overestimate analysis, `ComputeMayAlias-Module` may identify some spurious aliases that `ComputeMayAlias` does not identify. Using CMA-precise analysis, `ComputeMayAlias-Module` identifies precisely the aliases that `ComputeMayAlias` identifies.

To accommodate these three levels of analysis, we use modified versions of `ComputeMayAlias-Module` and `AnalyzeApplication` that have a precision level, *PrecisionLevel*, as a parameter. This parameter takes on one of the values "CMA-underestimate," "CMA-overestimate," or "CMA-precise." Fig. 8 and Fig. 9 show algorithms `ComputeMayAlias-Module`(*PrecisionLevel*) and `AnalyzeApplication`(*PrecisionLevel*), respectively. These algorithms are similar to `ComputeMayAlias-Module` and `Ana-`

`lyzeApplication`, respectively; in the figures, new or modified lines are marked with asterisks.

In applications where CMA-underestimate analysis is sufficient, `ComputeAliasInfo-Module`-(*Precision-Level*) simply does not introduce aliases introduced by multiple conditions. For example, if *PrecisionLevel* is "CMA-underestimate," and `ComputeAliasInfo-Module` analyzes the module of Fig. 5, the algorithm finds both $[(\texttt{D1}, (\texttt{*a}, \texttt{*x}))_{(\texttt{*r}, \texttt{*x})}, (\texttt{*a}, \texttt{*x})]$ and $[(\texttt{D1}, (\texttt{*q}, \texttt{*x}))_{(\texttt{*q}, \texttt{*x})}, (\texttt{*q}, \texttt{*x})]$, at $\texttt{D1}$, but does not introduce either $[(\texttt{D1}, (\texttt{*a}, \texttt{*x}))_{(\texttt{*r}, \texttt{*x})}, (\texttt{*a}, \texttt{*q})]$ or $[(\texttt{D1}, (\texttt{*q}, \texttt{*x}))_{(\texttt{*q}, \texttt{*x})}, (\texttt{*a}, \texttt{*q})]$. At the second call to C in `main`, inducement conditions $(\texttt{*r}, \texttt{*x})$ and $(\texttt{*q}, \texttt{*x})$ are both true. However, in this case, the algorithm misses alias pair $(\texttt{*a}, \texttt{*q})$, and computes a may alias set that is a subset of the set of may aliases computed by `ComputeMayAlias`. Thus, CMA-underestimate analysis is not safe: it may omit may aliases.

If a safe set of may aliases is required, but an overestimate is sufficient, `ComputeMayAlias-Module`-(*PrecisionLevel*) overestimates the set of conditional may aliases introduced by multiple conditions, by calculating all conditional may aliases that occur because of the existence of pairs of induced conditional may aliases in M, and adding them to *Worklist* and to *CondMayAlias–Module*. Lines 10.1, 10.2, and 10.3 in `ComputeMayAlias-Module`(*PrecisionLevel*) perform this action. In the example, when *PrecisionLevel* is "CMA-overestimate," `ComputeMayAlias-Module`(*PrecisionLevel*) introduces two new conditional may aliases: $[(\texttt{D1}, (\texttt{*a}, \texttt{*x}))_{(\texttt{*r}, \texttt{*x})}, (\texttt{*a}, \texttt{*q})]$ and $[(\texttt{D1}, (\texttt{*q}, \texttt{*x}))_{(\texttt{*q}, \texttt{*x})}, (\texttt{*a}, \texttt{*q})]$. In this case, in a

```
algorithm   AnalyzeApplication(PrecisionLevel)
input       P : a program
output      MayAlias : set of [N, PA], where PA may be aliased after N
declare     G : an interprocedural control flow graph (ICFG) for P
            Worklist : list of [(N, AA), PA]; initially empty
            CondMayAlias : set of [(N, (AA)), (PA)] for P
            CondMayAlias–LinkInfo: conditional may aliases for modules
            MayAlias–LinkInfo: alias link information for modules
begin
```
[1]   construct G, an ICFG for P
[2]   **foreach** N in G **do** /* compute conditional may alias introductions for P */
[3]   **if** N is an assignment to a pointer or a call node **then**
[4]   add conditional may aliases introduced by N to $Worklist$ and $CondMayAlias$
[5]   **if** N is an exit node of a separately analyzed module M **then**
[6]   **foreach** $[(N, \phi)_\phi, PA]$ in $CondMayAlias–LinkInfo$ for M **do**
[7]   add $[(N, \phi), PA]$ to $Worklist$ and $CondMayAlias$
[8]   **while** $Worklist$ is not empty **do** /* compute conditional may alias for P */
[9]   remove $[(N, AA), PA]$ from $Worklist$
[10]   **if** N is a call node **then**
[11]   propagate at N; add to $Worklist$ and $CondMayAlias$
[12]   **elseif** N is an entry node of a separately analyzed module M (with exit node X) **then**
[13]   **foreach** $[(X, AA')_{IC}, PA']$ in $CondMayAlias\text{-}LinkInfo$ such that $IC = PA$ **do**
[14]   add $[(X, AA'), PA']$ to $CondMayAlias$ and $Worklist$
[15]   **else** propagate at N and add to $Worklist$ and $CondMayAlias$
[15.1*] **if** $PrecisionLevel$ is "CMA-precise" **then**
[15.2*]   calculate and propagate aliases introduced by multiple conditions
[16]   **foreach** $[(N, AA), PA]$ in $CondMayAlias$ where $AA \neq (V, NA)$ **do** /* compute may alias for P */
[17]   add $[N, PA]$ to $MayAlias$
[18]   **foreach** may alias $[N_{IC}, PA]$ in $MayAlias\text{-}LinkInfo$ **do**
[19]   /* let E be the entry node of the module M that includes N */
[20]   **if** $IC = \phi$ **then** add $[N, PA]$ to $MayAlias$
[21]   **elseif** $PA = (V, NA)$
[22]   **foreach** $[E, (V, U)] \in MayAlias$ where U is visible in M **do** add $[N, (V, U)]$ to $MayAlias$
[23]   **elseif** $[(E, IC), IC] \in CondMayAlias$ add $[N, PA]$ to $MayAlias$
[24]   create aliases for (NA,NA) pairs where necessary
```
end
```

Fig. 9. Algorithm that uses separate analysis results to compute may alias information for an applications program, with three possible levels of precision. This algorithm is a modification of algorithm AnalyzeApplication of Fig. 6; new or modified lines are marked with asterisks.

calling context in which (*r, *x) and (*q, *x) do not both hold on entry to C, spurious may aliases may be identified. For example, if there were only one call to C in main, (*a, *q) would be a spurious may alias. Thus, CMA-overestimate analysis may yield results that are less precise than those calculated by ComputeMayAlias.

To obtain sets of conditional may aliases and may aliases that are identical to those computed by ComputeMayAlias, ComputeMayAlias-Module(PrecisionLevel) performs CMA-precise analysis. To do this, the algorithm postpones consideration of aliases introduced by multiple conditions (for those that depend on different inducement conditions) until M is linked with a calling program by AnalyzeApplication. In this case, lines 10.1 through 10.3 in ComputeMayAlias-Module(PrecisionLevel) are not executed; aliases that would otherwise be generated by these steps are not generated. Furthermore, the graph computed by ComputeMayAlias-Module for M is G, the entire ICFG for M⁴; lines 13 and 13.1 of the algorithm handle these actions. Finally, ComputeMayAlias-Module also includes, in CondMayAlias–LinkInfo, conditional may alias information for entry nodes and pointer variable assignment nodes (line 14). For exam-

ple, for module C and applications program main shown in Fig. 10, CondMayAlias–Link information is saved for only the nodes that are shaded.

When called with a PrecisionLevel other than "CMA-precise," AnalyzeApplication(PrecisionLevel) behaves exactly like AnalyzeApplication. When called with a PrecisionLevel of "CMA-precise," however, the algorithm performs additional actions. In this case, AnalyzeApplication(PrecisionLevel) inspects each entry or pointer assignment node N, and each $[(N, AA)_{IC}, PA]$ at N, to see if inducement condition IC is met in this calling environment. If there is a conditional may alias $[(entry(M), AA')_{IC'}, PA']$ such that $IC = PA'$, then the inducement condition IC that was used in the partial analysis now holds on some path to entry(M) in the calling environment; this implies that conditional may alias $[(N, AA)_{IC}, PA]$ can be rendered as conditional may alias $[(N, AA), PA]$, and added to CondMayAlias. The next step is to consider this new conditional may alias with other conditional may aliases at N, to see if any new aliases are introduced by multiple conditions. If so, these new aliases are added to CondMayAlias and to Worklist for further propagation. These actions produce the same analysis information as ComputeMayAlias because only those aliases that are introduced by multiple conditions that arise in the applications program are generated and propagated. How-

```
int *q, *r, *s, *x;

main() {          C() {              D(a, b) {
    int z;            int *p, y;         int *a, *b;

    s = &z;           p = x;             *a = *a+*b+*q
    r = x;            D(p, q);       }
    C();             if (*q > 0)
    C();                 q = r;
}                     s = &y;
                  }
```

main1: enter main
main2: s = &z
main3: r = x
main4: call C
main5: return C
main6: call C
main7: return C
main8: exit main

C1: enter C
C2: p = x
C3: call D
C4: return D
C5: *q > 0
C6: q = r
C7: s = &y
C8: exit C

D1: enter D
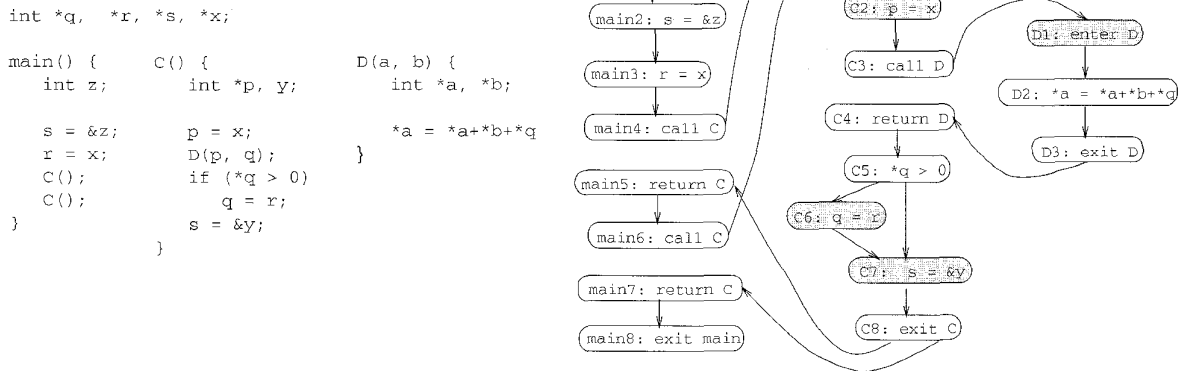D2: *a = *a+*b+*q
D3: exit D

Fig. 10. A module and its ICFG. To produce precise nodes with respect to aliases introduced by multiple conditions, ComputeMayAlias(*PrecisionLevel*) outputs *CondMayAlias–Module* information for the nodes that are shaded.

ever, we only need to propagate aliases introduced by multiple conditions because other previously computed alias information remains valid.

Alias pairs of the form (nonaccessible, nonaccessible) also may produce aliases induced by multiple conditions; these pairs are handled similarly.

To illustrate the actions of AnalyzeApplication (*PrecisionLevel*) when *PrecisionLevel* is "CMA-Precise," consider conditional may aliases [(D1, (*q, *x)){*q, *x} (*q, *x)], [(D1, (*a, *x)){*a, *x} (*a, *x)], and [(D1, (*b, *x)){*q, *x} (*b, *x)], that are saved by ComputeMayAlias-Module(*PrecisionLevel*) for entry node D1. On the first call to C in main4, (*r, *x) reaches the entry to C, and subsequently the call to D at C3. Alias pair (*p, *x) also reaches the call to D at C3. Because p is bound to a at the call, (*p, *a) is introduced at the call. Both (*p, *a) and (*p, *x) hold on entry to D, so AnalyzeApplication(*PrecisionLevel*) creates [(D1, (*a, *x)), (*a, *x)] and adds it to *CondMayAlias* and to *Worklist*. Because (*q, *r), created in C6, reaches the return from C at C8, and then reaches the second call to C in main6, it subsequently reaches D1. When AnalyzeApplication(*PrecisionLevel*) process-es this conditional may alias, it creates [(D1, (*q, *x)), (*q, *x)]. Then, in line 15.2, AnalyzeApplication(*PrecisionLevel*) notes the existence of these conditional may aliases, creates either [(D1, (*q, *x)), (*a, *q)] or [(D1, (*a, *x)), (*a, *q)], and adds it to *CondMayAlias* and to *Worklist* for later processing. Similarly, when the algorithm finds that both (*a, *x) and (*q, *x) reach D1, it creates either [(D1, (*a, *x)), (*a, *b)] or [(D1, (*q, *x)), (*a, *b)], and adds it to *CondMayAlias* and to *Worklist*.

3.4 Hybrid Analysis of Modules

We presented versions of our separate analysis and link algorithms for modules that were analyzed and then linked with complete programs. However, our technique can also be used for modules that are analyzed and then linked with other modules; this enables incremental analysis of a large system. Suppose we wish to perform may alias analysis

separately on each of procedures C and D of Fig. 5; Fig. 11 illustrates this situation. In this case, we first analyze module D, and compute may alias link information for it, using ComputeMayAlias-Module. Next, we wish to analyze C; however, we must do this in a manner that both makes use of alias link information for D, and outputs link information for C. To provide separate analysis of C in this case, we use a hybrid algorithm that incorporates actions from both ComputeMayAlias-Module and AnalyzeApplication. We call this algorithm ComputeMayAlias-Hybrid, and present it in Fig. 12.

ComputeMayAlias-Hybrid takes a module R, and may alias link information for a previously analyzed module M that is called by R.[5] The algorithm first constructs an ICFG for module R using the reduced ICFG for module M, in the same manner as AnalyzeApplication. The algorithm then constructs the *PASet* for R by considering the potential alias pairs that can reach the entry to R. Like ComputeMayAlias-Module, the algorithm next creates conditional may alias introductions at pointer assignment and call nodes in R; however, like AnalyzeApplication, the algorithm also introduces conditional may aliases at the exit node of the *ICFG-Module* for M.

Next, ComputeMayAlias-Hybrid, like Compute May-Alias-Module, uses a worklist to compute conditional may aliases; however, a conditional may alias computed to hold in M inherits the inducement condition from the conditional may alias in R that causes it to exist. Finally, ComputeMayAlias-Hybrid computes may alias link information for R, using a combination of tactics from AnalyzeApplication and ComputeMayAlias-Module; this preserves information on inducement conditions, and stores link information for later use. The may alias link information output by ComputeMayAlias-Hybrid may be used by AnalyzeApplication, or again by ComputeMayAlias-Hybrid, to analyze a program or module that calls R.

---

5. For simplicity, we present the algorithm for the case where R calls only one previously analyzed module M; however, the approach can be extended to handle multiple previously analyzed modules.
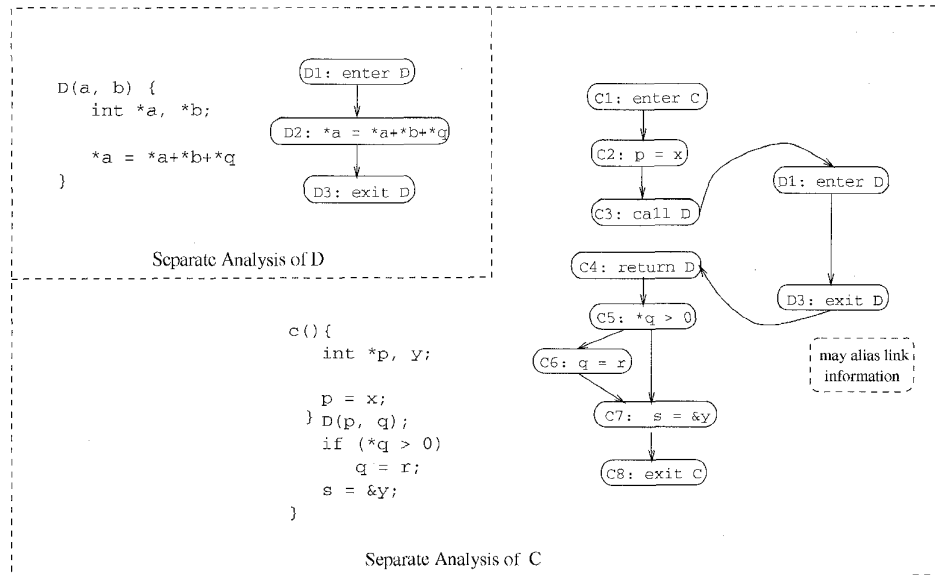
Fig. 11. An example showing analysis steps using the hybrid algorithm.

```
algorithm    ComputeMayAlias-Hybrid
input        alias link information for previously analyzed module M
             R : a module that calls M
output       CondMayAlias-LinkInfo: subset of CondMayAlias-Module
             MayAliasInfo-Link: subset of MayAlias-Module
             ICFG-Module: reduced ICFG for R
declare      G : an interprocedural control flow graph (ICFG) for M, with entry node E and exit node X
             Worklist : list of [(N, AA)_IC, PA], initially empty
             CondMayAlias-Module : set of [(N, (AA))_IC, (PA)]
             MayAlias-Module : set of [N_IC, (PA)]
             PASet : set of names potentially aliased in M
begin
[1]    construct G, an ICFG for R using ICFG-Module for M /* construct the ICFG for R */
[2]    compute PASet for R /* compute the PASet for M */
[3]    foreach PA in PASet do /* compute conditional may alias introductions for R */
[4]        add [(E, PA)_PA, PA] to Worklist and CondMayAlias-Module
[5]    foreach N in G do
[6]        if N is an assignment to a pointer or a call statement then
[7]            add conditional may aliases introduced by N to Worklist and CondMayAlias-Module
[8]        if N is an exit node of separately analyzed module M then
[9]            foreach [(N, φ)_IC, PA] in CondMayAlias-LinkInfo for M do
[10]               add [(N, φ), PA] to Worklist and CondMayAlias
[11]   while Worklist is not empty do /* compute conditional may alias information for M */
[12]       remove [(N, AA)_IC, PA] from Worklist
[13]       if N is a call node then
[14]           propagate at N and update Worklist and CondMayAlias-Module
[15]       elseif N is an entry node of M (with exit node X) then
[16]           foreach [(X, AA')_IC', PA'] in CondMayAlias-LinkInfo such that IC' = PA do
[17]               add [(X, AA')_IC, PA'] to CondMayAlias and Worklist
[18]       else propagate at N and add to Worklist and CondMayAlias
[19]   foreach [(N, AA)_IC, PA] in CondMayAlias-Module where AA ≠ (V, NA) do
[20]       add [N_IC, PA] to MayAlias-Module
[21]   ICFG-Module = node set {E, X} and edge set {(E, X)}
[22]   foreach may alias [N_IC, PA] in MayAlias-LinkInfo do
[23]       /* let E and X be the entry and exit nodes, respectively, of M */
[24]       if IC = φ then add [N, PA] to MayAlias-Module
[25]       elseif PA = (V, NA)
[26]           foreach [E, (V, U)] ∈ MayAlias-Module where U is visible in M do
[27]               add [N, (V, U)] to MayAlias-Module
[28]       elseif [(E, IC), IC] ∈ CondMayAlias-Module add [N, PA] to MayAlias-Module
[29]   foreach [(X, AA)_IC, PA] in CondMayAlias-Module do /* output may alias link information for M */
[30]       add [(X, AA)_IC, PA] to CondMayAlias-LinkInfo
[31]   foreach may alias [N_IC, PA] in MayAlias-Module do
[32]       add [N_IC, PA] to MayAlias-LinkInfo
[33]   output CondMayAlias-LinkInfo, MayAlias-LinkInfo, and ICFG-Module
end
```

Fig. 12. Algorithm for computing may alias link information for a module that calls another separately analyzed module.

3.5 Complexity

Landi and Ryder's `ComputeMayAlias` algorithm runs in time $O(n^3)$ for ICFGs of size n; thus, the algorithm is polynomial in program size [8]. Preliminary experimentation with `ComputeMayAlias` suggests, however, that in practice the algorithm runs in time linear in the size of the may alias solution [8].

`ComputeMayAlias` propagates only aliases that actually occur in a program. `ComputeMayAlias-Module`, in contrast, propagates all aliases in *PASet*. For modules that reference n global variables and parameters, *PASet* has size $(n * (n + 1))/2$; thus, in the worst case, the size of *PASet* is quadratic in the number of names in the program. However, the worst-case runtime analysis of `ComputeMayAlias` assumes that the number of aliases that occur in the program is quadratic in the number of names in the program; thus, the upper bound on the worst-case runtime of `ComputeMayAlias-Module` is the same as that for `ComputeMayAlias`. In practice, since `ComputeMayAlias-Module` propagates all aliases in *PASet*, the size of the may alias solution computed by `ComputeMayAlias-Module` for module M may exceed the size of the solution computed by `ComputeMayAlias` for module M.

Because design principles for reusable modules, such as ADTs, classes, and library routines, discourage the indiscriminate use of global variables, we expect that for most reusable modules, the number of global variables will be small. Furthermore, reusable modules designed with low coupling will have few parameters; thus, we expect that the number of parameters for well-designed reusable modules will be relatively small. For such modules, *PASet* size is small, facilitating separate analysis. We also expect that for certain well-designed systems, some submodules will have small *PASets*. These submodules can be candidates for separate analysis, facilitating incremental analysis of the modules from which they are called.

To demonstrate that our expectations for *PASet* size are reasonable, we analyzed a number of software modules to determine the sizes of the *PASets* for those modules. The results of our analyses are shown in Tables 14, 15, and 16.

Table 14 shows the results of our analysis of four ADT modules. The STACK, SET, and QUEUE modules are ADTs written in C, provided with the `Aristotle` program analysis system [6]. The STRING CLASS is an ADT written in C++, provided with a commercial compiler. Most of the modules in the ADTs make no use of global variables; those that use globals use at most two. No module uses more than three parameters. In the worst case, for these ADT modules, the size of the *PASet* is 10: small enough for our separate analysis technique to be practical.

Table 15 shows the results of our analysis for four library modules. The first three modules are part of the `Aristotle` program analysis system: the first is a library of hash functions, the second provides a set of routines that access a database, and the third provides a set of routines that insert probes into a program to trace the program's execution. The last library module is a mathematical function contained in a library of such functions obtained from Siemens corporation. Like the ADT modules, these library modules use global variables sparsely and use few pa-

rameters; the *PASet* for the modules contains at most 10 members. For such modules, *PASet* size is manageable.

TABLE 14
ANALYSIS RESULTS SHOWING PASET SIZES FOR ADT MODULES

Module Name	Number of Functions	Lines of Code	Number of Params	Number of Globals	Size of PASet
A STACK ADT:					
free_stack	1	10	1	0	1
init_stack	1	19	1	0	1
pop_stack	1	13	1	0	1
push_stack	1	11	2	0	3
empty_stack	1	9	1	0	1
A SET ADT:					
add_to_set	1	11	2	0	3
compare	1	33	2	0	3
copy_set	1	11	2	0	3
duplicate	1	16	1	0	1
intersection	1	33	3	0	6
is_in_set	1	11	2	0	3
is_subset	3	29	2	0	3
make_array_of_sets	1	24	2	0	3
make_set	1	19	1	0	1
print_set	1	15	1	0	1
remove_from_set	1	11	2	0	3
set_clear	1	21	1	0	1
set_diff	4	37	3	0	6
set_free	1	9	1	0	1
set_free_array	1	13	2	0	3
set_is_empty	1	13	1	0	1
set_union	1	12	3	0	6
A QUEUE ADT:					
dequeue	1	17	1	0	1
init_queue	1	15	0	0	0
release_queue	3	26	1	0	1
enqueue	1	26	2	0	3
is_empty	1	10	1	0	1
print_queue	1	16	1	0	1
STRING CLASS IN C++:					
string	1	9	1	2	6
isequal	1	5	2	2	10
isLT	1	4	2	1	6
hash	1	10	0	2	3
print	1	4	2	1	6
concat	1	14	2	2	10
len	1	4	0	2	3

TABLE 15
ANALYSIS RESULTS THAT SHOW PASET SIZES FOR LIBRARY MODULES

Module Name	Number of Functions	Lines of Code	Number of Params	Number of Globals	Size of PASet
A HASH FUNCTION LIBRARY:					
hash	2	52	3	1	10
hash_insert	4	106	3	1	10
hash_remove	3	85	2	1	6
hash_search	3	77	2	1	6
hash_table_init	1	39	1	1	3
THE CF HANDLER LIBRARY:					
cfbegin	1	32	1	0	1
cfcreate	1	40	1	0	1
cfend	1	28	1	0	1
cffree	1	58	1	0	1
cfgetedgelist	1	56	2	0	3
cfgetedgenumber	1	43	4	0	10
cfread	1	156	3	0	6
cfwrite	1	56	3	0	6
THE BRANCH TRACE FUNCTION LIBRARY:					
IPF_bt_SetBranch	1	24	4	0	10
IPF_bt_SetSWBranch	1	20	2	0	3
IPF_bt_main_Init	1	80	4	0	10
IPF_bt_proc_Init	1	29	4	0	10
IPF_bt_Term	1	42	0	0	0
IPF_bt_TestEdge	1	10	2	0	3
A MATHEMATICAL FUNCTION:					
QGamma	5	99	2	0	3

Finally, Table 16 shows the results of our analyses for three sets of software modules that either are reusable, or might be analyzed separately in order to incrementally analyze the systems in which they are contained. The first set of modules is a set of routines from the `Aristotle` program analysis system that implements computation of reverse control flow graphs, dominator trees [1], control

dependencies, and control dependence graphs [5]. These program analysis modules were designed to be reusable; they do not use global variables, and they have few parameters. The maximum size of a *PASet* for these modules is six. The modules are good candidates for separate analysis.

TABLE 16
ANALYSIS RESULTS SHOWING PASET SIZES
FOR POTENTIALLY REUSABLE MODULES

Module Name	Number of Functions	Lines of Code	Number of Params	Number of Globals	Size of PASet
PROGRAM ANALYSIS MODULES:					
cfg-reverse	2	154	2	0	3
dom-tree-construct	15	901	3	0	6
cdep-calculate	13	504	3	0	6
fow-build-cdg	37	3565	2	0	3
GAME MODULES:					
move	72	1760	9	34	946
fly	86	2010	5	23	406
CALCULATOR PROGRAM MODULES:					
calculator main menu	28	903	0	16	136
main event loop	58	1430	0	23	276

The second set of modules that are described in Table 16 contains two modules from an Internet-based game. These modules are reusable, and are called from multiple locations in the game software. However, the code for the game has evolved over several years, at the hands of numerous, independent coders who, presumably, did not make code reuse a priority. The modules make heavy use of global variables. *PASet* size for these modules is large; thus, these modules may not be favorable candidates for separate analysis.

The third set of modules that are described in Table 16 are submodules in a calculator whose source code is provided with a commercial compiler. These modules are not designed for reuse; however, they could, at first glance at the call graph for the system, be considered candidates for separate analysis if we wished to incrementally analyze the system. Unfortunately, the modules are coupled to the rest of the system strictly through global variables, and use many such variables. Thus, the *PASets* for the modules are large, and the modules may not be favorable candidates for separate analysis.

`AnalyzeApplication`, like `ComputeMayAlias`, propagates only aliases that actually occur in a program; thus, steps 1-17 of `AnalyzeApplication`, which compute the may alias solution for an application program P, run in time polynomial in the size of the (possibly reduced) ICFG for P. Lines 18-22 of `AnalyzeApplication` consider each may alias in *MayAlias-LinkInfo*. In the worst case, for each may alias, these lines consider each member of *CondMayAlias*. Because *MayAlias-LinkInfo* and *CondMay-Alias* have size polynomial in the size of *ICFG-Module*, the work done by `AnalyzeApplication` for these lines is polynomial in the size of *ICFG-Module*. An efficient implementation, in which *CondMayAlias* entries are organized in terms of inducement conditions, may yield a lower run time in practice.

3.6 Relation to Previous Work

Marlowe and Ryder [12] present a hybrid algorithm for data flow analysis that decomposes the control flow graph of a program into regions. Their algorithm first solves data flow problems within regions separately. Then, the algorithm propagates local data flow information throughout a condensed graph of the program, which consists of regions and their connections. The key to the solution of the local data flow problem within a region is to solve this problem using virtual data flow information to represent data flow information that is external to the region. Then, during propagation of data flow information throughout the condensed graph, no further propagation is required within regions—only the virtual data flow information must be updated. Marlowe and Ryder [13] extend their hybrid approach to handle aliases for Fortran programs. In this work, they introduce the idea of using one representative global variable to stand for any global variable aliased to a formal parameter at the entry node.

Landi and Ryder [9] present an algorithm to compute may alias information in the presence of pointer variables that uses alias assumptions at the entry to a function to compute this may alias information. A subsequent algorithm [10] uses a worklist approach that, instead of considering all alias assumptions at the entry to a function, computes the solution to the may alias problem for only the alias pairs that actually reach the function.

Our separate analysis algorithms are similar to the above work. First, like Marlowe and Ryder's, we compute data flow information for modules separately, and use representative data flow information to facilitate this computation. Then, when we compute data flow information for a module that calls a previously analyzed module, we need only update this precomputed information; we avoid complete reanalysis of the called module. However, instead of the global data flow problems and the Fortran aliasing problem that Marlowe and Ryder's hybrid algorithms solve, our separate analysis algorithms solve the may alias problem.

Second, like Landi and Ryder's initial may alias algorithm, our algorithm computes the effects of aliases that could reach a module in all possible contexts, by assuming aliases at the entry to a module. However, we put all these possible aliases on a worklist and propagate them using an approach similar to that of Landi and Ryder's subsequent algorithm. By using the features of these two approaches, and adding the concept of inducement conditions, we are able to compute may alias information for modules separately, in a manner that lets us reuse that information during the analysis of applications programs that use those modules.

4 CONCLUSIONS AND FUTURE WORK

We have presented a technique that permits separate analysis of a module M, and supports reuse of analysis information when analyzing a program, or another module, that calls M. We described our algorithms for the interprocedural may alias problem, for languages with reference parameters and pointers. However, a similar technique can be applied to the separate computation of other interprocedural analysis information such as reaching definitions. The main benefits of our approach are that it can amortize the cost of module analysis over all programs that use the

module, and facilitate incremental computation of may alias information for larger systems.

To demonstrate the practicality of our algorithms, we analyzed several ADT modules, library modules, and potentially reusable or separately analyzable modules to determine the sizes of the *PASet*—which is the dominant factor in the expense of our algorithms—for those modules. We found that in most cases, for well designed modules, the *PASet* is small. In several cases, where the modules were not designed for reusability, we found large *PASet*s.

We are implementing several tools for experimentation and future research. The first tool is a prototype implementation of our separate analysis and link algorithms. With this prototype, we will experiment with the practicality of our approach, and determine situations in which each of the three levels of precision is applicable. The second tool will let us automatically compute the size of a module's *PASet*. We will use our experiments to develop metrics to guide both the selection of appropriate algorithms for interprocedural analysis, and the design of modules that are amenable to separate analysis.

ACKNOWLEDGMENTS

REFERENCES

[1]   A.V. Aho, R. Sethi, and J.D. Ullman, *Compilers, Principles, Techniques, and Tools.* Reading, Mass.: Addison-Wesley, 1986.
[2]   D.R. Chase, M. Wegman, and F.K. Zadeck, "Analysis of Pointers and Structures," *Proc. ACM SIGPLAN '90 Conf. Programming Language, Design, and Implementation,* pp. 296-310, June 1990.
[3]   J.-D. Choi, M. Burke, and P. Carini, "Efficient Flow-Sensitive Interprocedural Computation of Pointer-Induced Aliases and Side Effects," *Proc. 20th Ann. ACM Symp. Principles of Programming Languages,* pp. 232-245, Jan. 1993.
[4]   M. Emami, R. Ghiya, and L.J. Hendren, "Context-Sensitive Interprocedural Points-To Analysis in the Presence of Function Pointers," *Proc. ACM SIGPLAN '94 Conf. Programming Language, Design, and Implementation,* pp. 242-256, June 1994.
[5]   J. Ferrante, K.J. Ottenstein, and J.D. Warren, "The Program Dependence Graph and its Use in Optimization," *ACM Trans. Programming Languages and Systems,* vol. 9, no. 3, pp. 319-349, July 1987.
[6]   M.J. Harrold, L. Larsen, J. Lloyd, D. Nedved, M. Page, G. Rothermel, M. Singh, and M. Smith, "Aristotle: A System for the Development of Program-Analysis-Based Tools," *Proc. 33rd Ann. Southeast Conf.,* pp. 110-119, ACM Press, Mar. 1995.
[7]   M.J. Harrold and G. Rothermel, "Separate Computation of Alias Information for Reuse," *Proc. ACM Int'l Symp. Software Testing and Analysis,* pp. 107-120, Jan. 1996.
[8]   W. Landi, *Interprocedural Aliasing in the Presence of Pointers,* PhD thesis, Rutgers Univ., 1992.
[9]   W. Landi and B.G. Ryder, "Pointer-Induced Aliasing: A Problem Classification," *Proc. 17th Ann. ACM Symp. Principles of Programming Languages,* pp. 93-103, Jan. 1990.
[10] W. Landi and B.G. Ryder, "A Safe Approximate Algorithm for Interprocedural Pointer Aliasing," *Proc. ACM SIGPLAN '92 Conf. Programming Language, Design, and Implementation,* pp. 235-248, June 1992.
[11] J.R. Larus and P.N. Hilfinger, "Detecting Conflicts Between Structure Accesses," *Proc. ACM SIGPLAN '88 Conf. Programming Language, Design, and Implementation,* pp. 21-34, June 1988.
[12] T.J. Marlowe and B.G. Ryder, "An Efficient Hybrid Algorithm for Incremental Data Flow Analysis," *Conf. Record 17th Ann. ACM Symp. Principles of Programming Languages,* pp. 184-196, Jan. 1990.
[13] T.J. Marlowe and B.G. Ryder, "Hybrid Incremental Alias Analysis," *Proc. 24th Hawaii Int'l Conf. System Sciences,* pp. 428-437, Jan. 1991.
[14] G. Murphy, D. Notkin, and E. Lan, "An Empirical Study of Static Call Graph Extractors," *Proc. 18th Int'l Conf. Software Engineering,* pp. 90-99, Mar. 1996.

Mary Jean Harrold received PhD and MS degrees in computer science from the University of Pittsburgh, and MA and BA degrees in mathematics from Marshall University. She is currently an assistant professor in the Department of Computer and Information Science at Ohio State University.

Dr. Harrold's research interests include program analysis and testing, testing of object-oriented software, and maintenance and testing environments. She is a recipient of the National Science Foundation's National Young Investigator award. She is a member of the IEEE Computer Society and the ACM.

Gregg Rothermel received a PhD in computer science from Clemson University, an MS in computer science from the State University of New York at Albany, and a BA in philosophy from Reed College. He is currently a senior research associate at Ohio State University. Previous positions included vice president for quality assurance and quality control at Palette Systems Inc.

Dr. Rothermel's research interests include software engineering and program analysis, with an emphasis on the application of program analysis techniques to problems in software maintenance and testing. He is a member of the IEEE Computer Society and the ACM.