RC 21055 (12/15/97)
Computer Science

IBM Research Report

Interprocedural Pointer Alias Analysis

Michael Burke, Paul R. Carini, Jong-Deok Choi

IBM Research Division
T.J. Watson Research Center
Yorktown Heights, New York

Michael Hind

Department of Mathematics and Computer Science
State University of New York at New Paltz
New Paltz, NY 12561
and IBM T.J. Watson Research Center

IBM Research Division
Almaden · T.J. Watson · Tokyo · Zurich

Interprocedural Pointer Alias Analysis

MICHAEL BURKE, PAUL CARINI and JONG-DEOK CHOI
IBM Thomas J. Watson Research Center
and
MICHAEL HIND
State University of New York at New Paltz and IBM Thomas J. Watson Research

---

We present practical approximation methods for computing and representing interprocedural aliases for a program written in a language that includes pointers, reference parameters, and recursion. We present a general framework for interprocedural pointer alias analysis, and the following: 1) a *flow-sensitive* interprocedural pointer alias analysis algorithm; 2) a *flow-insensitive* interprocedural pointer alias analysis algorithm that can incorporate *kill* information to improve precision, and use *deferred evaluation* to increase efficiency; 3) an algorithm for function pointer analysis that can be incorporated into the general framework; 4) a *compact representation* of alias information that is based on *named objects* rather than pointer expressions and accommodates cyclic data structures without resorting to a k-limiting scheme to limit the level of indirection; 5) techniques that improve the precision of alias analysis with respect to the *unrealizable path* problem and the *non-distributive combining* problem; 6) a technique for improving the precision of alias information for compact and graph-based representations; 7) an interprocedural naming technique for dynamically allocated objects that improves the precision of alias analysis for such objects; and 8) empirical measurements of the efficiency and precision of the three interprocedural alias analysis algorithms: the flow-sensitive, flow-insensitive, and flow-insensitive with precomputed kill algorithms.

This work was supported in part by the National Science Foundation under grant CCR-9633010, by IBM Research Contracts, and by a SUNY at New Paltz Research and Creative Project Award.
Authors' address: M. Burke, P. Carini, J.-D. Choi, IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598. email: {burkem,carini,jdchoi}@watson.ibm.com; M. Hind, Department of Mathematics and Computer Science, State University of New York at New Paltz, 75 South Manheim Blvd, Suite 6, New Paltz, New York 12561 and IBM Thomas J. Watson Research Center. email: hind@mcs.newpaltz.edu

Contents

1. INTRODUCTION

Data flow analysis computes information about the potential behavior of a program in terms of the definitions and uses of data objects. Such data flow information is important for compilers and programming environment tools [Allen et al. 1987; Choi and Ferrante 1994; Kennedy et al. 1991]. It can also provide compiler and run-time support for the parallel execution of programs originally written in sequential languages [Allen et al. 1987; Allen et al. 1987; Larus 1989; Gannon et al. 1990].

Numerous techniques have been successfully developed for data flow analysis of programs written in languages with only static data structures, such as FORTRAN. However, data flow analysis for programs written in languages with dynamically allocated data structures, such as C, C++, Fortran 90, Java, and LISP, has been less successful because of *pointer-induced aliasing*, which occurs when two or more pointer expressions refer to the same storage location.

Data flow analysis algorithms can be classified into two categories: *flow-sensitive* and *flow-insensitive* [Banning 1979; Marlowe et al. 1995]. Flow-sensitive algorithms consider intraprocedural control flow information during the analysis and, in general, are more precise than flow-insensitive algorithms. Flow-insensitive algorithms do not make use of intraprocedural control flow information during the analysis. As such, they can be more efficient than flow-sensitive algorithms and have been used primarily for the class of problems for which flow-sensitive algorithms do not provide increased precision. There exists, however, a range of trade-offs between efficiency and precision. Flow-insensitive analysis can also be used to improve efficiency, at the potential cost of precision, for the class of problems for which flow-sensitive analysis can yield better precision. Pointer-induced aliasing is a problem in this class.

The problem of exactly computing aliases in the presence of general pointers is known to be undecidable [Landi 1992; Ramalingam 1994]. In this paper, we describe approximation methods for computing interprocedural aliases for a program written in a language that includes pointers, reference parameters, and recursion. We present a general framework for interprocedural pointer alias analysis, and the following:

—a *flow-sensitive* interprocedural pointer alias analysis algorithm;

—a *flow-insensitive* interprocedural pointer alias analysis algorithm that can incorporate *kill* information to improve precision, and use *deferred evaluation* to increase efficiency;

—an algorithm for constructing the *program call graph* that accommodates function pointers while alias analysis is being performed and can be used with our interprocedural alias analyses;

—a *compact representation* of alias information that is based on *named objects* rather than pointer expressions and accommodates cyclic data structures without resorting to a k-limiting scheme to limit the level of indirection;

—Techniques that improve the precision of alias analysis with respect to the *unrealizable path* problem and the *non-distributive combining* problem;

—A technique for improving the precision of alias information for compact alias representations [Choi et al. 1993; Emami et al. 1994] and graph-based representations [Jones and Muchnick 1981; 1982; Larus and Hilfinger 1988; Ruggieri and Murtagh 1988; Horwitz et al. 1989; Chase et al. 1990].

—An interprocedural naming technique for dynamically allocated objects which improves the precision of alias analysis for such objects.

—Empirical measurements of the efficiency and precision of three interprocedural alias analysis algorithms: the flow-sensitive, flow-insensitive, and flow-insensitive with precomputed kill algorithms.

The rest of the paper is organized as follows. Section 2 defines terminology used in the paper and describes the background context. Section 3 presents a compact representation of alias information that enhances the space efficiency of alias analysis methods and naturally accommodates cyclic data structures. Section 4 presents our flow-sensitive and flow-insensitive interprocedural algorithms. Section 5 describes how to accommodate function pointers in our framework for interprocedural pointer alias analysis. Section 6 discusses issues pertaining to the precision of alias information during interprocedural and intraprocedural propagation. Section 7 discusses the correct treatment of parameters in alias analysis. Section 8 describes a technique for naming dynamically allocated memory locations and approaches for distinguishing different instances of named objects. The complexity of the alias analysis algorithms is presented in Section 9. Empirical results that measure and contrast the efficiency and precision of the three interprocedural pointer alias analysis algorithms described in Section 4 are provided in Section 10. A comparison of our method with relevant work is given in Section 11. We draw conclusions in Section 12.

2. BACKGROUND

Aliasing occurs when there exists more than one *access path* [Larus and Hilfinger 1988] to a storage location. An access path is the l-value of an expression that is constructed from variables, pointer indirection operators, and structure field selection operators. In C these expressions would include variables, the '$*$' dereference operator, the '.' field selection operator, and a combination of both operators: '$\rightarrow$'. Two access paths are *must-aliases* at a statement S if they refer to the same storage location in all execution instances of S. Two access paths are *may-aliases* at S if they refer to the same storage location in some execution instances of S. This paper addresses the computation of may-aliases, of which must-aliases are a subset. We will refer to may-aliases as aliases, whenever the meaning is clear from context.

Structural aliases result from program constructs such as C's `union` construct or FORTRAN's `EQUIVALENCE` statement. These aliases do not change within a program and can be computed from semantic information. We assume that the structural alias information is available and that our algorithms use this information as input.

We use $\langle x, y\rangle$ to denote that access paths x and y are may-aliases at a program point.[1] For example, consider the following statement:

$$p = \&r;$$

After this statement is executed, $*p$ and r refer to the same storage location and

[1] $\langle x, y\rangle \equiv \langle y, x\rangle$; i.e., it is an unordered pair of access paths.

thus become aliases of each other, which we express as the *alias relation* $\langle *p, r \rangle$. Note that the *trivial* alias $\langle x, x \rangle$ always holds for all *non-null* access paths x at all program points. Also note that, if x and y are non-null pointer-valued access paths, $\langle x, y \rangle$ *implies* $\langle *x, *y \rangle$. We refer to these aliases as *implied* aliases. An *implicit* alias is one that is either trivial or implied.

Interprocedural data flow analyses make use of the *program call graph* (PCG), which is a flow multigraph for which each procedure is represented by a single node and each call site by a unique edge.[2] The edge (f, g) represents a call site in function f that invokes g.

For pointer-induced aliasing, unlike reference-parameter aliasing, not only can the calling procedure affect the aliasing of the called procedure, but the called procedure can also affect the aliasing of the calling procedure.

The conventional methods for side-effect analysis [Banning 1979; Cooper and Kennedy 1988; Burke 1990] decompose the computation into separate direct side-effect and alias analyses. Aliasing information is factored into direct side effects after the latter has been computed. In the presence of pointers, it can be incorrect to perform side-effect analysis without using alias information because pointer aliasing information, unlike reference parameter aliasing, can change within a routine. In order to correctly perform side-effect analysis in the presence of pointers, alias information must be present and factored in when immediate MOD and USE information is collected [Choi et al. 1993; Horwitz et al. 1989; Landi et al. 1993]. During alias analysis we compute the side effects of a procedure using alias information by introducing a *representative parameter* r_i for each variable local to the caller whose address is passed as the ith actual parameter to a pointer in the called procedure [Choi et al. 1993].

We describe various alias analysis techniques for common imperative pointer languages such as C, C++, and Fortran 90. Our analysis supports such language features as the address operator, pointer assignment, pointer indirection, recursive calls, calls through function pointers, and dynamic memory allocation. A reference to an element of an aggregate data structure (such as a `struct` or `class`) is identified by its offset. A reference to an element of an array is considered to be a reference to the entire array. As described, our methods do not support pointer arithmetic and variable argument lists, although the methods can be extended to handle these features. We also do not support system features such as the Unix `longjmp` subroutine, or signal/exception handlers.

2.1 Transfer Function for Pointer Assignments

This section describes the behavior of the transfer function for pointer assignment statements in the control flow graph (CFG). Let In_N and Out_N be the set of alias relations holding at the entry and exit of node (statement) N, respectively. We use In and Out when the node N in question is clear from context. For a node N, In_N is the union of the Out sets of its predecessor nodes:

$$In_N = \bigcup_{P \in Preds(N)} Out_P$$

[2] In the presence of function pointers, a call site may correspond to multiple edges as described in Section 5.

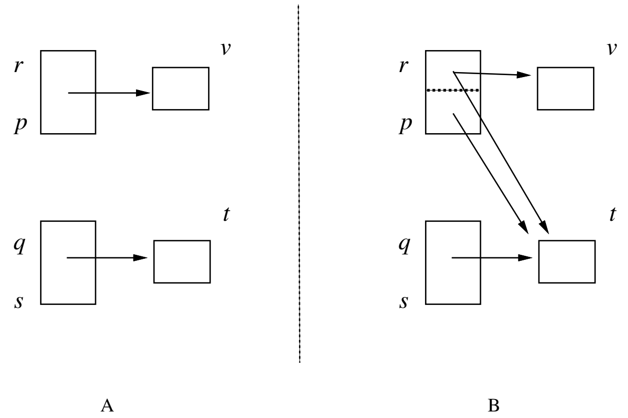

Fig. 1. Pictorial View of Aliasing

For an access path x, let $In(x)$ be the set of alias relations in In that contain access paths that include x. As an example, for

$$\begin{aligned} In \; = \; & \{\langle x, y\rangle, \langle *x, z\rangle, \langle w, z\rangle, \\ & \langle *x, *y\rangle, \langle x, x\rangle, \langle y, y\rangle, \\ & \langle *x, *x\rangle, \langle *y, *y\rangle, \langle w, w\rangle, \langle z, z\rangle\} \end{aligned}$$

$In(x)$ is $\{\langle x, y\rangle, \langle *x, z\rangle, \langle *x, *y\rangle, \langle x, x\rangle, \langle *x, *x\rangle\}$. The first two alias relations of $In(x)$ are *explicit* aliases, while the last four are implicit aliases. For brevity, we usually list only the explicit aliases of an alias set. The above alias sets become

$$In = \{\langle x, y\rangle, \langle *x, z\rangle, \langle w, z\rangle\}$$

and

$$In(x) = \{\langle x, y\rangle, \langle *x, z\rangle\},$$

respectively. Further details of our representation are given in Section 3.

We use $In(y/x)$ to denote the new set of alias relations obtained by replacing a single occurrence of access path x with access path y, for each alias relation in $In(x)$. For In in the above example,

$$In(q/x) = \{\langle q, y\rangle, \langle *q, z\rangle\}.$$

For the following assignment statement to an access path p, with a pointer type

$$N : p = q \;, \tag{1}$$

the transfer function TF_N is as follows:

$$Out_N = TF_N(In_N) = (In_N \; - \; \cup_{m \in M} In_N(*m)) \tag{2}$$

$$\cup \bigcup_{\langle p,u\rangle \in In_N} In_N(*u/*q), \tag{3}$$

where each $m \in M$ is a must-alias of p, which implicitly includes p. The *killing* rule (2) says that the value of each m changes after the assignment. The *generation*

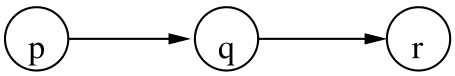

Fig. 2. Example Directed Alias Graph

rule (3) says that any may-alias of p, including trivial ones like $\langle p, p\rangle$, may point to what q may point to: if u is an alias of p, $*u$ becomes aliased to what $*q$ is aliased to. For example, assume the explicit aliases of In_N in (1) are

$$In_N = \{\langle p, r\rangle, \langle *r, v\rangle, \langle *p, v\rangle, \langle q, s\rangle, \langle *q, t\rangle, \langle *s, t\rangle\}, \tag{4}$$

which is shown in Figure 1A (along with the following implicit aliases: $\langle *p, *r\rangle$, $\langle *q, *s\rangle$, $\langle p, p\rangle$, $\langle *p, *p\rangle$, $\langle r, r\rangle$, $\ldots$).

Now the explicit aliases of Out_N are

$$\{\langle p, r\rangle, \langle *r, v\rangle, \langle q, s\rangle, \langle *q, t\rangle, \langle *s, t\rangle, \tag{5}$$

$$\langle *p, t\rangle, \langle *r, t\rangle, \langle *p, *q\rangle, \langle *r, *q\rangle, \langle *p, *s\rangle, \langle *r, *s\rangle\} \tag{6}$$

which is shown in Figure 1B. Each cell in Figure 1 represents a memory location. The elements of (5) remain after applying the killing rule (2) to In. The elements of (6) are generated by applying the generation rule (3) to the following subset of In:

$$\{\langle p, p\rangle, \langle p, r\rangle, \langle *q, t\rangle, \langle *q, *q\rangle, \langle *q, *s\rangle\},$$

among which $\langle p, p\rangle$, $\langle *q, *q\rangle$, and $\langle *q, *s\rangle$ are implicit aliases. Recall that, in the above example, $\langle p, r\rangle$ in In at S_i is a may-alias: p and r *may* refer to the same memory location. After the assignment to p in (1), r may still point to v.

If $\langle p, r\rangle$ is a must-alias, we can apply the killing rule to r as well as p [Aho et al. 1986]. This will delete $\langle *r, v\rangle$ from Out, resulting in better precision. This corresponds to the case when there is no edge from r to v in Figure 1B.

3. THE COMPACT REPRESENTATION OF ALIAS INFORMATION

For the representation of pointer-induced alias information, we use memory locations (variable names) aliased to a pointer expression rather than the pointer expression. We refer to memory locations associated with names as *named objects* [Horwitz et al. 1989; Chase et al. 1990].

Pointer-induced alias relations determine a directed *alias graph* as shown in Figure 2: $\langle *p, q\rangle$ implies that there is a (dereferencing) edge from object p to object q. Likewise, $\langle **p, r\rangle$ implies that there exists an object q such that there is an edge from p to q and one from q to r. Each named object that participates in a pointer-aliased relation is associated with a single node in the alias graph. Although we propose a method for naming dynamically allocated objects (malloc sites) in Section 8, any naming method may be employed.

We reduce the storage required by alias analysis by creating a *compact representation* for alias information such that for each alias relation

(1) each access path has no more than one level of dereferencing, and

(2) at least one of the pair of access paths has no dereferencing.

```
    /* Global Variables */
solution : Set of Variables

function ComputeAliases(Var, DerefLevel)
begin
    solution ← {}
    call Expand(Var, DerefLevel)
    return solution
end ComputeAliases

procedure Expand(Var, DerefLevel)
begin
    if DerefLevel = 0 then
        solution ← solution ∪ {Var}
    else
        foreach alias edge, e = (Var, Target), loop
            call Expand(Target, DerefLevel - 1);
        end loop
    end if
end Expand
```

Fig. 3. Algorithm for Answering an Alias Query

```
S1:    p = &x;
S2:    q = p;
S3:    *p = &y;
```

Fig. 4. Example for Compact Representation

Therefore, with a compact representation, all the alias relations are of the form $\langle *p, q\rangle$ ($\equiv \langle q, *p\rangle$) or $\langle p, q\rangle$, where p and q are variables (names of memory locations). For a given access path, the corresponding path(s) in the alias graph is (are) traversed to determine the named object(s) to which it is aliased. During this traversal, the number of dereferences encountered must be counted. Figure 3 presents a straightforward recursive algorithm to compute the aliases of an access path by traversing an alias graph. The alias information is recovered from the compact representation with an alias query, by calling the function ComputeAliases, passing two arguments: the named object of interest, and the number of dereferences of that variable desired.[3] The result of ComputeAliases(n, k) is all nodes in the alias graph along a path of length k from n. For example, to determine the aliases of $**p$ the following function call would be made: ComputeAliases(p, 2). For the alias graph of Figure 2, ComputeAliases(p, 2) would call Expand(p, 2), which would call Expand(q, 1), which would eventually return $\{r\}$.

In Figure 2, the named objects are p, q, and r. The *exhaustive* list of aliases holding for the example is $\{\langle *p, q\rangle, \langle *q, r\rangle, \langle **p, r\rangle, \langle **p, *q\rangle\}$. Notice that $\langle **p, r\rangle$

[3] If the variable is a pointer to an aggregate type, i.e., a structure, additional information concerning the fields for each level of dereference would also be required.

can be inferred from $\{\langle *p, q\rangle, \langle *q, r\rangle\}$, which is the compact representation.

Consider the code fragment listed in Figure 4. In the compact representation the alias relations at S2 are $\langle *q, x\rangle$ and $\langle *p, x\rangle$. The alias relations at S3 are $\langle *q, x\rangle, \langle *p, x\rangle$, and $\langle *x, y\rangle$. We generate the alias relation $\langle **q, y\rangle$ by applying ComputeAliases(q, 2) at S3. A full representation would require that the additional alias relation $\langle *p, *q\rangle$ be added at S2 in order to determine that the alias relation $\langle **q, y\rangle$ exists at S3. In Section 6.7, we provide examples that illustrate how each representation can be more precise than the other, and characterize the trade-offs in precision between the two representations [Marlowe et al. 1993]. We also describe methods to recover some of the precision lost due to the compact representation.

The compact representation, introduced in [Choi et al. 1993], is highly similar to the *points-to* representation of Emami et al. [1994]. The only difference between the representations is that the dereference of the first element is implied in the points-to representation. Although the points-to representation may be more intuitive, it cannot represent aliases between non-pointer objects. The compact representation can also represent reference parameter alias pairs, for example. Although the compact representation is not graph-based, it can easily be mapped to and from graph-based representations [Jones and Muchnick 1981]. The compact representation is not equivalent to performing a *transitive reduction* [Aho et al. 1972] over the alias graph during alias analysis in that transitive edges are not discarded.

We elaborate on the transfer function presented in (2) and (3) of Section 2 using the compact representation. Consider the assignment statement "$p_i = q_j$," where p_i is a pointer expression with i levels of indirection from variable p and q_j is a pointer expression with j levels of indirection from variable q. For example $p_2 = * * p$.[4]

$$Out_N = TF_N(In_N) = (In_N \; - \; Must(ComputeAliases(p,i)) \tag{7}$$

$$\cup \bigcup_{\substack{a \in ComputeAliases(p,i),\\ b \in ComputeAliases(q,j+1)}} \langle *a, b\rangle \tag{8}$$

where *ComputeAliases* uses the alias set associated with IN_N to answer its query. *Must* captures the set of must aliases returned by ComputeAliases. We follow the technique described in [Emami et al. 1994] and define *Must* in an optimistic manner: if during the analysis a pointer only points to one object, we treat it as a must alias. If later on in the analysis the pointer points to more than one object at the same program point, we no longer treat it as a must alias. Monotonicity is preserved because we use this information to remove aliases.

3.1 Cyclic Data Structures

In this section we illustrate how cyclic (recursive) structures are accommodated by the transfer function expressed in Rules 7 and 8 above. Consider the program segment and the cyclic data structure given in Figure 5, which shows the data structures in terms of r, *cell*1, and *cell*2 of the user program, resulting from the execution of statements S1 through S3, respectively, in the figure. For simplicity,

[4] We represent "&q" as q_{-1}.

```
S1:    cell1 = &cell2;
S2:    r = &cell1;
S3:    *cell1 = &cell1;
```

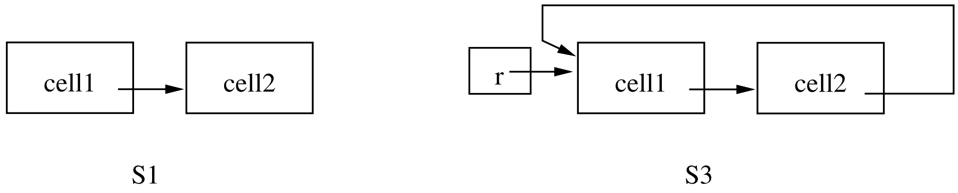

Fig. 5. Example for Cyclic Data Structure

we assume no non-trivial aliasing holds before S1. The alias relationship holding after S1, Out_{S1}, is thus $\langle *cell1, cell2\rangle$, and the alias relationship holding after S3 is as follows:

$$Out_{S3} = \{\langle *r, cell1\rangle, \langle *cell1, cell2\rangle, \langle *cell2, cell1\rangle\}. \tag{9}$$

This example illustrates that the transfer function combined with the compact representation naturally accommodates cyclic data structures without resorting to a k-limiting scheme to limit the level of indirection. Although the compact representation permits an arbitrary level of pointer indirection to be represented, it may lose precision at control flow join points. The precision of the compact representation is discussed in section 6.7. In the presence of a cyclic structure, an exhaustive representation that relies on k-limiting would summarize levels of dereference larger than some fixed constant value, resulting in a possible loss of precision [Jones and Muchnick 1981; Landi and Ryder 1992]. Other methods can accommodate some cyclic structures without such a loss [Deutsch 1994].

3.2 Data Flow Analysis and Aliasing

In classical data flow analysis without pointers, *uses* and *defs* at each statement are first computed in terms of (variable) names associated with the accessed memory locations. Then, alias information for each variable is used to identify additional variables potentially used or defined at each statement. While the association between a name and its memory location does not change within a name scope, the association between a pointer expression and its memory location can (repeatedly) change within a name scope. Our representation of aliases in terms of memory locations (named objects), rather than the pointer expression, precludes this problem.

4. INTERPROCEDURAL POINTER ALIAS ANALYSIS

Through pointer assignment, a called procedure can affect the aliasing of a calling procedure at the point just after the call site. Likewise, the aliasing of the calling procedure can affect the aliases that are created in the called procedure. Thus, aliases can be interprocedurally propagated forward from a call site to the entry point of a called procedure, and backward from a return point of a called procedure to the point just after the call site. This section presents a general interprocedural framework for computing pointer aliasing, along with two interprocedural algorithms that utilize this framework. The first algorithm is flow-sensitive, and the second is flow-insensitive.

The framework associates pointer alias information (represented as sets of intraprocedural aliases), with various points in the program. Two additional alias sets are associated with each procedure: one summarizing alias relations that hold on entry to the procedure, the other summarizing the effects of executing the procedure. Since these two sets are updated or used by other procedures, we refer to them as interprocedural alias sets.

The goal of the general interprocedural framework (as well as the flow-sensitive and flow-insensitive variants) is to compute the interprocedural alias sets. After the interprocedural algorithm terminates (the interprocedural alias sets converge), the (final) intraprocedural (statement-specific) alias information is computed.

During each interprocedural iteration, each procedure is visited and the (intermediate) intraprocedural sets for the procedure are computed. These sets are then used to update the corresponding interprocedural sets. After this update, the intermediate intraprocedural information is no longer required. Consequently, the space used to compute the intraprocedural sets can be reused during a subsequent intraprocedural phase of the interprocedural iteration. Thus, this method reduces the storage requirements of the interprocedural iteration by only requiring the presence of the intraprocedural sets of the current procedure.

The final intraprocedural alias information is computed by a performing a flow-sensitive pointer alias analysis pass over the PCG, using the converged interprocedural alias sets as input. This final pass can be deferred until other intraprocedural analyses are performed prior to generating code for each function. Once a function has been analyzed in this manner, the intraprocedural alias information does not need to be retained. In addition to reducing storage requirements, this method can also improve the precision of the flow-insensitive interprocedural algorithm, because flow-sensitive information is used intraprocedurally.

Figure 6 presents the general framework for interprocedural pointer alias analysis. Step S_1 builds the initial (optimistic) program call graph. Steps S_2 – S_4 initialize the interprocedural sets. While the algorithm iterates, optimistic assumptions are made concerning the contents of these sets. Structural and reference parameter aliases are incorporated as needed, for correctness. Iteration over the PCG occurs in topological order [5] (Steps S_6 – S_9) until the interprocedural sets and the PCG converge. During these steps the PCG is expanded (Step S_{10}) to accommodate additional call graph edges resulting from function pointers (Section 5). To improve the efficiency of this iteration (Steps S_5 – S_{14}), a worklist can be employed. Further, alternating iterations over the PCG between topological and reverse-topological order may reduce the number of iterations required for convergence.

The major characteristic of the general algorithm is the interleaving of the intraprocedural (Step S_7) and interprocedural phases (Step S_8). Sharir and Pnueli [1981] also employ interleaving in the first phase of their general functional framework for interprocedural analysis. The intraprocedural phase can be either flow-sensitive (Section 4.1) or flow-insensitive (Section 4.2).

[5] In the presence of PCG cycles, topological order can be defined by the removal of *back edges* [Kam and Ullman 1976].

S_1: build the initial PCG
S_2: foreach procedure, p, in the PCG, loop
S_3: initialize interprocedural alias sets of p
S_4: end loop
S_5: repeat
S_6: foreach procedure, p, in the PCG, loop
S_7: using the interprocedural alias sets (for entry of p and call sites in p), compute the intraprocedural alias sets of p
S_8: using the intraprocedural alias sets of p, update the interprocedural alias sets representing the effect of p on each procedure that calls or is called by p
S_9: end loop
S_{10}: update the PCG using new function pointer aliases
S_{11}: foreach new procedure p added to the PCG in step S_{10}, loop
S_{12}: initialize interprocedural alias sets of p
S_{13}: end loop
S_{14}: until the interprocedural alias sets and the PCG converge

Fig. 6. General Interprocedural Aliasing Framework

4.1 Flow-Sensitive Pointer Alias Analysis

This section presents a flow-sensitive alias analysis algorithm based on the interleaving framework. A flow-sensitive interprocedural alias analysis makes use of the intraprocedural control flow within individual procedures during the computation. The two phases of the flow-sensitive algorithm, interprocedural and intraprocedural, are described in Sections 4.1.1 and 4.1.2, respectively. Section 4.1.3 illustrates the algorithm with an example.

We use a sparse representation, the *sparse evaluation graph* (*SEG*) [Choi et al. 1991], for intraprocedural alias analysis. We define $SEG =< N_{SEG}, E_{SEG}, N_{Entry}, N_{Exit} >$, where N_{SEG}, the set of nodes in the Sparse Evaluation Graph, is the union of the following:

—$GenNodes$, the set of nodes where pointer variables [6] are potentially modified,

—$MeetNodes$, the set of nodes where alias information is merged at join points in the SEG, and

—N_{Entry} and N_{Exit}, two additional nodes such that each node is on a path from N_{Entry} to N_{Exit}, respectively.

E_{SEG} is the set of edges connecting nodes in N_{SEG}.

4.1.1 *Interprocedural Phase.* Consider Figure 7. For each procedure, p, the interprocedural phase supplies the sets $Entry_p$ and $Exit_p$, which correspond to the aliases that hold on entry to p and on exit from p, respectively. $Entry_p$ is initialized to the set of structural aliases for p (Step S_3).

In the figure, $ForwardBind_c$ maps the set of aliases holding at call site c into aliases holding in the called procedure, using the argument/parameter mappings

[6] Distinguishing between pointer and non-pointer variables is not required for correctness, but can result in improved efficiency.

S_1: build the initial PCG
S_2: foreach procedure p in the PCG, loop
S_3: $\quad Entry_p \leftarrow$ structural aliases for p
S_4: $\quad Exit_p \leftarrow \{\}$
S_5: end loop
S_6: repeat
S_7: $\quad$ foreach procedure, p, in the PCG, loop
S_8 $\quad\quad$ repeat
S_9: $\quad\quad\quad$ foreach SEG node, s, loop
S_{10}: $\quad\quad\quad\quad$ compute flow-sensitive intraprocedural aliasing:
S_{11}: $\quad\quad\quad\quad\quad$ if s is a call site, c, calling procedure r then
S_{12}: $\quad\quad\quad\quad\quad\quad$ compute Out_c using $BackwardBind_c(Exit_r)$ and In_c
S_{13}: $\quad\quad\quad\quad\quad$ else
S_{14}: $\quad\quad\quad\quad\quad\quad$ compute Out_s using In_s and its transfer function
S_{15}: $\quad\quad\quad$ end loop
S_{16}: $\quad\quad$ until the intraprocedural alias sets for p converge
S_{17}: $\quad\quad Exit_p \leftarrow Out_{N_{Exit}}$
S_{18}: $\quad\quad$ foreach call site, $c \in p$, calling procedure r loop
S_{19}: $\quad\quad\quad$ update $Entry_r$, using $ForwardBind_c(In_c)$
S_{20}: $\quad\quad$ end loop
S_{21}: $\quad$ end loop
S_{22}: $\quad$ update the PCG using new function pointer aliases
S_{23}: $\quad$ foreach new procedure p added to the PCG in step S_{22}, loop
S_{24}: $\quad\quad$ Initialize $Entry_p$ and $Exit_p$
S_{25}: $\quad$ end loop
S_{26}: until the interprocedural alias sets and PCG converge

Fig. 7. Flow-Sensitive Interprocedural Aliasing Algorithm

of c. We handle reference parameter aliases uniformly with pointer alias analysis. $BackwardBind_c$ maps aliases holding at the exit of the called procedure into aliases holding immediately following the call site c. Details of these mapping mechanisms are given in Section 7.2.

Traversals over the PCG are repeated in topological order (the outer loop of Steps S_7 – S_{21}). Step S_{17} updates the *Exit* set of the procedure being processed using the intraprocedural information. At Steps S_{18} – S_{20}, the alias information holding at a call site, say In_c, is propagated forward to the *Entry* set of the called routine, $Entry_r$. This is accomplished by performing the union of $ForwardBind_c(In_c)$ and the previous value of $Entry_r$.

4.1.2 *Intraprocedural Phase.* The intraprocedural alias computation, Steps S_8 – S_{16}, can be formulated as a data flow framework [Rosen 1979; Tarjan 1981], in which the solution at a given program point is related to the solution at other points. Thus, intraprocedural aliasing can be computed by applying well-known iterative techniques to the SEG. This computation requires three input components: the transfer function for each node that contains a (potential) pointer assignment, the set of aliases holding at the N_{Entry} node of the SEG (*Entry*), and the *BackwardBind* sets corresponding to call-site nodes in the SEG.

The transfer function for a general assignment to an access path of a pointer type is described in Sections 2.1 and 3. The aliases holding at the N_{Entry} node and the

```
S0:   char *u, a, b, c, ..., z;
S1:   P() {
S2:       u = &a;
S3:       *u = 'A';
S4:       Q();
S5:       *u = 'B';
S6:   }
S7:   Q() {
S8:       u = &b;
S9:   }
```

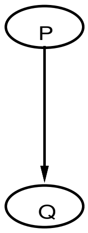

Fig. 8. Example Program Segment and its PCG

call-site nodes are interprocedurally computed, as described in Section 4.1.1.

If p calls a procedure r, $Exit_r$, which is computed during the previous iteration, is propagated back to the call site in p (Step S_{12}). Although a called procedure can affect the aliases holding on return to the call sites that invoke it, $Exit_r$ of a procedure r does not uniformly affect all the procedures that call r. Rather, the aliases holding on return from a call site are affected only by a subset of $Exit_r$. This subset includes only those aliases resulting from aliases holding at the call site, or generated in the called procedure independently of the aliases holding at the invoking call sites. Identifying call sites whose aliases have induced an alias relation in $Exit_r$ is discussed in Section 6.

After convergence, each node N has a solution Out_N that is consistent with In_N and the transfer function at every node.

Without an interprocedural phase, the flow-sensitive intraprocedural algorithm can be employed to sharpen alias information in a procedure. In this case, to preserve correctness, the sets of aliases holding at the N_{Entry} node and immediately after call-site nodes must contain all possible (interprocedurally induced) aliases.

4.1.3 *Example.* We illustrate our flow-sensitive algorithm with the program in Figure 8. Consider the first analysis of P. Since S2 generates the alias $\langle *u, a\rangle$, $Out_{S2} = Out_{S3} = \{\langle *u, a\rangle\}$. Since Q has not yet been analyzed, $Exit_Q = \{\}$. Thus, $Out_{S4} = Out_{S5} = Exit_P = \{\}$. Before Q is analyzed, the alias associated with the call site at S4 is propagated forward along the PCG edge to Q (Steps S_{18} – S_{20} in Figure 7), giving $Entry_Q = \{\langle *u, a\rangle\}$. A flow-sensitive analysis of Q sets $Exit_Q = \{\langle *u, b\rangle\}$, because of the pointer assignment at S8, which both kills the alias $\langle *u, a\rangle$, and generates the alias $\langle *u, b\rangle$. Thus, $Out_{S8} = Exit_Q = \{\langle *u, b\rangle\}$. This completes one iteration of the repeat loop (Steps S_6 – S_{26}). Since alias information has changed from the initial values, another iteration of this loop occurs.

During the second analysis of P, the alias $\langle *u, b\rangle$ is present in $Exit_Q$, and thus is propagated to Out_{S4}, Out_{S5}, and $Exit_P$. No further changes occur. At the conclusion of the flow-sensitive analysis, we have the following results:

$$Entry_P = \{\},$$

$$Out_{S2} = Out_{S3} = Entry_Q = \{\langle *u, a\rangle\},$$

$$Out_{S4} = Out_{S5} = Out_{S8} = Exit_P = Exit_Q = \{\langle *u, b\rangle\}.$$

4.2 Flow-Insensitive Pointer Alias Analysis

This section presents a flow-insensitive alias analysis algorithm, which is also based on the general interleaving framework. After describing the interprocedural and intraprocedural phases of the algorithm, we describe how to improve the precision of the interprocedural phase by incorporating kill information. This section concludes with a description of a technique called *deferred evaluation*, which can improve the efficiency of the intraprocedural phase.

To improve efficiency, flow-insensitive analyses do not rely on intraprocedural control flow information during the computation. Instead, the effects of a procedure are summarized and associated with either a node or edge of the PCG. One example of this is the computation of the set of variables modified by each procedure, the *PMod* computation. In this computation, the direct modifications (i.e., excluding call sites) of non-local variables within a procedure are summarized, and associated with each node in the PCG [Banning 1979] prior to the interprocedural propagation. This information is then used during the propagation over the PCG.

The flow-insensitive computation of pointer-induced aliases is similar to the $PMod$ computation, in that the effect of each PCG node is summarized. However, unlike the $PMod$ computation, the information directly generated at a node is dependent not only on the statements in the procedure, but also on the information that is propagated along the PCG edges to and from that node. Thus, the direct effects of a procedure cannot be determined without interprocedural information. Instead, the set of pointer-assignment statements in a procedure is associated with each PCG node, and their effect on alias information is computed during the analysis. The information describing how parameters are passed at a call site is associated with each edge. With this information for each node and each edge of the PCG as input, the flow-insensitive interprocedural alias analysis summarizes the effects of a procedure p, by computing $PGen_p$, the set of aliases *generated* by the invocation of p. This differs from the flow-sensitive algorithm, which summarizes the effects of a procedure by capturing the alias information that *holds* on exit from a procedure.

In addition to $PGen_p$, the flow-insensitive algorithm computes the following sets:

—$Entry_p$, the set of aliases that hold upon entry to procedure p,

—$Holds_p$, the set of aliases that may hold at any point in p,

—$IGen_p$, the set of aliases *generated* by all of the pointer-assignment statements[7] of procedure p,

—$CSGen_p$, the set of aliases *generated* by call sites in the procedure p.

This section describes the relationship among these sets and provides a set of data flow equations representing this relationship. As $Entry_p$ and $PGen_p$ are used or updated by other procedures, these interprocedural alias sets persist throughout the alias computation. The other intraprocedural sets, $IGen_p$, $CSGen_p$, and $Holds_p$,

[7] Storage allocation and deallocation statements for pointers, such as `malloc` and `free` in C, are regarded as pointer assignment statements.

S_1: build the initial PCG
S_2: foreach procedure, p, in the PCG, loop
S_3: $Entry_p \leftarrow$ structural aliases for p
S_4: $PGen_p \leftarrow \{\}$
S_5: end loop
S_6: repeat
S_7:   foreach procedure, p, in the PCG, loop
S_8:   using the interprocedural sets ($Entry_p$ and $PGen_r$ for all r called by p)
            compute intraprocedural sets, $CSGen_p$, $Holds_p$ and $IGen_p$,
S_9: $PGen_p \leftarrow CSGen_p \cup IGen_p$
S_{10}:   foreach call site, $c \in p$, calling procedure r loop
S_{11}:   update $Entry_r$ using $ForwardBind_c(Holds_p)$
S_{12}:   end loop
S_{13}:   end loop
S_{14}:   update the PCG using new function pointer aliases
S_{15}:   foreach new function, q, added to the PCG in step S_{14}, loop
S_{16}:   initialize $Entry_q$ and $PGen_q$ as in steps S_3 and S_4
S_{17}:   end loop
S_{18}: until the interprocedural alias sets and PCG converge

Fig. 9. Flow-Insensitive Interprocedural Aliasing Algorithm

can be discarded after their results are incorporated into the appropriate interprocedural set.

Figure 9 presents our flow-insensitive interprocedural algorithm. The computation of the intraprocedural sets (Step S_8) uses the interprocedural sets. Computing the interprocedural sets (Steps S_9 – S_{12}), in turn, uses the intraprocedural sets. Once again, the iterations over the PCG (Steps S_6 – S_{18}) handle this mutual dependence between the inter- and intraprocedural sets and are performed in topological order. The following sections provide details on how the interprocedural and intraprocedural sets are computed.

4.2.1 *Interprocedural Phase.* Using the intraprocedural sets, the interprocedural sets are computed as follows, where c represents a call site:

$$Entry_p = \bigcup_{c=(r,p)} ForwardBind_c(Holds_r), \text{for all } r \text{ that call } p \tag{10}$$

$$PGen_p = CSGen_p \cup IGen_p \tag{11}$$

$Entry_p$ is computed by propagating information along all call sites that invoke p (Steps S_{10} – S_{12}). $PGen_p$ summarizes what is generated by p (Step S_9).

4.2.2 *Intraprocedural Phase.* Using the interprocedural sets, the intraprocedural sets are computed by solving the following equations (Step S_8):

$$CSGen_p^c = BackwardBind_c(PGen_q), \text{where } p \text{ calls } q \text{ at } c\text{, i.e., } c = (p,q) \tag{12}$$

$$CSGen_p = \bigcup_{c \text{ in } p} CSGen_p^c \tag{13}$$

$$Holds_p = Entry_p \cup CSGen_p \cup IGen_p \tag{14}$$

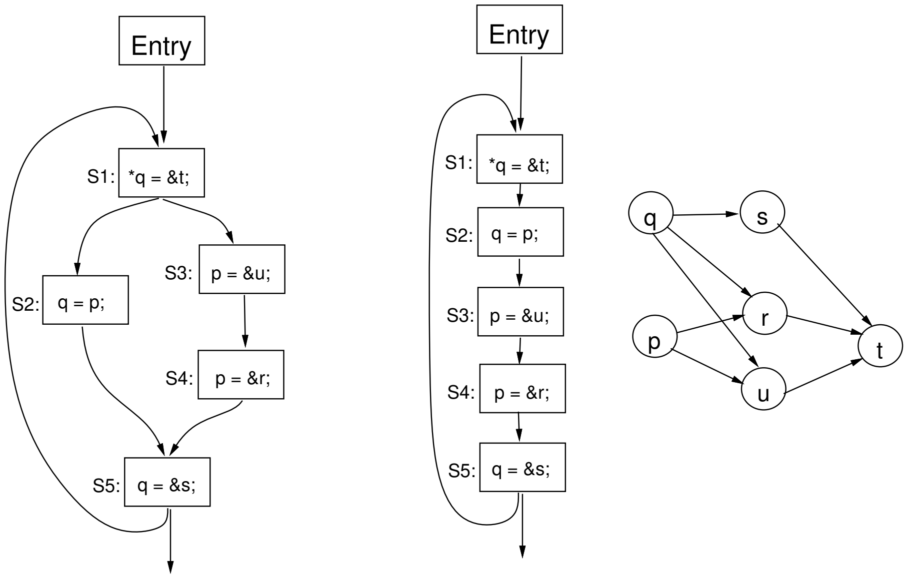

Fig. 10. Example Illustrating the Flow-Insensitive Intraprocedural Phase

$$IGen_p = \bigcup_{s \in p} IGen_p^s(Holds_p), \text{where } s \text{ is a pointer assignment} \tag{15}$$

$CSGen_p^c$, the component of $CSGen_p$ for call site c, is computed by propagating interprocedural information backward to a call site c in p. This information is then summarized for all call sites in p.

The cyclic dependence between Equations (14) and (15) requires iteration over the set of pointer-assignment statements in p. As a flow-insensitive analysis does not consider control flow information during the analysis, it must, for correctness, reflect all possible paths that can be constructed by the set of statements associated with p. This is modeled by treating each procedure as a single loop consisting of all the pointer-assignment statements of the procedure placed sequentially in a arbitrarily chosen order. The transfer function in Section 3 is modified to prevent killing any aliases.[8]

Figure 10 illustrates this technique. The control flow graph on the left represents the original program. The graph in the middle illustrates the manner in which the effect of all possible paths is captured. The order in which statements are chosen in this graph is arbitrary, and for this example the worst possible order for efficiency is chosen. The alias graph on the right specifies the aliases that result from the intraprocedural computation, assuming no aliases are in the *Entry* set.

Figure 11 provides further details of the intraprocedural phase of the flow-insensitive algorithm (Step S_8 of Figure 9). Steps S_1 – S_4 compute $CSGen_p$, the set of aliases that are generated from call sites of p. Step S_5 initializes $IGen_p$

[8]Since no killing can occur, an alternative "switch statement" representation has the same effect. In this representation, the statements are placed as alternatives in a switch statement structure contained in a loop, and killing is not allowed.

S_1: $CSGen_p \leftarrow \{\}$
S_2: foreach call site, c in p, calling q, loop
S_3: $CSGen_p \leftarrow CSGen_p \cup BackwardBind_c(PGen_q)$
S_4: end loop
S_5: $IGen_p \leftarrow \{\}$
S_6: $Holds_p \leftarrow Entry_p \cup CSGen_p$
S_7: repeat
S_8: foreach pointer assignment statement, s, in p, loop
S_9: compute the set, $A = IGen_p^s(Holds_p)$
S_{10}: $Holds_p \leftarrow Holds_p \cup A$
S_{11}: $IGen_p \leftarrow IGen_p \cup A$
S_{12}: end loop
S_{13}: until aliasing converges

Fig. 11. Refinement of Flow-Insensitive Intraprocedural Phase (Step S_8 in Figure 9)

optimistically. Step S_6 sets $Holds_p$ to be the aliases that reach p interprocedurally via $Entry_p$ and $CSGen_p$

Each iteration of the $S_8 - S_{12}$ loop traverses each pointer assignment statement, s, adding aliases generated by s to the $Holds_p$ and $IGen_p$ sets. Since the aliases generated depend on $Holds_p$, this traversal is repeated until no new aliases are generated (the $S_7 - S_{13}$ loop). For computations where killing is not considered, this method is equivalent to considering all possible paths. Section 4.2.4 describes how to limit the types of pointer-assignment statements that need be considered during the iteration (Steps $S_7 - S_{13}$).

As described in Section 3, the effect of a pointer-assignment statement, s, on aliasing can be described in terms of its alias transfer function (TF_s). The computation of $IGen_p$ presented treats the pointer-assignment statements of a procedure as one loop and iterates over this loop using the transfer function described in Section 3 *without* killing any aliases. This has the effect of defining a transfer function that includes all statements of a procedure in any possible order. Upon convergence, $Holds_p$ contains the result of this computation: the aliases assumed to hold at each point in p. Thus, when a fixed point is reached, the following equation is satisfied by $Holds_p$:

$$Holds_p = TF_s(Holds_p), \text{for all } s \text{ in } p.$$

4.2.3 *Using Kill Information to Improve Precision.* Flow-sensitive analyses utilize intraprocedural control information to *kill* alias information, which can improve both intraprocedural and interprocedural precision. This section describes how to improve the interprocedural precision of the flow-insensitive analysis by selectively utilizing *precomputed* kill information, without incurring the full overhead of flow-sensitive interprocedural analysis.

Consider Figure 12, where procedure r calls procedure p at S2, and $Entry_r$ is as given in the figure. By Equation (10) in Section 4.2.1, the aliases that are propagated to $Entry_p$ from S2 are those that hold at S2, i.e., $Holds_r$, which is: $\{\langle *x, a_1\rangle, \ldots, \langle *x, a_n\rangle, \langle *x, y\rangle, \langle *x, z\rangle\}$.

Although this is conservatively correct, the precision of this solution can be im-

```
    r() {
S1:     x = &y;
S2:     p();
S3:     x = &z;
    }
```

$Entry_r = \{\langle *x, a_1\rangle, \ldots, \langle *x, a_n\rangle\}$
$IGen_r = \{\langle *x, y\rangle, \langle *x, z\rangle\}$

Fig. 12. Example Illustrating the Usefulness of $ForwardKill$

proved by incorporating precomputed kill information to limit the aliases that are propagated interprocedurally. This is achieved by defining for each call site, c, contained in procedure r, $ForwardKill_c$ as follows:

$ForwardKill_c =$ the set of aliases killed along all paths from r's entry node to c.

As $ForwardKill_c$ is computed before alias information is available, it can only kill pointers where the left side of the assignment is known independently of alias information. For example, "x = ..." kills all aliases of $*x$. For a statement of the form "$*x = \ldots$", no aliases are included in $ForwardKill_c$.

However, it may be desirable to precede the kill computation by a flow-sensitive intraprocedural-only pointer alias analysis algorithm that makes pessimistic interprocedural assumptions. With this information computed, it may be possible to determine if a *strong update* [Chase et al. 1990] is performed, i.e., an assignment to a pointer expression that is an access path for only one memory location. In this case the statement can be considered as killing.

In order to determine which aliases should be propagated to the called procedure from c, $ForwardKill_c$ is used to remove aliases from $Entry_r$, before adding aliases that are generated by r ($IGen_r \cup CSGen_r$). In the example of Figure 12, $\{\langle *x, y\rangle, \langle *x, z\rangle\}$ is propagated to p, but not $\{\langle *x, a_1\rangle, \ldots, \langle *x, a_n\rangle\}$.

The updated equation for $Entry_p$ (Equation (10) of Section 4.2.1) is as follows:

$$Entry_p = \bigcup_{c=(r,p)} ForwardBind_c((Entry_r - ForwardKill_c) \cup CSGen_r \cup IGen_r),$$

where r is any procedure that calls p

Since $ForwardKill_c$ is constant, its computation is performed before the interprocedural propagation (S_6 of Figure 9), and used during the propagation. This allows the interprocedural algorithm to remain flow-insensitive, as no control flow information is required during the propagation.

In an analogous manner this technique can be used to improve the precision of information during the backward PCG propagation. Information is interprocedurally propagated from a called procedure to a calling procedure p at call site c via $CSGen_p^c$. To improve the precision of this set, we define for a call site c in p, the following:

$BackwardKill_c =$ the set of aliases killed along all paths from c to p's *exit* node.

```
    p() {
S1:     x = &y;
S2:     q();
S3:     x = &z;
    }
```

$CSGen_p^{S2} = \{\langle *x, b_1\rangle, \ldots, \langle *x, b_n\rangle\}$
$IGen_p = \{\langle *x, y\rangle, \langle *x, z\rangle\}$

Fig. 13. Example Illustrating the Usefulness of *BackwardKill*

This set is used when constructing $PGen_p$, by removing all aliases from $CSGen_p^c$ that are killed after the call. If any of these aliases are generated elsewhere in p, they are included in $PGen_p$ for correctness.

The updated equation for $PGen_p$ (Equation 11 of Section 4.2.2) is given below.

$$PGen_p = \bigcup_{c \in p} (CSGen_p^c - BackwardKill_c) \cup IGen_p \tag{16}$$

$CSGen_p$ is divided into its call site components to allow the addition of kill information.

Figure 13 presents an example in which using *BackwardKill* information improves the precision of $PGen_p$. Assume the contents of $CSGen_p^{S2}$ are as given in the figure. Without kill information we have

$$PGen_p = CSGen_p^{S2} \cup IGen_p = \{\langle *x, b_1\rangle, \ldots, \langle *x, b_n\rangle, \langle *x, y\rangle, \langle *x, z\rangle\}.$$

Using Equation 16 for $PGen_p$ gives:

$$PGen_p = \{\langle *x, y\rangle, \langle *x, z\rangle\}.$$

Like $ForwardKill_c$, $BackwardKill_c$ can be computed as a preprocessing step before the flow-insensitive iteration over the PCG has begun. Thus, techniques described for the computation of $ForwardKill_c$ for handling statements of the form "$*p = \ldots$" can also be employed for $BackwardKill_c$. A spectrum of killing criteria, based on the amount of control flow information required, can be employed when computing these sets. The computation of kill using only dominator trees is described in [Choi et al. 1993].

4.2.4 *Deferred Evaluation.* This section presents two techniques for improving the efficiency of the intraprocedural phase. First, a class of pointer-assignment statements is identified that need not be considered during the intraprocedural iteration. Second, a technique is introduced called *deferred evaluation*, which allows an additional class of pointer-assignment statements to be removed from the intraprocedural iteration. Both techniques rely on properties of the compact representation.

Consider statements S3, S4, and S5 of Figure 10. Because the aliases generated by these statements can be determined independently of the aliases that hold before executing them, these statements need not be included in the iteration over the

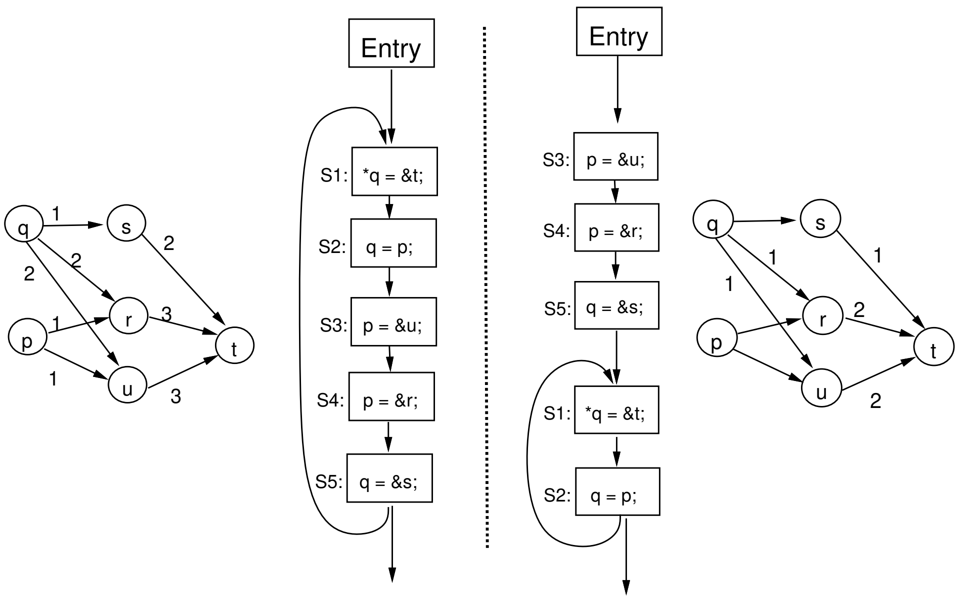

Fig. 14. Example Illustrating the Improved Flow-Insensitive Intraprocedural Phase

middle graph created for the flow-insensitive analysis.[9] These statements contribute a constant value to $IGen_p$; they do not depend on $Holds_p$. This class of statements is processed prior to the interprocedural propagation (S_6 of Figure 9) by recording the aliases they generate in $IGen_p$ and $Holds_p$. Steps S_5 and S_6 of Figure 11 are updated to use this precomputed information.

Figure 14 illustrates how the program in Figure 10 would be processed using this enhancement. In addition to the list of statements, we also include the alias graph representing $IGen_p$ upon convergence, once again assuming $Entry_p = \{\}$. The graphs (alias and control flow) on the left correspond to the original algorithm, while the graphs on the right uses the above enhancement. Each edge in the alias graph is labeled with the intraprocedural iteration number in which it was created. Unlabeled edges correspond to aliases that are created prior to the intraprocedural iteration. Using this enhancement, the number of iterations required for convergence is reduced from three to two, while still producing the same alias graph.

Determining the aliases generated by statements S1 and S2 requires information about the aliases holding at these statements ($Holds_p$). For statement S1 the aliases of $*q$ are required, i.e., for all x such that $\langle *q, x\rangle$ holds, an alias $\langle *x, t\rangle$ is created. Statement S2 requires the aliases of $*p$; for all y where $\langle *p, y\rangle$ holds, $\langle *q, y\rangle$ is created. Thus, it appears that both types of statements must be included in the intraprocedural iteration.

However, for the aliases generated by statement S2 the dereference component, $*q$, is known. Because the *source* node for edges in the alias graph that will be

[9] Assuming $\langle *s, p\rangle$ holds before these statements, a full representation of the alias solution would include $\langle **s, u\rangle$ as holding after considering S3.

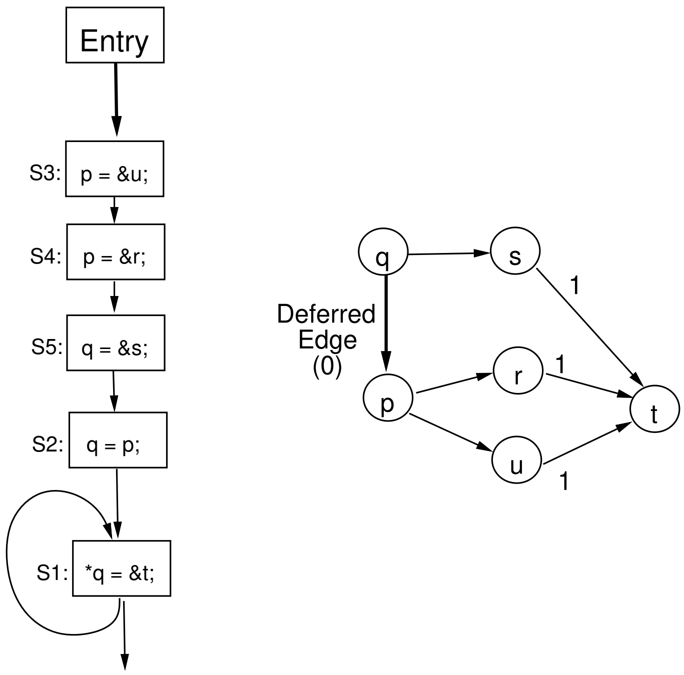

Fig. 15. Example Illustrating Flow-Insensitive Intraprocedural Phase with Deferred Evaluation

created by this statement is q, the algorithm defers the evaluation of statement S2 until after the intraprocedural iteration has completed, when the aliases of $*p$ are known. This is accomplished by creating a *deferred alias* from q to p, with a *deferred value* of 0. This signifies that $*q$ will be aliased to all objects that p can reach via a path of length 1 over the alias graph. If aliases of $*q$ are required during the iteration, the aliases the deferred alias represents are inferred. In this example, it represents edges from q to whatever p points to, i.e., $\{\langle *q, y\rangle|$ for all y such that $\langle *p, y\rangle\}$.

Thus, only statement S1 remains in the intraprocedural iteration. If the procedure does not contain any statements of this type, no intraprocedural iteration is required. After the intraprocedural iteration has completed, deferred aliases can be expanded into full aliases without a loss of precision. This approach is referred to as *deferred evaluation*.

Figure 15 shows how the example program would be processed using deferred evaluation. Notice that only one statement is contained in the loop, and thus only one iteration would be required for convergence. This compares favorably to the original analysis, in which three iterations are required before convergence.

The previous example is generalized by classifying pointer-assignment statements into three categories listed below. Type 1 statements have a constant *IGen* function and are processed first. For Type 2 statements, deferred aliases are created with a *deferred value* of r, where r corresponds to the number of dereferences that precede the variable on the right side of the assignment. This is performed before the iteration and then optionally expanded after the iteration converges. Type 3

(1) Add aliases from Type 1 statements to $IGen_p$ (Step S_5) and $Holds_p$ (Step S_6)
(2) Create deferred aliases for Type 2 statements (after Step S_6)
(3) Iterate over Type 3 statements only (Steps S_7 – S_{13})
(4) (Optional) Expand all deferred aliases (after Step S_{13})

Fig. 16. Enhancements to Flow-Insensitive Intraprocedural Algorithm

statements require iteration.

Type 1:. $p = \&q;$

Type 2:. $p = \underbrace{* \ldots *}_{r} q; r \geq 0$

Type 3:. $\underbrace{* \ldots *}_{l} p = \underbrace{* \ldots *}_{r} q; l \geq 1, r \geq -1,$ "$r = -1$" is equivalent to $\&q$.

An alias representation with deferred aliases can be viewed as an extension of the compact representation described in Section 3. Such a representation would extend the properties of an alias relation to include a value, which is either the deferred value of the alias edge it represents, or -1 for a regular alias edge. Using this representation requires an enhanced version of the ComputeAliases algorithm of Section 3, which is described in Appendix A.

Figure 16 summarizes the enhanced intraprocedural algorithm with the first technique (Step 1) and deferred evaluation (Steps 2 – 4). The first technique can be employed even without deferred evaluation. Both techniques decrease the number of pointer-assignment statements considered at S_8 of Figure 11, by processing some statements prior to entering the S_7 loop, and optionally expanding deferred aliases after this loop. This reduces the number of statements considered during the intraprocedural iteration, which can also reduce the number of iterations required for convergence of this phase.

Step 4 of this algorithm, expanding all deferred evaluation aliases after the intraprocedural iteration, is not required. In the absence of kill information, a representation with deferred aliases is as precise as if all deferred aliases were expanded after the intraprocedural iteration, which, in turn, is as precise as the flow-insensitive method without deferred evaluation. When interprocedural kill information (Section 4.2.3) is incorporated, the expansion of all deferred evaluation aliases (after the intraprocedural iteration for a procedure converges) is required for the method to be as precise as the flow-insensitive method without deferred evaluation.

Edges in the alias graph that represent deferred aliases may need to be interpreted correctly during alias analysis, or after alias analysis has completed. For example, to determine the aliases that are generated by the pointer-assignment statement "$*p = q$" requires the aliases of $*p$. Similarly, def-use analysis also requires alias information. Appendix A describes how alias edges should be interpreted when answering an alias query in the presence of deferred evaluation edges.

5. FUNCTION POINTER ANALYSIS

In this section we describe a method for constructing the PCG in the presence of function parameters, function variables, and function pointers. The method accommodates function parameters and arbitrary levels of function pointers by

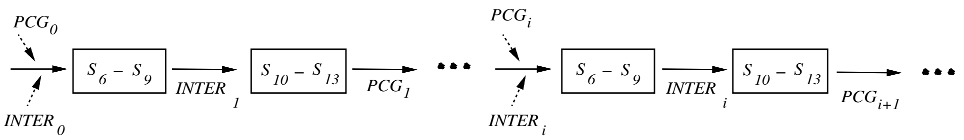

Fig. 17. Interleaving of Alias Phases and PCG Update

using the general interprocedural aliasing framework (Figure 6). Function pointer analysis and similar problems that are either more general or less general have been solved by others, as described in Section 11.4.

When embedded in the alias analysis, which generally occurs before any other phases that need the PCG, the method incurs minimal overhead. The techniques described in this section to accommodate function pointers can be applied to both flow-sensitive and flow-insensitive analysis. We will use the term "function pointers" to refer to function parameters, function variables, and function pointers. Likewise, we will use "regular pointers" to refer to pointers that are not function pointers. [10]

Alias analysis in the presence of function pointers is achieved by first handling function pointers in the same manner as regular pointers during the intraprocedural phase of the analysis, and then identifying new PCG edges from the alias information as the computation proceeds. Thus, the method *does not* compute aliasing from scratch as new PCG edges are found. Instead, it adds these new edges to construct the new PCG and incrementally continue the alias computation, using the alias information computed with the previous PCG as the initial condition.

Formally, this amounts to computing an additional *interprocedural* set of procedures, which is the set of aliases of a function pointer, fp, invoked at call site, c, as follows:[11]

$$FuncAlias_c = \{\,proc \mid \langle *fp, proc\rangle \in In_c,\ *fp \text{ is invoked at call site } c\}.$$

The algorithm then incrementally updates the PCG with the newly added edges identified from $FuncAlias_c$. These two steps are performed as the first step of an interprocedural phase.

Figure 17 shows the relationships among these alias analysis phases, interprocedural alias information, and program call graph construction. At the start, the algorithm computes the initial interprocedural alias information, $INTER_0$. At the same time, it also builds the initial program call graph, PCG_0, in which there is a single edge for each call site of *explicit* procedure calls, but no edges for call sites of function pointers (Step S_1 of the algorithm in Figure 6).

Given PCG_i and $INTER_i$, the algorithm applies Steps $S_6 - S_9$ to compute the interprocedural alias sets for each procedure, $INTER_{i+1}$. This involves computing intermediate intraprocedural alias sets for each procedure. $INTER_{i+1}$ provides a set of aliases for each function pointer invoked at call sites, from which the next generation PCG, PCG_{i+1}, is built, in Steps $S_{10} - S_{13}$. This process continues until the interprocedural information (i.e., alias and PCG) converges. With this

[10]Distinguishing between function pointers and regular pointers is not required for correctness, but can result in improved efficiency.
[11]If fp is a function parameter or variable, the '$*$' is not used.

approach, unlike conventional PCGs, a call site of a function pointer may have multiple PCG edges, one for each procedure aliased to the function pointer.

When applied only to function pointers, the algorithm provides a method for constructing the PCG.

6. THE EXECUTABLE PATH PROBLEM

Pointer alias analysis is known to be an undecidable problem [Landi 1992; Ramalingam 1994], and various analysis methods have been developed that approximate the solution with varying degrees of precision and efficiency. A source of inaccuracy is the generation of *invalid aliases* — aliases that cannot hold — by *combining* aliases that should not be combined because there is no execution instance in which they all occur.[12] We call this the *non-distributive combining* problem, which arises when non-distributive data flow solutions such as for pointer alias analysis are combined. The non-distributive combining problem occurs on forward paths over the PCG or CFG and we will call it the *forward path* problem for brevity.

The interprocedural forward path problem also exists between global variables and reference parameters, if alias information is not properly factored into data flow information [Burke and Choi 1992]. However, the problem becomes more pronounced for pointer aliasing because it can also occur at pointer-assignment statements within a routine.

Another source of inaccuracy in alias analysis is the propagation of an alias holding at the exit of a routine backward to a call site for which it is invalid. This is called the *unrealizable path* problem [Landi and Ryder 1992] and can occur for any interprocedural data flow problem. We will call it the *backward path* problem for brevity.

We divide path information into two components: *generation-path* information, which describes the path along which an alias is generated; and *propagation-path* information, which describes the path along which an alias is propagated after its generation.

The forward and backward path problems can be handled precisely with full path information [Sharir and Pnueli 1981], which can incur cost exponential in the path length. This section distinguishes alias relations by creating *alias instances* [Choi et al. 1993]. Alias instances can approximate full path information by capturing the manner in which an alias relation is generated and propagated. Alias instances offer a two-level framework for trade-offs between precision and efficiency. The first level is determined by the number and size of alias instances that are to be distinguished for each alias relation. It affects both the space and time efficiency of the analysis. The second level is determined by the amount of information that is utilized. It affects only the time efficiency of the analysis. Section 6.8 discusses this issue.

Alias instances are comprised of three components. The first component, *birth sites*, records the intraprocedural program point where an alias is generated. This technique can be used to prevent the combination of aliases unless they are all gen-

[12] In fact, all (combinations of) aliases are "valid" in that they make the analysis conservative, but not incorrect. The goal of the alias analysis here and in general is to make the set of "valid" aliases as small as possible. In this sense, invalid aliases are those that need not be in the solution set of a correct alias analysis.

```
B0:if (...) {
B1:     u = &v;
        p = &q;
    } else {
B2:     u = &w;
        p = &t;
    }
B3:*p = u;
```

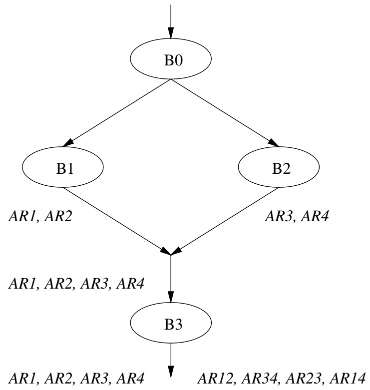

Fig. 18. Example Code Segment and its CFG

erated along the execution path of some execution instance (forward path problem).

The second component, *call sites*, records the most recent call site over which an alias is propagated interprocedurally. This information can be used to prevent the combination of aliases unless they are all propagated from the same call site (forward path problem). The call site component can also be used to ensure that aliases are propagated back only to the call site from which they are propagated (backward path problem).

Birth sites and call sites, respectively, capture only one level of generation and propagation path information. The third component, *source alias sets*, offers further precision, at increased cost, by capturing multiple levels of program points that contribute to the generation or propagation of an alias relation. Source alias sets are originally introduced in [Choi et al. 1993]. This section describes their computation and use.

The techniques in this section can be used to improve the precision of previous approximation methods [Jones and Muchnick 1981; Larus and Hilfinger 1988; Horwitz et al. 1989; Chase et al. 1990; Landi and Ryder 1992; Choi et al. 1993; Deutsch 1994; Burke et al. 1995; Emami et al. 1994; Wilson and Lam 1995] as well as help improve the precision of alias information for the compact representation [Choi et al. 1993; Emami et al. 1994] or *graph-based representations* [Jones and Muchnick 1981; Larus and Hilfinger 1988; Horwitz et al. 1989; Chase et al. 1990]. Section 6.7 describes how this is accomplished.

6.1 Example

Consider the example code segment and its CFG in Figure 18. We first illustrate the forward path problem by considering the two statements in (basic) block B1. After this block, $*u$ and v refer to the same memory location, generating the alias relation $AR1 = \langle *u, v\rangle$; and $*p$ and q refer to the same memory location, generating the alias relation $AR2 = \langle *p, q\rangle$. Likewise, in block B2, two alias relations, $AR3 = \langle *u, w\rangle$

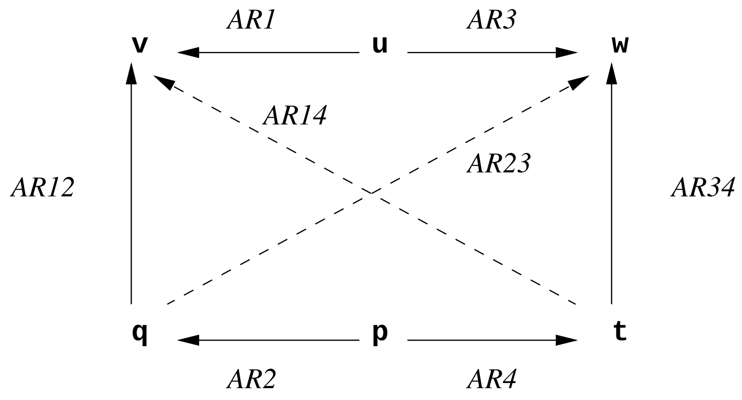

Fig. 19. Alias Graph of Figure 18

and $AR4 = \langle *p, t \rangle$, are generated. Figure 19 shows the corresponding alias graph of these alias relations. For example, $AR1 = \langle *u, v \rangle$ is represented by two nodes, u and v, and a directed edge from u to v labeled $AR1$, i.e., 'u points to v'.

These four alias relations are propagated to B3, where the two control flow edges, one from B1 and the other from B2, merge at block B3. The effect of the statement B3 on these alias relations is that aliases of $*p$ – i.e., q and t according to $AR2 = \langle *p, q \rangle$ and $AR4 = \langle *p, t \rangle$, respectively – are computed to point to what u points to, which is v or w, according to $AR1 = \langle *u, v \rangle$ or $AR3 = \langle *u, w \rangle$. In the alias graph in Figure 19, this creates edges from q and t to v and w. These edges are labeled $AR12$, $AR34$, $AR23$, and $AR14$ respectively, to represent the following new aliases generated at S3: $AR12 = \langle *q, v \rangle$, $AR34 = \langle *t, w \rangle$, $AR23 = \langle *q, w \rangle$, and $AR14 = \langle *t, v \rangle$. $AR12$ is generated by combining $AR1$, which is the edge from u to v, and $AR2$, which is the edge from p to q; $AR34$ is generated by combining $AR3$, which is the edge from u to w, and $AR4$, which is the edge from p to t; $AR23$ is generated by combining $AR2$ and $AR3$; and $AR14$ is generated by combining $AR1$ and $AR4$.

$AR23$ is an invalid alias: it is generated by combining $AR2$ and $AR3$, which are propagated by two different basic blocks along different execution paths. Likewise, $AR14$, generated by combining $AR1$ and $AR4$, is also an invalid alias. These are examples of the forward path problem. Landi and Ryder [1992] illustrate the forward path problem for pointer alias analysis with a similar example.

We now illustrate the backward path problem using the graph in Figure 18 as a PCG, not a CFG. (In this section, the graph in Figure 18 is generally used as a CFG in the intraprocedural context, and as a PCG in the interprocedural context.) Each node in the figure represents a procedure, and an edge between two nodes represents a call of a procedure: procedure $B0$ calls procedures $B1$ and $B2$, each of which in turn calls $B3$. Assume the four aliases propagated to the entry of $B3$ – $AR1$, $AR2$, $AR3$ and $AR4$ – still hold at the exit of $B3$. Note that $AR1$ and $AR2$ should be propagated back to only $B1$ from which they are originally propagated, and not to $B2$. Similarly, $AR3$ and $AR4$ should be propagated only back to $B2$, and not to $B1$. The path from $B1$ to $B3$ to $B2$, and the path from $B2$ to $B3$ to $B1$, are called (interprocedurally) unrealizable paths. These are examples of the backward path problem.

6.2 Birth Sites and Alias Instances

As illustrated in the previous section, aliases are generated at program points, propagated along CFG and PCG edges until they are killed at some program points, and possibly combined to generate new aliases. We use $\otimes$ to denote the explicit combination of aliases to generate a new alias. For example, in Figure 18, $AR12$ is generated at B3 by combining $AR1$ and $AR2$:

$$AR12 \Leftarrow AR1 \otimes AR2.$$

(Sometimes, more than two aliases are combined to generate a new alias, an example of which is given in Section 6.7.2.)

As mentioned earlier, a valid alias is one that results from a combination of aliases that can occur in the same execution instance. This section defines successive approximations of valid. Each approximation is more constraining than the previous.

A combination $AR12 \Leftarrow AR1 \otimes AR2$ is a *valid* combination at block B3 only if:

(1) $AR1$ and $AR2$ (are computed to) hold immediately before B3; and

(2) the semantics of B3 requires the combination of $AR1$ and $AR2$.

As these are the minimum conditions required for the combination of aliases by any non-trivial pointer alias analysis, we will assume in this section that these conditions are already satisfied whenever aliases are combined.[13]

We associate with each alias the unique program point (basic block) where it is generated, which we call the *birth site* of the alias.[14] For example, B1 is the birth site of both $AR1$ and $AR2$ in Figure 18. We use $birth\text{–}site(AR)$ to denote the birth site of alias relation AR. Since different program points can generate the same alias relation, we define an *alias instance* [Choi et al. 1993] as follows:

$$AI = [AR, \mathbf{BirthSite}],$$

where AR is an alias relation, and $BirthSite$ is $birth-site(AR)$. As such, a set of alias instances with different birth sites is defined for each alias relation holding at a program point in a routine. We also use $birth-site(AI)$ to denote the birth site component of alias instance AI. The birth site of an alias instance is the birth site of its alias relation. The birth site of an alias instance represents the generation-path information of the alias.

As each alias instance is defined with respect to an alias relation, an alias instance is generated or propagated when its alias relation is generated or propagated, respectively. Therefore, alias instances are combined at a program point to generate a new alias instance, written as:

$$AI_{ij} \Leftarrow AI_i \otimes AI_j$$

[13] New aliases can also be generated at the join nodes of the CFG with graph-based representations when aliases are merged [Chase et al. 1990].

[14] The entry node is designated as the generation point for aliases propagated into a called procedure and the call site is designated as the generation point for aliases propagated from a called procedure.

when alias relations of AI_i and AI_j are combined. A combination of two alias instances is *valid* if and only if the combination of the corresponding alias relations is valid.

The number of alias instances holding at a program point grows as the number of alias relations grows during the analysis. When a fixed point is reached, the alias relations holding at each program point are those of the alias instances holding at that point. Since the alias relation of each alias instance is uniquely defined, we will use "alias" for "alias instance" when the meaning is clear from the context.

6.3 Birth Sites and Forward Path Problem

This section describes how alias instances can improve precision with respect to the forward path problem by using the birth-site information described in the previous section to identify invalid combinations of aliases. Although limited, this simple method still improves precision over previous approaches. Later, we describe extensions of alias instances to include approximate information of the paths over which alias instances are generated and propagated. These extensions can still fail to identify invalid combinations of aliases unless full path information is kept. Appendix B provides an example of such a situation.

With alias instances tagged with their birth sites, we extend the definition of a valid combination. The combination

$$AI_{ij} \Leftarrow AI_i \otimes AI_j$$

at a block $\mathsf{B_k}$ is *valid* only if there exists a path, P_{ijk} or P_{jik}, over the CFG, such that

$$\begin{aligned} P_{ijk} &: birth{-}site(AI_i) \xrightarrow{*} birth{-}site(AI_j) \xrightarrow{*} \mathsf{B_k}, \text{ or} \\ P_{jik} &: birth{-}site(AI_j) \xrightarrow{*} birth{-}site(AI_i) \xrightarrow{*} \mathsf{B_k}, \end{aligned} \tag{17}$$

where $\xrightarrow{*}$ is a path of length 0 or more. Because AI_j and AI_i hold immediately before $\mathsf{B_k}$, an execution path $P_{jk} : birth{-}site(AI_j) \xrightarrow{*} \mathsf{B_k}$, and an execution path $P_{ik} : birth{-}site(AI_i) \xrightarrow{*} \mathsf{B_k}$ will always exist. Thus, Condition 17 simplifies to:

$$\begin{aligned} P_{ijk} &: birth{-}site(AI_i) \xrightarrow{*} birth{-}site(AI_j), \text{ or} \\ P_{jik} &: birth{-}site(AI_j) \xrightarrow{*} birth{-}site(AI_i) \end{aligned} \tag{18}$$

i.e., there is a path between $birth{-}site(AI_i)$ and $birth{-}site(AI_j)$. To formalize this condition, we first define the transitive relation *any–path* for nodes N_i and N_j of a graph as follows:

$$any{-}path(N_i, N_j) \stackrel{def}{=} (\exists P_{ij} : N_i \xrightarrow{*} N_j) \vee (\exists P_{ji} : N_j \xrightarrow{*} N_i). \tag{19}$$

$any{-}path(N_i, N_j)$ indicates whether there exists a path between N_i and N_j. We use *any–path* to define the transitive relation $alias{-}birth{-}path$ as follows:

$$alias{-}birth{-}path(AI_i, AI_j) \stackrel{def}{=} any{-}path(birth{-}site(AI_i), birth{-}site(AI_j)) \tag{20}$$

With this, Condition 18 can be expressed as:

$$alias{-}birth{-}path(AI_i, AI_j) \text{ holds.} \tag{21}$$

$alias\text{–}birth\text{–}path(AI_i, AI_j)$ holds if there exists a path between the birth sites of AI_i and AI_j. $alias\text{–}birth\text{–}path(AI_i, AI_j)$ captures the propagation-path information between the birth sites of AI_i and AI_j. This information can be precomputed for all the potential birth sites of aliases in $O((P \times (S_p)^K)$ time [Eve and Kurki-Suonio 1977], $2 \leq K \leq 3$, where P is the number of the procedures in the program, and S_p is the largest number of pointer-assignment statements in a procedure, using the sparse evaluation graph, instead of the CFG, as the intraprocedural program representation. The computed information incurs $O(P \times (S_p)^2)$ space, and can be stored in a table, which we call the *path table*. Once stored, the information in the path table can be retrieved in constant time during alias analysis.

Consider the code segment in Figure 18 (as a CFG), which illustrates the use of $alias - birth - path$ information to improve precision with respect to the forward path problem. In B3, we identify the combination of alias instances of $AR1$ and $AR2$ as valid, since $alias - birth - path(AI1, AI2)$ is true, where $AI1$ and $AI2$ are, respectively, the alias instances of $AR1$ and $AR2$ holding in B3. However, we identify $AI1 \otimes AI4$ as invalid, since $alias\text{–}birth\text{–}path(AI1, AI4)$ is not true: there is no execution path between $birth\text{–}site(AI1)$ (i.e., B1) and $birth\text{–}site(AI4)$ (i.e., B2).[15] In the above example, $AI4$ is the alias instance of $AR4$, holding in B3.

When $birth\text{–}site(AI_i)$ and $birth\text{–}site(AI_j)$ belong to different procedures, $alias\text{–}birth - path(AI_i, AI_j)$ is defined to be true since all procedures connected in the PCG are reachable from all other procedures when the bi-directional nature of the PCG is considered. Thus, birth sites cannot help with the interprocedural forward path problem.

6.4 Call Sites and Forward/Backward Path Problems

The interprocedural forward path problem results from combining aliases propagated from two distinct call sites. As an example, assume again that the graph in Figure 18 is a PCG, where each node represents a procedure. In procedure $B3$, aliases $AR1$ and $AR3$, or $AR2$ and $AR4$, should not be combined since they are propagated along different call paths. However, as stated in the previous section, birth sites cannot be used for the interprocedural forward path problem.

To address the interprocedural forward path problem, we associate with each alias instance the most recent call site of the execution path along which the alias is propagated:

$$AI = [AR, BirthSite, \mathbf{CallSite}].$$

Associating each alias instance with a call site this way is similar to previous interprocedural data flow analyses [Sharir and Pnueli 1981; Jones and Muchnick 1982] that distinguish information propagated along different interprocedural paths. In this section, we describe how to utilize the call site information for computing more precise alias instances: for detecting an invalid combination of alias instances, and also for determining the correct call site(s) to propagate an alias instance backward from a callee to its caller(s).

We use $call\text{–}site(AI)$ to denote the call site component of alias instance AI. The call site of an alias instance approximates the propagation path information of the

[15] We assume the completed graph does not contain a path between B1 and B2.

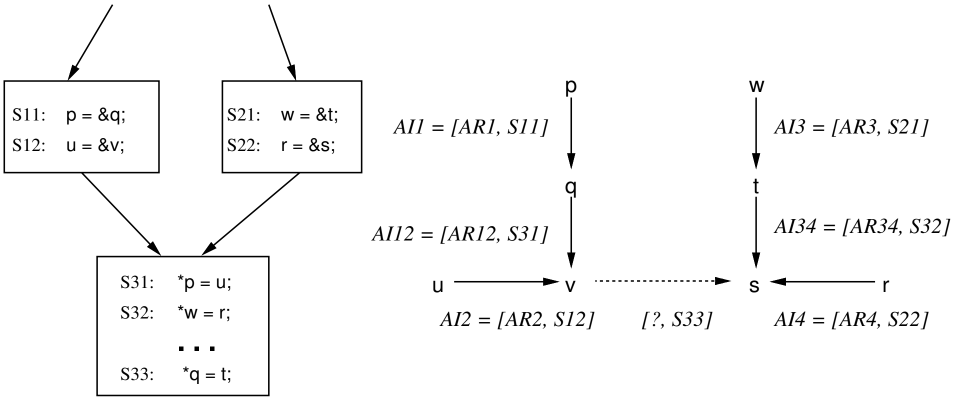

Fig. 20. Example Code Segment and Its Alias Graph

alias. The call site of an alias instance is determined as follows:

(1) An alias instance propagated along call site C_i to the entry of the invoked procedure, has C_i as its call site.

(2) An alias instance generated without combining any aliases has *null* as its call site. An example is an alias relation $\langle *p, q\rangle$ generated from statement p = &q.

(3) The combination $AI_{ij} \Leftarrow AI_i \otimes AI_j$ is *valid* only if the call sites of AI_i and AI_j are identical or at least one of them is *null*. In this case, we say their call sites are *equivalent*, and call the identical call site or the non-*null* call site the *equivalent call site* of the two, which becomes the call site of AI_{ij}.

The third case above extends the combining rule (Equation 21) using call sites, and thereby improves precision in handling the interprocedural forward path problem.

The call sites of alias instances also help improve the precision in handling the backward path problem by the following backward propagation rule:

RULE 1. An alias instance with a *null* call site holding at the exit of a procedure need be propagated to all the call sites of the procedure. An alias instance with call site C_i should be propagated only back to C_i.

In the example of Figure 18 (viewed as a PCG), $AI1$ and $AI2$, corresponding to $AR1$ and $AR2$ respectively, are propagated from $B1$ to $B3$. They have the corresponding call site as their call sites and are propagated back from the exit of $B3$ to $B1$, but not to $B2$. Also, $AI12$, generated by combining $AI1$ and $AI2$ in $B3$, has the same call site and is propagated correctly back to $B1$. Similarly, $AI3$, $AI4$, and $AI34$ are correctly propagated back to $B2$, but not $B1$. The call sites of $AI1$, $AI2$, $AI3$, and $AI4$ propagated back from $B3$ are those of the original alias instances propagated to $B3$. $AI12$ and $AI34$, however, do not have any corresponding original alias instances in $B1$ or $B2$, and have *null* as their call sites. The next section describes a technique that further improves precision by correctly identifying the call sites to be associated with $AI12$ and $AI34$.

6.5 Birth Site and Source Alias Sets

Birth-site information provides a limited form of alias generation history, i.e., of the generation-path information of an alias. This section addresses the issue of expanding alias generation-path information to improve the precision of alias analysis with respect to the forward path problem.

Consider Figure 20, which shows an example CFG with code segments, and the alias graph holding immediately after S33. The alias relation represented by the dotted arrow from v to s, $\langle *v, s\rangle$, is generated by combining $AI12$ and $AI34$ at S33: $[\langle *v, s\rangle, S33] \Leftarrow AI12 \otimes AI34$. Since there is an execution path from $birth-site(AI12)$, (S31), to $birth-site(AI34)$, (S32), this combination will not be detected as invalid by the method described in the previous sections. However, $AI12$ is valid only along an execution path on which both $AI1$ and $AI2$ are also valid (when the basic block of S31 is entered from that of S12), and $AI34$ is valid only along an execution path on which both $AI3$ and $AI4$ are also valid (when the basic block of S31 is entered from that of S22). Because the alias $[\langle *v, s\rangle, \mathrm{S33}]$ ($\Leftarrow AI12 \otimes AI34$) is valid only along an execution path on which both $AI12$ and $AI34$ are valid, $[\langle *v, s\rangle, \mathrm{S33}]$ is valid only along an execution path on which all of $AI1$, $AI2$, $AI3$, and $AI4$ are valid. As there is no such execution path leading to S33, $[\langle *v, s\rangle, \mathrm{S33}]$ is not a valid alias.

This example illustrates the conservativeness of a method that only requires the birth sites of aliases to be on a common execution path in order for them to be combined to generate a new alias.[16] From the example, it is clear that not only birth sites of aliases, but also the birth sites of the aliases on which these aliases depend, must be on the same execution path; we need to factor in this generation-path information. For this, we extend the definition of an alias instance AI to include the set of aliases called the *source alias set* (SAS), which is a collection of alias instances that combine to generate AI at a statement, as follows:

$$AI = [AR, BirthSite, CallSite, \mathbf{SAS}].$$

Alias instances generated independently of other aliases, such as $\langle *p, q\rangle$ at S11, have empty ($\{\}$) source alias sets. (They also have *null* call sites.) For $AI_{1k} \Leftarrow AI_1 \otimes \ldots \otimes AI_k$, we define the source alias set as follows:

$$SAS(AI_{1k}) = \{\ \&AI_1, \ldots, \&AI_k\},$$

where $\&AI_k$ is a *reference to* the alias instance AI_k. With source alias sets thus defined, the alias instances form a directed graph, called the *alias instance graph*, where a node represents an alias instance and an edge from node N_i to node N_j represents that N_j is in the source alias set of N_i.

As the elements in a source alias set are always references to, not actual copies of, alias instances, we drop the & notation from the source alias sets. With this, $SAS(AI_{1k}) = \{AI_1, \ldots, AI_k\}$.

We define the *alias history set* ($History$)[17] of an alias instance recursively as

[16] Call site information is irrelevant in this example because the aliases have the same *null* call site.

[17] Benjamin Cooper first defined the term *alias histories* in [Cooper 1989] to address a different path problem.

follows:

$$History(AI_{1k}) = \{AI_1, \ldots, AI_k\} \cup History(AI_1) \ldots \cup History(AI_k), \qquad (22)$$

where $SAS(AI_{1k}) = \{AI_1, \ldots, AI_k\}$. The alias history set (history set, for short) of an alias instance represents, in an approximate manner, the execution path over the CFG and PCG that has generated and/or propagated the alias instance.

With alias instances thus extended, we augment the valid combination condition (Condition 21 in Section 6.3) as follows:

$$\begin{aligned} AI_{ij} \Leftarrow\ & AI_i \otimes AI_j \text{ only if} \\ & alias{-}birth{-}path(AI_i, AI_j) \text{ holds}, \\ & AI_i \text{ and } AI_j \text{ have equivalent call sites, and} \\ & \forall a \in History(AI_i) \text{ and } b \in History(AI_j), \\ & \qquad alias{-}birth{-}path(a, b) \text{ holds} \end{aligned} \qquad (23)$$

Condition 23 can be further extended similarly to the call sites of alias instances in $History(AI_i)$ and $History(AI_j)$. Alias instances in these history sets can be categorized in two sets: the *intraprocedural* ones whose birth sites belong to the current procedure where AI_i and AI_j are combined, and the *interprocedural* ones whose birth sites belong to other procedures. The intraprocedural alias instances have call sites that are equivalent to both AI_i and AI_j, and an extension of Condition 23 to call sites of intraprocedural alias instances is redundant.

Extension of Condition 23 to interprocedural alias instances is equivalent in effect to capturing multiple levels of call sites, and can help identify invalid combinations of aliases propagated along different call paths. We call this the *eager invalidation* of interprocedural aliases. In Section 6.6, we describe an alternative, called the *lazy invalidation* of interprocedural aliases, which is more conservative but can be more efficient than the eager invalidation.

For the example in Figure 20, we will have the following alias instances:

$$\begin{aligned} AI1 &= [AR1, S11, null, \{\}] \\ AI2 &= [AR2, S12, null, \{\}] \\ AI3 &= [AR3, S21, null, \{\}] \\ AI4 &= [AR4, S22, null, \{\}] \\ AI12 &= [AR12, S31, null, \{AI1, AI2\}] \\ AI34 &= [AR34, S32, null, \{AI3, AI4\}] \end{aligned}$$

With this additional information we can identify $AI12 \otimes AI34$ as being invalid, because the last condition of Condition 23 does not hold with $History(AI_i)(= \{AI1, AI2\})$ and $History(AI_j)(= \{AI3, AI4\})$.

In Section 6.7.3 we show that the last condition of Condition 23 is equivalent to the existence of a path that covers all the birth sites of the alias instances in $History(AI_i)$ and $History(AI_j)$.

The size of a history set can be the same as the total number of alias instances during the alias analysis. The time efficiency of the analysis can be improved, at the cost of reduced precision, by limiting the total number of alias instances. A more detailed description of this will be given in Section 6.8.

6.6 Call Site and Source Alias Sets

As stated before, source alias sets can also improve precision with respect to the backward path problem. Consider Figure 18 again as a PCG with each node representing a procedure. We denote the edge from procedure $B1$ to procedure $B3$ as C_1, and the edge from $B2$ to $B3$ as C_2. Two alias instances generated in $B1$ are

$$AI_1 = \{AR1, BS_1, null, \{\}\} \text{ and } AI_2 = \{AR2, BS_2, null, \{\}\}, \text{ for some } BS_1, BS_2.$$

Upon being propagated to $B3$, they generate the following at the entry of $B3$:

$$\widehat{AI_1} = \{AR1, entry, C_1, \{AI_1\}\} \text{ and } \widehat{AI_2} = \{AR2, entry, C_1, \{AI_2\}\}.$$

Likewise, the following two alias instances generated in $B2$:

$$AI_3 = \{AR3, BS_3, null, \{\}\} \text{ and } AI_4 = \{AR4, BS_4, null, \{\}\}, \text{ for some } BS_3, BS_4.$$

generate the following at the entry of $B3$:

$$\widehat{AI_3} = \{AR3, entry, C_2, \{AI_3\}\} \text{ and } \widehat{AI_4} = \{AR4, entry, C_2, \{AI_4\}\}.$$

$AI_{12} \Leftarrow \widehat{AI_1} \otimes \widehat{AI_2}$ is $\{AR12, BS_{12}, C_1, \{\widehat{AI_1}, \widehat{AI_2}\}\}$, and $AI_{34} \Leftarrow \widehat{AI_3} \otimes \widehat{AI_4}$ is $\{AR34, BS_{34}, C_2, \{\widehat{AI_3}, \widehat{AI_4}\}\}$. At the exit of $B3$, six alias instances hold: $\widehat{AI_1}$, $\widehat{AI_2}$, $\widehat{AI_3}$, $\widehat{AI_4}$, AI_{12}, and AI_{34}. Among them, $\widehat{AI_1}$ and $\widehat{AI_2}$ ($\widehat{AI_3}$ and $\widehat{AI_4}$) indicate that AI_1 and AI_2 (AI_3 and AI_4) are not killed by $B3$ and should be regarded as holding immediately after the corresponding call site C_1 (C_2) in $B1$ ($B2$). AI_{12} is a newly generated alias and is propagated to C_1, which is its call site. There, it will be converted into $AI_{12} = \{AR12, C_1, C_m, \{AI_1, AI_2\}\}$, where C_m is the new call site determined by examining AI_1 and AI_2.[18] In this case, AI_1 and AI_2 have a compatible call site, $null$, and thus C_m is $null$ by rule 3.

Assume, however, that AI_1 and AI_2 do not have compatible call sites: AI_1 and AI_2 might have been propagated from two different call sites to $B1$. In that case, we would identify AI_{12} as an invalid alias in $B1$. Note that there can be multiple levels of calls between the procedure where AI_{12} is generated ($B3$ in this case) and the procedure where AI_{12} is found to be not valid by this mechanism.

When an alias instance is found not to be valid by this mechanism, the source alias set information can be used again to identify all the (invalid) alias instances that have been generated with the assumption that this alias is valid, by recursively identifying the alias instances that have this alias in their (transitive) source alias sets. This process of identifying invalid aliases that have been assumed to be valid is similar to computing the history set of an alias instance except that this time the alias instance graph is traversed from an alias to the set of aliases that have the former in their source alias sets.

We call this lazy invalidation, in contrast to the eager invalidation described in Section 6.5. Lazy invalidation works only when an invalid alias generated in a called procedure gets propagated to a calling procedure without being killed in the called procedure. If it gets killed in the called procedure, it cannot trigger the lazy invalidation process.

[18] Note that AI_{12} in the context of $B1$ has the call site C_1 as its birth site.

S_1: x = &y;
S_2: q = &x;
S_3: p = q;
S_4: x = &z;

Fig. 21. Compact Representation Improves Precision

6.7 Precision of the Compact Representation

The compact representation (Section 3) requires less storage and also can be more precise in the presence of alias kills than the full representation. However, it can be less precise than the full representation when aliases are merged at a join node of the PCG or CFG [Marlowe et al. 1993]. This section illustrates this trade-off in precision and describes how to improve the precision of the compact representation by using techniques from the previous sections.

Once again, consider the alias graph in Figure 19, which illustrates the alias information kept by compact and graph-based representations. In this graph, alias relation $\langle **p, v\rangle$ holding after $B3$ can be inferred from alias relations $\langle *p, q\rangle$ and $\langle *q, v\rangle$ holding at the same place. By contrast, a full representation would represent $\langle **\,p, v\rangle$ explicitly [Landi and Ryder 1992; Deutsch 1992; 1994; Sagiv et al. 1990]. Recall from Section 3 that identifying aliases of $**p$ after $B3$ with a compact representation is equivalent to finding all (alias) paths of length two starting at p over the alias graph. (In this section we first use alias relations, instead of alias instances, to facilitate comparisons of different methods.)

Traversing the alias graph to find paths of length n occurs when aliases of an access path with dereference level n need be computed during alias analysis or when alias information is factored into data flow information [Burke and Choi 1992; Cytron and Gershbein 1993]. Traversing the alias graph along the aliases $\langle *p, q\rangle$ and $\langle *q, v\rangle$ in computing the aliases of $**p$ can be regarded as *implicitly* combining these aliases. We distinguish a combination based on the alias representation from an explicit combination based on the program by using the operator, $\hat{\otimes}$, rather than $\otimes$, i.e.,

$$\langle **p, v\rangle \Leftarrow \langle *p, q\rangle \hat{\otimes} \langle *q, v\rangle.$$

6.7.1 *Compact Representation Yields More Precision.* Consider the program segment of Figure 21. Assuming no aliases hold before S_1, $\langle *x, y\rangle$ is the only alias holding after S_1. After S_3, the full representation of alias relations holding are as follows:

$$\begin{aligned}\{ \quad & \langle *x, y\rangle, \langle *p, x\rangle, \langle *q, x\rangle, \langle *p, *q\rangle, \\ & \langle **p, y\rangle, \langle **q, y\rangle, \langle **p, *x\rangle, \langle **q, *x\rangle, \\ & \langle **p, **q\rangle\},\end{aligned} \tag{24}$$

all of which can be represented by the following compact representation:

$$\{\langle *p, x\rangle, \langle *q, x\rangle, \langle *x, y\rangle\}. \tag{25}$$

After S_4, the compact representation becomes:

$$\{\langle *p, x\rangle, \langle *q, x\rangle, \langle *x, z\rangle\}, \tag{26}$$

from which the full information can be computed:

$$\begin{aligned} \{ \quad & \langle *x, z\rangle, \langle *p, x\rangle, \langle *q, x\rangle, \langle *p, *q\rangle, \\ & \langle **p, z\rangle, \langle **q, z\rangle, \langle **p, *x\rangle, \langle **q, *x\rangle, \\ & \langle **p, **q\rangle\}. \end{aligned} \tag{27}$$

The difference between (24) and (27) is that every access path y in (24) has been replaced by z in (27).

However, the full alias information computed by applying S_4 directly to (24) is as follows:

$$\begin{aligned} \{ \quad & \langle *x, z\rangle, \langle *p, x\rangle, \langle *q, x\rangle, \langle *p, *q\rangle, \\ & \langle **p, z\rangle, \langle **q, z\rangle, \langle **p, *x\rangle, \langle **q, *x\rangle, \\ & \langle **p, **q\rangle, \langle **p, y\rangle, \langle **q, y\rangle \quad \}, \end{aligned} \tag{28}$$

which contains additional alias relations (28), which are carried over from (24).[19] These alias relations originate from $\langle *x, y\rangle$ at S_1 and $\{\langle *p, x\rangle, \langle *q, x\rangle\}$ at S_3. They are carried over after S_4, although S_4 kills $\langle *x, y\rangle$. Therefore, the compact representation not only improves the storage efficiency of the analysis, but can improve the precision as well. In general, the compact representation improves precision when one of the base relations that enabled derivation of an alias relation is killed and the derived alias is not.

Note that the same improved precision can also be obtained with the full representation by keeping must-alias information: if we keep the information that $\langle *p, x\rangle$ and $\langle *q, x\rangle$ are must-aliases, we can regard $*p$ and $*q$ as being modified at S_4, and can apply the killing rule to them as well as x.

Figure 22 shows an analogue of Figure 21 for flow-sensitive interprocedural analysis. The improved precision of the compact representation relies on the analysis recognizing the assignment in SubC as killing the alias relation $\langle *x, y\rangle$.

6.7.2 *Compact Representation Yields Less Precision.* The compact representation can also reduce the precision of the analysis [Marlowe et al. 1993]. Consider the program segment in Figure 23. With the compact representation, two implicit combinations of aliases, $\langle *p, q\rangle\hat{\otimes}\langle *q, z\rangle$ and $\langle *p, q\rangle\hat{\otimes}\langle *q, y\rangle$, are performed at S27 to identify z and y as the aliases of $**p$ holding at S27. From these implicit combinations, each coupled by an *explicit combination* with $\langle *x, w\rangle$ at S27, we derive two new aliases: $\langle *z, w\rangle(\Leftarrow \langle **p, z\rangle \otimes \langle *x, w\rangle)$ and $\langle *y, w\rangle(\Leftarrow \langle **p, y\rangle \otimes \langle *x, w\rangle)$. The implicit combination $\langle **p, y\rangle \Leftarrow \langle *p, q\rangle\hat{\otimes}\langle *q, y\rangle$, however, is invalid because $\langle *p, q\rangle$ and $\langle *q, y\rangle$ cannot occur along the same executable path, and thus $\langle *y, w\rangle$ is also invalid.

With a full alias representation, an implicit combination is performed explicitly on each control flow path, before the alias relations holding on each are merged.

[19] Note that $\langle **p, *x\rangle$ and $\langle **q, *x\rangle$ in (28) are implicit aliases derived from $\langle *p, x\rangle$ and $\langle *q, x\rangle$, respectively.

```
main() {
    x = &y;
    SubB();
}

SubB() {
    q = &x;
    p = q;
    SubC();
}

SubC() {
    x = &z;
}
```

Fig. 22. Interprocedural Analogue for Figure 21

```
S20:  x = &w;
S21:  q = &z;
S22:  if (...)
S23:      p = &q;
S24:  else
S25:      q = &y;
S26:  ...
S27:  **p = x;
```

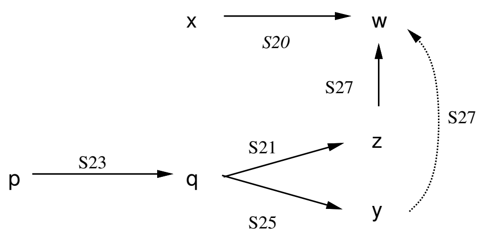

Fig. 23. Compact Representation Worsens Precision

Therefore, with a full representation, the set of aliases computed to hold at S27 is

$$\{\langle *x, w\rangle, \langle *q, z\rangle, \langle *q, y\rangle, \langle *p, q\rangle, \langle **p, z\rangle\},$$

from which only $\langle *z, w\rangle$ will be computed as a new alias at S27. With the compact representation, the implicit combination is deferred until after the join point, thereby coupling alias relations generated along different paths, and in effect losing control flow path information.

As illustrated in Section 6.1, a loss of control flow information can occur independently of the compact representation. As shown in Figure 18, the explicit combination of aliases such as $\langle *u, v\rangle \otimes \langle *p, t\rangle$ after join points of control flow can generally result in a loss of precision, unless full path information is kept.

In previous sections, we have shown how birth sites and history sets can reduce the loss of precision for explicit combinations of aliases without keeping full path information. In the following section, we describe how these techniques can also reduce the loss of precision for implicit combinations of aliases after the join points of control paths.

6.7.3 *Improving the Precision of the Compact Representation.* In Section 6.3 we described techniques to identify invalid combinations of aliases. This section shows how similar techniques can improve the precision of compact alias representations [Choi et al. 1993; Emami et al. 1994] and graph-based representations [Jones and Muchnick 1981; Larus and Hilfinger 1988; Ruggieri and Murtagh 1988; Horwitz et al. 1989; Chase et al. 1990]. When only birth sites are considered, the cost of this technique is quadratic in the maximum level of pointer dereferencing, which in practice is a small constant. The precision can be further improved, at increased cost, by also considering call sites and alias history sets.

The compact and graph-based representations can lose precision when aliases that cannot occur along an execution path are implicitly combined. To avoid such *implicit* combinations that are invalid, we can apply the same techniques we have already applied to identify invalid *explicit* combinations: i.e., the test in Equation (23).

Formally, a path $P_A : AI_1 \circ AI_2 \circ \ldots \circ AI_N$ of alias instances of an alias graph is a valid path only if their call sites are equivalent and there exists a path in the CFG over the birth-site components in $History(AI_1) \cup History(AI_2) \ldots \cup History(AI_N)$. Naively applying this condition to a set of N nodes (birth-site components in the alias history sets) can incur cost exponential in the number of nodes. However, an $O(N^2)$ algorithm can be derived from the following observation:

> Let P be a path of length $k > 0$ in a directed alias graph. There exists a path containing all nodes of P and another node N_{k+1} *if and only if* $any\!-\!path(N, N_{k+1})$ is true for all nodes, N, on P.

This observation is used in Figure 24, which shows a simple enhancement of the ComputeAliases algorithm of Section 3. This algorithm uses alias instances to verify valid implicit combinations of aliases over the alias graph. A similar enhancement can be applied to the version presented in Section 4.2.4.

```
    /* Global Variables */
solution : Set of Variables

function ComputeAliases(Var, DerefLevel)
begin
    solution ← {}
    call Expand(Var, DerefLevel, {})
    return solution
end ComputeAliases

procedure Expand(Var, DerefLevel, PathSet)
begin
    if DerefLevel = 0 then
        solution ← solution ∪ {Var}
    else
        foreach alias instance edge, e = (Var, Target), loop
            if e ⊗̂ e_k is valid using equation 23, for all e_k ∈ PathSet then
                call Expand(Target, DerefLevel - 1, PathSet ∪ {e});
            end if
        end loop
    end if
end Expand
```

Fig. 24. Alias Query Algorithm for Improved Precision

6.8 Efficiency Issues

In the previous sections we presented techniques to identify invalid aliases resulting from both generating and propagating aliases. The source alias set of an alias captures the set of program points that result in the generation and/or propagation of the alias. The total number of alias instances for a particular alias relation holding in a procedure p is bounded by $num(BirthSites) \times num(CallSites) \times num(SAS)$, where $num(BirthSites)$, $num(CallSites)$, and $num(SAS)$ are the numbers of birth sites in p, call sites invoking p, and source alias sets, respectively. The number of source alias sets is linear in the number of alias instances, which can be exponential in the size of the program. This section first defines a property that allows the elimination of some alias instances for the same alias relation without a loss of precision. Next, it describes how to limit the size of the source alias set component of an alias instance, with a potential loss of precision. Finally, it describes how to approximate call sites and birth sites to further reduce the number of alias instances for the same alias relation, with a potential loss of precision.

6.8.1 *Summarizing Alias Instances.* We define a property between two program points that will enable the elimination of some alias instances without a loss of precision:

Definition 1. A program point P_i *subsumes* a program point P_j if for any program point $\hat{P}$, $any\!-\!path(P_i, \hat{P}) \Longrightarrow any\!-\!path(P_j, \hat{P})$.

Some examples of where P_i subsumes P_j are if P_j dominates P_i, or if P_j post-dominates P_i.

```
    if (...) {
S1:     p = &q;
S2:     r = &s;
    }
S3: *p = r;
```

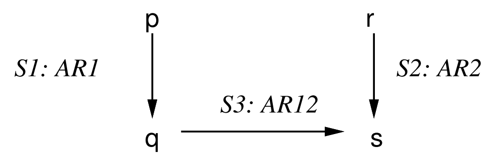

Fig. 25. Example Code Segment and its Alias Graph

Consider two alias instances containing the same alias relation, source alias sets, and equivalent call sites, but with different birth sites. If one birth site subsumes the other, the alias instance with the subsumed birth site can be removed without a loss of precision.

6.8.2 *Approximating Alias Instances.* We now consider techniques that can improve time and space efficiency with a potential loss of precision. These techniques can reduce the size and number of alias instances. Consider the example in Figure 25. The alias instance

$$AI_{12} = [\langle *q, s\rangle, \mathrm{S3}, C_1, \{AI_1, AI_2\}], \text{ for some } C_1$$

holds after S3. AI_{12} is valid only along a path where both AI_1 and AI_2 are valid. Entries in a source alias set impose a *conjunctive* condition: a subset of a source alias set imposes fewer restrictions than the full source alias set. Thus, a correct approximation can be made by limiting the size of source alias sets for a set of alias instances that have the same alias relation, equivalent birth sites, and equivalent call sites. In this example, we could substitute $\{AI_1\}$, $\{AI_2\}$, or $\{\}$ for the source alias set $\{AI_1, AI_2\}$ in AI_{12}.

Although each alias instance has one birth site and one call site, the same alias relation may exist in alias instances with different birth sites and/or call sites. Since such instances impose a *disjunctive* condition, simply eliminating some would result in incorrect analysis.

However, the *null* call site of an alias instance imposes the least restriction in inferring the interprocedural context of the alias instance. An alias instance with *null* call site is considered to be created in the current procedure and needs be propagated to *all* call sites which invoke the current procedure. In this sense, *null* call site is "equivalent" to all call sites, which is the most conservative in the interprocedural context, and can always replace any non-*null* call sites in the alias instances without affecting correctness, at the cost of precision.

To further reduce the number of alias instances with the same alias relation, but different birth sites, beyond the property mentioned in Section 6.8.1, one can use the entry point of a procedure. This point has a path to every program point in the procedure, and can replace other birth sites in the alias instances without affecting correctness, at the cost of precision.

6.8.3 *Limiting the Alias Instances Considered.* The previous techniques improve time and space efficiency by reducing the size and number of alias instances in the program. A second level of approximation of alias instances can be achieved by once again utilizing the conjunctive property of source alias sets when evaluating Condition 23. Such a method would improve time efficiency by limiting the number of history sets visited.

7. PARAMETERS AND LOCAL VARIABLES

This section presents a method for handling parameters and local variables. For incorporating parameters into the alias computation, we describe two functions which map alias sets to alias sets. The first function, $ForwardBind_c$, captures the effects of propagating aliases forward over the PCG from a call site, c, to the called procedure. The second function, $BackwardBind_c$, captures the effects of propagating aliases backward over the PCG from a called procedure to the call site c. These functions are used in the interprocedural algorithms of Section 4. Unless stated otherwise, the functions $ForwardBind_c$ and $BackwardBind_c$ include their input aliases in their result, i.e., $x \in ForwardBind_c(x)$ and $y \in BackwardBind_c(y)$.

This section assumes that the compact representation of Section 3 is used to represent alias information. It also assumes that structural alias information is available. Alias instances that result from structural aliases have empty birth site and history components, and are used to initialize $Entry$ as described in Section 4. First, this section describes methods for incorporating call-by-value and reference parameters into the alias computation. It then describes how to correctly handle interactions between local variables and the alias computation in the presence of recursion.

7.1 Call-by-Value/Copy-in-Copy-out Parameters

Let a_i and f_i be the ith actual and formal parameters corresponding to a call site c that invokes P. With call-by-value (or copy-in) parameters, the only case to consider is when the value contained in an actual parameter (say a_i) is the address of an object (say p):

$$AR_{IN} = \langle *a_i, p \rangle,$$

where p also can be an actual parameter. In this case, f_i gets the value of a_i, and $ForwardBind_c(AR_{IN}) = \langle *f_i, p \rangle$, and

$$ForwardBind_c(AI_{IN}) = [\langle *f_i, p \rangle, BirthSite, History]$$

where the $BirthSite$ and $History$ can be computed by considering the call site as an assignment statement of actual parameter a_i to formal parameter f_i.

With call-by-value parameters, all the alias relations whose access paths contain variables local to the called procedure can be ignored upon its termination unless the calling procedure is also reachable from the called procedure: i.e., there is a recursion. Therefore, $BackwardBind_c$ is defined so that alias instances with such alias relations, in the absence of a recursion involving the calling and the called procedures, are not propagated back to the call sites of the calling procedure. Recursively called procedures are identified by detecting cycles in the call graph.

With copy-out parameters, alias relations of the form $\langle *f_i, p \rangle$ are converted by $BackwardBind_c$ into $\langle *a_i, p \rangle$, where p is not a local variable of the called procedure. We model values returned by functions in a manner similar to copy-out parameters, where the formal parameter is the return value, and the actual parameter is the variable on the left side of the assignment statement at the call site.

7.2 Reference Parameters

We define $ap(v)$ for a variable v to denote an access path including v that is eligible to be a part of a compact representation of an alias relation: $ap(v)$ is either v or $*v$. An alias relation, AR, can always be expressed as $\langle ap_1(v_1), ap_2(v_2)\rangle$ for some variables (names) v_1 and v_2. For example, $\langle *q, p\rangle$ can be expressed as $\langle ap_1(q), ap_2(p)\rangle$, where $ap_1(q) = *q$ and $ap_2(p) = p$.

$ForwardBind_c(IN_c)$ is described by considering an alias instance, $AI_{IN} \in IN_c$, where $AI_{IN} = [AR_{IN}, BirthSite, History]$. There are three possibilities to consider for AR_{IN}:

(1) $\langle ap_i(a_i), ap_j(a_j)\rangle$,
(2) $\langle ap_k(a_i), ap_l(p)\rangle$,
(3) $\langle ap_m(q), ap_n(r)\rangle$,

where p, q, and r are not actual parameters of call site c. Among these we consider only the first two cases, which involve parameters.

Applying $ForwardBind_c$ to each of the first two cases generates the following additional sets of alias relations, respectively:

(1) $\{\langle ap_i(f_i), ap_j(a_j)\rangle, \langle ap_i(a_i), ap_j(f_j)\rangle, \langle ap_i(f_i), ap_j(f_j)\rangle\}$,
(2) $\{\langle ap_k(f_i), ap_l(p)\rangle\}$

Each of these results in the following set of alias instances

$$\{[AR, BirthSite, History] \mid AR \in ForwardBind_c(AR_{IN})\},$$

which will be added in $Entry_P$ and where the $BirthSite$ and $History$ are computed as in section 7.1.

In computing $ForwardBind_c$, aliases that contain access paths not reachable in the called procedure P are not explicitly removed because such aliases may become reachable aliases of a procedure (transitively) called by P [Burke and Cytron 1986].[20] These aliases can be removed after the interprocedural alias analysis is done. However, the alias relations whose access paths contain only local variables of the called procedure can be ignored upon its termination when such storage is reclaimed unless the calling and the called procedures are involved in a recursion. In this context local variables do not include the formal parameters of the procedure.

To compute $BackwardBind_c(AI_{Exit})$: convert each reference to f_i in the alias instances in AI_{Exit} into that of a_i, where AI_{Exit} is the alias set computed to hold at the exit of P. This step can generate multiple identical alias instances, of which only one need be kept.

7.3 Local Variables

During the lifetime of a program, a global variable name designates a unique object. However, in the presence of recursion the name of a local variable or call-by-value parameter can designate multiple objects in the run-time stack at a given time [Larus

[20] This can occur due to the scoping rules of the language, such as invisible common blocks in Fortran 90, file scope variables in C and C++, and hidden variables in Ada and Pascal.

```
p = malloc(...);
while( ) {
    p→next = malloc(...);
    p = p→next;
}
```

Fig. 26. Example of Dynamically Allocated Recursive Structure

and Hilfinger 1988; Ruf 1995].[21] Thus, an assignment to a local variable l at a stack location ST_i does not affect the values held by the same local variable at a stack location $ST_j, j < i$. (Stack location $ST_k, i < k$, does not exist, and is, therefore, irrelevant.) In other words, an assignment to l at ST_i should not kill any alias $\langle *l, p\rangle$ for any p at $ST_j, j < i$. Ignoring this distinction when killing a local variable of a recursive function can result in an incorrect data flow result.[22]

This situation is handled conservatively, by regarding all the assignments to local variables and parameters of a recursively called procedure as *may-kill* assignments; they only add new aliases without killing any reaching the assignment statements.

8. NAMING ANONYMOUS OBJECTS

The use of aliased memory locations rather than pointer expressions is possible only when pointers point to named objects. For *anonymous* objects, objects created in memory by a dynamic storage allocator such as the C malloc, LISP cons, Pascal and C++ new, or Fortran 90 allocate statement, we use a naming scheme based on the place (statement) in the program where an anonymous object is created as in [Jones and Muchnick 1981; Horwitz et al. 1989; Chase et al. 1990; Choi et al. 1993; Wilson and Lam 1995]. We thus ensure that a pointer expression is always associated with at least one named object. The named object, instead of the access path of the expression, is used for data flow analysis.

Section 3.1 demonstrates that the compact representation supports arbitrary levels of indirection, and can precisely represent the aliases generated by cyclic data structures in the absence of control flow join points. However, the naming scheme described above does not distinguish different *instances* [Chase et al. 1990] of anonymous objects that can be created by the same malloc statement along the same or different execution paths. This situation can arise along the same execution path when recursive structures are dynamically allocated. An example is provided in Figure 26, where the malloc statement is contained within a loop.

We improve the precision of a location based naming scheme for dynamically allocated recursive structures by combining the naming scheme above with the *named instances* approach, which allows up to k instances of the object to have distinct names for each statement, where k is a pre-determined constant. This approach is similar to loop unrolling, and equivalent to duplicating the malloc statement k times. Providing a name for each instance of a dynamically allocated

[21] The issue of name scopes such as file scopes for globals or block scopes for locals can be easily handled by uniquely renaming each variable to reflect its scope.

[22] It is also desirable to not generate aliases involving l at ST_j. However, not making this distinction results in a less precise, but not incorrect, result.

```
integer *p, *q, *r, *s, *t;

main() {                     int *sub1() {
        p = sub1();              r = sub3();
        q = sub2();              sub4();
}                                return r;
                             }

int *sub2() {                int *sub3() {
        s = sub3();              t = malloc(...);
        sub4();                  sub4();
        return s;                return t;
}                            }
```

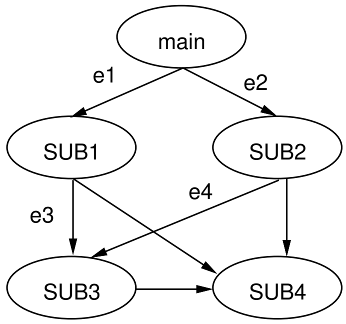

Fig. 27. Example Program and its PCG

object can be more precise if the object's predecessor in the alias graph is aliased with more than one storage location. Where more than k instances of an object may be dynamically allocated at a statement, one of the names will represent more than one object instance. Consequently, the corresponding alias graph node, n, will have an edge to itself, forming a cycle in the alias graph. This cycle, which results from merging multiple nodes into a single node, can be distinguished from a cycle in the alias graph that represents a true cyclic data structure.

8.1 Qualified Names

A potential drawback of this naming scheme is that different instances of anonymous objects created at the same malloc statement, but along different paths in the PCG, become indistinguishable, as shown in Figure 27.

Sharir and Pnueli [1981] use *call strings* to encode the interprocedural call context and discriminate different instances of the same procedure. Jones and Muchnick [1982] develop a more general abstraction of the call stack, which is used to name activation records. Other methods [Harrison III 1989; Chow and Harrison 1992] present a naming technique for dynamically allocated objects based on *procedure strings*, a generalization of call strings that includes the call stack and its evolution. Deutsch [1990] names dynamically allocated objects based on an arbitrary abstraction of the machine state at allocation time. Hudak [1986] presents a technique for naming dynamically allocated objects based on the top k call stack elements.

We do not encode the interprocedural context in the names of dynamically allocated objects when the names are created. Rather, we qualify these named anonymous objects by appending to the name an additional name string that captures PCG edge information *only* as these names are propagated *backward* in the PCG along acyclic paths from the node containing the dynamic allocation. With this scheme, we identify two qualified names as identical if and only if one qualified name is a prefix substring of the other.[23]

Our naming scheme may be defined as follows. Here a *backward edge* with respect to a node is one for which the node is a target. A *forward edge* with respect to a node is one for which the node is a source. A *backward path* from a node n is a sequence of backward edges, where n is the target of the first edge. A *forward path* from a node n is a sequence of forward edges, where n is the source of the first edge. From a malloc site S in function sub, for all backward acyclic paths from sub in the PCG, propagate the name S, appending an edge identification string for each edge that is traversed. This yields an association of some nodes in the PCG with at least one name for this malloc site. These names are propagated along all forward paths from each such node in the PCG, without modification. Without loss of precision, this forward propagation for a dynamically allocated object does not traverse an edge which has already been traversed for the object.

The above naming scheme can be incorporated into the alias analysis framework. The binding functions for call sites must include the appropriate name mapping for each name which designates a dynamically allocated object. This requires that each such name have a flag that indicates whether the current node is on a backward or a forward path with respect to that name. $ForwardBind_c$ for a call site c maps each name to itself. $BackwardBind_c$ excludes from its domain any name that is on a forward path. A name that is on a backward path is mapped to the same name appended with a name string identifying c.

In the program shown in Figure 27, all five pointer variables — p, q, r, s, t — point to objects dynamically allocated by the malloc statement in $sub3$. We first name the object allocated in $sub3$ as, say, $S1$ based on the malloc statement. At the call site of $sub3$ in $sub1$, the name becomes $S1_{e_3}$, and $*r$ becomes aliased to $S1_{e_3}$; and at the call site of $sub1$ in $main$, the name becomes $S1_{e_3e_1}$, and $*p$ becomes aliased to $S1_{e_3e_1}$. At the call site of $sub3$ in $sub2$, the name becomes $S1_{e_4}$, and $*s$ becomes aliased to $S1_{e_4}$; and at the call site of $sub2$ in $main$, the name becomes $S1_{e_4e_2}$, and $*q$ becomes aliased to $S1_{e_4e_2}$. Note that names are appended with PCG edges when they are propagated from the callee to the caller, but not when and after they are propagated from the caller to the callee. Therefore, object names, propagated from a caller to a callee, do not change when they are propagated back from the callee to the caller.

In $main$, we identify $*r$ and $*p$ as accessing the same named object since $*r$ is aliased to $S1_{e_3e_1}$, which is a prefix substring of (in fact identical to) the named object to which $*p$ is aliased. Note that $*r$ is aliased to $S1_{e_3e_1}$ in $main$, while it is aliased to $S1_{e_3}$ in $sub1$. We also identify $*s$ and $*q$ as aliases for the same reason. However, $*p$ and $*q$ are not aliases since the object name to which $*p$ is aliased is not a prefix substring of the object name to which $*q$ is aliased, and vice versa. In $sub4$, we identify $*t$ and $*r$ as accessing the same named object since $*t$ is aliased to $S1$, which is a prefix substring of $S1_{e_3}$, to which $*r$ is aliased.

We handle call chains with cycles (due to recursive calls) by not allowing a name string to have duplicate call graph edges in it. With this scheme, the number and length of name strings are bounded by the program size, while the capability of distinguishing different instances of the objects created at the same place is

[23] An algorithm that prepends edge information and performs a suffix substring test is equivalent.

compromised. The size of the name strings can be kept below some fixed value by limiting the number of appended PCG edges to some small constant, *size*.

The complexity of our technique for qualifying the names of anonymous objects is $O(V_{alloc} * W^K)$, where W is the maximum indegree or outdegree of a call graph node, K is the minimum of the predetermined constant *size* and the height of the call graph (i.e., the length of its longest acyclic path), and V_{alloc} is the total number of named dynamically allocated objects in the program.

9. COMPLEXITY ANALYSIS

This section describes the complexity analysis of our methods assuming the compact representation (Section 3) is used. Below are listed the variables that are used throughout this section. V is the total number of distinct objects, i.e., variables and named dynamically allocated objects. Since this also corresponds to the number of nodes in the alias graph, it is an upper on the number of aliases for any access path.

$A_{ap} \leq V$. – the maximum number of aliases for any access path explicitly referenced;

$A_{var} \leq V$. – the maximum number of aliases for any named object due to a single level of dereference;

$A_{stmt} \leq V * A_{var}$. – the maximum number of aliases holding at any statement;

CFE. – the total number of edges in the control flow graphs of the procedures in the program;

CFE_p. – the largest number of edges in a control flow graph in any procedure;

CFN_p. – the largest number of nodes in a control flow graph in any procedure;

D. – the total number of Type 3 (Section 4.2.4) pointer-assignments statements in the program;

L. – the maximum number of dereferences contained in an access path explicitly referenced in the program;

M. – the total number of *MeetNodes* in all the SEGs;

P. – the total number of nodes/procedures in the PCG;

PCE. – the total number of edges in the PCG; (This can be more than the number of call sites if function pointers are present.)

$PCE_{fp} \leq PCE$. – the total number of edges in the PCG whose associated call site invokes a function pointer;

S. – the total number of *GenNodes* in all the SEGs, i.e., the total number of pointer-assignments statements in the program; (D, M, and S can grow linearly with the size of the program.)

SEG_p. – the largest number of SEG nodes in any procedure;

S_p. – the largest number of pointer assignment statements in any procedure;

$V_{alloc} \leq V$. – the total number of named dynamically allocated objects;

w. – the largest *width* [Dhamdhere and Khedker 1993] of the program's control flow graphs.[24]

[24] Dhamdhere and Khedker [1993] show that the classical bound on the number of iterations for obtaining the fixed point solution of a data flow problem, $d + 1$, where d is the *depth* of the flow graph [Aho et al. 1986], is a loose bound. They provide a tighter bound, $w + 1$, where w is the width of the flow graph.

We first describe our base interprocedural algorithms and then address the enhancements.

9.1 Base Algorithms

This section first describes the complexity of techniques that are required for both types of interprocedural analysis algorithms.

ComputeAliases:. Both algorithms call ComputeAliases to determine the aliases of an access path using the compact representation. The worst-case time complexity of the basic ComputeAliases algorithm (Section 3) is dependent on the alias graph and the second parameter, DerefLevel. The worst-case cost of such a query is $O((A_{var})^L)$, which we denote as $O(A_{var}^L)$ in the remainder of this section. Note that L is typically a small constant.

Function Pointers:. The number of iterations performed for analysis of function pointers as described in Section 5 is bounded by PCE_{fp}. Each iteration entails an interprocedural and intraprocedural pointer alias analysis. The complexity of these iterations depends on the algorithms on which the analysis is based.

Anonymous Objects:. From Section 8, the complexity of the naming scheme is $O(V_{alloc} * D^K)$, where D is the maximum indegree or outdegree of a call graph node, and K is the minimum of the predetermined constant *size* (described in Section 8), and the height of the call graph (i.e., the length of its longest acyclic path).

9.1.1 *Flow-Sensitive Interprocedural Algorithm.* There are three distinct phases of the flow-sensitive alias algorithm:

—the SEG construction phase, during which the SEG for intraprocedural alias analysis is constructed for each procedure;
—the interprocedural phase during which, by iteration over the PCG, $Entry_p$ and $Exit_p$ for each procedure, p, are computed;
—the final intraprocedural phase, during which aliasing is computed for each statement in each procedure.

The time (and space) complexity to construct the SEG for a procedure is $\Theta(E)$, where E is the number of edges in the control flow graph [Sreedhar and Gao 1995].[25] Thus, the complexity for the first phase is $\Theta(P * CFE_p)$.

Since each of P procedures may have up to A_{stmt} aliases holding in their *Entry* and *Exit* sets, the space complexity during the interprocedural phase is $O(V * A_{var} * P)$, which can also be expressed as $O(A_{stmt} * P)$. The interprocedural phase dominates the time complexity. The worst case requires $O(A_{stmt} * P)$ iterations over the PCG. During each iteration, call sites, N_{Entry} nodes, N_{Exit} nodes, and $MeetNodes$ propagate alias sets, the size of which are bounded by A_{stmt}. The union operation performed at $MeetNodes$ is also bounded by A_{stmt}. Thus, this process is $O(A_{stmt} * (PCE + P + M))$. At each $GenNode$, at most two alias queries, calls to ComputeAliases, are made, one for each side of the assignment statement. From the discussion above, the cost of such a call is $O(A_{var}^L)$. The resulting sets from these calls are used to create new aliases using the transfer

[25] An $\Theta(E * \alpha(E))$ algorithm is given in [Cytron and Ferrante 1995].

function of Section 3. The worst-case size of each set is A_{ap}, making the cost of each $GenNode$ be $O(2 * A_{var}^L + A_{ap}^2) = O(A_{var}^L + A_{ap}^2)$, and the cost of processing all $GenNodes$, $O(S * (A_{var}^L + A_{ap}^2))$. Since these nodes, ($MeetNodes$, $GenNodes$, call sites, N_{Entry}, and N_{Exit}), will be considered at most $A_{stmt} * SEG_p$ times, the time complexity to compute new $Entry$ and $Exit$ sets for the procedures for each iteration over the PCG is $O((A_{stmt} * SEG_p) * (A_{stmt} * (PCE + P + M) + S * (A_{var}^L + A_{ap}^2)))$ and $O((A_{stmt} * P) * (A_{stmt} * SEG_p) * (A_{stmt} * (PCE + P + M) + S * (A_{var}^L + A_{ap}^2)))$ overall. This can be rewritten as $O(SEG_p * A_{stmt}^3 * P^2 + SEG_p * A_{stmt}^3 * P * PCE + SEG_p * A_{stmt}^3 * P * M + SEG_p * A_{stmt}^2 * P * S * (A_{var}^L + A_{ap}^2))$. A simpler but looser bound results from replacing A_{ap} and A_{var} terms by V, and A_{stmt} by V^2: $O(SEG_p * P * V^6 * (P + PCE + M + S * V^{L-2}))$.

The space complexity during the final phase is $O(P * SEG_p * A_{stmt})$. Using the logic from above, the time complexity of the final intraprocedural phase is $O((A_{stmt} * SEG_p) * (A_{stmt} * (PCE + M) + S * (A_{var}^L + A_{ap}^2)))$. Since this term is included in the time complexity for the interprocedural iteration, the overall (inter- and intraprocedural) time complexity of the flow-sensitive algorithm is as given above for the interprocedural iteration.

9.1.2 *Flow-Insensitive Interprocedural Algorithm.* The flow-insensitive alias algorithm contains three distinct phases:

—the initial collection of intraprocedural information (pointer-assignment statements)

—the interprocedural iteration over the PCG,

—and the final flow-sensitive intraprocedural analysis phase.

The space and time complexity of the first phase is $O(P * S_p)$. The space complexity of the analysis during the second phase is $O(A_{stmt} * P) = O(V * A_{var} * P)$. Once again the interprocedural phase dominates the time complexity. As above, there can be $O(A_{stmt} * P)$ iterations over the PCG. Each iteration requires P flow-insensitive intraprocedural analyses. During each intraprocedural analysis, as many as $O(S_p * A_{stmt})$ iterations of the artificial loop can occur. During each iteration, $O(S_p)$ statements are processed. The worst-case cost of processing each statement is the same as the flow-sensitive case: $O(A_{var}^L + A_{ap}^2)$. Thus, the cost of an intraprocedural phase is $O((S_p * A_{stmt}) * S_p * (A_{var}^L + A_{ap}^2))$, and for all intraprocedural phases $O(P * (S_p * A_{stmt}) * S_p * (A_{var}^L + A_{ap}^2))$. Since there are at most $O(A_{stmt} * P)$ such phases, the time complexity for the second phase is $O((A_{stmt} * P) * P * (S_p * A_{stmt}) * S_p * (A_{var}^L + A_{ap}^2)) = O(A_{stmt}^2 * P^2 * S_p^2 * (A_{var}^L + A_{ap}^2))$.

The space and time complexity during the final phase is the same as with the flow-sensitive algorithm, and is analyzed above: $O(P * SEG_p * A_{stmt})$ and $O((A_{stmt} * SEG_p) * (A_{stmt} * (PCE + M) + S * (A_{var}^L + A_{ap}^2)))$, respectively.

The overall time complexity of the flow-insensitive algorithm is, thus, $O(A_{stmt}^2 * P^2 * S_p^2 * (A_{var}^L + A_{ap}^2) + A_{stmt} * SEG_p * (A_{stmt} * (PCE + M) + S * (A_{var}^L + A_{ap}^2))$. A simpler but looser bound results from replacing A_{ap} and A_{var} terms by V, and A_{stmt} by V^2: $O(V^6 * P^2 * S_p^2 * (V^{L-2} + 1) + V^4 * SEG_p * (PCE + M + S + V^{L-2} * S))$.

9.2 Flow-Insensitive Enhancements

In Sections 4.2.3 and 4.2.4 two techniques were introduced to improve the efficiency and precision, respectively, of the flow-insensitive algorithm. This section presents the complexity of each of these enhancements over the base flow-insensitive algorithm.

9.2.1 *Using Precomputed Kill Information.* The time to precompute intraprocedural kill information for each procedure is bounded by $O(w * CFE_p)$. Thus, the worst-case time complexity for this precomputation is $O(P * w * CFE_p)$.

9.2.2 *Deferred Evaluation.* The worst-case time complexity using deferred evaluation is the same as the flow-insensitive algorithm, except during the intraprocedural iteration. In an attempt to reduce the number of statements that are considered during this iteration, deferred alias edges are introduced. Although one would expect the number of statements considered during the intraprocedural iteration to decrease, which in turn could reduce the number intraprocedural iterations required, the worst-case complexity for these quantities is still the same. This expected improvement comes at a cost during the alias queries, i.e., calls to ComputeAliases.

The ComputeAliases algorithm of Appendix A is comprised of three steps. The worst-case time complexity for Steps 1 and 3 of this algorithm is $O(A_{var}^L)$. Step 2 of the algorithm dominates the complexity. Its loop will iterate until up to $V * A_{var}$ regular alias edges are created, and D deferred evaluation edges are added to Expand_List, resulting in a total of $O(V * A_{var} + D)$ iterations. During each iteration, at most D deferred edges are considered, each of which can result in $O(A_{var}^L)$ edges being traversed. Thus, the time complexity of Step 2, and of the complete ComputeAliases algorithm given in Appendix A, is $O(V * A_{var} * D * A_{var}^L + D^2 * A_{var}^L)$. However, when $D = 0$, the complexity is the same as without deferred evaluation: $O(A_{var}^L)$.

As the worst-case time complexity of the ComputeAliases algorithm of Appendix A is $O(V * A_{var} * D * A_{var}^L + D^2 * A_{var}^L)$, the worst-case time complexity of the flow-insensitive algorithm with deferred evaluation is

$$
\begin{aligned}
O(A_{stmt}^2 * P^2 * S_p^2 * (V * A_{var} * D * A_{var}^L + D^2 * A_{var}^L + A_{ap}^2)) = \\
O(A_{stmt}^2 * P^2 * S_p^2 * V * A_{var} * D * A_{var}^L \\
+ A_{stmt}^2 * P^2 * S_p^2 * D^2 * A_{var}^L \\
+ A_{stmt}^2 * P^2 * S_p^2 * A_{ap}^2)
\end{aligned}
$$

9.3 Path Enhancements

In Section 6 several techniques were introduced to improve the precision of pointer alias analysis. Some of these techniques require precomputing *Path Tables*, which can be performed in $O((P \times (S_p)^K)$ time [Eve and Kurki-Suonio 1977], $2 \leq K \leq 3$, and is stored in $O(P \times (S_p)^2)$ space. Once the tables are computed, the generation of each alias at a pointer-assignment statement is augmented with a constant time query of the table.

These techniques replace alias relations with alias instances, which contain alias relations along with two other components: a birth-site and an alias history set.

The first item requires a constant amount of space. The worst-case size of the second is the number of potential birth sites, which is $O(P \times S_p)$. Approximations can bound this information to a constant size as described in 6.8. This additional information is utilized during an alias query. The additional time cost for an alias query using birth sites and alias history is $O(N^2)$ (by observation of Section 6.7), where N is the size of the set of birth sites being considered. Using the analysis from Section 9.1, this results in the total cost of an alias query being $O(A^L_{var} * N^2)$, and the cost for alias propagation being $O(A_{stmt} * (PCE * N^2 + P + M))$.

10. EMPIRICAL RESULTS

This section provides results for the fundamental algorithms presented in this paper. In particular, it reports on the three algorithms discussed in Section 4: flow-sensitive, flow-insensitive, and flow-insensitive with kill. Empirical evidence of the precision and efficiency of these analyses is provided.

10.1 Benchmarks

Our benchmark suite includes 14 C programs used by other researchers [Landi et al. 1993; Emami et al. 1994; Ruf 1995] as well as two programs from the SPEC CINT92 [Balan and Bays 1992] and CINT95 [SPEC 1995] benchmarks. Table I describes characteristics of the suite. The third column contains the number of lines in the source and header files reported by the Unix utility `wc`.[26] The next two columns report the number of functions and CFG nodes, respectively.

A dummy `main` function was added, which called the benchmark's `main` function, simulating the effects of `argc` and `argv`. This function also initialized the `_iob` array, used for standard I/O. The added function is the same as the one added by Ruf [1995], which is similar to the one used by Landi et al. [1993]. This function is included in the statistics reported in Table I. Global variables initializations are automatically modeled as assignment statements in the dummy `main` function, and thus are included in the count of CFG nodes.

10.2 Analyses

The three analyses have been implemented in an experimental program analysis system, called NPIC [Hind and Pioli 1997], written in C++. The system utilizes a C++ compiler from IBM [Soroker et al. 1997; Nackman 1997] as a front end, which constructs an abstract syntax tree.[27] This tree is transformed into a PCG and a CFG for each function, which serve as input to our algorithms. Since no CFG is built for library functions, a call to a library function is modeled based on its semantics. Library calls that cannot affect the value of a pointer are treated as the identity transfer function. A conservative SEG representation is used [Hind and Pioli 1997], which may include additional phi-nodes not included in the SEG defined by Choi et al. [1991].

---

[26] A few program names are different than those reported by Ruf [1995]. The SPEC CINT92 program 052.alvinn was named backprop in Todd Austin's benchmark suite [Austin 1995]. Ruf referred to `ks` as `part`, and `ft` as `span`.

[27] Since our benchmark suite contains C programs, some of them had to be syntactically modified to satisfy C++'s stricter type checking semantics.

Name	Source	LOC	Functions	CFG Nodes
allroots	Landi	227	7	157
052.alvinn	SPEC92	272	9	223
anagram	Austin	650	16	332
lex315	Landi	733	17	527
ks	Austin	782	14	513
vor	McCat	1,406	52	876
loader	Landi	1,539	30	687
ft	Austin	2,156	38	732
football	Landi	2,354	58	2,710
compiler	Landi	2,360	409	1,723
assembler	Landi	3,446	52	1,540
yacr2	Austin	3,979	59	2,030
simulator	Landi	4,639	111	2,684
099.go	SPEC95	29,637	373	16,822

Table I. Benchmark Suite

The compact representation (Section 3) is used to represent alias relations. All three analyses are implemented using a worklist of functions, where the worklists are implemented as a priority queue using a topological order of the functions as a priority.[28] The flow-sensitive analysis also uses a priority-based worklist of SEG nodes for each intraprocedural iteration (the inner loop of Figure 7). Priority is based on a topological order of the SEG nodes.

We have implemented the preprocessing of type 1 statements for flow-insensitive analyses, but have not yet implemented deferred evaluation (Section 4.2.4).

Function pointer analysis is implemented as described in Section 5. Techniques described in Section 6 have not yet been implemented; thus, all analyses are context-insensitive. Anonymous objects are named based on the statement where they are allocated, but the interprocedural naming technique described in Section 8 is not applied. Array elements and field components are not distinguished. Currently, `setjmp/longjmp`[29] and `goto` statements are not supported. The implementation also assumes that pointer values will only exist in pointer variables, and that pointer arithmetic does not result in the pointer going beyond array boundaries.

10.3 Description of Experiment

This section presents precision and efficiency results. Table II is similar to Table 3 in [Emami et al. 1994] and Figure 4 in [Ruf 1995]. For each benchmark and each analysis (flow-sensitive, flow-insensitive with kill, flow-insensitive), we report the analysis time, and the average number of objects a dereferenced pointer can point to. (The precision results for the FIK analysis is exactly the same as the FI analysis for all benchmarks. To save space we report the FIK analysis times in parentheses with the FI analysis times.)

The analysis time is reported in seconds. It does not include the time to build the

[28] We have found the worklist implementation to be on average three times more efficient than the corresponding iterating implementation [Hind and Pioli 1997].

[29] Although one program in our benchmark suite, anagram, does syntactically contains a call to `longjmp`, the code is unreachable.

			Access	Num Objects								
Program	Type	Time	Type	Avg	Tot	1	2	3	>3	Mx	Md	
allroots	FS	0.2	read	1.60	40	16	24	0	0	2	2.0	
			write	1.00	2	2	0	0	0	1	1.0	
227 LOC	FI	0.2	read	1.60		16	24	0	0	2	2.0	
7 Functions		(0.3)	write	1.00		2	0	0	0	1	1.0	
052.alvinn	FS	0.2	read	1.00	25	25	0	0	0	1	1.0	
			write	1.00	9	9	0	0	0	1	1.0	
272 LOC	FI	0.2	read	1.00		25	0	0	0	1	1.0	
9 Functions		(0.3)	write	1.00		9	0	0	0	1	1.0	
anagram	FS	1.0	read	1.13	23	20	3	0	0	2	1.0	
			write	1.00	10	10	0	0	0	1	1.0	
650 LOC	FI	0.9	read	1.13		20	3	0	0	2	1.0	
16 Functions		(1.5)	write	1.00		10	0	0	0	1	1.0	
lex315	FS	0.9	read	1.95	22	1	21	0	0	2	2.0	
			write	1.83	6	1	5	0	0	2	2.0	
733 LOC	FI	1.5	read	1.95		1	21	0	0	2	2.0	
17 Functions		(2.5)	write	1.83		1	5	0	0	2	2.0	
ks	FS	3.4	read	1.74	72	19	53	0	0	2	2.0	*
			write	1.62	42	16	26	0	0	2	2.0	*
782 LOC	FI	1.8	read	1.79		15	57	0	0	2	2.0	
14 Functions		(2.9)	write	1.86		6	36	0	0	2	2.0	
vor	FS	28.0	read	1.68	242	102	115	25	0	3	2.0	*
			write	1.32	158	119	28	11	0	3	1.0	*
1,406 LOC	FI	23.9	read	1.75		85	132	25	0	3	2.0	
52 Functions		(35.1)	write	1.35		113	34	11	0	3	1.0	
loader	FS	9.5	read	1.43	75	48	26	0	1	7	1.0	*
			write	2.22	37	24	7	1	5	9	1.0	*
1,539 LOC	FI	3.7	read	1.65		46	23	0	6	7	1.0	
30 Functions		(7.1)	write	2.38		24	5	1	7	9	1.0	
ft	FS	14.0	read	2.94	79	22	9	0	48	4	4.0	*
			write	2.03	61	38	3	0	20	4	1.0	*
2,156 LOC	FI	8.4	read	3.20		21	0	0	58	4	4.0	
38 Functions		(12.8)	write	2.18		37	0	0	24	4	1.0	
football	FS	5.9	read	1.56	354	256	5	91	2	8	1.0	*
			write	1.69	16	10	1	5	0	3	1.0	*
2,354 LOC	FI	5.2	read	1.62		249	4	91	10	8	1.0	
58 Functions		(10.2)	write	2.81		4	1	5	6	4	3.0	
compiler	FS	14.0	read	1.00	18	18	0	0	0	1	1.0	
			write	0.00	0	0	0	0	0	0	0.0	
2,360 LOC	FI	3.5	read	1.00		18	0	0	0	1	1.0	
40 Functions		(5.8)	write	0.00		0	0	0	0	0	0.0	
assembler	FS	26.9	read	1.60	211	171	13	0	27	9	1.0	*
			write	2.16	103	62	16	0	25	9	1.0	*
3,446 LOC	FI	14.6	read	1.73		166	10	0	35	9	1.0	
52 Functions		(28.2)	write	2.37		58	14	0	31	9	1.0	
yacr2	FS	33.5	read	1.11	315	281	34	0	0	2	1.0	
			write	1.16	114	97	16	1	0	3	1.0	
3,979 LOC	FI	18.9	read	1.11		279	36	0	0	2	1.0	
59 Functions		(36.3)	write	1.25		87	26	1	0	3	1.0	
simulator	FS	26.1	read	1.59	129	105	4	12	8	13	1.0	*
			write	2.05	104	67	11	16	10	13	1.0	*
4,639 LOC	FI	12.3	read	1.67		103	1	14	11	13	1.0	
111 Functions		(20.9)	write	2.09		67	9	16	12	13	1.0	
099.go	FS	21.2	read	18.35	57	16	7	3	31	100	5.0	
			write	13.64	47	6	9	9	23	100	3.0	
29,637 LOC	FI	20.1	read	18.35		16	7	3	31	100	5.0	
373 Functions		(29.9)	write	13.64		6	9	9	23	100	3.0	
Total	FS	—	read	2.13	1,662	1,100	314	131	117	100	—	
		—	write	2.47	709	461	122	43	83	100	—	
	FI	—	read	2.20		1,060	318	133	151	100	—	
		—	write	2.59		424	139	43	103	100	—	

Table II. Efficiency/precision measurements: numbers in "()" are the FIK analysis times. An entry with a "$*$" indicates an increase in precision for the flow-sensitive analysis.

PCG and CFGs, but does include any preprocessing that is specific to the analysis, such as constructing the SEG from the CFG. The results were collected on an IBM PowerPC 604 machine with 256MB RAM and 240MB paging space, running AIX 4.1.5.

To collect precision information, the system inspects the source visiting each dereference containing one or more dereferences of a pointer and, using the computed alias information, reports how many named objects are aliased to the pointer expression. We report the total number of such dereferences for both reads and writes, the average number of objects, and how often 1, 2, 3, or more than 3 objects are returned. The maximum and median number of objects returned are also given. Statements such as "`(*p)++`" and "`*p += increment`" are treated as both a read and a write of `*p`.

A pointer is considered to be dereferenced if the variable is declared as a pointer or an array formal parameter, and one or more of the "*", "->", or "[ ]" operators are used with that variable. Formal parameter arrays are included because their corresponding actual parameter could be a pointer. The use of the "[ ]" operator on arrays that are not formal parameters is not counted because the resulting "pointer" (the array name) is constant.

The manner in which the heap is modeled must be considered in evaluating these numbers. For example, a model that uses several names for objects in the heap may seem less precise when compared to a model that uses fewer names [Ruf 1995]. As mentioned above we distinguish heap objects based on their allocation site. Similarly, analyses that represent invisible objects (objects not lexically visible in the current procedure) aliased to a formal parameter as a single object [Landi and Ryder 1992; Emami et al. 1994][30] may report fewer objects. Our analyses do not use this technique.

For a correct input program, each pointer dereference will correspond to at least one object at run-time, and thus one serves as a lower bound for the average. Although a precision result close to one demonstrates the analysis is precise (modulo heap and invisible object naming), a larger number could reflect one or more of the following: an imprecise algorithm, a limitation of static analysis, or a pointer dereference where a pointer may point to different addresses over the program's execution.

10.4 Analysis

The results indicate that the flow-insensitive analysis with kill (FIK) is not beneficial. On our benchmark suite it is never more precise than the flow-insensitive analysis, and on some occasions requires more analysis time than the flow-sensitive analysis. One explanation for this could be that an alias relation created to simulate a reference parameter, where the formal points to the actual, typically is not killed in the called routine, i.e., the formal parameter is not modified, but rather is used to access the passed actual. Thus, programs containing only these alias relations will not benefit from the precomputed kill information.

One surprising result is the overall precision of the flow-insensitive analysis. In half of our benchmarks (those not marked with a "*" in Table II) it is as precise

[30] This modeling technique can increase the possibility of strong updates [Chase et al. 1990].

as the flow-sensitive analysis. This seems to suggest that considering control flow within a function is not significant for those benchmarks, at least where pointers are dereferenced. One explanation for this could be that a large number of alias relations are created at call sites because of actual/formal parameter bindings. For example, using a context- and flow-sensitive algorithm Emami et al. [1994] report that 95% of the points-to relations at pointer indirections in their benchmark suite are dereferencing a formal parameter. Thus, the lack of a significant difference in precision between our flow-sensitive and flow-insensitive analyses may be because both algorithms rely on the same (context-insensitive) mapping mechanism at call sites.

Another possible explanation for the similar precision is that pointer variables are often not assigned more than one distinguished object within the same function; therefore, the benefit of distinguishing program points is not realized. An exception to this is in the function `InitLists` in the `ks` benchmark, where the same list pointer is used in two list-creating loops. This is reflected in the precision results of Table II.

The flow-insensitive analysis can be up to four times faster than the flow-sensitive analysis (`compiler`); in one benchmark (`lex315`) it was slower.

The precision results for `099.go` report an average of 18.35 and 13.64 objects for reads and writes, respectively, with a maximum of 100 for each. This program contains six small list processing functions (using an array-based "cursor" implementation) that accept a pointer to the head of a list as a parameter. One of these functions, `addlist`, is called 404 times, creating 100 different actuals for one of the formals. However, because the scope of the formals is limited to this function, these relations are not propagated to any other function. Therefore, these relations do not suffer the effects of the unrealizable path problem, discussed in Section 6.

A context-sensitive analysis of this function may seem to offer better precision because it would analyze the `addlist` function for each calling context. However, such an analysis would require that code for 100 separate clones of this function be generated to utilize the more precise analysis information.

The results also suggest that analysis time is not only a function of program size; it also depends on the amount of alias relation propagation along the PCG and SEGs. For example, the flow-sensitive analysis of `099.go`, despite being our largest program and having a pointer aliased with 100 objects, is analyzed at the fastest rate (1,398 LOC/second) because these relations are not propagated throughout the program.

As suggested by Shapiro and Horwitz [1997a] and Diwan [1997], a more precise and time-consuming alias analysis may not be as inefficient as it may appear because the time required to obtain increased precision may reduce the time required by subsequent analyses that utilize mod-use information, and thus pointer alias information, as their input. As the previous paragraph suggests, this can also be true about pointer alias analysis itself, which also utilizes pointer alias information during its analysis.

10.5 Comparison with Other Results

Landi et al. [1993] report precision results for the computation of the MOD problem using a flow-sensitive pointer alias algorithm with limited context-sensitive information. Among the metrics they report is the number of "thru-deref" assigns, which correspond to the "write" metrics reported in Table II. However, since their results included compiler-introduced temporaries in their "thru-deref" count [Landi 1997], a direct comparison is not possible.

Stocks et al. [1997] use the same metric without including temporaries. Using the flow-sensitive context-sensitive analysis of Landi and Ryder [1992], the average number of objects ranges from 1.0 to 2.0 on the benchmarks we have in common (`allroots`, `lex315`, `loader`, `football`, `compiler`, `assembler`, `simulator`). On these benchmarks our flow-sensitive context-insensitive analysis ranges from 1.0 to 2.22. Two possible explanations for the slightly less precise results are 1) their algorithm is context-sensitive; 2) the underlying representation is not identical, and thus pointer dereferences may not be counted in the same manner in all cases. For example, in [Landi and Ryder 1992] invisible objects aliased to a formal parameter are represented as a single object, and statements such as `cfree(TP)` located in `allroots` are treated as modifying the structure freed, and thus as a pointer dereference [Landi 1997], where our analysis does not. On the three programs (`allroots`, `lex315`, `simulator`) in which our analysis reports the same, or close to the same, number of "writes" as "thru-derefs," our precision is identical to that reported in [Stocks et al. 1997].

The relative precision of the flow-insensitive analysis compared to the flow-sensitive analysis is in contrast to the study of Stocks et al. [1997], which compares the flow-sensitive analysis mentioned above with a flow-insensitive analysis described in [Zhang et al. 1996]. For the seven common benchmarks, the relative precision of our flow-insensitive algorithm ranges from 1.0 to 2.38, compared to 1.0 to approximately 6.3 for the flow-insensitive analysis they studied. We attribute this to the nature of our flow-insensitive analysis, which does not treat pointer-related assignments as symmetric, as is done in [Zhang et al. 1996].

Emami et al. [1994] report precision results for a context and flow-sensitive algorithm. Their results range from 1.0 to 1.77 for all indirect accesses, using a heap naming scheme that represents all heap objects with one name. Because we were unable to obtain the benchmarks from their suite, a direct comparison with our results is not possible.

Ruf [1995] reports both read and write totals for a flow-sensitive context-insensitive analysis. However, unlike our analysis he counts use of the "[ ]" operator on arrays that are not formal parameters as a dereference [Ruf 1997]. Since such an array will always point to the same place, the average number of objects is improved. To facilitate comparisons we have also counted in this manner and report the results in Table III.[31] For the benchmarks in common,[32] Ruf reports an overall read and write average of 1.35 and 1.43, respectively. Using these common benchmarks, our averages are 1.29 and 1.53. We attribute the slight differences to the difference in representations. As Ruf [1995] states, "the VDG intermediate representation often

[31] Once again, the precision numbers for the flow-insensitive analysis with kill are identical to the version without kill and are omitted.

[32] The common benchmarks are `allroots`, `052.alvinn`, `anagram`, `lex315`, `ks`, `loader`, `compiler`, `assembler`, `yacr2`, and `simulator`. Although, [Ruf 1995] reports results for `ft` (under the name span), our version of the benchmark is substantially larger than the one Ruf analyzed, and thus is not comparable.

coalesces series of structure or array operations into a single memory write."

Shapiro and Horwitz [1997b] present an empirical comparison of three flow-insensitive algorithms. All three algorithms are less precise (but probably more efficient) than the algorithms studied in this paper. Precision is reported in terms of the total number of points-to relations. We agree with [Emami et al. 1994; Ruf 1995] that a more meaningful metric is to measure where the points-to information is used as we have done in Table II.

11. RELATED WORK

Pointer alias analysis is known to be an undecidable problem [Landi 1992; Ramalingam 1994], and various analysis methods have been developed that approximate the solution with varying degrees of precision and efficiency.

11.1 Interprocedural Context

Context-sensitive algorithms preserve the calling context along each path in the PCG, which may require that each procedure be analyzed for each call path. For instance, Sharir and Pnueli [1981] augment each alias relation with a string of call paths, called a *call string*, that represents full call path information. Such algorithms avoid the interprocedural forward and return path problems, but incur potentially exponential space and time overhead.

Other algorithms, including the algorithm described in this paper, merge the calling context and, thus, in a single analysis of a procedure, account for the multiple call paths to it. Such algorithms can annotate each alias relation with call path information to avoid unrealizable execution paths. For instance, Landi and Ryder's [1991; 1992] interprocedural pointer alias analysis associates a *reaching alias set* with each alias in a procedure [Landi et al. 1993]. This set contains the aliases needed to hold on entry to the procedure to infer an alias at the given program point. An alias holding at a procedure exit point will be valid on return to any call site that passes a superset of its reaching alias set. To avoid exponential cost, they use reaching alias sets of size one.

Choi et al. [1993] also merge the calling context and perform at most a single analysis of each procedure during a traversal over the PCG. They annotate each alias relation with a *source alias set* along with one-level call site information to address the return path problem.

Emami et al. [1994] present a context-sensitive approach that represents all invocation paths (in the absence of recursion), which is similar to the method by Sharir and Pnueli. They claim through empirical evidence that exponential behavior is not seen in practice and suggest, without details, the use of a memoization scheme to avoid redundant analyses.

Following the approach of Emami et al., Wilson and Lam [1995] construct *partial transfer functions* that capture the impact of invocations of the same function from multiple call sites. This method summarizes the effect of a procedure call for an input subset that can be reused at other call sites of the procedure and may eliminate the cost of reanalyzing the procedure at each call site.

Ruf [1995] examines the effect of full call path information on the precision of alias analysis by measuring the precision of alias information with a context-insensitive analysis and a maximally context-sensitive version of the same analysis. He con-

		Access			Num Objects						
Program	Analysis	Type	Avg	Tot	1	2	3	>3	Mx	Md	
allroots	FS	read	1.56	43	19	24	0	0	2	2.0	
		write	1.00	4	4	0	0	0	1	1.0	
227 LOC	FI	read	1.56		19	24	0	0	2	2.0	
7 Functions		write	1.00		4	0	0	0	1	1.0	
052.alvinn	FS	read	1.00	44	44	0	0	0	1	1.0	
		write	1.00	22	22	0	0	0	1	1.0	
272 LOC	FI	read	1.00		44	0	0	0	1	1.0	
9 Functions		write	1.00		22	0	0	0	1	1.0	
anagram	FS	read	1.06	49	46	3	0	0	2	1.0	
		write	1.00	28	28	0	0	0	1	1.0	
650 LOC	FI	read	1.06		46	3	0	0	2	1.0	
16 Functions		write	1.00		28	0	0	0	1	1.0	
lex315	FS	read	1.81	26	5	21	0	0	2	2.0	
		write	1.50	10	5	5	0	0	2	1.5	
733 LOC	FI	read	1.81		5	21	0	0	2	2.0	
17 Functions		write	1.50		5	5	0	0	2	1.5	
ks	FS	read	1.47	113	60	53	0	0	2	1.0	*
		write	1.39	66	40	26	0	0	2	1.0	*
782 LOC	FI	read	1.55		51	62	0	0	2	2.0	
14 Functions		write	1.56		29	37	0	0	2	2.0	
vor	FS	read	1.68	242	102	115	25	0	3	2.0	*
		write	1.31	160	121	28	11	0	3	1.0	*
1,406 LOC	FI	read	1.75		85	132	25	0	3	2.0	
52 Functions		write	1.35		115	34	11	0	3	1.0	
loader	FS	read	1.41	78	51	26	0	1	7	1.0	*
		write	1.96	47	34	7	1	5	9	1.0	*
1,539 LOC	FI	read	1.63		49	23	0	6	7	1.0	
30 Functions		write	2.09		34	5	1	7	9	1.0	
ft	FS	read	3.09	92	22	9	0	61	4	4.0	*
		write	2.16	70	41	3	0	26	4	1.0	*
2,156 LOC	FI	read	3.32		21	0	0	71	4	4.0	
38 Functions		write	2.33		39	0	0	31	4	1.0	
football	FS	read	1.12	1,629	1,531	5	91	2	8	1.0	*
		write	1.05	201	195	1	5	0	3	1.0	*
2,354 LOC	FI	read	1.14		1,524	4	91	10	8	1.0	
58 Functions		write	1.14		189	1	5	6	4	1.0	
compiler	FS	read	1.15	97	82	15	0	0	2	1.0	
		write	1.18	51	42	9	0	0	2	1.0	
2,360 LOC	FI	read	1.15		82	15	0	0	2	1.0	
40 Functions		write	1.18		42	9	0	0	2	1.0	
assembler	FS	read	1.59	213	173	13	0	27	9	1.0	*
		write	1.98	121	80	16	0	25	9	1.0	*
3,446 LOC	FI	read	1.73		168	10	0	35	9	1.0	
52 Functions		write	2.17		76	14	0	31	9	1.0	
yacr2	FS	read	1.11	315	281	34	0	0	2	1.0	*
		write	1.16	116	99	16	1	0	3	1.0	*
3,979 LOC	FI	read	1.11		279	36	0	0	2	1.0	
59 Functions		write	1.24		89	26	1	0	3	1.0	
simulator	FS	read	1.24	460	425	4	12	19	13	1.0	*
		write	1.65	229	179	11	16	23	13	1.0	*
4,639 LOC	FI	read	1.28		423	1	14	22	13	1.0	
111 Functions		write	1.72		179	9	16	25	13	1.0	
go	FS	read	1.13	7,896	7,855	7	3	31	100	1.0	
		write	1.48	1,236	1,195	9	9	23	100	1.0	
29,637 LOC	FI	read	1.13		7,855	7	3	31	100	1.0	
373 Functions		write	1.48		1,195	9	9	23	100	1.0	
Total	FS	read	1.17	11,297	10,696	329	131	141	100	—	
		write	1.47	2,361	2,085	131	43	102	100	—	
	FI	read	1.19		10,651	338	133	175	100	—	
		write	1.51		2,046	149	43	123	100	—	

Table III. Results counting all uses of "*", "− >", "[ ]". An entry with a "*" indicates an increase in precision for the flow-sensitive analysis.

cludes that the context-insensitive analysis incurs little to no precision penalty at the program points where pointers are dereferenced. However, he suggests that the benchmarks used may not be representative of general programs.

As in Choi et al. [1993], we merge the calling context of each procedure and distinguish alias relation into alias instances, which contain birth site, call site, source alias sets, and (indirectly) history sets components. These components are used to address both the forward and return path problem at varying degrees of precision and efficiency. Alias instances offer a two-level framework for trade-offs between precision and efficiency of alias analysis. The first level affects both the space and time efficiency of the analysis. The second level affects only the time efficiency of the analysis.

11.2 Non-distributive Combining Problem

None of the previous work considers the intraprocedural execution path along which an alias is generated when combining aliases. A combined use of birth sites and alias histories provides a spectrum of trade-offs between precision and cost, from birth-site information only, to alias histories with unlimited size. The approaches that explicitly represent all invocation paths [Emami et al. 1994; Wilson and Lam 1995] avoid this problem on the interprocedural level. Choi et al. [1993] use call-site information to address this problem at the interprocedural level, and suggest the use of CFG path information to address the intraprocedural version of this problem.

11.3 Flow-Insensitive Analysis

Larus [1989] presents a flow-insensitive intraprocedural algorithm to compute aliases in LISP programs. This algorithm uses alias graphs, which are similar to ours, but serve both as values propagated to solve data flow equations and as representations of statements' effects on propagated values. He uses the fastness closure technique of Graham and Wegman [1976] to process the alias graphs.

Andersen [1994] defines both context-sensitive and context-insensitive flow-insensitive algorithms. The algorithms are constraint-based and are solved in an iterative manner. The context-insensitive algorithm is polynomial in the size of the program.

Steensgaard [1996] presents a flow-insensitive points-to algorithm that conservatively treats alias relations as reflexive and transitive. He utilizes a fast union/find algorithm, which results in an almost linear time complexity for his algorithm, making it less precise, but more efficient than Andersen's algorithm.

Shapiro and Horwitz [1997b] provide two flow-insensitive algorithms. The first can be tuned so that its precision and worst-case time and space complexity can vary between Steensgaard's and Andersen's. They provide experimental results that vary the value of the tuning parameter. Based on their observations, they devise a second algorithm that uses their first algorithm as a subroutine. The worst-case running time of the second algorithm is $\log N$ slower than Steensgaard's algorithm, but is more accurate.

Horwitz [1997] proves that precise flow-insensitive may-alias analysis with arbitrary levels of pointers and an arbitrary number of pointer dereferences and no dynamic memory allocation is NP-Hard.

Our flow-insensitive algorithm does not treat alias relations as reflexive and transitive, and thus, like Andersen's algorithm, is more precise than Steensgaard's and Shapiro and Horwitz's second algorithm, at the cost of increased worst-case complexity. Although our algorithm relies on data flow iteration, we have described techniques that can reduce the number of statements involved in this iteration. We have described how limited kill information can be used to improve precision.

11.4 PCG Construction

Methods for constructing the PCG in the presence of function parameters and function variables are given in [Weihl 1980; Burke 1987; Callahan et al. 1990; Hall and Kennedy 1992; Lakhotia 1993]. Our method accommodates function parameters and arbitrary levels of function pointers. As such, it is more general than methods for constructing the PCG in the presence of function parameters [Ryder 1979; Burke 1987; Callahan et al. 1990] or function variables [Hall and Kennedy 1992; Lakhotia 1993]. By using the framework of pointer-induced aliasing, it is also more precise than Weihl's method, which performs transitive closure of the alias relations [Weihl 1980].

Emami et al. [1994] independently proposed an algorithm similar to ours for constructing the PCG for the same programming model we consider. Shivers [1988] presents a technique for deriving control flow in Scheme programs, where functions are first class objects. To build control flow in this context, it is necessary to compute, for every function call, the set of functions it could be bound to. In general where functions are first class objects, control flow analysis includes a function binding analysis that is essentially equivalent to function pointer analysis. Solutions for performing such a control flow analysis are given in [Shivers 1988; Neirynck et al. 1989; Sestoft 1989; Harrison III 1989; Mogensen 1989; Deutsch 1990].

Chow and Harrison [1994] present an algorithm for analyzing programs with pointers and closures that is conceptually similar to our algorithm for analyzing function pointers. Our iterative algorithm uses an approximation of the PCG to perform an analysis that either indicates convergence or provides a better approximation of the PCG. Chow and Harrison's iterative algorithm uses approximate read/write sets to perform a side-effect analysis that either indicates convergence or provides a better approximation of the read/write sets.

Our algorithm can be easily extended to use a worklist, as done in [Landi and Ryder 1992; Hall and Kennedy 1992]. It can also be extended to build the *Interprocedural Control Flow Graph* (ICFG) [Myers 1981; Callahan 1988; Harrold and Soffa 1994; Landi and Ryder 1992], instead of the PCG.

11.5 Alias Representation

The compact representation, introduced in [Choi et al. 1993], is highly similar to the *points-to* representation of Emami et al. [1994]. The only difference between the representations is that the dereference of the first element is implied in the points-to representation. Although the points-to representation may be more intuitive, it cannot represent aliases between non-pointer objects. The compact representation can also represent reference parameter alias pairs, for example. The compact representation associates a name with every storage location. Although the compact representation is not graph-based, it can easily be mapped to and from graph-based representations [Jones and Muchnick 1981].

The compact representation, as well as graph-based representations [Jones and Muchnick 1981; Larus and Hilfinger 1988; Horwitz et al. 1989; Chase et al. 1990], suffer imprecision as a result of the transitivity effect described in Section 6.7. The techniques described in Section 6.7 help improve the precision for such representations. Previous work has not addressed this loss of precision.

The compact representation accommodates cyclic data structures without resorting to a k-limiting scheme to limit the level of indirection. The compact representation permits an arbitrary level of pointer indirection to be represented without loss of precision, except due to the transitivity effect. In the presence of a cyclic structure, an exhaustive representation that relies on k-limiting would summarize levels of dereference larger than some fixed constant value, resulting in a possible loss of precision [Jones and Muchnick 1981; Landi and Ryder 1992]. Other methods can represent the full solution for some cyclic structures without such a loss [Deutsch 1994].

11.6 Heap Analysis

In our treatment of dynamically allocated structures, we use a naming scheme based on the statement in the program where an anonymous object is created as in [Horwitz et al. 1989; Chase et al. 1990], combined with the named instance approach, as described in Section 8. We qualify named anonymous objects with a name string that captures backward call path information, distinguishing different instances of dynamically allocated objects. Hudak [1986] distinguishes different instances of dynamically allocated objects by directly applying the Sharir and Pnueli [1981] interprocedural naming technique. Whereas Hudak abstracts a call string by k truncation, we abstract a call string by truncating it just before a repeated call edge. Other approaches toward handling dynamically allocated recursive structures [Jones and Muchnick 1981; Larus and Hilfinger 1988; Ruggieri and Murtagh 1988; Horwitz et al. 1989; Hendren and Nicolau 1990; Landi and Ryder 1992] do not name all instances of anonymous objects. However, [Jones and Muchnick 1981; Larus and Hilfinger 1988; Ruggieri and Murtagh 1988; Horwitz et al. 1989; Landi and Ryder 1992] use the *k-limiting* approach [Jones and Muchnick 1981], and thus distinguish up to k objects created at the same malloc statement. In Emami et al. [1994],Wilson and Lam [1995], and Ruf [1995], only one name is used to represent all dynamically allocated objects.

The method proposed by Chase et al. [1990] uses the program structure to determine when to summarize anonymous objects. For each malloc statement, they classify the anonymous objects created into *interesting* and *summary* nodes. An interesting node is an anonymous object that has exactly one variable pointing to it. A summary node represents all other anonymous objects allocated at a particular statement. This distinction is made to preserve the *strong update* characteristic of interesting nodes. For each malloc statement, up to $V+1$ anonymous objects can be created: V interesting nodes (where V is the number of variables in the program) and one summary node. An extension using reference counting is described that can discover data structures that are "true lists" and "true trees."

Hendren and Nicolau [1990] represent recursive data structures with a *path matrix* whose entries are path expressions. For each pair of nodes (a, b) in a data structure, the corresponding path expression conservatively represents the set of all paths from *a* to *b*. For a programming language that includes dynamically allocated recursive data structures, pointer assignments, structured control flow, and procedure calls, they develop the data flow analysis for generating path matrices for binary trees and DAGs. The method can be generalized to apply to any recursive, structured data type.

Ghiya and Hendren [1995] provide a context-sensitive *connection analysis* that can determine if two stack pointers can reach a common heap element. In [Ghiya and Hendren 1996], the same authors describe a context-sensitive *shape analysis*, which estimates the shape of data structures accessible from a stack pointer as one of *Tree, DAG,* or *Cyclic Graph.* In both cases they report results as implemented in the McCAT compiler. In the latter case they conclude that the results show the analysis provides accurate results for programs that build simple data structures.

Sagiv et al. [1996] develop a shape analysis algorithm that performs destructive updates on heap-allocated storage, using a finite, but potentially exponential (in the number of pointer variables), shape-graph to approximate the possible shapes of a heap-allocated structure. Their method is accurate for certain programs that update cyclic data structures.

Deutsch [1994] improves the accuracy of alias analysis in the presence of recursive pointer data structures by capturing position-dependent alias relation properties. Aliases are represented as pairs of symbolic access paths, which are access paths qualified by integer coefficients representing iteration factors. The alias lattice is parameterized by a numeric lattice, which determines which class of relations between positions in aliased data structures can be captured. Deutsch's method can yield precise information where other methods utilize the approximation techniques of k-limiting or collapsing multiple dynamically allocated objects into a single one. The precision and efficiency of Deutsch's method depends on the choice made for the numeric lattice.

12. CONCLUSIONS

Without interprocedural alias analysis, compilers must make worst-case assumptions about pointers, formal parameters, and variables global to a procedure. These assumptions impede optimizations and can result in inefficient code and increased compilation time, particularly for languages like C and C++, which encourage the frequent use of pointers and procedure calls.

We have provided practical approximation methods for computing and representing interprocedural aliases for a program written in a language that includes pointers, reference parameters, and recursion, like C, C++, Fortran 90, Java, and LISP. These methods include a flow-sensitive interprocedural alias analysis and a flow-insensitive interprocedural alias analysis that can incorporate kill information to improve precision, and use deferred evaluation to improve efficiency. These methods rely on an algorithm for function pointer analysis that constructs a program call graph during alias analysis.

We have reported empirical measurements to contrast both the efficiency and precision of the three interprocedural pointer alias analysis algorithms: the flow-sensitive, flow-insensitive, and flow-insensitive with precomputed kill algorithms. After accounting for differences in program representation and metrics, we have found that the precision of most flow-sensitive analysis algorithms are roughly equivalent. We have also shown that a flow-sensitive analysis can be efficiently implemented to handle programs on the order of 30,000 LOCs. The precision of the flow-insensitive analysis is shown to be the same as the flow-sensitive analysis on half of our benchmarks.

We have also introduced a compact representation for alias information that is based on named objects rather than pointer expressions and accommodates cyclic data structures without resorting to a k-limiting scheme to limit the level of indirection.

We have presented a number of techniques for improving the precision of alias analysis with respect to the unrealizable path and non-distributive combining problems; a technique for improving the precision of alias information for compact and graph-based representations; and an interprocedural naming technique for dynamically allocated objects that improves the precision of alias analysis for such objects.

ACKNOWLEDGEMENTS

We thank Lee Nackman and Fran Allen for their support of this work. We thank Todd Austin, Bill Landi, and Rakesh Ghiya for making their benchmarks available. Bill Landi, Erik Ruf, and Bob Wilson provided useful details concerning their implementations. David Bacon, John Field, Yong-fong Lee, Anthony Pioli, G. Ramalingam, Barbara Simons, Harini Srinivasan, Laureen Treacy, and the anonymous referees provided many useful comments on earlier drafts of this paper.

Anthony Pioli played a significant role in the design and implementation of the NPIC system. Harini Srinivasan designed and implemented the initial control flow graph builder. David Bacon participated in several useful discussions of the program call graph representation. Robert Culley, Lynne Delesky, Lap Chung Lam, Giampaolo Lauria, Mark Nicosia, Keith Sanders, Truong Vu, and Ming Wu assisted with the implementation and testing of the system.

REFERENCES

Aho, A. V., Garey, M. R., and Ullman, J. D. 1972. The transitive reduction of a directed graph. *SIAM Journal on Computing 1,* 2, 131–137.

Aho, A. V., Sethi, R., and Ullman, J. D. 1986. *Compilers: Principles, Techniques, and Tools.* Addison-Wesley, Reading, MA.

Allen, F., Burke, M., Charles, P., Cytron, R., and Ferrante, J. 1987. An overview of the PTRAN analysis system for multiprocessing. In *1987 International Conference on Supercomputing.* Springer-Verlag, 194–211. Also published in The Journal of Parallel and Distributed Computing, October, 1988, 5(5):617–640.

Allen, R., Callahan, D., and Kennedy, K. 1987. Automatic decomposition of scientific programs for parallel execution. In *14th Annual ACM Symposium on the Principles of Programming Languages.* 63–76.

Andersen, L. O. 1994. Program analysis and specialization for the C programming language. Ph.D. thesis, DIKU, University of Copenhagen. Available at ftp.diku.dk/pub/diku/semantics/papers/D-203.dvi.Z.

Austin, T. 1995. Pointer-intensive benchmark suite, version 1.1. http://www.cs.wisc.edu/~austin/ptr-dist.html.

Balan, S. and Bays, W. 1992. Spec announces new benchmark suites cint92 and cfp92. Tech. rep., Systems Performance Evualation Cooperative. March. *SPEC Newsletter* 4(1).

Banning, J. 1979. An efficient way to find the side effects of procedure calls and the aliases of variables. In *6th Annual ACM Symposium on the Principles of Programming Languages.* 29–41.

Burke, M. 1987. An interval-based approach to exhaustive and incremental interprocedural data flow analysis. Tech. rep., IBM Research. August. Report RC12702.

Burke, M. 1990. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems 12,* 3 (July), 341–395.

Burke, M., Carini, P., Choi, J.-D., and Hind, M. 1995. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Lecture Notes in Computer Science, 892*, K. Pingali, U. Banerjee, D. Gelernter, A. Nicolau, and D. Padua, Eds. Springer-Verlag, 234–250. Proceedings from the *7th International Workshop on Languages and Compilers for Parallel Computing.* Extended version published as Research Report RC 19546, IBM T. J. Watson Research Center, September, 1994.

Burke, M. and Choi, J.-D. 1992. Precise and efficient integration of interprocedural alias information into data-flow analysis. *ACM Letters on Programming Languages and Systems 1,* 1 (March), 14–21.

Burke, M. and Cytron, R. 1986. Interprocedural dependence analysis and parallelization. In *SIGPLAN '86 Symposium on Compiler Construction.* ACM, 162–175. *SIGPLAN Notices, 21(7).*

Callahan, D. 1988. The program summary graph and flow-sensitive interprocedural data flow analysis. In *SIGPLAN '88 Conference on Programming Language Design and Implementation.* 47–56. *SIGPLAN Notices, 23(7).*

Callahan, D., Carle, A., Hall, M. W., and K.Kennedy. 1990. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering 16,* 4 (Apr.), 483–487.

Chase, D. R., Wegman, M., and Zadeck, F. K. 1990. Analysis of pointers and structures. In *SIGPLAN '90 Conference on Programming Language Design and Implementation.* 296–310. *SIGPLAN Notices* 25(6).

Choi, J.-D., Burke, M., and Carini, P. 1993. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages.* 232–245.

Choi, J.-D., Cytron, R., and Ferrante, J. 1991. Automatic construction of sparse data flow evaluation graphs. In *18th Annual ACM Symposium on the Principles of Programming Languages.* 55–66.

Choi, J.-D. and Ferrante, J. 1994. Static slicing in the presence of goto statements. *ACM Transactions on Programming Languages and Systems 16,* 4 (July), 1097–1113.

Chow, J. H. and Harrison, W. L. 1992. Compile-time analysis of parallel programs that share memory. In *19th Annual ACM Symposium on the Principles of Programming Languages.* 130–141.

Chow, J. H. and Harrison, W. L. 1994. State space reduction in abstract interpretation of parallel programs. In *Proceedings of the IEEE International Conference on Computer Languages.* 277–288.

Cooper, B. 1989. Ambitious data flow analysis. M.S. thesis, University of Minnesota.

Cooper, K. D. and Kennedy, K. 1988. Interprocedural side-effect analysis in linear time. In *SIGPLAN '88 Conference on Programming Language Design and Implementation.* 57–66. *SIGPLAN Notices, 23(7).*

Cytron, R. and Ferrante, J. 1995. Efficiently computing phi-nodes on-the-fly. *ACM Transactions on Programming Languages and Systems 16,* 3 (May), 487–506.

Cytron, R. and Gershbein, R. 1993. Efficient accomodation of may-alias information in SSA form. *SIGPLAN '93 Conference on Programming Language Design and Implementation*, 36–45. *SIGPLAN Notices* 28(6).

Deutsch, A. 1990. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *17th Annual ACM Symposium on the Principles of Programming Languages.* San Francisco, 157–168.

Deutsch, A. 1992. A storeless model of aliasing and its abstractions using finite representations of right-regular equivalence relations. In *IEEE'92 International Conference on Computer Languages.*

DEUTSCH, A. 1994. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*. 230–241. *SIGPLAN Notices, 29(6)*.

DHAMDHERE, D. M. AND KHEDKER, U. P. 1993. Complexity of bi-directional data flow analysis. In *20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*. 397–408.

DIWAN, A. 1997. Personal communication.

EMAMI, M., GHIYA, R., AND HENDREN, L. J. 1994. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*. 242–256. *SIGPLAN Notices, 29(6)*.

EVE, J. AND KURKI-SUONIO, R. 1977. On computing the transitive closure of a relation. *Acta Informatica 8*, 303–314.

GANNON, D., GUARNA JR., V. A., AND LEE, J. K. 1990. Static analysis and runtime support for parallel execution of C. In *Languages and Compilers for Parallel Computing*, D. Gelernter, A. Nicolau, and D. Padua, Eds. MIT Press, 254–274. Selected papers from the Second Workshop on Languages and Compilers for Parallel Computing at Urbana, Illinois in August 1989.

GHIYA, R. AND HENDREN, L. J. 1995. Connection analysis: A practical interprocedural heap analysis for C. In *Proceedings of the Eighth Workshop on Languages and Compilers for Parallel Computing*.

GHIYA, R. AND HENDREN, L. J. 1996. Is it a tree, a dag or a cyclic graph? A shape analysis for heap-directed pointers in C. In *23nd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*.

GRAHAM, S. L. AND WEGMAN, M. 1976. A fast and usually linear algorithm for global flow analysis. *Journal of the Association for Computing Machinery 23*, 1 (January), 172–202.

HALL, M. W. AND KENNEDY, K. 1992. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems 1*, 3 (Sept.), 227–242.

HARRISON III, W. L. 1989. The interprocedural analysis and automatic parallelisation of Scheme programs. *Lisp and Symbolic Computation 2*, 3 (Oct.), 176–396.

HARROLD, M. J. AND SOFFA, M. L. 1994. Efficient computation of interprocedural definition – use chains. *ACM Transactions on Programming Languages and Systems 16*, 2 (Mar.), 175–204.

HENDREN, L. J. AND NICOLAU, A. 1990. Parallelizing programs with recursive data structures. *IEEE Transactions on Parallel and Distributed Systems 1*, 1 (Jan.), 35–47.

HIND, M. AND PIOLI, A. 1997. An empirical comparison of interprocedural pointer alias analyses. submitted for publication.

HORWITZ, S. 1997. Precise flow-insensitive may-alias analysis is NP-Hard. *ACM Transactions on Programming Languages and Systems 19*, 1 (Jan.), 1–6.

HORWITZ, S., PFEIFFER, P., AND REPS, T. 1989. Dependence analysis for pointer variables. In *SIGPLAN '89 Conference on Programming Language Design and Implementation*. 28–40. *SIGPLAN Notices* 24(6).

HUDAK, P. 1986. A semantic model of reference counting and its abstraction. In *Conference Record of the 1986 ACM Symposium of LISP and Functional Programming*. 351–363.

JONES, N. D. AND MUCHNICK, S. S. 1981. Flow analysis and optimization of lisp-like structures. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice-Hall, Chapter 4, 102–131.

JONES, N. D. AND MUCHNICK, S. S. 1982. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *9th Annual ACM Symposium on the Principles of Programming Languages*. 66–74.

KAM, J. B. AND ULLMAN, J. D. 1976. Global data flow analysis and iterative algorithms. *Journal of the ACM 23*, 1 (January), 158–171.

KENNEDY, K., MCKINLEY, K. S., AND TSENG, C. 1991. Interactive parallel programming using the parascope editor. *IEEE Transactions on Parallel and Distributed Systems 2*, 3 (July), 329–341.

LAKHOTIA, A. 1993. Constructing call multigraphs using dependence graphs. In *20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*. 273–284.

LANDI, W. 1992. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems 1,* 4 (Dec.), 323–337.

LANDI, W. 1997. Personal communication.

LANDI, W. AND RYDER, B. 1991. Pointer-induced aliasing: A problem classification. In *18th Annual ACM Symposium on the Principles of Programming Languages.* 93–108.

LANDI, W. AND RYDER, B. 1992. A safe approximate algorithm for interprocedural pointer aliasing. In *SIGPLAN '92 Conference on Programming Language Design and Implementation.* 235–248. *SIGPLAN Notices* 27(6).

LANDI, W., RYDER, B., AND ZHANG, S. 1993. Interprocedural modification side effect analysis with pointer aliasing. *SIGPLAN '93 Conference on Programming Language Design and Implementation*, 56–67. *SIGPLAN Notices* 28(6).

LARUS, J. R. 1989. Restructuring symbolic programs for concurrent execution on multiprocessors. Ph.D. thesis, University of California. Technical Report No. UCB/CSD 89/502.

LARUS, J. R. AND HILFINGER, P. N. 1988. Detecting conflicts between structure accesses. In *SIGPLAN '88 Conference on Programming Language Design and Implementation.* 21–34.

MARLOWE, T., LANDI, W., RYDER, B., CHOI, J.-D., BURKE, M., AND CARINI, P. 1993. Pointer-induced aliasing: A clarification. *SIGPLAN Notices 28,* 9 (Sept.), 67–70.

MARLOWE, T. J., RYDER, B. G., AND BURKE, M. G. 1995. Defining flow sensitivity in data flow problems. Tech. Rep. RC 20138 (89049), IBM Thomas J. Watson Research Center. July.

MOGENSEN, T. 1989. Binding time analysis for polymorphically typed higher-order languages. In *Lecture Notes in Computer Science, 352.* Springer-Verlag, 298–312. Proceedings of TAPSOFT.

MYERS, E. W. 1981. A precise inter-procedural data flow algorithm. In *8th Annual ACM Symposium on the Principles of Programming Languages.* 219–230.

NACKMAN, L. R. 1997. Codestore and incremental C++. *Dr. Dobbs Journal*, 92–95.

NEIRYNCK, A., PANANGADEN, P., AND DEMERS, A. J. 1989. Effect analysis in higher-order languages. *International Journal of Parallel Programming 18,* 1, 1–17.

RAMALINGAM, G. 1994. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems 16,* 6 (Nov.), 1467–1471.

ROSEN, B. K. 1979. Data flow analysis for procedural languages. *Journal of the ACM 26,* 2 (Apr.), 322–344.

RUF, E. 1995. Context-insensitive alias analysis reconsidered. In *SIGPLAN '95 Conference on Programming Language Design and Implementation.* 13–22. *SIGPLAN Notices, 30(6).*

RUF, E. 1997. Personal communication.

RUGGIERI, C. AND MURTAGH, T. P. 1988. Lifetime analysis of dynamically allocated objects. In *15th Annual ACM Symposium on the Principles of Programming Languages.* 285–293.

RYDER, B. 1979. Constructing the call graph of a program. *IEEE Software Engineering.*

SAGIV, M., REPS, T., AND WILHELM, R. 1996. Solving shape-analysis problems in languages with destructive updating. In *23nd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages.*

SAGIV, S., FRANCEZ, N., RODEH, M., AND WILHELM, R. 1990. A logic-based approach to data flow analysis. In *Lecture Notes on Computer Science, 456.* Springer-Verlag, 277–292. *Programming Language Implementation and Logic Programming.*

SESTOFT, P. 1989. Replacing function parameters by global variables. In *Conference on Functional Programming Languages and Computer Architecture.* ACM Press, London, 39–53.

SHAPIRO, M. AND HORWITZ, S. 1997a. The effects of the precision of pointer analysis. In *Lecture Notes in Computer Science, 1302*, P. V. Hentenryck, Ed. Springer-Verlag, 16–34. Proceedings from the *4th International Static Analysis Symposium.*

SHAPIRO, M. AND HORWITZ, S. 1997b. Fast and accurate flow-insensitive point-to analysis. In *24th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages.*

SHARIR, M. AND PNUELI, A. 1981. Two approaches to interprocedural data flow analysis. In *Program Flow Analysis: Theory and Applications*, S. S. Muchnick and N. D. Jones, Eds. Prentice-Hall, Chapter 7, 189–234.

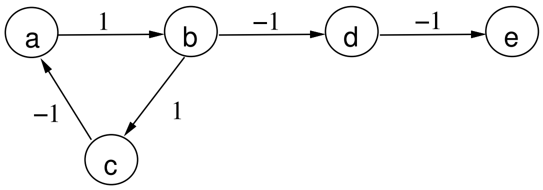

Fig. 28. Problematic Alias Graph

Shivers, O. 1988. Control flow analysis in Scheme. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*. 164–174. *SIGPLAN Notices, 23(7)*.

Soroker, D., Karasick, M., Barton, J., and Streeter, D. 1997. Extension mechanisms in Montana. In *8th IEEE Israeli Conference on Software and Systems*.

SPEC. 1995. SPEC CPU95, Version 1.0. Standard Performance Evaluation Corporation, http://www.specbench/org.

Sreedhar, V. C. and Gao, G. R. 1995. A linear time algorithm for placing ϕ-nodes. In *22nd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*. 62–73.

Steensgaard, B. 1996. Points-to analysis in almost linear time. In *23nd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*.

Stocks, P. A., Ryder, B. G., Landi, W. A., and Zhang, S. 1997. Comparing flow and context sensitivity on the modifications-side-effects problem. Tech. Rep. DCS-TR-335, Department of Computer Science, Rutgers University. Aug.

Tarjan, R. 1981. Fast algorithms for solving path problems. *Journal of the Association for Computing Machinery 28*, 3, 594–614.

Tarjan, R. E. 1972. Depth-first search and linear graph algorithms. *SIAM Journal of Computing 1*, 146–160.

Weihl, W. 1980. Interprocedural data flow analysis in the presence of pointer, procedure variables and label variables. In *7th Annual ACM Symposium on the Principles of Programming Languages*. 83–94.

Wilson, R. P. and Lam, M. S. 1995. Efficient context-sensitive pointer analysis for C programs. In *SIGPLAN '95 Conference on Programming Language Design and Implementation*. 1–12. *SIGPLAN Notices, 30(6)*.

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

APPENDIX

A. INTERPRETING ALIAS GRAPHS USING DEFERRED EVALUATION

Section 3 presents a straightforward recursive algorithm for answering alias queries, called ComputeAliases. This section describes an enhanced version of this algorithm that is required when the flow-insensitive analysis with deferred evaluation is performed (Section 4.2.4).

Recall that to determine the aliases of $**p$, a call to ComputeAliases is made using pointer variable p and 2 levels of indirection, i.e., ComputeAliases(p, 2). In this algorithm, as each edge is traversed, the dereference level is decremented until it becomes zero. This is equivalent to adding the value of this edge, which is -1, to the value of the original query as the edge is traversed.

A simple extension to the recursive algorithm of Figure 3 in Section 3 would attempt to accommodate deferred alias edges by using the weight of each edge as the delta value on subsequent recursive calls to Expand; i.e., regular alias edges still add -1 to DerefLevel, but deferred alias edges add their associated value. This simple extension is not correct because it may not terminate in the presence of cycles in the alias graph. Consider the alias graph of Figure 28, which could result from the following statements:

$$
\begin{aligned}
a &= *b \\
b &= *c \\
c &= \&a \\
b &= \&d \\
d &= \&e
\end{aligned}
$$

This graph is problematic for the extended algorithm because it contains a cycle in which the sum of alias edges is non-negative. For example, the call ComputeAliases(a, 1) could result in recursive calls that visit a node with a DeferLevel not smaller than the previous visit. As this can happen for an unbounded number of traversals, termination is not guaranteed.

The algorithm presented in Figures 29 and 30 addresses this problem and correctly interprets deferred evaluation alias edges with no loss of precision. The following observation is instrumental to the formulation of the algorithm:

> Since there are a fixed number of nodes, N, (the number of named objects) in an alias graph, there is an upper bound, N^2, on the number of regular alias edges that all deferred alias edges can represent.[33]

The goal of the algorithm is to answer the alias query, by expanding all deferred alias edges encountered into the regular alias edges they represent. Recall that the intraprocedural iteration (Steps S_7 – S_{13} of Figure 9 in Section 4.2.4) does not utilize intraprocedural control flow information, and instead assumes all paths are possible. As the full effects of a deferred alias edge may not be realized until this iteration converges, a deferred alias edge, expanded as part of an alias query during this iteration, cannot be removed because all of the aliases it represents may not be present.

If deferred alias edges are expanded after the intraprocedural iteration, as suggested in Step 4 of Figure 21, then they can be removed.

The expansion of a deferred alias edge can depend on the expansion of another deferred alias edge. As a dependence cycle may arise, care must be taken in formulating the termination condition.

The algorithm will terminate when either:

(1) there are no more deferred alias edges to expand to answer the query, or
(2) the deferred alias edges needed to answer the query that are apparently not fully expanded cannot be expanded further. This occurs when an attempt to expand all such edges results in no new regular alias edges, and in not finding any new deferred alias edges that require expansion. The alias graph of Figure 28 illustrates such a case.

[33] Although the alias instance representation described in Section 6 allow more than one edge (alias relation) between two nodes (named objects), the number of such edges can be bounded.

```
    /* Global Variables */
solution : Set of Variables
expand_list : List of Deferred Alias Edges

function ComputeAliases(Var, DerefLevel) returns Set of Variables
begin
    solution ← {}
    expand_list ← {}

    /* Step 1 */
    if Expand(Var, DerefLevel, False, Null) = Success then
        return solution
    end if

    /* Step 2 */
    repeat
        foreach edge, e = (s, t) with value v on expand_list loop
            if Expand(t, v + 1, False, s) = Success then
                remove e from expand_list
                mark e in the alias graph as "expanded"
            end if
        end loop
    until expand_list does not change and no regular alias edges are created
    Mark any remaining edges on expand_list as "expanded"

    /* Step 3, only the side effects of the following call are used.
        It will always return success. */
    Expand(Var, DerefLevel, False, Null)
    Unmark all "expanded" edges in the alias graph
    return solution
end ComputeAliases
```

Fig. 29. Enhanced ComputeAliases Algorithm

```
function Expand(Var, DerefLevel, Traversed_Deferred, SourceNode) returns boolean
    status : boolean
begin
    status ← Success
    if DerefLevel = 0 then                    /* found a node of interest */
        if SourceNode = Null then         /* original query call (Step 1 or 3) */
            solution ← solution ∪ {Var}
        else                                      /* Step 2 call */
            create regular (−1) alias edge from SourceNode to Var
        end if
        return status
    end if

    /* DerefLevel > 0 */
    foreach alias edge, e = (Var, Target) with value v and not marked "expanded", loop
        if v = −1 then
            if Expand(Target, DerefLevel - 1, Traversed_Deferred, SourceNode) = Fail then
                status ← Fail
            end if
        else                /* v ≥ 0, i.e., deferred alias edge */
            if not Traversed_Deferred then
                /* Incorporate v, but pass "True" since we've seen a deferred edge */
                if Expand(Target, DerefLevel + v, True, SourceNode) = Fail then
                    status ← Fail
                    if e is not already present on expand_list then
                        add e to expand_list
                    end if
                end if
            else            /* 2nd deferred edge, could be a cycle, so abandon this path */
                status ← Fail
                if e is not already present on expand_list then
                    add e to expand_list
                end if
            end if
        end if
    end loop
    return status
end Expand
```

Fig. 30. Enhanced Expand Algorithm

As before, the algorithm consists of two functions: ComputeAliases and Expand, which are presented in Figures 29 and 30, respectively. Note that Expand has two additional parameters, which will be discussed later in this section.

ComputeAliases is comprised of three steps, each of which calls Expand. The second and third steps are only performed if a non-negative cycle is detected. The first step calls Expand to satisfy the initial alias query. An unsuccessful initial query results in at least one deferred alias edge being placed on expand_list, a list of edges that need to be expanded to answer the query.

During the second step, deferred alias edges on expand_list are used to generate new alias queries via calls to Expand. Once a deferred alias edge has been successfully expanded, it is marked as "expanded." Marking a deferred alias edge as "expanded" has the effect of temporarily deleting the edge for this call to ComputeAliases because it will not be considered on all subsequent calls to Expand, including those from Step 3. The iteration over the edges on expand_list continues until a pass is made that results in no new regular alias edges being created, and in expand_list being unchanged (including the case in which expand_list is empty). Any deferred alias edges remaining on expand_list do not represent any regular alias edges. For this reason, all such edges are marked "expanded" at the conclusion of Step 2. Consequently, once the second step completes, all deferred alias edges that are required for the original query have been marked "expanded."

The third step involves reissuing the initial query by calling Expand. As all deferred edges of interest have been expanded, it is guaranteed to succeed. The last part of Step 3 resets all such edges so that they are no longer marked "expanded." This results in these edges being considered on subsequent alias queries.

Figure 30 contains the code for Expand. As in the previous algorithm Expand continues recursively until a non-negative cycle is detected. If such a cycle is detected, each deferred alias edge that has been traversed on the current path is placed on the list of edges to be expanded, called expand_list, if it is not already on this list. Then, Fail is returned to ComputeAliases. If no non-negative cycle is detected, the query has been satisfied, and Success is returned. If any paths through the alias graph satisfy the original query, the appropriate variables are placed in the set result as in the previous algorithm.

Any non-negative cycle detection algorithm can be used in Expand. Since this algorithm is only used to trigger the enhanced version of Expand, a simple conservative approach to detecting a non-negative cycle is employed. The third parameter to Expand (the flag Traversed_Deferred) is used to signify whether a deferred alias edge has been traversed on the current path. If a non-negative edge is considered in the foreach loop of Expand,[34] and Traversed_Deferred is True, a non-negative cycle may exist, and the actions mentioned above are performed. This algorithm ensures that all non-negative cycles are detected. A more precise (and costly) cycle detection algorithm [Tarjan 1972; Eve and Kurki-Suonio 1977] can be used to potentially reduce the number of cycles detected, which, in turn, can improve the overall efficiency of the ComputeAliases algorithm. However, the choice of cycle detection algorithm does not affect the precision of the ComputeAliases algorithm.

Some other characteristics of the algorithm are:

[34] This includes considering the same non-negative edge previously traversed.

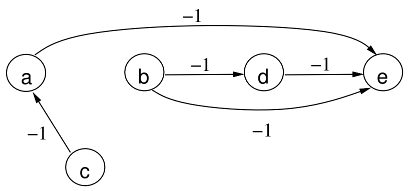

Fig. 31. Resulting Alias Graph

```
S1:    q = &x;
S2:    if (...) {
S3:        q = &y;
S4:        p = &z;
S5:    }
S6:    *p = q;
```

Fig. 32. Example Where Birth Site and Alias History Do Not Prevent Invalid Combination

—If no deferred alias edges are traversed, the algorithm behaves as the algorithm presented in Figure 3.

—The 4th parameter to Expand is used when deferred alias edges are expanded (Step 2 of ComputeAliases). It represents the source node of any regular alias edge that is created by Expand.

—Calls to Expand may result in new deferred alias edges being added to expand_list. These new edges can either be considered during the iteration of expand_list when they are added, or during the next iteration.

The above algorithm applied to the alias graph of Figure 28 for the alias query ComputeAliases(a, 1) results in the alias graph given in Figure 31. This results in $\{e\}$ being returned, signifying that a points to e. This result is consistent when considering the effect of the original five statements, in any order, assuming no alias relations hold before these statements and no aliases are killed.

The ComputeAliases algorithm presented in this section (Figures 29 and 30) is only required when deferred evaluation edges are present. Deferred evaluation edges are created to help improve the efficiency of the intraprocedural iteration of the flow-insensitive algorithm. If this technique is not employed, the simple algorithm of Section 3 will suffice.

B. INVALID COMBINATION EXAMPLE

Section 6 illustrated how birth site and alias history information can refine alias information to improve the precision of alias analysis by preventing invalid alias combinations. However, as mentioned in Section 6.3, there are cases where this information still fails to detect invalid combinations. This section provides such an example.

Consider Figure 32. Alias instance $AI1 = [\langle *q, x\rangle, S1, null, \{\}]$, generated at $S1$ is propagated through $S2$ to $S3$ and to $S6$. It is killed at $S3$ by the assignment to

q, which generates a new alias instance, $AI2 = [\langle *q, y\rangle, S3, null, \{\}]$. $S4$ generates the alias instance $AI3 = [\langle *p, z\rangle, S4, null, \{\}]$, which along with $AI2$ is propagated to $S6$. Because $AI1$ propagated to $S3$ is killed by $S3$, $AI1$ holding at the entry of $S6$ is the one propagated from $S2$. $AI3$ holding at the entry of $S6$ is propagated from $S4$.

The assignment statement at $S6$ potentially combines $AI1$ and $AI3$, $AI1 \otimes AI3$, which is an invalid combination. However, because there exists an execution path between the birth sites of $AI1$ and $AI3$, i.e., $alias{-}birth{-}path(AI1, AI3)$ holds, the method of using alias instances with birth sites and histories fails to identify that $AI1$ and $AI3$ should not be combined, even though they are not propagated along the same execution path.

Recording full path information would allow the detection of this invalid combination, but can be exponential in the size of the CFG. A similar example can be constructed interprocedurally using the PCG.