Interprocedural Pointer Alias Analysis

MICHAEL HIND, MICHAEL BURKE, PAUL CARINI, and JONG-DEOK CHOI
IBM Thomas J. Watson Research Center

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 the following contributions: (1) a framework for interprocedural pointer alias analysis that handles function pointers by constructing the program call graph while alias analysis is being performed; (2) a *flow-sensitive* interprocedural pointer alias analysis algorithm; (3) a *flow-insensitive* interprocedural pointer alias analysis algorithm; (4) a *flow-insensitive* interprocedural pointer alias analysis algorithm that incorporates *kill* information to improve precision; (5) empirical measurements of the efficiency and precision of the three interprocedural alias analysis algorithms.

Categories and Subject Descriptors: D.3.4 [**Programming Languages**]: Processors—*Optimization*

General Terms: Algorithms

Additional Key Words and Phrases: Interprocedural analysis, pointer aliasing, program analysis

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 optimizing compilers, program environments, and understanding tools. It can also be used in a software-testing system or to provide compiler and run-time support for the parallel execution of programs originally written in sequential languages.

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

This work was performed when the first author was at SUNY at New Paltz and IBM Research. The work by the first author was supported in part by the National Science Foundation under grant CCR-9633010, by IBM Research, and by SUNY at New Paltz Research and Creative Project Awards.
Authors' address: IBM Thomas J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598; email: {hind;burkem;carini}@watson.ibm.com; jdchoi@us.ibm.com.

allocated data structures, such as C, C++, Fortran 90, Java, and LISP, is more challenging 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, and therefore must compute a summarized solution for all program points in a procedure, which reflects any possible control flow path. Flow-insensitive analyses can be more efficient than flow-sensitive algorithms and have been primarily used for problems for which flow-sensitive algorithms do not provide increased 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 article, we describe approximation methods for computing interprocedural aliases for a program written in a language that includes pointers, reference parameters, and recursion. We present the following contributions:

—a framework for interprocedural pointer alias analysis that handles function pointers by constructing the program call graph while alias analysis is being performed;

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

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

—a *flow-insensitive* interprocedural pointer alias analysis algorithm that incorporates *kill* information to improve precision;

—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 article is organized as follows. Section 2 defines terminology used in the article and describes the background context, including our alias representation. Section 3 presents our analysis framework and the flow-sensitive and flow-insensitive interprocedural algorithms, as well as an extension of the flow-insensitive algorithm that incorporates kill information. Section 4 provides empirical results that measure and contrast the efficiency and precision of the three interprocedural pointer alias analysis algorithms. Section 5 presents the complexity of the algorithms. Section 6 provides comparisons of our methods and results with relevant work. Section 7 draws conclusions. Appendix A illustrates precision trade-offs between alias representations.

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 dereference operators, and structure field selection operators. In C such an expression would include a variable with a possibly empty sequence of the following operators: "$*$" (dereference), "." (field selection), and "$\rightarrow$" (dereference and field selection). 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 article addresses the computation of may-aliases, which includes must-aliases as 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` or Fortran's `EQUIVALENCE` statement. These aliases do not change within a program and can be computed from semantic information. Our algorithms assume that structural alias information is available and use this information as input.

We use $\langle x, y\rangle(\equiv \langle y, x\rangle)$ to denote that access paths x and y are may-aliases at a program point. For example, after the statement

$$a = \&b;$$

is executed, $*a$ and b refer to the same storage location and thus become aliases of each other, which can be expressed as the *alias relation* $\langle *a, b\rangle$. The trivial alias $\langle x, x\rangle$ holds for all access paths x, provided x does not result in a dereference of the null pointer. If x and y are nonnull pointer-valued access paths, $\langle x, y\rangle$ implies $\langle *x, *y\rangle$.

Interprocedural data flow analyses make use of the *program call graph* (PCG), which is a flow multigraph in which each procedure is represented by a single node and in which an edge (f, g) represents a potential call of procedure g from a call site in procedure f. In the presence of function pointers or virtual methods a call site may contribute multiple edges to different procedures.

The conventional methods for side-effect analysis [Banning 1979; Burke 1990; Cooper and Kennedy 1988] 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; 1998].

We describe various alias analysis techniques for features occurring in common imperative pointer languages such as C, C++, and Fortran 90. Our analysis supports such language features as the address operator, pointer assignment, pointer dereference, recursive calls, calls through function pointers, dynamic memory allocation, and aggregates (arrays and structs). As described, our methods do not support pointer arithmetic, variable argument lists, and system features such as the Unix `longjmp` subroutine and signal/exception handlers. Our methods treat a reference to an element of an array as a reference to the entire array.

2.1 The Compact Representation of Alias Information

Our representation refers to memory locations associated with names as *named objects* [Chase et al. 1990; Horwitz et al. 1989]. These names can be either variable

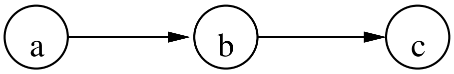

Fig. 1. Example directed alias graph.

names or synthetic names created by the analysis that represent a set of memory locations, such as parts of the heap. Any method for naming dynamically allocated objects, such as the *named instance approach* [Burke et al. 1997] summarized in Section 6.6.2, may be employed.[1] An element of a `struct` or `class` is identified by its offset.

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

(1) at least one access path component is a named object, i.e., it does not contain a dereference, and
(2) neither access path component has more than one level of dereferencing.

Therefore, with a compact representation, all alias relations are of the form $\langle *a, b\rangle$ ($\equiv \langle b, *a\rangle$) or $\langle a, b\rangle$ ($\equiv \langle b, a\rangle$), where a and b are named objects.

Although the representation is defined in terms of alias relations, relations containing a pointer dereference can be mapped to and from a (directed) graph-based representation. We refer to such a graph as an *alias graph*. For example, consider the graph in Figure 1, where each node represents a named object, and an edge $a \rightarrow b$ represents the alias relation $\langle *a, b\rangle$. The explicit list of aliases, referred to as the *explicit representation*, holding for this example is $\{\langle *a, b\rangle, \langle *b, c\rangle, \langle **a, c\rangle, \langle **a, *b\rangle\}$. However, $\langle ** a, c\rangle$ and $\langle ** a, *b\rangle$ can be inferred from $\{\langle *a, b\rangle, \langle *b, c\rangle\}$, which is the compact representation of these alias relations.

Figure 2 presents a straightforward recursive algorithm that uses the compact representation to compute the explicit aliases of an access path. Two arguments are passed to the function *ComputeAliases*: the named object of interest and the number of dereferences. (If the variable is a pointer to a `struct`, additional information concerning the field offsets for each level of dereference would also be required.) This algorithm traverses the graph implied by alias relations with a pointer dereference. The result of $ComputeAliases(n, k)$ is the set of all nodes (named objects) in the graph along a path of length k from n. For example, to determine the aliases of $**a$ the following function call would be made: $ComputeAliases(a, 2)$. For the graph of Figure 1, $ComputeAliases(a, 2)$ would call $Expand(a, 2)$, which would call $Expand(b, 1)$, which would eventually return $\{c\}$.

The Appendix provides examples that illustrate how the compact representation and the explicit representation are incomparable—each can be more precise than the other—and characterizes the trade-offs in precision between the two representations [Marlowe et al. 1993].

The compact representation, introduced in Choi et al. [1993], is highly similar to the *points-to* representation [Emami et al. 1994; Ghiya 1992]. The major difference

[1] The implementation described in Section 4 names objects based on their allocation site [Chase et al. 1990; Horwitz et al. 1989; Hudak 1986; Jones and Muchnick 1981; Ruggieri and Murtagh 1988; Wilson and Lam 1995].

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

procedure Expand(Var, DerefLevel, Solution (by reference))
begin
    if DerefLevel = 0 then
        Solution ← Solution ∪ {Var}
    else
        foreach alias relation ⟨*Var, Target ⟩ /* or alias edge, e = (Var, Target) */, loop
            call Expand(Target, DerefLevel - 1, Solution);
        end loop
    end if
end Expand
```

Fig. 2. Algorithm for answering an alias query.

between the representations is that the dereference of the first element is implied in the points-to representation, i.e., $\langle *a, b\rangle$ is written as (a, b, P), where P signifies the (may) alias is possible. The compact representation can also represent reference parameter alias pairs.

The compact representation, like the points-to representation, is not equivalent to performing a *transitive reduction* [Aho et al. 1972] over the alias graph during alias analysis. The compact and points-to representations both consider path length information, i.e., number of dereferences, when combining alias relations. A transitive reduction representation does not capture this information.

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. Although 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 pointer expressions, precludes this problem.

2.2 The Precision of Alias Analysis

A less precise alias analysis will conservatively report more alias relations representing real memory locations than a more precise analysis. This spurious information can lead to less precise and efficient client analyses that use alias information [Pioli 1999; Shapiro and Horwitz 1997a].

There are several factors that can affect the precision (and cost) of alias analysis. These include the following:

—the use of flow sensitivity;
—the use of context sensitivity;

—the manner in which aggregates (arrays and structs) are modeled;

—the manner in which the heap is modeled;

—the alias representation.

For many of these factors a spectrum of choices exists. For example, full context sensitivity is used in Emami et al. [1994] and Wilson and Lam [1995]; limited context sensitivity (of different types) is used in Landi and Ryder [1992], Choi et al. [1993], and Burke et al. [1997]; and no context sensitivity is used in Burke et al. [1995], Steensgaard [1996], and Shapiro and Horwitz [1997b].

3. 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 hold in the called procedure. Thus, aliases can be interprocedurally propagated from a call site to the entry point of a called procedure, and from a return point of a called procedure to the point just after the call site. This section presents three interprocedural algorithms for computing pointer aliasing that utilize a common framework. The first algorithm is flow-sensitive; the second is flow-insensitive; the third utilizes precomputed flow-sensitive information during the flow-insensitive analysis.

The framework associates pointer alias information (represented as sets of aliases) with various points in the program. Two additional alias sets are associated with each procedure: one containing alias relations that hold on entry to the procedure, the other representing the potential 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 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. The framework can be used with various levels of *context sensitivity* [Burke et al. 1997], i.e., the degree in which a function is analyzed with knowledge of its calling context.

During an 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. Thus, this method reduces the storage requirements of the interprocedural iterations by only requiring the presence of the intraprocedural sets of the currently analyzed procedure.

The final intraprocedural alias information is computed by 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 to be 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, the final pass can also improve the precision of the flow-insensitive interprocedural

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

Fig. 3. Interprocedural aliasing framework.

algorithm, because flow-sensitive information is used intraprocedurally.

Figure 3 presents the framework for interprocedural pointer alias analysis. Step S_1 builds the initial (optimistic) program call graph, which is discussed further in Section 3.1. Steps S_2–S_4 initialize the interprocedural sets. While the algorithm iterates, optimistic assumptions are made concerning the contents of these sets, i.e., they initially contain no alias relations. Structural and reference parameter aliases are incorporated as needed, for correctness. Iteration over the PCG occurs in topological order[2] (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 3.1). To improve the efficiency of this iteration (Steps S_5–S_{14}), a worklist can be employed.

The major characteristic of the algorithm is the interleaving of the intraprocedural (Step S_7) and interprocedural phases (Step S_8). The intraprocedural phase can be either flow-sensitive (Section 3.2) or flow-insensitive (Section 3.3).

3.1 Function Pointer Analysis

This section describes 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 using the interprocedural aliasing framework (Figure 3). Function pointer analysis and similar problems that are either more general or less general have been solved by others and are described in Section 6.4.

When embedded in the alias analysis, which generally occurs before any other phases that need the PCG, the method incurs minimal overhead. We will use

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

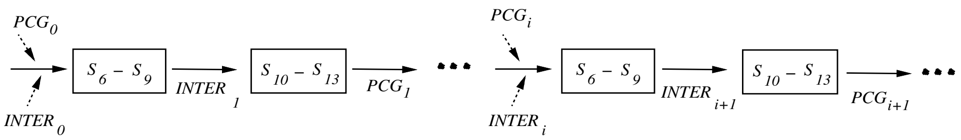

Fig. 4. Interleaving of alias phases and PCG update.

the term *function pointers* to refer to function parameters, function variables, and function pointers.

PCG construction is performed in an optimistic manner; all indirect calls are initially assumed to call no functions. The technique first handles function pointers in the same manner as other pointers. However, it uses function pointer alias information to identify new PCG edges as the computation proceeds. The method does not compute aliasing from scratch as new PCG edges are found, but instead adds these new edges to the PCG and incrementally continues the alias computation, using the alias information computed with the previous PCG.

Formally, this amounts to computing an additional *interprocedural* set of pairs, $(c, proc)$, where c is a call site invoked through a function pointer expression, fp,[3] and $proc$ is a called procedure. Each (called) procedure component in this set is aliased to fp at c, i.e.,

$$FuncAlias = \{\, (c, proc) \mid \langle fp, proc \rangle \in \text{ alias set for } c,\ fp \text{ is invoked at call site } c \}.$$

In Step S_{10} of Figure 3 the PCG is updated by adding new call edges identified from $FuncAlias$. In the case where an edge results in adding a new procedure to the PCG, the alias sets for this procedure are optimistically initialized by Steps S_{11}–S_{13}.

Figure 4 shows the relationships among these alias analysis phases, interprocedural alias information, and PCG construction. At the start, the algorithm computes the initial interprocedural alias information, $INTER_0$, and builds the initial PCG, PCG_0, in which there is a single edge for each call site of direct procedure calls, and no edges for call sites of function pointers (Step S_1).

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. $INTER_{i+1}$ may update $FuncAlias$, from which the new PCG, PCG_{i+1}, is built (Steps S_{10}–S_{13}). This process continues until the interprocedural alias information and $FuncAlias$ set converge. With this approach, unlike conventional PCGs, a call site using a function pointer may contribute multiple edges to the PCG, one for each procedure aliased to the function pointer.

When applied only to function pointers, the algorithm provides a method for constructing the PCG. This algorithm can be extended to handle virtual method calls in languages such as C++ and Java as described in Carini et al. [1995].

[3]In the case of function variables or parameters fp would not contain any dereferences.

3.2 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 3.2.1 and 3.2.2, respectively. Section 3.2.3 illustrates the algorithm with an example.

We use a sparse representation, the *sparse evaluation graph* (SEG) [Choi et al. 1991], for intraprocedural alias analysis, which is intuitively a subset of the original CFG containing only "interesting" CFG nodes and the edges needed to connect them. More formally, we define $SEG = \langle N_{SEG}, E_{SEG}, N_{Entry}, N_{Exit}\rangle$, where N_{SEG}, the set of nodes in the sparse evaluation graph, is the union of the following:

—$GenNodes$, the set of CFG nodes where alias information is potentially modified;[4]

—$MeetNodes$, the set of CFG 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} and therefore represents control flow in the SEG.

3.2.1 *Interprocedural Phase.* Consider Figure 5. For each procedure, f, the interprocedural phase supplies the sets $Entry_f$ and $Exit_f$, which correspond to the aliases that hold on entry to f and on exit from f, respectively. $Entry_f$ is initialized to the set of structural aliases for f (Step S_3). $Exit_f$ is initialized to the empty set.

In the figure, $ForwardBind_f^c$ maps the set of aliases holding at call site c of procedure f into aliases holding in the called procedure, using the argument/parameter mappings of c. $BackwardBind_f^c$ maps aliases holding at the exit of the called procedure into aliases holding immediately following c. Details of these mapping mechanisms are given in Section 3.5. Reference parameter aliases are handled uniformly with pointer alias analysis.

Traversals over the PCG are repeated until convergence (the outer loop of Steps S_6–S_{33}). On each traversal Step S_{22} updates the $Exit$ set of the procedure being processed using the intraprocedural information. At Steps S_{23}–S_{27}, the alias information immediately before a call site, say In_c, is propagated forward to the $Entry$ set of each called routine, $Entry_g$. (Recall that indirect calls can lead to multiple called routines.) This is accomplished by performing the union of $ForwardBind_f^c(In_c)$ and the previous value of $Entry_g$.

3.2.2 *Intraprocedural Phase.* The intraprocedural alias computation, Steps S_8–S_{21}, can be formulated as a data flow framework [Rosen 1979; Tarjan 1981]. 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

[4] 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, f, in the PCG, loop
S_3: $Entry_f \leftarrow$ structural aliases for f
S_4: $Exit_f \leftarrow \{\}$
S_5: end loop
S_6: repeat
S_7: foreach procedure, f, in the PCG, loop
S_8 repeat
S_9: foreach SEG node, s, loop
compute flow-sensitive intraprocedural aliasing:
S_{10}: compute In_s using Equation (1)
S_{11}: if s is a call site then
S_{12}: foreach called procedure, g, at s, loop
S_{13}: compute Out_s using $BackwardBind_f^s(Exit_g)$ and In_s
S_{14}: end loop
S_{15}: else if s is a pointer assignment then
S_{16}: compute Out_s using Equation (2)
S_{17}: else
S_{18}: $Out_s = In_s$
S_{19}: end if
S_{20}: end loop
S_{21}: until the intraprocedural alias sets for f converge
S_{22}: $Exit_f \leftarrow Out_{N_{Exit}}$
S_{23}: foreach call site, c, in f, loop
S_{24}: foreach called procedure, g, at c, loop
S_{25}: update $Entry_g$, using $ForwardBind_f^c(In_c)$
S_{26}: end loop
S_{27}: end loop
S_{28}: end loop
S_{29}: update the PCG using new function pointer aliases
S_{30}: foreach new procedure, f, added to the PCG in Step S_{29}, loop
S_{31}: Initialize $Entry_f$ and $Exit_f$
S_{32}: end loop
S_{33}: until the interprocedural alias sets and PCG converge

Fig. 5. Flow-sensitive interprocedural aliasing algorithm.

of aliases holding at the N_{Entry} node of the SEG (the *Entry* for the procedure), and the *BackwardBind* sets corresponding to call-site nodes in the SEG. The *BackwardBind* sets are constructed from the *Exit* sets of the called procedures.

Let In_n and Out_n be the set of alias relations holding immediately before and immediately after SEG node n, respectively. For a node n, In_n is the union of the *Out* sets of its predecessor nodes:

$$In_n = \bigcup_{p \in Preds(n)} Out_p \qquad (1)$$

We now describe the transfer function (using the compact representation) for an assignment statement to an access path of a pointer type. Consider the assignment statement "$p_i = q_j$," where p_i is a pointer expression with i levels of dereference from variable p and q_j is a pointer expression with j levels of dereference from variable q.[5] For example, $p_2 = **\,p$. The transfer function is

$$Out_n = (In_n - Must(ComputeAliases(p,i)) \cup \bigcup_{\substack{a \in ComputeAliases(p,i),\\ b \in ComputeAliases(q,j+1)}} \langle *a, b\rangle \tag{2}$$

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 define *Must* in an optimistic manner: if during the analysis a pointer expression is aliased to exactly one object that represents one run-time memory location, we treat it as a must alias.[6] To preserve monotonicity, if *ComputeAliases* returns no objects, *all* alias relations are killed, i.e., we define $Must(\{\})$ to be all alias relations.[7]

As described in (2), two calls to *ComputeAliases* may occur when applying the transfer function, each returning a list of named objects. The transfer function creates alias pairs from these lists by combining each element of one with every element of the other.

Consider the assignment "$*p = p$," with the aliases $\langle *p, a\rangle, \langle *p, b\rangle$ holding before the assignment. Applying the transfer function in (2) would result in the following aliases being generated:

$$\langle *a, a\rangle, \langle *b, b\rangle, \langle *a, b\rangle, \langle *b, a\rangle$$

However, the last two aliases cannot hold because p can only point to a or b, but not both, during a single execution. Thus, a more precise solution can be obtained by refining the transfer function to consider the alias paths traversed in the two calls to *ComputeAliases*.[8]

The aliases holding at the N_{Entry} node and the call-site nodes are interprocedurally computed, as described in Section 3.2.1. If a procedure f calls a procedure g, $Exit_g$, which is computed during the previous iteration, is propagated back to the call site in f (Step S_{13}). Although a called procedure can affect the aliases holding on return to the call sites that invoke it, $Exit_g$ of a procedure g does not uniformly affect all the procedures that call g. Rather, the aliases holding on return from a call site are affected only by a subset of $Exit_g$. 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.

---

[5] We represent "$\&q$" as q_{-1}.

[6] Excluded from this treatment are aggregates, if the individual elements of such structures are not distinguished.

[7] For programs that do not dereference uninitialized pointers, *ComputeAliases* returns no objects only when the statements that create aliases for the pointer expression of interest have not yet been processed. Defining $Must(\{\})$ to be all alias relations results in $Out_n = \{\}$ (the "Gen" component will be empty), which is consistent with the optimistic nature of our algorithm. A similar effect occurs when processing a call to a procedure that has not been analyzed; the procedure's initial *Exit* set, $\{\}$, is used as the *Out* set for the call.

[8] This imprecision can occur in a more general manner, such as a statement "$**x = *y$," where $\langle *x, p\rangle$ and $\langle *y, p\rangle$ hold along with the above alias relations.

```
        char *u, a, b;
        f() {
S1:         u = &a;
S2:         *u = 'A';
S3:         g();
S4:         *u = 'B';
        }
        g() {
S5:         u = &b;
        }
```

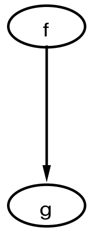

Fig. 6. Example program segment and its PCG.

Identifying call sites whose aliases have induced an alias relation in $Exit_g$ is related to identifying unrealizable execution paths and is discussed in Burke et al. [1997].

After convergence, each node n has a solution Out_n that is consistent with In_n and the transfer function at every node.

The flow-sensitive intraprocedural algorithm can also be employed when analyzing incomplete programs, such as libraries. To preserve correctness, conservative assumptions (all possible interprocedurally induced aliases) would be used at points where no information is available, such as the N_{Exit} node of unavailable procedures.[9] Scoping rules can be used to improve these worst-case assumptions.

3.2.3 *Example.* We illustrate our flow-sensitive algorithm with the program in Figure 6. During the first analysis of f, S1 generates the alias $\langle *u, a\rangle$, resulting in $Out_{S1} = In_{S2} = Out_{S2} = In_{S3} = \{\langle *u, a\rangle\}$. Since g has not yet been analyzed, $Exit_g = \{\}$. Thus, $Out_{S3} = In_{S4} = Out_{S4} = Exit_f = \{\}$. Before g is analyzed, the alias associated with the call site at S3 is propagated forward along the PCG edge to g (Steps S_{21}–S_{23} in Figure 5), giving $Entry_g = \{\langle *u, a\rangle\}$. The analysis of g sets $Exit_g = \{\langle *u, b\rangle\}$, because of the pointer assignment at S5, which both kills the alias $\langle *u, a\rangle$, and generates the alias $\langle *u, b\rangle$. Thus, $Out_{S5} = Exit_g = \{\langle *u, b\rangle\}$. This completes one iteration of the repeat loop (Steps S_6–S_{29}). Since alias information has changed from the initial values, another iteration of this loop occurs.

During the second analysis of f, the alias $\langle *u, b\rangle$ is present in $Exit_g$, and thus is propagated to Out_{S3}, In_{S4}, Out_{S4}, and $Exit_f$. No further changes occur. At the conclusion of the flow-sensitive analysis, we have the following results:

$$Entry_f = In_{S1} = \{\},$$

$$Out_{S1} = In_{S2} = Out_{S2} = In_{S3} = Entry_g = In_{S5} = \{\langle *u, a\rangle\},$$

$$Out_{S3} = In_{S4} = Out_{S4} = Exit_f = Out_{S5} = Exit_g = \{\langle *u, b\rangle\}.$$

This example assumed that all statements were SEG nodes, resulting in identical alias information being saved in more than one set. A SEG representation would not include nodes S2 and S4 because they do not affect pointer aliasing, effectively

[9] Worst-case assumptions would also be made for the entry sets of any procedure that can be called by a missing procedure. Examples of such called procedures are those passed as a parameter to a library.

reducing both storage requirements and analysis time. A further optimization is to share certain common alias sets, such as Out_{S1} and In_{S3}. The combination of these techniques results in a 64–83% reduction of alias sets and a 1.85–4.50 speedup in analysis time in our benchmark suite [Hind and Pioli 1998a; Pioli 1999]. This combination allows the association of alias sets with non-SEG nodes without explicitly visiting such nodes, and it serves the same purpose as the mapping function described in Choi et al. [1991]

3.3 Flow-Insensitive Pointer Alias Analysis

This section presents a flow-insensitive alias analysis algorithm, which is also based on the interleaving framework. An earlier formulation of this algorithm appears in Burke et al. [1995]. After describing the interprocedural and intraprocedural phases of the algorithm, we describe how to improve the precision of the interprocedural phase by incorporating precomputed kill information.

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. For pointer-induced aliasing, the information directly generated at a PCG 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 PCG edge. With this information for each node and each edge of the PCG as input, the flow-insensitive interprocedural alias analysis represents the effects of a procedure f, by computing $PGen_f$, the set of aliases *generated* by the invocation of f.[10] This differs from the flow-sensitive algorithm, which represents the effects of a procedure by recording the alias information that *holds* on exit from a procedure.

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

—$Entry_f$, the set of aliases that hold upon entry to procedure f;

—$Holds_f$, the set of aliases that may hold at any point in f;

—$IGen_f$, the set of aliases *generated* by all of the pointer-assignment statements of procedure f;

—$CSGen_f$, the set of aliases *generated* by call sites in the procedure f.

This section describes the relationship among these sets and provides a set of data flow equations representing this relationship. As $Entry_f$ and $PGen_f$ are used or updated by other procedures, these interprocedural alias sets persist throughout the alias computation. The other intraprocedural sets, $IGen_f$, $CSGen_f$, and $Holds_f$, can be discarded after their results are incorporated into the appropriate interprocedural set.

Figure 7 presents our flow-insensitive interprocedural algorithm. The computation of the intraprocedural sets (Step S_8) uses the interprocedural sets. Computing

[10] The computation of *kill* information requires control flow information, which is generally not available in a flow-insensitive analysis.

```
S1:  build the initial PCG
S2:  foreach procedure, f, in the PCG, loop
S3:      Entry_f ← structural aliases for f
S4:      PGen_f ← {}
S5:  end loop
S6:  repeat
S7:      foreach procedure, f, in the PCG, loop
S8:          using the interprocedural sets (Entry_f and PGen_g for all g called by f)
                 compute intraprocedural sets, CSGen_f, Holds_f and IGen_f,
S9:          PGen_f ← CSGen_f ∪ IGen_f
S10:         foreach call site, c, in f, loop
S11:             foreach called procedure, g, at c, loop
S12:                 update Entry_g using ForwardBind^c_f(Holds_f)
S13:             end loop
S14:         end loop
S15:     end loop
S16:     update the PCG using new function pointer aliases
S17:     foreach new function, f, added to the PCG in Step S16, loop
S18:         initialize Entry_f and PGen_f as in Steps S3 and S4
S19:     end loop
S20: until the interprocedural alias sets and PCG converge
```

Fig. 7. Flow-insensitive interprocedural aliasing algorithm.

the interprocedural sets (Steps S_9–S_{14}), in turn, uses the intraprocedural sets. Once again, the iterations over the PCG (Steps S_6–S_{20}) handle this mutual dependence between the inter- and intraprocedural sets. The following sections provide details on how the interprocedural and intraprocedural sets are computed.

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

$$Entry_f = \bigcup_{c=(g,f)} ForwardBind^c_g(Holds_g), \text{for all } g \text{ that call } f \quad (3)$$

$$PGen_f = CSGen_f \cup IGen_f \quad (4)$$

$Entry_f$ is computed by propagating information along all call sites that invoke f (Steps S_{10}–S_{14}). $PGen_f$ summarizes what is generated by f (Step S_9).

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

$$CSGen^c_f = BackwardBind^c_f(PGen_g), \text{where } f \text{ calls } g \text{ at } c, \text{ i.e., } c = (f,g) \quad (5)$$

$$CSGen_f = \bigcup_{c \text{ in } f} CSGen^c_f \quad (6)$$

$$Holds_f = Entry_f \cup CSGen_f \cup IGen_f \quad (7)$$

$$IGen_f = \bigcup_{s \in f} IGen^s_f(Holds_f), \text{where } s \text{ is a pointer assignment} \quad (8)$$

$CSGen_f^c$, the component of $CSGen_f$ for call site c, is computed by propagating interprocedural information backward to a call site c in f. This information is then summarized for all call sites in f giving $CSGen_f$. $IGen_f^s$ is the "gen" component of the flow-sensitive transfer function given in (2) of Section 3.2.2:

$$IGen_f^s(Holds_f) = \bigcup_{\substack{a \in ComputeAliases(p,i),\\ b \in ComputeAliases(q,j+1)}} \langle *a, b\rangle \tag{9}$$

for a statement "$p_i = q_j$," where the subscript specifies the number of dereferences.

The discussion in Section 3.2.2 concerning alias paths traversed in the two calls to *ComputeAliases* also applies to (9).

To satisfy the cyclic dependence between (7) and (8), the algorithm iterates over the set of pointer assignment statements in f. As mentioned in Section 1, a flow-insensitive analysis does not consider control flow information during the analysis, and must, for correctness, capture any possible path that can be traversed by the set of statements associated with f. Our algorithm satisfies this requirement by placing the statements in a "switch" statement structure that is enclosed in a loop, and by not allowing killing of any alias relations. We conjecture that an extension to our algorithm that refines (9) to consider the alias paths traversed during the two calls to *ComputeAliases* at the same statement would satisfy the definition of a *precise* flow-insensitive alias analysis described by Horwitz [1997].

Figure 8 illustrates this technique. The CFG labeled **(A)** represents the original program. The CFG labeled **(B)** illustrates the manner in which the effect of all possible paths is captured. (This graph is not actually built.) The alias graph labeled **(C)** specifies the aliases that result from the intraprocedural computation, assuming no aliases are in the *Entry* set.

Figure 9 provides further details of the intraprocedural phase of the flow-insensitive algorithm (Step S_8 of Figure 7). Steps S_1–S_6 compute $CSGen_f$, the set of aliases that are generated from call sites of f. Step S_7 initializes $IGen_f$ optimistically. Step S_8 sets $Holds_f$ to be the aliases that reach f interprocedurally via $Entry_f$ and $CSGen_f$

Each iteration of the S_{10}–S_{14} loop traverses each pointer assignment statement, s, adding aliases generated by s to the $Holds_f$ and $IGen_f$ sets. Since the aliases generated depend on $Holds_f$, this traversal is repeated until no new aliases are generated (the S_9–S_{15} loop). This method is equivalent to considering all possible paths.

Consider statements S3, S4, and S5 of Figure 8. 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 by the flow-insensitive analysis. These statements contribute a constant value to $IGen_f$; they do not depend on $Holds_f$. This class of statements is processed prior to the interprocedural propagation (S_6 of Figure 7) by recording the aliases they generate in $IGen_f$ and $Holds_f$. Steps S_7 and S_8 of Figure 9 are updated to use this precomputed information.

In general, this optimization is not possible if an explicit representation of alias information is used, because the aliases generated with this representation are al-

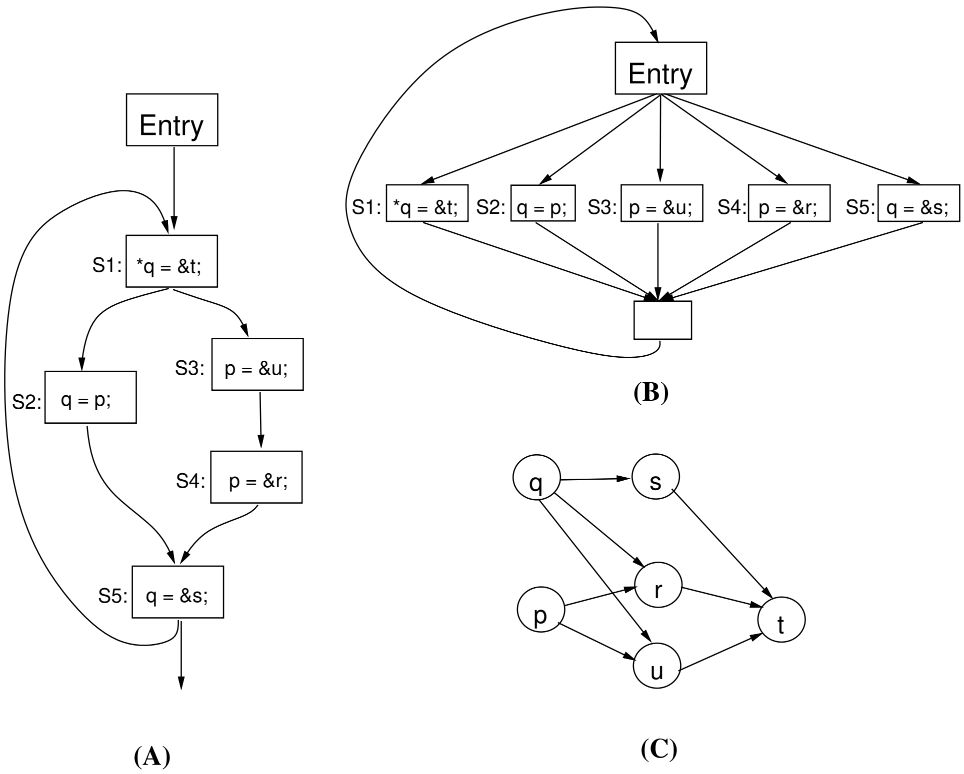

Fig. 8. Example illustrating the flow-insensitive intraprocedural phase.

S_1: $CSGen_f \leftarrow \{\}$
S_2: foreach call site, c, in f, loop
S_3:   foreach called procedure, g, at c, loop
S_4: $CSGen_f \leftarrow CSGen_f \cup BackwardBind_f^c(PGen_g)$
S_5:   end loop
S_6: end loop
S_7: $IGen_f \leftarrow \{\}$
S_8: $Holds_f \leftarrow Entry_f \cup CSGen_f$
S_9: repeat
S_{10}:   foreach pointer assignment statement, s, in f, loop
S_{11}:   compute the set, $A = IGen_f^s(Holds_f)$
S_{12}: $Holds_f \leftarrow Holds_f \cup A$
S_{13}: $IGen_f \leftarrow IGen_f \cup A$
S_{14}:   end loop
S_{15}: until aliasing converges

Fig. 9. Refinement of flow-insensitive intraprocedural phase (Step S_8 in Figure 7).

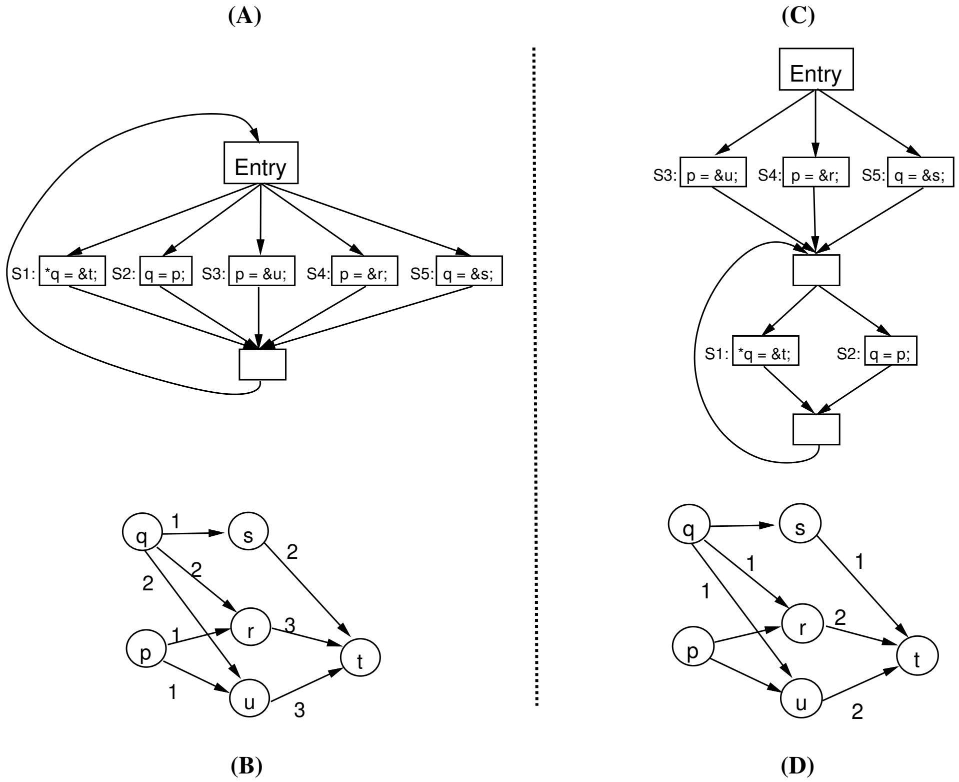

Fig. 10. Example illustrating the improved flow-insensitive intraprocedural phase.

ways dependent on what holds when they are generated. For example, if the alias relation $\langle *a, p\rangle$ holds, S3 (of Figure 8) would also generate $\langle ** \, a, u\rangle$ using an explicit representation.

Figure 10 illustrates how the program in Figure 8 would be processed using this enhancement. In addition to the implied CFG (graph **(A)**), we also include the alias graph (**B**), representing $IGen_f$ upon convergence, once again assuming $Entry_f = \{\}$. Graphs **(A)** and **(B)** correspond to the original algorithm; graphs **(C)** and **(D)** use the above enhancement. Each edge in the alias graphs (**B**) and **(D)** is labeled with the intraprocedural iteration number in which it was created, assuming statements are visited in the order S1, S2, ..., S5. 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 ($Holds_f$). For the aliases generated by statements like S2 a technique called *deferred evaluation* [Burke et al. 1997] can be used to move these statements prior to the loop. The technique, which does not lose precision, uses special alias graph edges to represent the effects of such statements, and relies on an enhanced version of the *ComputeAliases* algorithm of

```
    g() {
S1:     x = &y;
S2:     f();
S3:     x = &z;
    }
```

$Entry_g = \{\langle *x, a_1\rangle, \ldots, \langle *x, a_n\rangle\}$
$IGen_g = \{\langle *x, y\rangle, \langle *x, z\rangle\}$

Fig. 11. Example illustrating the usefulness of $ForwardKill$.

Section 2.1 to interpret these edges.

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 $IGen_f$ treats the pointer assignment statements of a procedure as one loop body, and the computation iterates over this loop body using the transfer function described in (9). This has the effect of defining a transfer function that includes all statements of a procedure in any possible order. Upon convergence, $Holds_f$ contains the result of this computation: the aliases assumed to hold at each point in f. Thus, when a fixed point is reached, the following equation is satisfied by $Holds_f$:

$$Holds_f = TF_s(Holds_f), \text{for all } s \text{ in } f$$

3.4 Using Kill Information to Improve Precision

Flow-sensitive alias analyses utilize intraprocedural control flow 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 11, where procedure g calls procedure f at S2, and $Entry_g$ is as given in the figure. By (3) in Section 3.3.1, the aliases that are propagated to $Entry_f$ from S2 are those that hold at S2, i.e., $Holds_g$, which is $\{\langle *x, a_1\rangle, \ldots, \langle *x, a_n\rangle, \langle *x, y\rangle, \langle *x, z\rangle\}$.

The precision of this solution can be improved 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, $ForwardKill_g^c$ as follows:

$$ForwardKill_g^c = \text{the set of aliases killed along all paths from } g\text{'s entry node to } c \tag{10}$$

As $ForwardKill_g^c$ 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 $ForwardKill_g^c$.

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. If this information is available during the $ForwardKill_g^c$

```
    f() {
S1:     x = &y;
S2:     g();
S3:     x = &z;
    }
```

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

Fig. 12. Example illustrating the usefulness of *BackwardKill*.

computation, it may be possible to determine if a strong update is performed at an assignment statement through a pointer dereference.

To determine which aliases should be propagated to the called procedure from c, $ForwardKill_g^c$ is used to kill aliases from $Entry_g$, before adding aliases that are generated by g ($IGen_g \cup CSGen_g$). In the example of Figure 11, $\{\langle *x, y\rangle, \langle *x, z\rangle\}$ is propagated to f, but not $\{\langle *x, a_1\rangle, \ldots, \langle *x, a_n\rangle\}$.

The updated equation for $Entry_f$ ((3) of Section 3.3.1) is as follows:

$$Entry_f = \bigcup_{c=(g,f)} ForwardBind_g^c((Entry_g - ForwardKill_g^c) \cup CSGen_g \cup IGen_g), \tag{11}$$

where g is any procedure that calls f

Since $ForwardKill_g^c$ is constant, its computation is performed before the interprocedural propagation (S_6 of Figure 7) and is 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 f at call site c via $CSGen_f^c$. To improve the precision of this set, we define for a call site c in f the following:

$$BackwardKill_g^c = \text{the set of aliases killed along all paths from } c \text{ to } f\text{'s } exit \text{ node} \tag{12}$$

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

The updated equation for $PGen_f$ ((4) of Section 3.3.2) divides $CSGen_f$ into its call site components to allow the incorporation of kill information.

$$PGen_f = \bigcup_{c \in f} (CSGen_f^c - BackwardKill_g^c) \cup IGen_f \tag{13}$$

Figure 12 presents an example in which using *BackwardKill* information improves the precision of $PGen_f$. Assume the contents of $CSGen_f^{S2}$ are as given in the figure. Without kill information we have

$$PGen_f = CSGen_f^{S2} \cup IGen_f = \{\langle *x, b_1\rangle, \ldots, \langle *x, b_n\rangle, \langle *x, y\rangle, \langle *x, z\rangle\}.$$

Using (13) for $PGen_f$ gives

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

Like $ForwardKill_g^c$, $BackwardKill_g^c$ can be computed as a preprocessing step before the flow-insensitive iteration over the PCG has begun. Thus, techniques described for the computation of $ForwardKill_g^c$ for handling statements of the form "$*p = \ldots$" can also be employed for $BackwardKill_g^c$. 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].

3.5 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 that map alias sets to alias sets. The first function, $ForwardBind_g^c$, captures the effects of propagating aliases forward over the PCG from a call site, c, in procedure g to the called procedure. The second function, $BackwardBind_g^c$, captures the effects of propagating aliases backward over the PCG from a called procedure to the call site c in procedure g. These functions are used in the interprocedural algorithms described in this section. Unless stated otherwise, the functions $ForwardBind_g^c$ and $BackwardBind_g^c$ include their input aliases in their result, i.e., $x \in ForwardBind_g^c(x)$ and $y \in BackwardBind_g^c(y)$.[11]

This section assumes that the compact representation is used to represent alias information, and structural alias information is available. Alias relations that result from structural aliases are used to initialize $Entry$ as described in Section 3. This section first describes methods for incorporating call-by-value and reference parameters into the alias computation. It then describes how to correctly handle local variables in the presence of recursion.

3.5.1 *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 in procedure g that invokes procedure h.

ForwardBind. 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 x):

$$AR = \langle *a_i, x\rangle,$$

where x also can be an actual parameter. In this case, f_i gets the value of a_i, and $ForwardBind_g^c(AR) = \langle *f_i, x\rangle$. This includes the case where the actual explicitly contains the address of an object, such as "&x."

[11] The efficiency of the analysis can be significantly increased by refining $ForwardBind_g^c$ to filter out irrelevant alias relations [Hind and Pioli 1998a; Pioli 1999].

BackwardBind. 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, $BackwardBind_g^c$ is defined so that 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. Handling local variables in the presence of recursive calls will be described in Section 3.5.3.

With copy-out parameters, alias relations of the form $\langle *f_i, x\rangle$ are converted by $BackwardBind_g^c$ into $\langle *a_i, x\rangle$, where x 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.

3.5.2 *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, $\langle *x, y\rangle$ can be expressed as $\langle ap_1(x), ap_2(y)\rangle$, where $ap_1(x) = *x$ and $ap_2(y) = y$.

For a set of aliases, S, $ForwardBind_g^c(S)$ is described by considering an alias relation $AR \in S$. There are three possibilities for AR:

(1) $\langle ap_i(a_i), ap_j(a_j)\rangle$,
(2) $\langle ap_k(a_i), ap_l(x)\rangle$,
(3) $\langle ap_m(y), ap_n(z)\rangle$,

where x, y, and z are not actual parameters of call site c. The third case does not involve parameters, so we need only consider the first two cases.

Applying $ForwardBind_g^c$ 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) $\{\langle ap_k(f_i), ap_l(x)\rangle\}$

each of which will be added in $Entry_p$.

In computing $ForwardBind_g^c$, aliases that contain nonlocal named objects that are 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]. 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. These aliases can be removed after the interprocedural alias analysis is performed. 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 $BackwardBind_g^c$, convert each reference to f_i in the alias relations of the called routine's alias set into that of a_i. The called routine's alias set will be $Exit_p$ for the flow-sensitive analysis and $PGen_p$ for the flow-insensitive analysis.

This step can generate multiple identical alias relations, of which only one need be kept.

3.5.3 *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 formal parameter can designate multiple objects in the run-time stack at a given time.[12] Thus, killing an alias $\langle *l, x\rangle$ for a local variable l at SA_j based on an assignment to l of stack activation SA_i can result in an incorrect data flow result.

A local variable l of stack activation SA_i can be accessed at stack activation SA_j only if there is an access path valid in SA_j that leads to l of SA_i. This can happen only if the "address-taken" operator ("&") is applied to l. We call such locals *escaping* locals.[13] Therefore, we distinguish two types of locals of a procedure involved in a recursive call: escaping locals and nonescaping locals.

Escaping locals of a recursive procedure are handled conservatively by regarding an assignment to such a local variable as a weak update. Such a statement only adds new aliases; it does not kill any aliases reaching the statement. At a recursive call site, aliases involving escaping locals are propagated across procedure call boundaries.

Nonescaping locals of a recursive procedure are handled more precisely as follows: (1) an assignment to a nonescaping local is regarded as a strong update;[14] and (2) the transfer function of a procedure call is the identity function with respect to the nonescaping locals of the calling procedure: the callee cannot affect aliases of the nonescaping locals of the caller. This treatment can also be applied to nonescaping locals of nonrecursive procedures.

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

4. EMPIRICAL RESULTS

This section provides results for the three pointer alias analysis algorithms discussed in Section 3: flow-sensitive (FS), flow-insensitive (FI), and flow-insensitive with kill (FIK).

4.1 Implementation

The three analyses have been implemented in the NPIC system [Hind and Pioli 1998b; Pioli 1999], an experimental program analysis system written in C++. A prototype version of the IBM VisualAge C++ compiler [Nackman 1997; Soroker et al. 1997] is used as the front end. The abstract syntax tree constructed by the front end is transformed into a PCG and a CFG for each function, which serve as input to the alias analyses. No CFG is built for library functions. We model a call to a library function based on its semantics, thereby providing the benefits of context-sensitive analysis of such calls. Library calls that cannot affect

[12] In the rest of this section we use the term *local variable* to refer to local variables or formal parameters.

[13] A more sophisticated analysis can be used to identify locals that are not escaping despite having the "&" operator applied to them.

[14] Assuming the local variable represents one run-time location, i.e., it is not an array.

the value of a pointer are treated as the identity transfer function. A conservative SEG representation [Hind and Pioli 1998a; Pioli 1999] is used for the flow-sensitive analysis. This representation includes a CFG node in the SEG if (1) it may affect a pointer value; (2) it has more than one predecessor that is a SEG node; or (3) it has an incoming back edge. This representation may include additional ϕ-nodes not included in the SEG defined by Choi et al. [1991].

The compact representation (Section 2.1) is used to represent alias relations. All analyses are context-insensitive. All three analyses are implemented using a worklist of functions, where each worklist is implemented as a priority queue based on a topological order of the functions. We have found the worklist implementation to be three times more efficient on average than the corresponding iterative implementation [Hind and Pioli 1998a]. The flow-sensitive analysis also uses a priority-based worklist of SEG nodes for each intraprocedural iteration (the inner loop of Figure 5). Priority is based on a topological order of the SEG nodes.

The implementation includes the optimization of moving pointer assignment with constant *Gen* sets out of the intraprocedural iteration of flow-insensitive analysis, as described in Section 3.3. Both flow-insensitive algorithms do not include the final flow-sensitive pass as described in Section 3. Instead, for each function the union of the function's *Entry* and *PGen* sets is used to determine what alias relations hold anywhere in the function.

Anonymous objects are named based on the statement where they are allocated. Array elements and field components are not distinguished. 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. Our results do not include extensions to our techniques described in Burke et al. [1997], such as deferred evaluation or the interprocedural naming of heap-allocated storage.

4.2 Benchmarks

Our benchmark suite contains 24 C programs, 21 provided by other researchers [Landi et al. 1993; Emami et al. 1994; Ruf 1995; Rutgers PROLANGS 1999] and 3 from the SPEC CINT92 [Balan and Bays 1992] and CINT95 [SPEC 1995] benchmarks.[15] 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`.[16] The fourth column reports the number of CFG nodes, which include nodes created by the initialization of globals. The fifth column reports the number of user-defined functions (nodes in the PCG). The next two columns give the number of call sites, distinguished between user and library function calls. The next column reports the percentage of CFG nodes that are considered as pointer assignment nodes. The current analysis treats an assignment as a pointer assignment if the variable involved in the pointer expression on the left side of the assignment is declared to be a pointer. The last two columns report the number of recursive functions (functions that are in PCG cycles) and heap-allocation sites in each program. The last

[15] Some programs had to be syntactically modified to satisfy C++'s stricter type-checking semantics.

[16] 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`.

Table I. Benchmark Suite and Static Characteristics

Name	Source	LOC	CFG Nodes	Fcts	Call Sites User	Call Sites Lib	Ptr-Asgn Nodes%	Rec Fcts	Alloc Sites
allroots	Landi	227	159	7	19	35	1.3%	2	1
052.alvinn	SPEC92	272	229	9	8	13	10.0%	0	0
01.qbsort	McCat	325	170	8	9	25	24.1%	1	5
06.matx	McCat	350	245	7	18	37	13.5%	0	9
15.trie	McCat	358	167	13	19	21	23.4%	3	5
04.bisect	McCat	463	175	9	11	18	9.7%	0	2
fixoutput	PROLANGS	477	299	6	12	85	4.4%	0	3
17.bintr	McCat	496	193	17	27	28	8.8%	5	1
anagram	Austin	650	346	16	22	38	9.5%	1	2
lex315	Landi	733	569	17	102	52	6.5%	0	3
ks	Austin	782	526	14	17	67	27.4%	0	5
05.eks	McCat	1,202	677	30	62	49	4.0%	0	3
08.main	McCat	1,206	793	41	68	53	20.9%	3	8
09.vor	McCat	1,406	857	52	174	28	28.6%	5	8
loader	Landi	1,539	691	30	79	102	8.8%	2	7
129.compress	SPEC95	1,934	17,012	25	35	28	0.2%	0	0
ft	Austin	2,156	775	38	63	55	18.6%	0	5
football	Landi	2,354	2,854	58	257	274	1.8%	1	0
compiler	Landi	2,360	1,767	40	349	107	5.1%	14	0
assembler	Landi	3,446	1,845	52	247	243	16.6%	0	16
yacr2	Austin	3,979	2,070	59	158	169	6.6%	5	26
simulator	Landi	4,639	2,929	111	447	226	6.3%	0	4
flex	PROLANGS	7,659	7,107	88	375	239	5.2%	4	10
099.go	SPEC95	29,637	31,788	373	2,054	22	1.5%	1	0
Average							11.0%		

row of the table reports the average pointer assignment node percentage, which is computed by averaging the corresponding value over the 24 benchmarks.

An artificial `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 similar to the one added by Ruf [1995] and Landi et al.[1993; 1998]. This function is included in the statistics reported in Table I. Global variable initializations (both implicit and explicit) are automatically modeled as assignment statements in the artificial `main` function, and thus are included in the count of CFG nodes. The large number of CFG nodes for the `129.compress` benchmark is due to many such array initializations.

4.3 Description of Experiment

This section presents precision and efficiency results. For each benchmark and each analysis, we report the analysis time, the high-water mark in memory usage, and the average number of objects a dereferenced pointer can point to. The precision results for the FIK (flow-insensitive with kill) analysis are exactly the same as the FI (flow-insensitive) analysis for all benchmarks. Thus, we do not explicitly include this analysis in our precision data.

The analysis time is reported in seconds on a 333MHz IBM RS/6000 PowerPC 604e with 512MB RAM and 1GB paging space, running AIX 4.1.5. The executable was built using IBM's compiler (xlC) with "-O3." We report the time spent in each alias analysis, which includes any analysis-specific preprocessing, such as building the SEG from the CFG in the FS analysis. The times do not include the time to build the initial PCG and CFGs because this time is constant for all analyses. This information is displayed in the left/bottom chart in Figure 13. The right/top chart of this figure reports the high-water mark in memory usage. This information was obtained by using the "`ps v`" command under `AIX 4.1.5`.

To collect precision information, the system traverses the representation by visiting each expression containing a pointer dereference and, using the computed alias information, reports how many named objects are aliased to the pointer expression. We report the average number of such dereferences for both reads and writes. This form of counting, also used by Emami et al. [1994], Ruf [1995], and Zhang et al. [1998], provides a precision metric based on the use of alias information, and therefore can be more meaningful than recording the number of aliases at all program points. Further precision information is provided in Hind and Pioli [1998b] and Pioli [1999].

A pointer expression containing multiple dereferences (such as "`***p`") is counted as multiple dereference expressions, one for each dereference. The intermediate dereferences (`*p and **p`) are counted as reads. The last dereference (`***p`) is counted as a read or write, depending on the context of the expression. Statements such as "`(*p)++`" and "`*p += increment`" are treated as both a read and a write of `*p`.

We consider a pointer to be dereferenced if the variable is declared as a pointer or an array formal parameter and if one or more of the "*", "->", or "[ ]" operators are used with that variable. Each formal parameter array is included because one or more of its corresponding actual parameters could be a pointer. We do not count the use of the "[ ]" operator on arrays that are not formal parameters because the resulting "pointer" (the array name) is constant, and therefore counting it may skew results.

The manner in which the heap is modeled must be considered in evaluating precision 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 the invisible objects (objects not lexically visible in the current procedure) aliased to a formal parameter (because their address is passed as the corresponding actual parameter) as a single object[17] may report fewer objects. Our analyses do not use this technique. However, to reduce the number of string objects, we model all strings as one object. This differs from the model in Burke et al. [1997] and Hind and Pioli [1998a], where each string literal is modeled as a separate object.

Assuming a correct input program, each pointer dereference should 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 1 demonstrates the analysis is precise (modulo heap and invisible object naming), a larger number could reflect an imprecise algorithm, a limitation of static analysis, or a pointer dereference that corresponds to different memory locations over the program's execution.

[17] This modeling technique can increase the possibility of strong updates.

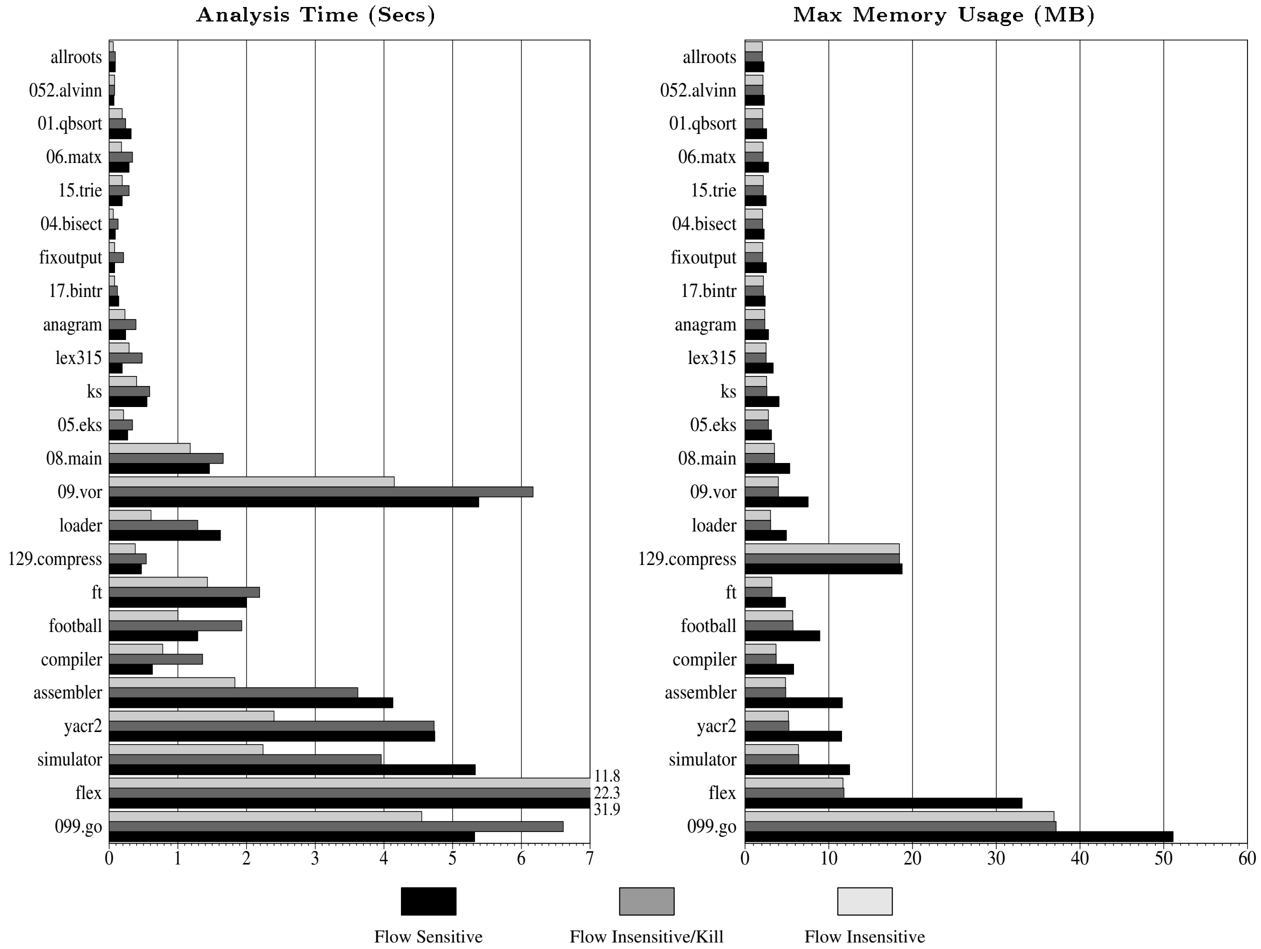

Fig. 13. Analysis time (in seconds) and memory usage (in MB).

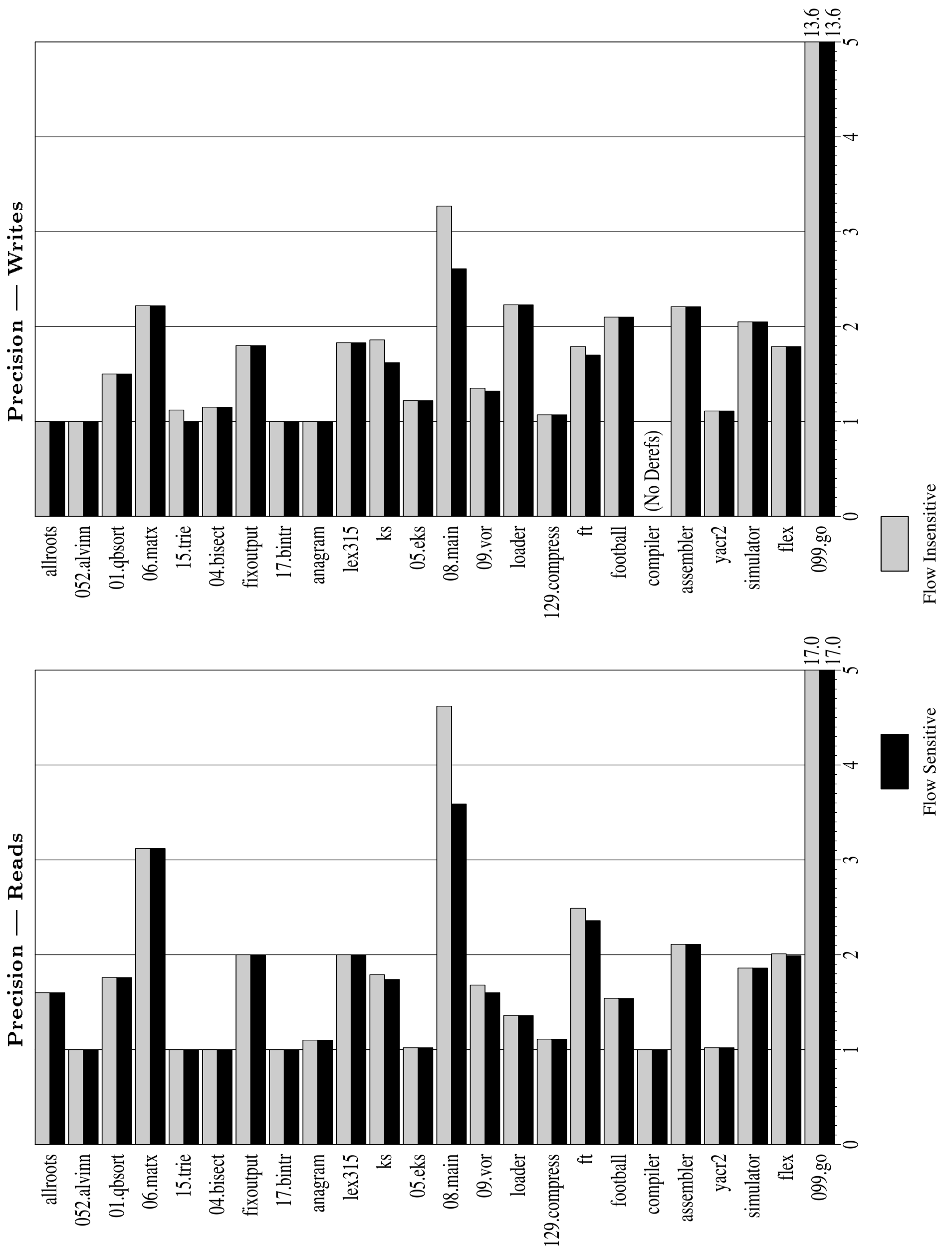

Fig. 14. Precision results (number of objects pointed to by a dereferenced pointer).

4.4 Discussion

The results indicate that the FIK analysis is not beneficial. On our benchmark suite it is never more precise than the FI analysis, and on some occasions requires more analysis time than the FS analysis. One explanation for this could be that an alias relation created to simulate a reference parameter, in which 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. We have seen that approximately 50% of pointers that are dereferenced are formal parameters.

One surprising result is the overall precision of the FI analysis. In 75% (18 of 24) of the benchmarks the FI analysis is as precise as the FS analysis.[18] This seems to suggest that the added precision obtained by the FS analysis in considering control flow within a function is not significant for those benchmarks, at least where pointers are dereferenced. We offer two possible explanations:

(1) Pointer variables are often not assigned more than one distinguished object within the same function. Thus, distinguishing program points within a function, a key difference between the FS and FI analyses, does not often result in an increase in precision. We have seen exceptions to this in the function `InitLists` of the `ks` benchmark and in the function `InsertPoint` in the `08.main` benchmark. In both cases the same pointer is reused in two list-traversing loops.

(2) It appears that a large number of alias relations are created at call sites because of actual/formal parameter bindings. These bindings are created independently of the FS and FI analyses, and thus do not provide an opportunity for a difference in precision.

The FI analysis can be close to three times faster than the FS analysis (`flex`), and was faster than the FS analysis in 92% (22 of 24) of the benchmarks.

One would expect the memory usage for the FS analysis to be orders of magnitude larger than the FI analysis because the FS analysis can use many more alias sets.[19] Although the initial implementation did have this property, several storage-saving schemes, such as using an SEG, sharing alias sets among nodes, and applying a filtering technique in *ForwardBind*, significantly reduced the storage requirements and analysis time of the FS analysis [Hind and Pioli 1998a; Pioli 1999]. It remains to be seen if these techniques will keep the storage requirements of FS comparable with FI when larger programs are analyzed.

The precision results for `099.go` merit discussion. An average of 17.03 and 13.64 objects are returned for reads and writes, respectively, with a maximum of 100. 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, passing the address of 100 different actuals for the list header, resulting in 100 aliases for the formal parameter.

[18] Prior results [Burke et al. 1997; Hind and Pioli 1998a] reported more programs with differences because strings were modeled as individual objects.
[19] The FS analysis *potentially* uses two alias sets for every CFG node in the program plus two alias sets for each function. The FI analysis uses two alias sets for each function.

However, because the lifetime of the formal is limited to this function (it does not call any other function), these relations are not propagated to any other function. Therefore, these relations do not suffer the effects of the *infeasible* or *unrealizable* path problem [Landi and Ryder 1992; Sharir and Pnueli 1981].

The results also confirm that the 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, `099.go`, despite being our largest program and having a pointer aliased with 100 objects at one point in the program, is analyzed by the flow-sensitive analysis at one of the fastest rates (5,571 LOC/second, 3,162 CFG nodes/second). This is because a significant number of aliases involve a formal parameter whose scope is one function (`addlist`), and thus these aliases are propagated only within this function.

A more precise and time-consuming alias analysis may not be as inefficient as it appears 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 [Pioli 1999; Shapiro and Horwitz 1997a]. The previous paragraph suggests that this can also be true about pointer alias analysis itself, which also utilizes pointer alias information during its analysis.

Section 6.5 discusses how our results compare with empirical results of other researchers.

5. COMPLEXITY ANALYSIS

This section describes the complexity analysis of our methods assuming the compact representation (Section 2.1) 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 bound 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;

—$A_{max} \leq V * A_{var}$: the maximum number of aliases holding in any alias set;

—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;

—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 (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.[20]

This section first describes the complexity of techniques that are common to the 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 2.1) 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 $O(A^L_{var})$ in the remainder of this section. Note that L is typically a small constant.

Function Pointers. The number of iterations performed for the analysis of function pointers as described in Section 3.1 is bounded by PCE_{fp}. Each iteration entails an interprocedural and intraprocedural pointer alias analysis. Below we analyze the complexity of each iteration for the three algorithms described in this article. The full complexity for each algorithm is the product of each algorithm's complexity and PCE_{fp}.

5.1 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 (as well as intermediate In and Out information for each SEG node);
—the final intraprocedural phase, during which final alias information is computed for each statement in each procedure.

The 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]. Since each of P procedures may have up to A_{max} 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 $O(A_{max} * P)$. The space complexity for the final phase dominates and is $O(P * SEG_p * A_{max})$.

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

[20] Dhamdhere and Khedker [1993] show that the bound on the number of iterations for obtaining the fixed-point solution of a unidirectional *bit vector* data flow problem is $w + 1$, where w is the width of the control flow graph. This provides a tighter bound than the classical bound of $d + 1$, where d is the *depth* of the flow graph [Aho et al. 1986], i.e., the maximum number of back edges along any acyclic path. Width identifies only those back edges that require additional iterations.

The second phase dominates the time complexity. The worst case requires $O(A_{max} * P)$ 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_{max}. The union operation performed at $MeetNodes$ is also bounded by A_{max}. Thus, this process is $O(A_{max} * (PCE + P + M))$. 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 $O(A^L_{var})$. The resulting sets from these calls are used to create new aliases using the transfer function of Section 2.1. The worst-case size of each set is A_{ap}, making the cost of each $GenNode$ be $O(2 * A^L_{var} + A^2_{ap}) = O(A^L_{var} + A^2_{ap})$, and the cost of processing all $GenNodes$, $O(S * (A^L_{var} + A^2_{ap}))$. Since these nodes ($MeetNodes$, $GenNodes$, call sites, N_{Entry}, and N_{Exit}) will be considered at most $A_{max} * SEG_p$ times, the time complexity to compute new $Entry$ and $Exit$ sets for the procedures for each iteration over the PCG is $O((A_{max} * SEG_p) * (A_{max} * (PCE + P + M) + S * (A^L_{var} + A^2_{ap})))$ and $O((A_{max} * P) * (A_{max} * SEG_p) * (A_{max} * (PCE + P + M) + S * (A^L_{var} + A^2_{ap})))$ overall. This can be rewritten as $O(SEG_p * A^3_{max} * P^2 + SEG_p * A^3_{max} * P * PCE + SEG_p * A^3_{max} * P * M + SEG_p * A^2_{max} * P * S * (A^L_{var} + A^2_{ap}))$. A simpler but looser bound results from replacing A_{ap} and A_{var} terms by V, and A_{max} by V^2: $O(SEG_p * P * V^6 * (P + PCE + M + S * V^{L-2}))$.

Using the logic from above, the time complexity of the final intraprocedural phase is $O((A_{max} * SEG_p) * (A_{max} * (PCE + M) + S * (A^L_{var} + A^2_{ap})))$. 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.

Worst-case complexity is often not an accurate indicator of pointer analysis performance. An integral part of our flow-sensitive algorithm is the number of times a function is visited during the interprocedural phase. The average number of such visits varies from 3.1 (`129.compress`) to 14.7 (`09.vor`) for our benchmark suite. The average over all benchmarks is 7.3. Another useful metric is how often an SEG node is visited, i.e., its transfer function is evaluated. The average number of visits ranged from 2.1 (`052.alvinn`) to 12.1 (`09.vor`), with a benchmark average of 5.9. These values do not correlate to program size for our benchmarks, but do correlate to the analysis rate of each program (SEG nodes/sec).

Goyal [1999] describes an incremental adaption of the transfer function (Section 3.2.2), which reduces the worst-case time complexity at the cost of increased storage.

5.2 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 complexity of the first phase is $O(P * S_p)$. The space complexity of the analysis during the second phase is $O(A_{max} * P) = O(V * A_{var} * P)$. The space complexity during the final phase is the same as with the flow-sensitive algorithm and is analyzed above: $O(P * SEG_p * A_{max})$.

The time complexity of the first phase is $O(P * S_p)$. Once again the interprocedural phase dominates the time complexity. As above, there can be $O(A_{max} * P)$ iterations over the PCG. Each iteration requires P flow-insensitive intraprocedural analyses. During each intraprocedural analysis, as many as $O(S_p * A_{max})$ 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 $O((S_p * A_{max}) * S_p * (A_{var}^L + A_{ap}^2))$, and for all intraprocedural phases $O(P * (S_p * A_{max}) * S_p * (A_{var}^L + A_{ap}^2))$. Since there are at most $O(A_{max} * P)$ such phases, the time complexity for the second phase is $O((A_{max} * P) * P * (S_p * A_{max}) * S_p * (A_{var}^L + A_{ap}^2)) = O(A_{max}^2 * P^2 * S_p^2 * (A_{var}^L + A_{ap}^2))$.

The time complexity during the final phase is the same as with the flow-sensitive algorithm: $O((A_{max} * SEG_p) * (A_{max} * (PCE + M) + S * (A_{var}^L + A_{ap}^2)))$.

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

5.3 Flow-Insensitive Interprocedural Algorithm with Kill

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 $O(P * w * CFE_p)$.

Kill information is applied twice at each call site during a visit to a procedure. Each procedure is visited $O(A_{max})$ times, resulting in $O(PCE * A_{max})$ uses of the precomputed kill information. Each use is $O(A_{max})$. Thus, the flow-insensitive algorithm with kill requires an additional complexity of $O(P * w * CFE_p + C * P * A_{max} * A_{var})$.

6. RELATED WORK

Pointer alias analysis has been shown 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.

6.1 The Interprocedural Framework

The major characteristic of our interprocedural framework is the interleaving of the intraprocedural and interprocedural phases. During an 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, which reduces the storage requirements of the interprocedural iterations. The only intraprocedural sets required are those of the currently analyzed procedure.

Sharir and Pnueli [1981] also employ interleaving in the first phase of their general functional framework for interprocedural analysis. Their first phase computes, for each node of each procedure, a summary function for all paths from the entry node of that procedure to that node. Computing summary functions at call sites utilizes the summary function for paths from the entry to the exit of the called procedure. In the second phase, the solution at the entry node of each procedure is computed using the summary functions at call sites computed in the first phase. The solution at each point in the procedure is then computed using the summary functions from the first phase. Our framework does not utilize summary functions. Representing and computing such functions is not straightforward in pointer alias analysis, where transfer functions are dependent upon inputs.

Chatterjee et al. [1999] do compute summary functions for pointer alias analysis by specifying the relevant calling context conditions necessary for an alias relation to hold. The only intraprocedural sets needed in memory are those of the currently analyzed strongly connected component of the PCG.

Landi and Ryder [1992] use a single worklist to interprocedurally propagate may-hold relations, which contain the program point to which the relations apply. The analyses of Emami et al. [1994] and Wilson and Lam [1995] follow the control flow of the program through the call site to analyze the called procedure. This differs from our approach, which postpones analyzing called procedures until the analysis of the current procedure converges.

6.2 Flow-Sensitive Analysis

Flow-sensitive pointer analyses propagate alias information over a control flow graph representation by applying a transfer function at each node in the graph. Our analysis employs the *sparse evaluation graph* [Choi et al. 1991] to improve space and time efficiency.[21]

Our flow-sensitive analysis uses the compact representation [Choi et al. 1993] to represent alias information. This representation is similar to the points-to representation [Emami et al. 1994; Ghiya 1992], which has been used by a number of researchers [Andersen 1994; Hasti and Horwitz 1998; Ruf 1995; Shapiro and Horwitz 1997b; Steensgaard 1996; Wilson and Lam 1995; Zhang et al. 1998]. Landi and Ryder [1992] use a representation that explicitly describes all alias relations at the cost of increased storage. The Appendix illustrates how the compact and explicit representations have incomparable precision. Similar discussions can be found in Marlowe et al. [1993], Landi et al. [1998], and Pioli [1999].

To kill alias relations at assignment statements whose left side contains pointer dereferences, "must" information about the left-side expression is required. Our treatment optimistically kills alias relations if the current alias information resolves the pointer expression to one nonsummary object. This technique was described by Aho et al. [1986] for single-level nonheap pointers and by Chase et al. [1990] for heap-directed pointers and is used by Wilson and Lam [1995] and Ruf [1995]. Emami et al. [1994] distinguish this information explicitly by augmenting each points-to relation with a predicate that describes if the relation is "possible" (may) or "definite" (must). Landi and Ryder [1992] only kill alias information when the left side does not contain a pointer dereference.

[21] Hind and Pioli [1998a] report an average reduction of 74% in the number of alias sets, resulting in a 2.8 times reduction in analysis time.

6.3 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. It is similar to the FI algorithm described in Section 3.3 in that iteration without killing is used for correctness. It differs in that it computes one solution for the whole program rather than for each function.

Steensgaard [1996] presents a flow-insensitive points-to algorithm that conservatively groups all objects pointed to by a variable into one object. 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.

Zhang et al. [1996] present an algorithm, which shares a property of Steensgaard's analysis [Steensgaard 1996] in that it groups all objects pointed to by a variable into an equivalence class. This algorithm and Steensgaard's algorithm have been shown to be fast in practice [Shapiro and Horwitz 1997b; Steensgaard 1996; Zhang et al. 1998].

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.

Hasti and Horwitz [1998] present a pessimistic algorithm that attempts to increase the precision of a flow-insensitive analysis by iterating over the flow-insensitive analysis and an SSA [Cytron et al. 1991] construction. No empirical results are reported.

Horwitz [1997] provides a definition of precise flow-insensitive may-alias analysis and proves that with arbitrary levels of pointers and an arbitrary number of pointer dereferences, computing such a solution is NP-hard even in the absence of dynamic memory allocation. We conjecture that our flow-insensitive algorithm, extended to consider two calls to *ComputeAliases* at the same statement, satisfies the definition of precise as described by Horwitz [1997]. If this is true, assuming the number of pointer dereferences at each statement is bounded, our algorithm is an example of a polynomial-time algorithm to compute a precise flow-insensitive solution in the absence of dynamic memory allocation. Horwitz [1997] provides as an open question whether such an algorithm exists.

Our flow-insensitive algorithm does not group all objects pointed to by a variable into one object, 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, which could also reduce the number of iterations. We have described how limited kill information can be used to potentially improve precision.

6.4 PCG Construction

The method described in this article 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 [Burke 1987; Callahan et al. 1990; Ryder 1979] 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 [1980] method, which performs a transitive closure of the alias relations.

Ghiya [1992] and Emami et al. [1994] independently proposed an algorithm similar to ours for constructing the PCG in a flow-sensitive algorithm for the same programming model we consider. Like our algorithm, their algorithm optimistically grows the PCG as new function pointer aliases are discovered. 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 [Deutsch 1990; Harrison 1989; Mogensen 1989; Neirynck et al. 1989; Sestoft 1989; Shivers 1988].

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.

6.5 Empirical Comparisons

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 corresponds to the "write" metric reported in Figure 14. However, since their results included compiler-introduced temporaries in their "thru-deref" count [Landi 1997], a direct comparison is not possible.

Stocks et al. [1998] and Landi et al. [1998] 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 and (2) the underlying representation is not identical, and thus pointer dereferences may not be counted in the same manner in all cases. For example, 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 either identical or close to that reported in Stocks et al. [1998].

The relative precision of the flow-insensitive analysis compared to the flow-sensitive analysis is in contrast to the study of Stocks et al. [1998], which compares the flow-sensitive analysis mentioned above with a flow-insensitive analysis described in Zhang et al. [1996]. For the eight common benchmarks, our flow-insensitive algorithm ranges from 1.0 to 2.8 objects on average for a write dereference, compared to 1.0 to approximately 6.3 for the less precise flow-insensitive analysis they studied.

Emami et al. [1994] report precision results for a context- and flow-sensitive algorithm. Their results range from 1.0 to 1.77 objects for all indirect accesses for their benchmark suite, using a heap-naming scheme that represents all heap objects with one name. Because we were unable to obtain the benchmarks from the suite they used in this study, 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 1997b]. Since such an array will always point to the same place, the average number of objects is improved.[22] For the 11 benchmarks in common,[23] Ruf reports an overall read and write average of 1.33 and 1.38, respectively. To facilitate comparisons, we have also counted in this manner. The results for the common benchmarks are averages of 1.35 and 1.47 for the FS analysis and 1.41 and 1.54 for the FI analysis. We attribute the slight differences in the FS analysis to the difference in intermediate representations. As Ruf [1995] states, "the VDG intermediate representation often coalesces series of structure or array operations into a single memory write." This coalescing can skew results in either direction.

Shapiro and Horwitz [1997a] present an empirical comparison of four flow-insensitive algorithms. The first algorithm simply records all variables whose address has been taken. The remaining three algorithms [Andersen 1994; Shapiro and Horwitz 1997b; Steensgaard 1996] can be less precise and more efficient than the algorithms studied in this article. The authors measure the precision of these analyses by implementing three data flow analyses (GMOD, live variables, and truly live variables) and an interprocedural slicing algorithm. In addition to these alias analysis clients, the authors also report the direct precision of the alias analysis algorithms in terms of the total number of points-to relations. We agree with Emami et al. [1994] and Ruf [1995] that a more meaningful metric is to measure where the points-to information is used, such as where a pointer is dereferenced. They conclude that

[22] The best illustration of this is in 099.go, which has a large number of array references, but a low number of pointer dereferences. In this program, the average changed from 17.03 to 1.13 for reads and from 13.64 to 1.48 for writes when all uses of the [ ] operator were counted.

[23] The common benchmarks are allroots, 052.alvinn, anagram, lex315, ks, loader, 129.compress, 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.

(1) a more precise flow-insensitive analysis in general leads to increased precision by the subsequent analyses that use this information with varying magnitudes; (2) metrics measuring the alias analysis precision tend to be good predictors on the precision of subsequent analyses that use alias information; and (3) more precise flow-insensitive analysis can also improve the efficiency of subsequent analyses that use this information.

Diwan et al. [1998] provide static and dynamic measurements of the effectiveness of three flow-insensitive analyses for a type-safe language (Modula-3). All three algorithms are less precise than the algorithms presented in this article.

Ruf [1997a] describes a program-partitioning technique that is used for a flow-sensitive points-to analysis, achieving a storage savings of 1.3–7.2 over existing methods.

Zhang et al.[1996; 1998] investigate the effectiveness of a program decomposition technique for pointer aliasing of well-typed C programs. After the program is decomposed, they investigate various combinations of three pointer alias analyses, one flow- and context-sensitive and two different flow- and context-insensitive analyses.

Hind and Pioli [1998a] provide further details of the implementation described in this article. In addition to contrasting the precision and efficiency of these analyses and an "address-taken" analysis, they provide analysis-time speed-up results for various implementation techniques for the flow-sensitive analysis. Hind and Pioli [1999] expand on this work by also comparing the precision and efficiency of implementations of Steensgaard's [1996] and Andersen's [1994] analyses.

Pioli [1999] illustrates how Wegman and Zadeck [1991]'s conditional constant propagation algorithm can be combined with various pointer analysis algorithms and provides an algorithm for a synthesized version of the flow-sensitive algorithm presented in this work and conditional constant propagation. Empirical results are presented for all combinations. Pioli [1999] also expands on the work of Hind and Pioli [1999] by providing efficiency and precision results of client analyses of alias information such as mod-ref, live variables, dead assignments, conditional constant propagation, and unreachable code.

6.6 Other Related Work

6.6.1 *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 *unrealizable-execution-path* problem, but incur potentially exponential space and time overhead.

Other algorithms 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 generates a graph representing all invocation paths (in the absence of recursion). 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.

Wilson and Lam [1995] and Wilson [1997] optimize the computation of full context-sensitive information by constructing *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 and reuses this summary at other call sites that invoke the procedure with the same inputs, eliminating the cost of reanalyzing the procedure at such call sites.

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 concludes 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.

Burke et al. [1997] merge the calling context of each procedure and distinguish alias relations 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* problems 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.

Chatterjee et al. [1999] describe a technique for incorporating relevant context information into a data flow analysis and illustrate their approach for points-to analysis. Empirical results on C++ programs are provided. Chatterjee et al. [1998] describes how the technique can handle exceptions and a special case of incomplete programs, such as libraries.

6.6.2 *Heap Analysis.* Jones and Muchnick [1981], Larus and Hilfinger [1988], Ruggieri and Murtagh [1988], Horwitz et al. [1989], and 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], only one name is used to represent all dynamically allocated objects. This name can be refined by applying a more sophisticated heap analysis [Ghiya and Hendren 1996a; 1996b].

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 [1990] and 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.

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.

Ghiya and Hendren [1996a] provide a context-sensitive *connection analysis* that can determine if two stack pointers can reach a common heap element. Ghiya and Hendren [1996b] 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; 1998] 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.

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 Jones and Muchnick [1981], Hudak [1986], Ruggieri and Murtagh [1988], Horwitz et al. [1989], Chase et al. [1990], and Wilson and Lam [1995]. The *named-instance approach* [Burke et al. 1997] is an extension that qualifies named anonymous objects with a name string that captures backward call path information, distinguishing different instances of dynamically allocated objects.

7. 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 such as C and C++, which encourage

```
S1:    x = &y;
S2:    q = &x;
S3:    p = q;
S4:    x = &z;
```

Fig. 15. Compact representation improves precision.

the frequent use of pointers and procedure calls.

We have provided a framework and practical approximation methods for computing and representing interprocedural aliases for a program written in a language that includes pointers, reference parameters, and recursion, such as C, C++, Fortran 90, Java, and LISP. These methods include a flow-sensitive interprocedural alias analysis, a flow-insensitive interprocedural alias analysis, and a flow-insensitive interprocedural alias analysis that incorporates kill information to improve precision. These methods employ a technique 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. The precision of the flow-insensitive analysis is the same as the flow-sensitive analysis in 18 of 24 benchmarks. The flow-insensitive analysis with kill did not improve precision over the flow-insensitive analysis on any of the benchmarks. After accounting for differences in program representation and metrics, we have found that the precision of most flow-sensitive analysis algorithms are roughly equivalent. Although the flow-sensitive analysis efficiently analyzed a program on the order of 30,000 LOCs, further benchmarks are needed to see if this property generalizes.

A. PRECISION OF THE COMPACT REPRESENTATION

The compact and points-to representations require less storage than the explicit representation. For flow-sensitive analyses, these nonexplicit representations can be more precise in the presence of alias kills, but they can also be less precise than the explicit 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 using two intraprocedural examples; the trade-offs also occur for interprocedural analysis.

A.1 Compact Representation Yields More Precision

Consider the program segment of Figure 15. Assuming no aliases hold before S1, $\langle *x, y\rangle$ is the only alias holding after S1. After S3, the explicit representation of alias relations holding are as follows:

$$\{\langle *x, y\rangle, \langle *p, x\rangle, \langle *q, x\rangle, \langle *p, *q\rangle, \langle **p, y\rangle, \langle **q, y\rangle, \langle **p, *x\rangle, \langle **q, *x\rangle, \langle **p, **q\rangle\}, \tag{14}$$

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

$$\{\langle *p, x\rangle, \langle *q, x\rangle, \langle *x, y\rangle\} \tag{15}$$

After S4, the compact representation becomes

$$\{\langle *p, x\rangle, \langle *q, x\rangle, \langle *x, z\rangle\}, \tag{16}$$

from which the explicit information can be computed:

$$\{\langle *x, z\rangle, \langle *p, x\rangle, \langle *q, x\rangle, \langle *p, *q\rangle, \langle **p, z\rangle, \langle **q, z\rangle, \langle **p, *x\rangle, \langle **q, *x\rangle, \langle **p, **q\rangle\} \tag{17}$$

The difference between (14) and (17) is that every access path y in (14) has been replaced by z in (17).

However, the explicit alias information computed by applying S4 directly to (14) is as follows:

$$\{\langle *x, z\rangle, \langle *p, x\rangle, \langle *q, x\rangle, \langle *p, *q\rangle, \langle **p, z\rangle, \langle **q, z\rangle, \langle **p, *x\rangle, \langle **q, *x\rangle, \langle **p, **q\rangle, \tag{18}$$

$$\langle **p, y\rangle, \langle **q, y\rangle\}, \tag{19}$$

which contains additional alias relations (19) that are carried over from (14). $\langle **p, *x\rangle$ and $\langle **q, *x\rangle$ in (19) are implicit aliases derived from $\langle *p, x\rangle$ and $\langle *q, x\rangle$, respectively. These alias relations originate from $\langle *x, y\rangle$ at S1 and $\{\langle *p, x\rangle, \langle *q, x\rangle\}$ at S3. They are carried over after S4, although S4 kills $\langle *x, y\rangle$, resulting in less precision. In general, the compact and points-to representations improve precision when one of the base relations that enabled derivation of an alias relation is killed, while the derived alias is not.

The same improved precision can also be obtained with the explicit 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 S4 and can apply the killing rule to them as well as x.

A.2 Explicit Representation Yields More Precision

Before illustrating how the explicit representation can improve precision, we define two binary operators that will be used in this section. We use $\otimes$ to denote the explicit combination of aliases to generate a new alias. For example, a statement such as "$*a = b$" will generate the alias relation $\langle *x, y\rangle$ if $\langle *a, x\rangle$ and $\langle *b, y\rangle$ hold. We represent this as $\langle *x, y\rangle \Leftarrow \langle *a, x\rangle \otimes \langle *b, y\rangle$.

We distinguish a combination based on the compact representation from an explicit combination based on the program by using the operator, $\hat{\otimes}$, rather than $\otimes$. For example, the alias relation $\langle **p, v\rangle$ is inferred from the alias relations $\langle *p, q\rangle$ and $\langle *q, v\rangle$. We write this as

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

Consider the program segment in Figure 16 and the alias graph showing what a flow-sensitive analysis using the compact representation would compute as holding after S8. 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 S8 to identify z and y as the aliases of $**p$ holding at S8. From these implicit combinations, each coupled by an explicit combination with $\langle *x, w\rangle$ at S8, 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.

```
S1:   x = &w;
S2:   q = &z;
S3:   if (...)
S4:       p = &q;
S5:   else
S6:       q = &y;
S7:   ...
S8:   **p = x;
```

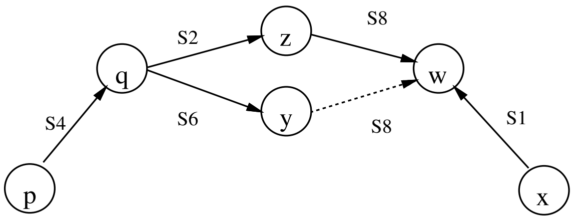

Fig. 16. Explicit representation improves precision.

```
B1:   if (...) {
B2:       u = &v;
          p = &q;
      }
      else {
B3:       u = &w;
          p = &t;
B4:   }
B5:   *p = u;
```

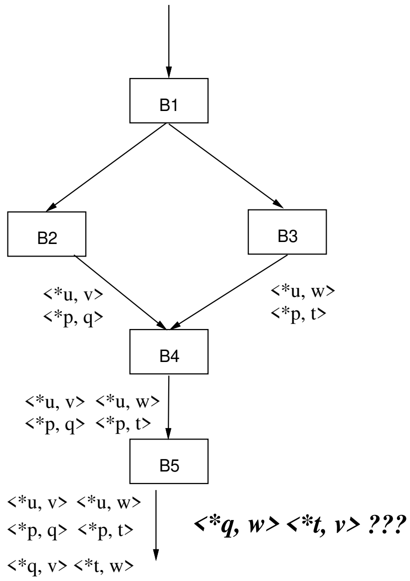

Fig. 17. Example Code Segment and its CFG.

With an explicit alias representation, alias relation combinations are performed explicitly on each control flow path, before the alias relations holding on each path are merged. Therefore, with an explicit representation, the set of aliases computed to hold at S8 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 S8. With the compact and points-to representations, 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.

A loss of control flow information can occur independently of the compact or points-to representations. As shown in Figure 17, a loss of precision can result from the explicit combination of aliases by the assignment at B5. For example, the invalid alias $\langle *t, v\rangle$ (or $\langle *q, w\rangle$) is created by combining alias relations $\langle *u, v\rangle$ and $\langle *p, t\rangle$ (or $\langle *u, w\rangle$ and $\langle *p, q\rangle$). This occurs at join points of control flow unless full path information is kept.

Approximation techniques to address this problem, as well as improve the precision of compact and points-to representations, are given in Burke et al. [1997].

ACKNOWLEDGMENTS

We thank Michael Karasick, Vivek Sarkar, Lee Nackman, Fran Allen, and Mary Lou Soffa for their support of this work. Anthony Pioli played a significant role in the design, implementation, and testing of the algorithms described in this article. We thank Todd Austin, Bill Landi, and Rakesh Ghiya for making their benchmarks available. Bill Landi, Laurie Hendren, Erik Ruf, Barbara Ryder, and Bob Wilson provided useful details concerning their implementations. Tom Marlowe suggested formulating the flow-insensitive intraprocedural phase as a switch statement. Discussions with Manuel Fähndrich led to reporting intermediate read dereferences, which were not considered in Burke et al. [1997]. David Bacon, John Field, Deepak Goyal, Yong-fong Lee, Anthony Pioli, G. Ramalingam, Barbara Simons, Harini Srinivasan, Laureen Treacy, and the anonymous referees provided many useful comments on earlier drafts.

We are grateful to NPIC group members at SUNY New Paltz and IBM researchers who have 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.

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 Evaluation 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 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., CARINI, P., CHOI, J.-D., AND HIND, M. 1997. Interprocedural pointer alias analysis. Research Report RC 21055, IBM T. J. Watson Research Center. Dec.

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., CARLE, A., HALL, M. W., AND KENNEDY, K. 1990. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering 16,* 4 (Apr.), 483–487.

CARINI, P., HIND, M., AND SRINIVASAN, H. 1995. Flow-sensitive interprocedural type analysis for C++. Research Report RC 20267, IBM T. J. Watson Research Center. Nov.

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).*

CHATTERJEE, R., RYDER, B. G., AND LANDI, W. A. 1998. Relevant context inference. Tech. Rep. DCS-TR-360, Department of Computer Science, Rutgers University. Aug.

CHATTERJEE, R., RYDER, B. G., AND LANDI, W. A. 1999. Relevant context inference. In *26th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages.*

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.

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, 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., FERRANTE, J., ROSEN, B. K., WEGMAN, M. N., AND ZADECK, F. K. 1991. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems 13,* 4 (Oct.), 451–490.

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. 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., MCKINLEY, K. S., AND MOSS, J. E. B. 1998. Type-based alias analysis. In *SIGPLAN '98 Conference on Programming Language Design and Implementation.* 106–117. *SIGPLAN Notices, 33(5).*

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).*

GHIYA, R. 1992. Interprocedural aliasing in the presence of function pointers. ACAPS Technical Memo 62, McGill University, Dec.

GHIYA, R. AND HENDREN, L. J. 1996a. Connection analysis: A practical interprocedural heap analysis for C. *International Journal of Parallel Programming 24,* 6, 547–578.

GHIYA, R. AND HENDREN, L. J. 1996b. Is it a tree, a dag or a cyclic graph? A shape analysis for heap-directed pointers in C. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages.* 1–15.

GOYAL, D. 1999. An improved intra-procedural may-alias analysis algorithm. Tech. Rep. TR1999-777, New York University. Feb.

GRAHAM, S. L. AND WEGMAN, M. 1976. A fast and usually linear algorithm for global flow analysis. *Journal of the ACM 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, W. L., III. 1989. The interprocedural analysis and automatic parallelisation of Scheme programs. *Lisp and Symbolic Computation 2,* 3 (Oct.), 176–396.

HASTI, R. AND HORWITZ, S. 1998. Using static single assignment form to improve flow-insensitive pointer analysis. In *SIGPLAN '98 Conference on Programming Language Design and Implementation.* 97–105. *SIGPLAN Notices, 33(5).*

HENDREN, L. 1990. Parallelizing programs with recursive data structures. Ph.D. thesis, Cornell University. Technial Report TR 90-1114.

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. 1998a. Assessing the effects of flow-sensitivity on pointer alias analyses. In *Lecture Notes in Computer Science, 1503*, G. Levi, Ed. Springer-Verlag, 57–81. Proceedings from the *5th International Static Analysis Symposium.*

HIND, M. AND PIOLI, A. 1998b. Assessing the effects of flow-sensitivity on pointer alias analyses (extended version). Research Report 21251, IBM T. J. Watson Research Center. June. Also available as SUNY at New Paltz Technical Report #98-104.

HIND, M. AND PIOLI, A. 1999. Evaluating the effectiveness of pointer alias analyses. Tech. Rep. RC 21510, IBM T. J. Watson Research Center. Mar.

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.

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

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. In *SIGPLAN '93 Conference on Programming Language Design and Implementation.* 56–67. *SIGPLAN Notices* 28(6).

LANDI, W. A., RYDER, B. G., STOCKS, P. A., ZHANG, S., AND ALTUCHER, R. 1998. A schema for interprocedural modification side-effect analysis with pointer aliasing. Tech. Rep. DCS-TR-336, Department of Computer Science, Rutgers University. May.

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. *SIGPLAN Notices, 23(7).*

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, IBM T. 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.

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.

PIOLI, A. 1999. Conditional pointer aliasing and constant propagation. M.S. thesis, SUNY at New Paltz. Available at http://www.mcs.newpaltz.edu/tr as Technical Report # 99-102.

RAMALINGAM, G. 1994. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems 16,* 5 (Sept.), 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. 1997a. Partitioning dataflow analyses using types. In *24th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*. 15–26.

RUF, E. 1997b. 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.

RUTGERS PROLANGS. 1999. http://www.prolangs.rutgers.edu/public.html.

RYDER, B. 1979. Constructing the call graph of a program. *IEEE Transactions on Software Engineering 5,* 3 (May), 216–226.

SAGIV, M., REPS, T., AND WILHELM, R. 1996. Solving shape-analysis problems in languages with destructive updating. In *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*. 16–31.

SAGIV, M., REPS, T., AND WILHELM, R. 1998. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems 20,* 1 (Jan.), 1–50.

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*. 1–14.

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.

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*. 119–128.

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 *23rd Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*. 32–41.

STOCKS, P. A., RYDER, B. G., LANDI, W. A., AND ZHANG, S. 1998. Comparing flow and context sensitivity on the modifications-side-effects problem. In *International Symposium on Software Testing and Analysis*. 21–31.

TARJAN, R. 1981. Fast algorithms for solving path problems. *Journal of the ACM 28,* 3, 594–614.

WEGMAN, M. N. AND ZADECK, F. K. 1991. Constant propagation with conditional branches. *ACM Transactions on Programming Languages and Systems 13,* 2, 181–210.

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. 1997. Efficient context-sensitive pointer analysis for C programs. Ph.D. thesis, Stanford University.

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 *4th Symposium on the Foundations of Software Engineering.* 81–92.

ZHANG, S., RYDER, B. G., AND LANDI, W. 1998. Experiments with combined analysis for pointer aliasing. In *ACM SIGPLAN-SIGSOFT Workshop on Program Analysis for Software Tools and Engineering.* 11–18.

Received August 1998; revised February 1999; accepted April 1999