Lecture Notes in Computer Science, 892, Springer-Verlag, 1995
Proceedings from the 7th International Workshop on
Languages and Compilers for Parallel Computing

Flow-Insensitive Interprocedural Alias Analysis in the Presence of Pointers

Michael Burke, Paul Carini, Jong-Deok Choi
IBM T. J. Watson Research Center

Michael Hind
State University of New York at New Paltz
IBM T. J. Watson Research Center

Abstract. Data-flow analysis algorithms can be classified into two categories: *flow-sensitive* and *flow-insensitive*. To improve efficiency, flow-insensitive interprocedural analyses do not make use of the intraprocedural control flow information associated with individual procedures. Since pointer-induced aliases can change within a procedure, applying known flow-insensitive analyses can result in either incorrect or overly conservative solutions. In this paper, we present a *flow-insensitive data-flow analysis algorithm* that computes interprocedural pointer-induced aliases. We improve the precision of our analysis by (1) making use of certain types of *kill* information that can be precomputed efficiently, and (2) computing aliases generated in each procedure instead of holding at the exit of each procedure. We improve the efficiency of our algorithm by introducing a technique called *deferred evaluation*.
Interprocedural analyses, including alias analysis, rely upon the *program call graph* (PCG) for their correctness and precision. The PCG becomes incomplete or overly imprecise in the presence of function pointers. This paper also describes a method for constructing the PCG in the presence of function pointers.

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 in providing compiler and run-time support for the parallel execution of programs originally written in sequential languages [2, 3, 25, 14]. It is also important for compilers and programming environment tools [2, 10, 21]. Data-flow analysis for programs written in languages with only static data structures (i.e., arrays), such as FORTRAN, enjoys numerous techniques developed for it. However, data-flow analysis for programs written in languages

with dynamically-allocated data structures, such as C, C++, LISP, and Fortran 90, has been less successful due to pointer-induced aliasing.

Aliasing occurs when there exists more than one *access path* [24] to a storage location. Access paths are l-value expressions that are constructed from variables, pointer indirection operators, and structure field selection operators. In C these expressions would include the '$*$' indirection operator, and the '$->$' and '.' field select operators. Two access paths are *must-aliases* at a statement S if they refer to the same storage location in all execution instances of S. Two access paths are *may-aliases* at S if they refer to the same storage location in some execution instances of S. This paper addresses the computation of may-aliases, of which must-aliases are a subset. We will refer to may-aliases as aliases, whenever the meaning is clear from context. For example, consider statement S_1 in Figure 1. After the statement is executed, $*p$ and r refer to the same storage location and thus become aliases of each other, which we express as the *alias relation* $<*p, r>$. In the figure, we denote the alias relations holding *after* each statement by placing it on the same line with the statement.

SubA() {	Flow-Sensitive	Flow-Insensitive
S_1: `p = &r;`	$\{<*p, r>\}$	$\{<*p, r>, <*q, s>, <*q, r>, <*q, t>\}$
`if (...)`		
S_2: `q = p;`	$\{<*p, r>, <*q, r>\}$	$\{<*p, r>, <*q, s>, <*q, r>, <*q, t>\}$
`else`		
S_3: `q = &s;`	$\{<*p, r>, <*q, s>\}$	$\{<*p, r>, <*q, s>, <*q, r>, <*q, t>\}$
S_4: ...	$\{<*p, r>, <*q, s>, <*q, r>\}$	$\{<*p, r>, <*q, s>, <*q, r>, <*q, t>\}$
S_5: `q = &t;`	$\{<*p, r>, <*q, t>\}$	$\{<*p, r>, <*q, s>, <*q, r>, <*q, t>\}$
}		

Fig. 1. Example Program Segment and Its Aliases

Data-flow analysis algorithms can be classified into two categories: *flow-sensitive* and *flow-insensitive* [4, 28]. Flow-sensitive algorithms consider intraprocedural control flow information during the analysis and, in general, are more precise than flow-insensitive algorithms. Flow-insensitive algorithms do not make use of intraprocedural control flow information during the analysis. As such, they can be more efficient than flow-sensitive algorithms and have been used primarily for the class of problems for which flow-sensitive algorithms do not provide increased precision. There exists, however, a range of trade-offs between efficiency and precision [9]. 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.

In this paper, we present a flow-insensitive interprocedural algorithm that

computes may-aliases in the presence of general pointers for languages like C, C++, LISP, and Fortran 90, and that can provide information with comparable precision to flow-sensitive analyses. The algorithm, which has been implemented in a Fortran 90 research prototype, is based on the general framework for flow-insensitive analysis described in [9] without algorithmic elaborations. We also present a method for constructing the *program call graph* (PCG) in the presence of function pointers. The method accommodates function pointers precisely and efficiently in the framework of pointer-induced alias analysis. It is applicable to both flow-sensitive and flow-insensitive frameworks for computing pointer-induced aliases.

Consider the example in Figure 1, which illustrates the major difference between flow-sensitive and flow-insensitive analyses. Assume that no aliases hold before S_1 and that S_4 does not modify any pointer variables. Flow-sensitive analysis [9] will compute alias relations $<*p, r>$ and $<*q, t>$, but not $<*q, r>$ or $<*q, s>$, as holding immediately after S_5.[1] Since no intraprocedural control information is used in a flow-insensitive analysis, it does not distinguish the execution order among statements within a procedure. As shown in the third column, it will thus compute all four aliases as holding at each point in the procedure, except at the procedure entry. This example shows that for pointer-induced aliasing, flow-sensitive analysis can provide better precision than flow-insensitive analysis.

However, the difference in precision between the two analyses can be negligible for certain programs. Consider the effect of the `if` statement in Figure 1. If we exclude statement S_5, the same three alias relations are computed to hold immediately after S_4 in both a flow-sensitive and a flow-insensitive analysis. Branches and loops tend to diminish the difference in precision between the two analyses. Further, when a flow-sensitive analysis produces a large number of alias relations, our flow-insensitive analysis can provide a more efficient alternative.

Notice that in Figure 1 the assignment to q at S_5 *kills* any alias relations of $*q$ arriving at S_5. A flow-sensitive analysis makes use of this kill information in computing the alias relations holding after S_5. This is possible since it can determine what nodes follow S_5 using control flow information. In Section 3, we show how to improve the precision of our flow-insensitive analysis, without incurring the full overhead of flow-sensitive interprocedural analysis, by making use of certain types of kill information that can be computed efficiently.

For correctness and precision, interprocedural data-flow analyses make use of the PCG. Figure 2 illustrates the PCG for an example program segment, where nodes represent procedures and edges represent call sites. In the presence of function pointers and parameters, the PCG of a program is incomplete, resulting in either an incorrect or overly conservative analysis. Call site $c3$ illustrates how this can occur. Without precise knowledge of the aliases of `*funcPtr()`, a PCG edge must be inserted to each PCG node for correctness. Our method constructs the PCG of a program in the presence of function pointers precisely

[1] We do not include the alias relation $<*p, *r>$ holding at S_2, since it can be inferred by the alias relations $<*p, r>$ and $<*q, r>$ holding there (Section 2).

and efficiently in the framework of pointer-induced alias analysis. The precision of the method is the same as the precision of the underlying pointer-induced alias analysis method.

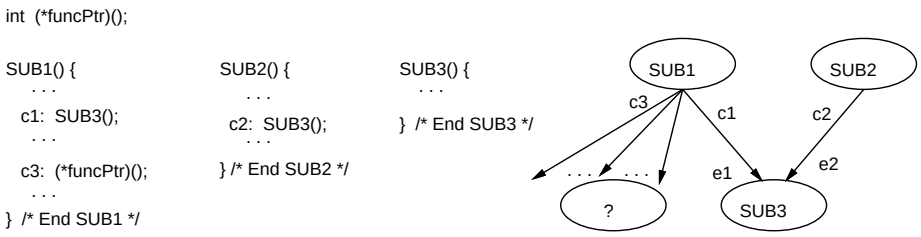

Fig. 2. Example Program Segment and Its PCG

The rest of the paper is organized as follows. Section 2 defines terminology used in the paper and describes the background context. Section 3 describes the framework for our flow-insensitive interprocedural analysis of pointer-induced aliasing. Section 4 gives further details concerning the intraprocedural component of our analysis. Using *deferred evaluation*, we show how to reduce the number of statements that are considered during the iteration step of the intraprocedural component. Section 5 describes how to accommodate function pointers in the framework of alias analysis. Section 6 compares our work with others, and Section 7 draws conclusions.

2 Background

We ignore the computation of *static* aliases that occur due to the C `union` construct and Fortran `EQUIVALENCE` statement. Static aliases do not change within a program and can be computed by the front-end of the compiler. In this paper, we focus on computing pointer-induced aliasing, while ignoring parameters (reference, call-by-value, or copy-in-copy-out). Complete discussions on the interactions between parameters and pointer-induced aliasing can be found in [23, 9].

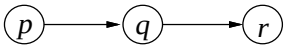

Fig. 3. Example Alias Graph

Pointer-induced alias relations determine a directed graph, called an *alias graph*, as shown in Figure 3: $< *p, q >$ implies that there is a (de-referencing) edge from object p to object q. Likewise, $< **p, r >$ implies that there exists an

object q such that there is an edge from p to q and one from q to r. Our method assumes that every memory location which is referenced or dereferenced has a unique name. Each named object that participates in a pointer-aliased relation is associated with a single node in the directed graph. Although we have proposed a method for naming dynamically allocated objects (malloc sites) in [9], any naming method may be employed.

The *exhaustive* list of aliases holding for the example is $\{<*p, q>, <*q, r>, < **p, r >, < **p, *q >\}$. Notice that $< **p, r >$ and $< **p, *q >$ can be inferred from $\{<*p, q>, <*q, r>\}$, which corresponds to the *transitive reduction* [1] of the directed graph. This *compact representation* [9], which we utilize in our pointer-induced alias analysis method, combines two techniques to reduce the size of the alias sets. It discards alias pairs that do not have at least one named object or that involve more than one level of dereferencing. This representation enables deferred evaluation to be used during the intraprocedural component of our flow-insensitive analysis. A discussion of this technique is provided in Section 4. Tradeoffs between this representation and a more exhaustive approach [23] are discussed in [27].

3 Flow-Insensitive Interprocedural Alias Analysis

To improve efficiency, a flow-insensitive analysis does not rely on intraprocedural control flow information. Instead, the effects of a procedure are captured in a summarized form and associated with either a node or edge of the PCG. In the computation of pointer-induced aliasing, the information directly generated at a node is dependent not only on the statements in the procedure, but also on the information that is propagated along the PCG edges to and from that node. Thus, we cannot determine the direct effects of a procedure without interprocedural information. Instead, the set of pointer-assignment statements in a procedure is associated with each node. Their effect on alias information will be computed during the analysis. The information describing how parameters are passed at a call site is once again associated with each edge.

With the above information as input, the flow-insensitive interprocedural alias analysis computes $PGen_p$, the set of alias relations *generated* by the invocation of procedure p, for each procedure in the program. This approach differs from previous approaches for flow-sensitive analysis, which summarize alias information of a procedure by capturing what *holds* on exit from the procedure [23, 9].

When the summarized procedure information for flow-insensitive analysis is the set of aliases that *hold* on exit of the procedure, alias information can be propagated along *unrealizable* control paths. The advantage of our approach for flow-insensitive analysis is that only the aliases that are generated in the procedure are propagated back to a call site which invokes the procedure.

For the example in Figure 2, aliases propagated from $Sub1$ to $Sub3$ along $e1$ can be propagated, if not killed in $Sub3$, not only to call site $c1$ in $Sub1$, but also to call site $c2$ in $Sub2$ via $e2$, yielding conservative information. Using our approach, only aliases that are generated in $Sub3$ are propagated back to

$Sub2$. This does not entirely eliminate the unrealizable control path problem, since aliases generated by a procedure can depend on the aliases that hold in that procedure. (Section 3.3 provides further details.) Techniques that handle this problem can be incorporated into our algorithm to handle these indirect effects and further refine precision [9, 23].

In addition to $PGen_p$, we compute the following sets:

$Entry_p$**:** the set of alias relations that hold upon entry to procedure p,
$Holds_p$**:** the set of alias relations that are presumed to hold at each point in p, except at the entry of p,
$IGen_p$**:** the set of alias relations *generated* by the pointer-assignment statements[2] of procedure p,
$CSGen_p$**:** the set of alias relations *generated* by call sites in the procedure p.

In this section, we describe the relationship among these sets and provide a set of data-flow equations representing this relationship. We classify these five sets according to the manner in which information is propagated when computing them. Two categories of sets exist: *interprocedural* ($Entry_p$ and $CSGen_p$) and *intraprocedural* ($IGen_p$, $Holds_p$, and $PGen_p$).

$Entry_p$ represents the alias information that *holds* upon entry to p. It is captured by propagating the alias information that holds at a call site of p *forward* along the corresponding PCG edge. This information is unioned with the alias information propagated from other call sites which invoke p.

$CSGen_p$ represents the alias information *generated* at the call sites in p. This information is computed by propagating the $PGen$ of the procedures called in p *backward* along the corresponding PCG edges. Since information is propagated forward and backward along the PCG edges, pointer-induced aliasing over the PCG is a *bi-directional* problem.

In flow-sensitive analyses *kill* information is used to reduce the size of alias sets propagated forward and backward along the PCG edges as well as within each procedure [23, 9]. In Section 3.4, we describe how to improve the precision of the interprocedural sets by selectively utilizing kill information without incurring the full overhead of flow-sensitive interprocedural analysis.

3.1 Overview of the Algorithm

Figure 4 shows a high-level description of our algorithm for interprocedural alias analysis.[3] The major characteristic of the algorithm is the *interleaving* of the intraprocedural (Steps S_6 and S_7) and interprocedural phases (Steps S_8 and S_9). Each phase is followed by the other, which uses the results of the previous phase as its input. The intraprocedural phase can be flow-sensitive or flow-insensitive. The algorithm that we present here is flow-insensitive. This interleaving removes

[2] We regard storage allocation and deallocation statements for pointers, such as `malloc` and `free` in C, as pointer assignment statements.

[3] In the presence of PCG cycles, topological order is defined by the removal of *back edges* [20].

S_1: foreach procedure p in the PCG
S_2: set $Entry_p = STATIC\_ALIASES$;
S_3: set $CSGen_p = \{\}$;
S_4: build initial PCG;
S_5: repeat
S_6: foreach procedure p in the PCG
S_7: compute intraprocedural alias sets of p, using interprocedural alias sets;
S_8: foreach procedure p in the PCG
S_9: compute interprocedural alias sets of p, using intraprocedural alias sets;
S_{10}:until aliasing converges;

Fig. 4. Algorithm to Compute Interprocedural Aliasing

the requirement that the iterations alternate topological directions when solving the bi-directional problem. An interleaving paradigm similar to this is used in our previous work for flow-sensitive analysis of interprocedural aliasing [9].

During an intraprocedural phase, the following three *intraprocedural sets* are computed for each procedure p: $PGen_p$, $Holds_p$, and $IGen_p$. In computing these intraprocedural sets, the two interprocedural sets, ($Entry_p$ and $CSGen_p$) computed during the previous interprocedural phase, are used. Computing the interprocedural sets in turn uses the intraprocedural sets computed during the previous intraprocedural phase. The iterations over the PCG (Steps S_5 through S_{10}) handle this mutual dependence between the inter- and intraprocedural sets.

The basic algorithm presented in Figure 4 can be enhanced in a number of ways. The interprocedural computation, Steps S_8 and S_9, can be incorporated into the intraprocedural phase. This can be accomplished by performing the update of the interprocedural information of each procedure, Step S_9, as soon as the relevant intraprocedural information is computed at S_7. Using this enhancement, a topological traversal of the PCG is generally more efficient than visiting procedures in arbitrary order.[4] Finally, the iterative algorithm can be implemented with a worklist to improve efficiency.

In the following sections, we describe how to compute the interprocedural and intraprocedural sets. These descriptions would require no modifications to incorporate the enhancements described above.

[4] Alternating iterations between topological and reverse-topological order may further improve efficiency.

3.2 Interprocedural Phase

With the intraprocedural sets computed, the interprocedural sets can be computed as follows:

$$Entry_p = \bigcup_c ForwardBind_c(HoldsBefore_c), \text{where call site } c \text{ calls } p \quad (1)$$

$$CSGen_p^c = BackwardBind_c(PGen_q), \text{where } p \text{ calls } q \text{ at call site } c \quad (2)$$

$$CSGen_p = \bigcup_{c \in p} CSGen_p^c \quad (3)$$

$Entry_p$ is computed by propagating information along all call sites to p. $ForwardBind_c$ maps the set of alias relations holding at call site c (referred to as $HoldsBefore_c$) into alias relations holding in the called routine, using the arguments of c. With flow-insensitive analysis, $HoldsBefore_c$ for a call site c in q is the same as $Holds_q$. $BackwardBind_c$, which is similar to $ForwardBind_c$, maps alias relations holding at the called routine into alias relations holding immediately following the call site c, using the arguments of c. (Details of these mapping mechanisms are given in [9].)

3.3 Intraprocedural Phase

With the interprocedural sets computed, the intraprocedural sets can be computed by solving the following equations:

$$Holds_p = Entry_p \cup CSGen_p \cup IGen_p \quad (4)$$

$$IGen_p = \bigcup_s IGen_s(Holds_p), \text{where statement } s \text{ in } p \text{ is a ptr. assignment} \quad (5)$$

$$PGen_p = CSGen_p \cup IGen_p \quad (6)$$

The cyclic dependences between equations (4) and (5) require iteration over the set of pointer-assignment statements in p. Since our flow-insensitive analysis does not use control flow information, it must, for correctness, reflect all possible paths that can be constructed by the set of statements associated with p. Thus, we treat each procedure as a single (potentially large) loop consisting of all the pointer-assignment statements of the procedure in some arbitrary order. Figure 5 illustrates our technique. The control flow graph on the left (Figure 5-A) represents the original program. The graph in the middle (Figure 5-B) illustrates the manner in which we capture the effect of all possible paths. The order in which statements are chosen in this graph is arbitrary, and for this example is the worst possible order. A total of three iterations are required to construct the eight alias relations that result from this graph [7].

Figure 6 specifies our intraprocedural algorithm. Initially, we assume aliases that reach p interprocedurally via $Entry_p$ and $CSGen_p$ are in $Holds_p$ (S_1). We also assume $IGen_p$ is empty (S_2). The loop from S_4 to S_6 traverses each pointer assignment statement, adding alias relations generated by each statement to the

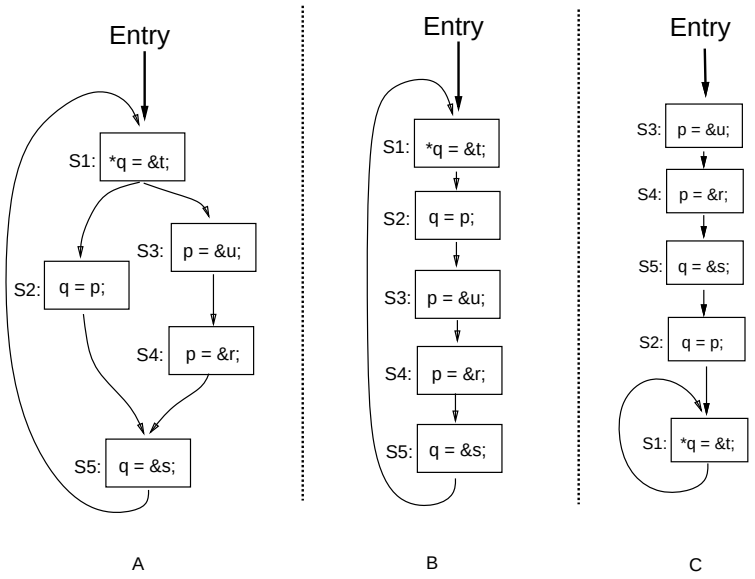

Fig. 5. Intraprocedural Example

$Holds_p$ and $IGen_p$ sets for the procedure of interest. Since the aliases generated depend on $Holds_p$, this traversal is repeated until no new aliases are generated. Note that computing $PGen_p$ is not part of the iteration; $PGen_p$ is computed upon convergence. Where kill information is not considered, this method is equivalent to considering all possible paths. Section 4 describes how to identify certain types of pointer-assignment statements that need not be considered during the iteration (Steps S_4 to S_6).

S_1: $Holds_p = Entry_p \cup CSGen_p$
S_2: $IGen_p = \{\}$
S_3: repeat
S_4: foreach pointer assignment statement, s, in p
S_5: compute the set, $A = IGen_s(Holds_p)$
S_6: add A to $Holds_p$ and $IGen_p$
S_7: until aliasing converges;
S_8: $PGen_p = CSGen_p \cup IGen_p$

Fig. 6. Intraprocedural Algorithm

In [9], the effect of a pointer-assignment statement s on aliasing is described in terms of its alias *transfer function* (TF_s) as follows:

$$HoldsAfter_s = TF_s(HoldsBefore_s),$$

where $HoldsBefore_s$ and $HoldsAfter_s$ are the aliases holding immediately before and after s, respectively. $IGen_s$ in Equation 5 can be derived from TF_s. With flow-insensitive analysis, $Holds_p$ is the alias information assumed to hold at each point in p, including immediately before and after s. Thus, when a fixed point is reached, the following equation is satisfied by $Holds_p$:

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

3.4 Using Kill Information

In this section, we describe how to improve the interprocedural precision by utilizing precomputed kill information. Consider Figure 7, where procedure r calls procedure p at S_2, and the assumed values for $Entry_r$ and $CSGen_p^{S_2}$ are provided in the figure. By Equation 1, the alias relations that are propagated to $Entry_p$ from S_2 are those that hold at S_2. From our discussion above, this is $Holds_r$, which is: $\{<*x, a_1>, \ldots, <*x, a_n>, <*x, y>, <*x, z>\}$.

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

$Entry_r = \{<*x, a_1>, \ldots, <*x, a_n>\}$
$IGen_r = \{<*x, y>, <*x, z>\}$
$IGen_p = \{<*x, y>, <*x, z>\}$
$CSGen_p^{S_2} = \{<*x, b_1>, \ldots, <*x, b_n>\}$

Fig. 7. $ForwardKill$ and $BackwardKill$ Example

Although this is correct, one can improve the precision of this solution by selectively using kill information to limit the alias relations that are propagated interprocedurally [9]. We adopt this approach by defining for each call site c, contained in procedure r, $ForwardKill_c$, which represents alias relations that are killed along all paths from the entry point of r to c. We use $ForwardKill_c$ to remove alias relations from $Entry_r$ and then factor in any aliases that are generated by r ($IGen_r \cup CSGen_r$), for correctness, in computing the aliases

to be propagated at c. In the example of Figure 7, $\{< *x, y >, < *x, z >\}$ is propagated to p, but not $\{< *x, a_1 >, \ldots, < *x, a_n >\}$.

Below we give the updated equation for $Entry_p$ (Equation 1 of Section 3.2). Along with the addition of kill information, $HoldsBefore_c$ has been expanded.

$$Entry_p = \bigcup_c ForwardBind_c(Entry_r - ForwardKill_c \cup IGen_r \cup CSGen_r),$$

where call site c, located in r, calls p

The computation of $ForwardKill_c$ can be performed before the interprocedural propagation (S_6 of Figure 4), and used when needed. This allows the interprocedural algorithm to remain flow-insensitive, as no control flow information is required during the propagation. A pointer variable is killed when it appears without any indirection operations on the left side of an assignment statement or in a `malloc` or `free` function call.[5] Killing a pointer variable via indirection, i.e. $*x = \ldots$ requires must alias information for $*x$. Thus, for kill information to be used for this kind of statement, must-alias information is required.

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 routine to a calling routine p at call site c via $CSGen_p^c$. To improve the precision of this set, we define the *BackwardKill* set for a call site c in p, ($BackwardKill_c$) to represent alias relations that are killed along all paths from c to the *exit* node of p. We use this set when constructing $PGen_p$, by removing all alias relations from $CSGen_p^c$ that are killed after the call. If any of these aliases are generated elsewhere in p, they are included in $PGen_p$ for correctness.

Below we give the updated equation for $PGen_p$ (Equation 6 of Section 3.3). We divide $CSGen_p$ into its call site components to allow the addition of kill information.

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

Figure 7 also gives an example where using the *BackwardKill* information improves the precision of $PGen_p$. Without kill information we have,

$$PGen_p = CSGen_p \cup IGen_p = \{< *x, b_1 >, \ldots, < *x, b_n >, < *x, y >, < *x, z >\}.$$

By using the new equation for $PGen_p$, we get

$$PGen_p = \{< *x, y >, < *x, z >\}.$$

Like $ForwardKill_c$, $BackwardKill_c$ can be computed as a preprocessing step before the flow-insensitive iteration over the PCG has begun. 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 [9].

[5] It is possible to broaden this definition, and improve precision, by allowing user function calls to kill pointer variables [7].

4 Deferred Evaluation

Section 3.3 describes how to capture the effect of all possible intraprocedural paths by treating a procedure as a single loop consisting of the pointer-assignment statements of the procedure in some arbitrary order. In this section we improve the efficiency of the intraprocedural phase by identifying certain types of pointer assignments that need not be considered during the intraprocedural iteration. For example, consider statements S_3, S_4, and S_5 of Figure 5. Since the alias relations generated by these statements can be computed independently of the alias relations that hold before executing them, these statements need not be included in the iteration.[6] These statements have a constant $IGen$ set; they only contribute to the procedure $IGen$ set and do not depend on what aliases hold. Statements with a constant $IGen$ set can be treated like $ForwardKill$ and $BackwardKill$ information, in that the alias relations they generate can be precomputed prior to the interprocedural propagation (S_5 of Figure 4). This enhancement reduces the number of intraprocedural iterations required for convergence from three to two.

Determining the alias relations generated by statements S_1 and S_2 requires information about the alias relations holding at these statements. In statement S_1 the aliases of $*q$ are required. In particular, for every x such that $<*q, x>$ holds, we will create an alias relation $<*x, t>$. Statement S_2 requires the aliases of $*p$. It will create alias relations $<*q, y>$ for all y where $<*p, y>$ holds. Thus, it appears that both types of statements must be included in our intraprocedural iteration.

However, for the aliases generated by statement S_2, the dereference component is known: i.e., $*q$. Furthermore, q is the *source* node of the edges that represent these aliases in the alias graph. Therefore, our algorithm is able to defer the evaluation of statement S_2 until after the intraprocedural iteration has completed, when the aliases of $*p$ are known. This is accomplished by creating a *deferred alias relation* from q to p, with a *deferred value* of 0. This signifies that $*q$ will be aliased to all objects that p can reach via a path of length 1 over the alias graph. If aliases of $*q$ are required during the iteration, we infer the alias relations the deferred alias relation represents. In this example, it represents edges from q to whatever p points to, i.e. $\{<*q, y> \mid \text{for all } y \text{ such that } <*p, y>\}$.

Thus, only statement S_1 remains in the intraprocedural iteration. If the procedure does not contain any statements of this type, no intraprocedural iteration is required. After the intraprocedural iteration has completed, deferred alias relations are expanded into full alias relations without a loss of precision. We refer to this approach as *deferred evaluation*.

Figure 5-C shows how the example program would be processed using deferred evaluation. Notice that only one statement is contained in the loop and thus, only one iteration would be required for aliasing to converge. This com-

[6] This is not the case with *exhaustive representations* [23], which, assuming $<*s, p>$ holds, would include $<** s, u>$ as holding after considering S_3.

pares favorably to the original analysis, where three iterations are required before convergence. (See [7] for more details.)

We generalize the previous example by classifying pointer-assignment statements into three categories listed below. Type 1 statements have a constant $IGen$ function and are processed first. For Type 2 statements we create deferred alias relations with a *deferred value* of r. This is performed before the iteration and then expanded after the iteration converges. We iterate over Type 3 statements.

Type 1: $p = \&q;$
Type 2: $p = \underbrace{* \ldots *}_{r} q; r \geq 0$
Type 3: $\underbrace{* \ldots *}_{l} p = \underbrace{* \ldots *}_{r} q; l \geq 1, r \geq -1$, "$r = -1$" is equivalent to $\&q$.

Below we summarize the enhanced intraprocedural algorithm using deferred evaluation. This algorithm decreases the number of pointer assignment statements considered in S_4 of Figure 6, by considering some statements at S_2 and expanding deferred alias relations after S_8. This reduces the number of statements considered during the intraprocedural iteration, which can also reduce the number of iterations required for convergence of this phase. No precision is lost using this technique.

1. Add alias relations from Type 1 statements to $IGen_p$ (S_2)
2. Create deferred alias relations for Type 2 statements (S_2)
3. Iterate over Type 3 statements only (S_4)
4. Expand deferred alias relations (after S_9)

5 Incremental Analysis for Function Pointers

In this section, we describe a method for constructing the PCG in the presence of function parameters, variables, and pointers. Our method accommodates function parameters and arbitrary levels of function pointers by using the framework of pointer-induced aliasing. When embedded in the alias analysis, which generally occurs before any other phases that need the PCG, the method incurs minimal overhead. Although we focus on flow-insensitive analysis in this paper, the techniques described in this section to accommodate function pointers can also be applied to flow-sensitive analysis. We will use the term "function pointers" to refer to function parameters, function variables, and function pointers. Likewise, we will use "regular pointers" to refer to pointers that are not function pointers.

Recall from Figure 4 the *interleaving* of the intraprocedural (Steps S_6 and S_7) and interprocedural phases (Steps S_8 and S_9). Interleaving these two phases makes our algorithm particularly well suited for accommodating function pointers via *incremental* interprocedural data-flow analysis. Incremental data-flow analysis addresses the issue of updating data-flow information, without a full

recomputation, when part of the program is changed [26, 6]. In our case, the program itself does not change, but its graphical representation, the PCG, changes as the analysis proceeds. Such changes in the representation of the program warrant the use of incremental analysis for efficiency.

Incremental alias analysis in the presence of function pointers is achieved by first handling function pointers the same way as regular pointers during the intraprocedural phase of the analysis (Steps S_6 and S_7), and then identifying new PCG edges from the alias information as the computation proceeds. This way, our method *does not* compute aliasing from scratch as new PCG edges are found. Instead, it adds these new edges to construct the new PCG and incrementally continues the alias computation, using the alias information computed with the previous PCG as the initial condition.

Formally, this amounts to computing an additional *interprocedural* set of procedures, which is the set of aliases of a function pointer, fp, invoked at call site, c, as follows:[7]

$$FuncAlias_c = \{ proc \mid < *fp, proc > \in HoldsBefore_c,\ *fp \text{ is invoked at } c \}.$$

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

Alias analysis generally occurs before any other phases that need the PCG, and the embedded PCG construction will incur minimal overhead. When applied only to function pointers, our algorithm still constructs the same PCG without incurring the overhead of full alias analysis. This way, it can be used as a general method to construct the PCG, independently of any other data-flow analysis such as the alias analysis of regular pointers described in this paper. More details are provided in [7].

6 Related Work

Larus [25] gives 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 [25]. He uses the fastness closure technique of Graham and Wegman [15] to correctly process the alias graphs.

Flow-sensitive interprocedural algorithms for pointer-induced aliases are given in [23, 9, 12, 13]. They utilize kill information and can provide improved precision over a flow-insensitive analysis that does not utilize kill information during the analysis. Our flow-insensitive algorithm utilizes precomputed kill information to improve interprocedural precision. Complete discussions on the interactions between reference parameters and pointer-induced aliasing are also given in [9].

[7] If fp is a function parameter or variable, the '$*$' is not required.

A framework for flow-insensitive interprocedural analysis of pointer-induced aliasing is described in [9] without algorithmic elaborations. The algorithm described in this paper is based on that framework.

Methods for constructing the PCG in the presence of function parameters and function variables are given in [34, 5, 8, 16, 22]. Our method accommodates function parameters and arbitrary levels of function pointers. As such, it is more general than methods for constructing the PCG in the presence of function parameters [5, 31, 8] or function variables [16, 22]. By using the framework of pointer-induced aliasing, it is also more precise than Weihl's method, which performs transitive closure of the alias relations [34].

Emami, Ghiya, and Hendren [13] independently proposed an algorithm similar to ours for constructing the PCG for the same programming model we consider. Also, a similar problem to constructing the PCG is performing control flow analysis in functional languages, where functions are first class objects [33]. Solutions to this problem are given in [33, 32, 19, 29, 11, 30]

Hall and Kennedy [16] suggested a method, similar to Lakhotia's method [22], to improve Weihl's approach. Lakhotia performs interprocedural constant propagation analysis over the *system dependence graph* [18]. We believe his method for handling *label variables* can be easily incorporated into our algorithm. Our algorithm can be easily extended to use a work list, as done in [23, 16]. It can also be extended to build the *Interprocedural Control Flow Graph* (ICFG) [8, 17, 23], instead of the PCG.

7 Conclusions

In this paper, we have presented a flow-insensitive algorithm that computes interprocedural pointer-induced aliases. We enhance the precision of the interprocedural phase of our algorithm by (1) utilizing precomputed kill information, and (2) computing aliases generated in each procedure rather than aliases holding at the exit of each procedure. We preserve correctness during the intraprocedural phase of the algorithm by iterating over those statements that are dependent on the current alias relations. A technique called deferred evaluation is introduced to improve the efficiency of the intraprocedural phase by identifying a class of pointer-assignment statements that can be excluded from the intraprocedural iteration.

We have also presented a method for constructing the PCG in the presence of function pointers. This method can be applied to either a flow-sensitive or flow-insensitive analysis. It uses incremental analysis and is based on our alias analysis framework.

Our flow-insensitive alias analysis algorithm has been implemented in a Fortran 90 research prototype. We plan to measure the efficiency and precision of our algorithms on Fortran 90, C, and C++ programs.

Acknowledgements

We thank the referees for their helpful comments and Laureen Treacy for her careful proofreading.

References

1. A. V. Aho, M. R. Garey, and J. D. Ullman. The transitive reduction of a directed graph. *SIAM Journal on Computing*, 1(2):131–137, 1972.
2. Frances Allen, Michael Burke, Philippe Charles, Ron Cytron, and Jeanne Ferrante. An overview of the ptran analysis system for multiprocessing. *Proceedings of the ACM 1987 International Conference on Supercomputing*, 1987. Also published in The Journal of Parallel and Distributed Computing, Oct., 1988, 5(5) pages 617-640.
3. Randy Allen, David Callahan, and Ken Kennedy. Automatic decomposition of scientific programs for parallel execution. *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 63–76, January 1987.
4. John Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. *6th Annual ACM Symposium on the Principles of Programming Languages*, pages 29–41, January 1979.
5. Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data flow analysis. Technical report, IBM Research, August 1987. Report RC12702.
6. Michael Burke. An interval-based approach to exhaustive and incremental interprocedural data-flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.
7. Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Efficient flow-insensitive alias analysis in the presence of pointers. Technical report RC 19546, IBM T. J. Watson Research Center, September 1994.
8. D. Callahan, A. Carle, M. W. Hall, and K.Kennedy. Constructing the procedure call multigraph. *IEEE Transactions on Software Engineering*, 16(4):483–487, April 1990.
9. Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, January 1993.
10. Jong-Deok Choi and Jeanne Ferrante. Static slicing in the presence of GOTO statements. *ACM Transactions on Programming Languages and Systems*, 1994. To appear.
11. Alain Deutsch. 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*, pages 157–168, San Francisco, January 1990. ACM Press.
12. Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.
13. Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, 1994.

14. Dennis Gannon, Vincent A. Guarna, Jr., and Jenq Kuen Lee. Static analysis and runtime support for parallel execution of C. *Proceedings of the Second Workshop on Languages and Compilers for Parallel Computing*, August 1989.
15. Susan L. Graham and Mark Wegman. A fast and usually linear algorithm for global flow analysis. *Journal of the Association for Computing Machinery*, 23(1):172–202, January 1976.
16. Mary W. Hall and Ken Kennedy. Efficient call graph analysis. *ACM Letters on Programming Languages and Systems*, 1(3):227–242, September 1992.
17. Mary Hean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition – use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.
18. Susan Horwitz, Thomas Reps, and David Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1):26–60, January 1990.
19. W. L. Harrison III. The interprocedural analysis and automatic parallelisation of Scheme programs. *Lisp and Symbolic Computation*, 2(3):176–396, October 1989.
20. John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *JACM*, 23,1:158–171, January 1976.
21. K. Kennedy, K. S. McKinley, and C. Tseng. Interactive parallel programming using the parascope editor. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):329–341, July 1991.
22. Arun Lakhotia. Constructing call multigraphs using dependence graphs. In *20th Annual ACM SIGACT-SIGPLAN Symposium on the Principles of Programming Languages*, pages 273–284. ACM, January 1993.
23. William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.
24. J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. *Proceedings of the ACM SIGPLAN '88 Conference on Programming Language Design and Implementation*, 23(7):21–34, July 1988.
25. James Richard Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors.* PhD thesis, University of California, 1989. Technical Report No. UCB/CSD 89/502.
26. T. J. Marlowe and B. Ryder. An efficient hybrid algorithm for incremental data flow analysis. *17th Annual ACM Symposium on the Principles of Programming Languages*, pages 184–196, January 1990.
27. Thomas Marlowe, William Landi, Barbara Ryder, Jong-Deok Choi, Michael Burke, and Paul Carini. Pointer-induced aliasing: A clarification. *SIGPLAN Notices*, 28(9):67–70, September 1993.
28. T.J. Marlowe, B.G. Ryder, and M.G. Burke. Defining flow-sensitivity in data flow problems. *In Preparation*, 1994.
29. Torben Æ Mogensen. Binding time analysis for polymorphically typed higher-order languages. In *Proceedings TAPSOFT*, volume 352 of *Lecture Notes in Computer Science*, pages 298–312. Springer Verlag, 1989.
30. A. Neirynck, P. Panangaden, and A. J. Demers. Effect analysis in higher-order languages. *International Journal of Parallel Programming*, 18(1):1–17, 1989.
31. Barbara Ryder. Constructing the call graph of a program. *IEEE Software Engineering*, May 1979.
32. Peter Sestoft. Replacing function parameters by global variables. In *Conference on Functional Programming Languages and Computer Architecture*, pages 39–53,

London, September 1989. ACM Press.
33. Olin Shivers. Control flow analysis in Scheme. In *SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 164–174, June 1988.
34. William Weihl. Interprocedural data flow analysis in the presence of pointer, procedure variables and label variables. *Conf. Rec. Seventh ACM Symposium on Principles of Programming Languages*, 1980.

This article was processed using the LaTeX macro package with LLNCS style