Efficient Flow-Insensitive Alias Analysis in the Presence of Pointers

Michael Burke* Paul Carini* Jong-Deok Choi* Michael Hind†*

Abstract

Data-flow analysis computes the information required by compilers to support the parallel execution of programs written in sequential languages. Data-flow analysis for programs written in languages such as FORTRAN has been well developed. However, due to the presence of pointers, data-flow analysis has been less successful for languages such as C, C++, and LISP.

Flow-insensitive interprocedural methods do not make use of the intraprocedural control flow information associated with individual procedures. As such, they are in practice generally more efficient than *flow-sensitive* methods. Since pointer-induced aliases can change within a procedure, applying known flow-insensitive methods can result in incorrect or overly conservative solutions. In this paper, we present the first *flow-insensitive algorithm* that efficiently computes interprocedural pointer-induced aliases and that gives comparable information to flow-sensitive analyses. This is achieved by (1) making use of certain types of *killing* information that can be computed efficiently, and (2) computing aliases generated in each procedure instead of holding at the exit of each procedure.

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 [ABC+87, ACK87, Lar89, GGL89]. It is also important for compilers and programming environment tools [ABC+87, CF94, KMT91]. 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 aliasing*, which occurs when two or more pointer expressions refer to the same storage location. 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* $< *p, r >$.

Data-flow analysis algorithms can be classified into two categories: *flow-sensitive* and *flow-insensitive* [MRB94]. Flow-sensitive algorithms consider control flow information and, in general,

*IBM T. J. Watson Research Center P. O. Box 704 Yorktown Heights, NY 10598

†State University of New York, New Paltz, New York 12561

SubA ()
{

		Flow-Sensitive	Flow-Insensitive
S_1:	`p = &r;`		
		$\{<*p, r>\}$	$\{<*p, r>, <*q, s>, <*q, r>\}$
S_2:	`q = p;`		
		$\{<*p, r>, <*q, r>\}$	$\{<*p, r>, <*q, s>, <*q, r>\}$
S_3:	`q = &s;`		
		$\{<*p, r>, <*q, s>\}$	$\{<*p, r>, <*q, s>, <*q, r>\}$
S_4:	...		

}

Figure 1: Example Program Segment and Its Aliases.

are more precise than flow-insensitive algorithms. Flow-insensitive algorithms do not make use of control flow information. As such, they are more efficient than flow-sensitive algorithms and have been used mostly for the class of problems for which flow-sensitive algorithms do not provide any better precision. There exists, however, a range of trade-offs between efficiency and precision [CBC93]. 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 the first interprocedural *flow-insensitive* algorithm that efficiently computes aliases in the presence of general pointers for languages like LISP, C, C++, or Fortran 90, and can give comparable information to flow-sensitive analyses. The algorithm is based on the general framework for flow-insensitive analysis described in [CBC93] without algorithmic elaborations. We have implemented this algorithm in a Fortran 90 research prototype.

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 [CBC93] will compute alias relations $< *p, r >$ and $< *q, s >$, but not $< *q, r >$, as holding immediately before S_4.[1] A flow-insensitive analysis will compute all three of the aliases as holding immediately before S_4. In fact, it will compute these aliases as holding at each point in the procedure, except at the entry of the procedure. 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 Figure 2, where an `if` statement is added to the statements of Figure 1. In the presence of this conditional, the same three alias relations are computed to hold immediately before

[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. Thus, we employ the *compact representation* of [CBC93].

```
SubB ()
{
        S1:     p = &r;

                if (...) {
        S2:             q = p;

                else
        S3:             q = &s;

                }

        S4:     ...
}
```

Flow-Sensitive

$\{<*p, r>\}$

$\{<*p, r>, <*q, r>\}$

$\{<*p, r>, <*q, s>\}$

$\{<p, r>, <*q, s>, <*q, r>\}$

Figure 2: Example Program Segment and Its Aliases.

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.

In fact, a flow-sensitive analysis can produce a large number of alias relations holding at each statement [Lan92]. An example of such a program is the **make.c** program, for which 675 alias relations are computed to hold per statement on average, and about 2000 alias relations are computed to hold for one statement, by a flow-sensitive method [Lan92]. For such programs, our flow-insensitive analysis may be as precise as, while more efficient than, flow-sensitive analysis, and should provide a useful alternative to flow-sensitive analysis.

Notice that in Figure 2 the assignment to `q` at S_3 *kills* any alias relations of $*q$ arriving at S_3. A flow-sensitive analysis makes use of this kill information in computing the alias relations holding after S_3. This is possible since it can determine what nodes follow S_3 using control flow information. In Section 3, we show how to improve the precision of flow-insensitive analysis, without incurring the full overhead of flow-sensitive analysis, by making use of certain types of killing information that can be computed efficiently.

A well-known example of flow-insensitive analysis is computing *(interprocedural) reference parameter aliases* [CK89, Ban79], which is generated by the mechanism of passing *reference* parameters at call sites. Another well-known example is computing *PMod* for a procedure: the set of non-local variables *potentially* updated during the execution of a procedure, excluding the effect of aliasing [Ban79]. As with reference parameter aliasing, flow-insensitive analysis of $PMod$ is as precise as flow-sensitive analysis.[2]

[2] A similar, but flow-sensitive analysis is the *Kill* information of a procedure: the set of non-local variables that are *definitely* updated during any execution instance of the procedure.

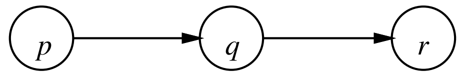

Figure 3: Example Aliasing.

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 the statements that are considered during the intraprocedural iteration. Section 5 compares our work with others, and Section 6 draws conclusions.

2 Background

Aliasing occurs when there exists more than one *access path* [LH88] 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 p if they refer to the same storage location in all execution instances of p. Two access paths are *may-aliases* at p if they refer to the same storage location in some execution instances of p. 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.

We ignore the computation of *static* aliases that occur due to constructs such as the union construct in C and Fortran `EQUIVALENCE`. Static aliases do not change within a program and can be computed by the front-end of the compiler. In this paper we concentrate 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 [LR92, CBC93].

There are alternative ways of representing aliases holding at a statement. Consider the example in Figure 3. The *exhaustive* list of aliases holding for the example is $\{< *p, q >, < *q, r >, < **p, r >\}$ [LR92]. Notice that $< **p, r >$ can be inferred from $\{< *p, q >, < *q, r >\}$, which forms the *transitive reduction* of the exhaustive list. This *compact representation* [CBC93], $\{< *p, q >, < *q, r >\}$, is employed by our alias analysis method. This compact representation allows for fewer iterations during the intraprocedural components of flow-insensitive analysis. More details of this aspect are described in Section 4. [MLR$^+$93] discuss tradeoffs between the two representations.

Interprocedural data-flow analyses make use of the *program call graph* (PCG). Figure 4 illustrates the PCG for an example program segment, where nodes represent procedures and edges represent call sites.[3] Unlike reference-parameter aliasing, pointer-induced aliasing is a *bi-directional* data-flow problem over the PCG: not only can the calling procedure affect the aliasing of the called

[3] We handle function pointers and parameters by treating them the same as other pointer variables. We incrementally expand the PCG as new aliases of function pointers and parameters are identified.

```
int  (*funcPtr)();

SUB1() {               SUB2() {               SUB3() {
   . . .                  . . .                  . . .
  c1:  SUB3();          c2:  SUB3();          }  /* End SUB3 */
   . . .                  . . .
}  /* End SUB1 */      } /* End SUB2 */
```

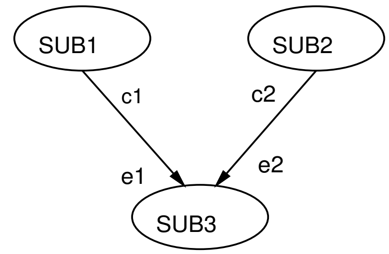

Figure 4: Example Program Segment and Its *PCG*

procedure, but the called procedure can also affect the aliasing of the calling procedure. When the summarized procedure information is the aliases that *hold* on exit of the procedure [LR92, CBC93], alias information can be propagated along *unrealizable* control paths. For the example in Figure 4, 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 $e2$, which should not be allowed.

Our algorithm (presented in Section 3) reduces the significance of this problem by specifying our summarized information in terms of what is *generated* by a procedure. Using this approach, only aliases that are generated in $Sub3$ are propagated back to $Sub2$. However, techniques which handle the unrealizable control paths problem can be used in our algorithm to further refine our precision [LR92, CBC93].

3 Flow-Insensitive Interprocedural Alias Analysis

Since a flow-insensitive analysis does not rely on control flow information, it must capture the effects of a procedure in a summarized form. Depending on the type of information summarized, an interprocedural analysis will associate this information with either a node or edge of the PCG.

Prior to the interprocedural propagation of the *PMod* computation, the direct modifications (i.e. excluding call sites) of non-local variables within a procedure are summarized ($IMod$), and associated with each node is the PCG [Ban79]. This information is then used during the propagation over the PCG.

Consider the computation of *reference parameter aliasing*, where aliases are created by passing a global variable or the same variable in two different positions. For this problem, the information of interest is present at each call site. Since each call site has a corresponding PCG edge, information concerning how parameters are passed is associated with each PCG edge.

The computation of pointer-induced aliases is similar to the $PMod$ computation, in that the effect of each PCG node is summarized. However, unlike the $PMod$ computation, the information directly generated at a node is dependent not only on the statements in the procedure, but also on the information that is propagated along the PCG edges to 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, our *flow-insensitive interprocedural alias analysis* computes $PGen_p$, the set of alias relations *generated* by invocation of procedure p, for each procedure in the program. This approach differs from previous approaches which summarize alias information of a procedure by capturing what *holds* on exit from the procedure [LR92, CBC93]. The advantage of this approach is that only the aliases that are generated in the procedure are propagated back to a call site which invokes the procedure.

From $PGen_p$, we can compute the following sets:

- $Entry_p$: the set of alias relations that hold upon entry to procedure p,
- $IGen_p$: the set of alias relations *generated* by the pointer-assignment statements[4] of procedure p,
- $CSGen_p$: the set of alias relations *generated* by call sites in the procedure p, and
- $Holds_p$: the set of alias relations that hold at each point in p, except at the entry of p.

Computing $PGen_p$ in turn needs the above four sets; there exists a dependence among the five sets which requires iteration over the PCG until a fixed point is reached. 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 into two categories: *interprocedural* and *intraprocedural*. $Entry_p$ and $CSGen_p$ are interprocedural sets. $IGen_p$, $Holds_p$, and $PGen_p$ are intraprocedural sets.

$Entry_p$ represents the alias information which *holds* upon entry to p. It is captured by propagating the alias information which 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 [LR92, CBC93]. In section 3.4, we describe how to improve the precision of our flow-insensitive analysis by selectively utilizing kill information without incurring the full overhead of flow-sensitive analysis.

3.1 Overview of Our Algorithm

Figure 5 shows a high-level description of our algorithm for interprocedural alias analysis.[5] The major characteristic of the algorithm is the *interleaving* of the intraprocedural (Steps S_6 and S_7)

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

[5] In the presence of PCG cycles, topological order is defined by the removal of *back edges* [KU76].

S_1: foreach procedure p

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 visited in a topological traversal over the PCG;

S_7: compute the intraprocedural alias sets of p, using the interprocedural alias sets;

S_8: compute the interprocedural alias sets for each p, using the intraprocedural alias sets;

S_9: until aliasing converges;

Figure 5: Algorithm to Compute Interprocedural Aliasing

and interprocedural phases (Step S_8). 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-sensitive. This interleaving removes the requirement that the iterations alternate topological directions when solving the bi-directional problem. In fact, an interleaving paradigm similar to this is used in our previous work for flow-sensitive analysis of interprocedural aliasing [CBC93].

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 following two *interprocedural sets*, computed during the previous interprocedural phase, are needed: $Entry_p$ and $CSGen_p$. Computing the interprocedural sets in turn requires the three intraprocedural sets computed during the previous intraprocedural phase. The iterations over the PCG (steps S_5 through S_9) handle this mutual dependence between the inter- and intraprocedural sets.

The basic algorithm presented in Figure 5 can be enhanced in a number of ways. First, alternating the iterations over the PCG (steps S_6 and S_7) between topological and reverse-topological order can be more time-efficient in general. Second, the interprocedural phase, step S_8, can be incorporated into the intraprocedural phase by computing the interprocedural information of each procedure as soon as the relevant intraprocedural information is updated at S_7. Finally, the iterative algorithm can be implemented with a worklist to improve efficiency.

In the following sections, we first describe how to compute the interprocedural sets, assuming the intraprocedural sets have been computed. We then describe how to compute the intraprocedural sets using the interprocedural sets. The descriptions which follow are based on the algorithm presented in Figure 5. These descriptions would require no modifications to incorporate the enhancements described above.

3.2 Interprocedural Phase

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

(Step 1) : $Entry_p = \bigcup_c ForwBind_c(HoldsBefore_c)$, where call site c calls p,

(Step 2) : $CSGen_p^c = BackBind_c(PGen_q)$, where call site c in p calls q,

(Step 3) : $CSGen_p = \bigcup_c CSGen_p^c$, for each call site c is in p.

$Entry_p$ is computed by propagating information along all call sites to p. $ForwBind_c$ 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 $Holds_q$. $BackBind_c$, which is similar to $ForwBind_c$, 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 [CBC93].)

3.3 Intraprocedural Phase

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

$$\begin{aligned} PGen_p &= CSGen_p \cup IGen_p && (1) \\ Holds_p &= Entry_p \cup PGen_p && (2) \\ IGen_p &= \bigcup_s IGen_s(Holds_p), \text{where statement } s \text{ in } p \text{ is a pointer assignment} && (3) \end{aligned}$$

The cyclic dependences among the three equations necessitate 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. We treat each procedure as a single (potentially large) loop consisting of all the pointer-assignment statements of the procedure in some arbitrary order, which can be handled by simple iterative methods. 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.

In [CBC93], 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 3 can be easily derived from TF_s. With flow-insensitive analysis, $Holds_p$ is the alias information assumed to hold at each place in p, including immediately before and after s. Thus, when a fixed point is reached, the following equation is satisfied by $Holds_p$:

$$Holds_p = TF_s(Holds_p), \text{for all } s \text{ in } p. \quad (4)$$

```
p()
{
        S_1:    x = &y;

        S_2:    q();
        ⋮
        S_n:    x = &z;
}
```

Flow-Insensitive Solution

$Entry_p = \{<*x, a_1>, \ldots, <*x, a_n>\}$

$\{<*x, y>, <*x, z>\}$

Figure 6: An Example Illustrating the Usefulness of $ForwardKill$

3.4 Using Kill Information

Consider Figure 6, where procedure p calls procedure q at S_2. By Step 1, the alias relations that are propagated to $Entry_q$ from S_2 are those that hold at S_2. From our discussion above, this is $Holds_p$, which is: $\{<*x, a_1>, \ldots, <*x, a_n>, <*x, y>, <*x, z>\}$.

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 [CBC93]. We adopt this approach by computing for each call site c, $ForwardKill_c$, which contains any alias relation that is killed along all paths from the entry point of p to the call site c. We use $ForwardKill_c$ to remove alias relations from $Entry_p$ and then factor in any aliases that are generated via $PGen_p$, for correctness. In our example, $\{<*x, y>, <*x, z>\}$ is propagated to q, but not $\{<*x, a_1>, \ldots, <*x, a_n>\}$.

The computation of $ForwardKill_c$ can be performed before the interprocedural propagation (before S_5 of Figure 5), and used when needed.[6] This allows the interprocedural algorithm to remain flow-insensitive, as no control flow information is required during the propagation.

This same 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 via $PGen_p$. To improve the precision of this set, we define the *Backward Kill* set for a call site c in p, ($BackwardKill_c$) to be all 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.

Like $ForwardKill_c$, $BackwardKill_c$ can be computed as a preprocessing step before the flow-insensitive iteration over the PCG is begun. In [CBC93] we introduce *killing criteria* which can

[6]We only kill a pointer variable when it appears on the left side of an assignment statement or in a `malloc` or `free` function call. Killing a pointer variable via indirection, i.e. $*x = \ldots$ cannot be precomputed since it requires the alias information of $*x$.

$$
\begin{array}{ll}
S_1: & \mathsf{p} = \&\mathsf{q}; \\
S_2: & \mathsf{p} = \underbrace{*\ldots*}_{r}\mathsf{q};\ r \geq 0 \\
S_3: & \underbrace{*\ldots*}_{l}\mathsf{p} = \underbrace{*\ldots*}_{r}\mathsf{q};\ l \geq 1, r \geq -1
\end{array}
$$

Figure 7: Three Types of Pointer-Assignment Statements

be employed when computing these sets. These criteria are based on the amount of control flow information that is used when computing a kill set.

4 Deferred Evaluation

Section 3.3 describes how to improve intraprocedural efficiency by treating a procedure as a single (potentially large) loop consisting of all the statements of the procedure in some arbitrary order. We further improve the efficiency of the intraprocedural phase by first identifying certain types of pointer assignments and then *deferring* the computation of their effects on alias information until a fixed point in aliasing is reached. As an example, consider statement S_1 of Figure 7. Since the alias relations generated by statements of type S_1 can be computed independently of the alias relations that hold before executing S_1,[7] these types of statements need not be included in the iteration. Statements of this form are processed first[8] since they only contribute to the $IGen$ set and do not depend on it.

Statements of type S_2 and S_3 do require information about the alias relations that hold to determine what new alias relations are generated after these statements are executed. In both types of statements the aliases of $\underbrace{*\ldots*}_{r}\mathsf{q}$ are required[9], and in statement S_3 the aliases of $\underbrace{*\ldots*}_{l}\mathsf{p}$ are required. Thus, it appears that both types of statements must be included in our intraprocedural iteration.

However, our algorithm defers the evaluation of statements of type S_2, by postponing evaluation of the aliases of $\underbrace{*\ldots*}_{r}\mathsf{q}$ until after the intraprocedural iteration has completed. This is done by creating a *deferred alias relation* between p and q, with a *deferred value* of r. This signifies that $*p$ will be aliased to all objects that q can reach via a path of length $r+1$ over the alias relations. We refer to this approach as *deferred evaluation.* Thus, only statements of type S_3 are needed in the intraprocedural iteration. If the procedure does not contain any statements of this type, no

[7] This is not the case with *exhaustive representations* [LR92], which, assuming $<*s, p>$ holds at S_1, would include $<**s, q>$ as holding immediately before S_2. With our *compact representation* [CBC93], factoring in this is also deferred until a fixed point is reached. Factoring in the reference parameter aliases, if any, of q is also deferred.

[8] In fact, they can be treated like $ForwardKill$ and $BackwardKill$ information, in that they can be precomputed prior to *interprocedural* iteration.

[9] Note that &q is represented by "$r = -1$".

intraprocedural iteration is required. Deferred alias relations can be expanded on demand into full alias relations without a loss of precision.

5 Related Work

Flow-sensitive interprocedural algorithms for pointer-induced aliases are given in [LR92, CBC93]. They utilize kill information and can provide improved precision over a flow-insensitive analysis, which in general does not utilize kill information. Our flow-insensitive algorithm also utilizes, in a restricted way, kill information to improve precision. Complete discussions on the interactions between reference parameters and pointer-induced aliasing are also given in [CBC93].

A framework for flow-insensitive analysis of pointer-induced aliasing is described in [CBC93] without algorithmic elaborations. The algorithm described in this paper is based on that framework and is, to our knowledge, the first interprocedural flow-insensitive algorithm that efficiently computes pointer-induced aliasing correctly, without being overly conservative as in [Wei80].

The compact representation of Choi, Burke, and Carini [CBC93] for alias relations is used in this work. In addition to the obvious benefit of being smaller than the exhaustive representation [LR92], the compact representation also allows the technique of deferred evaluation to be used. However, each representation has advantages and disadvantages over the other [MLR+93].

Larus gives a flow-insensitive algorithm to compute aliases in LISP programs [Lar89]. This algorithm uses alias graphs, which serve as both values propagated to solve data-flow equations and as representations of statements' effects on propagated values, a property shared with the $Holds_p$ set of equation 4. He uses the fastness closure algorithm of Graham and Wegman [GW76] to correctly process the alias graphs.

6 Conclusions

In this paper, we have presented the first *flow-insensitive* algorithm that efficiently and correctly computes interprocedural pointer-induced aliases, and can give comparable information to flow-sensitive analyses. This is achieved by (1) making use of certain types of *killing* information that can be computed efficiently, and (2) computing aliases generated in each procedure instead of 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. By employing deferred evaluation, we are able to identify a particular class of pointer-assignment statements that can be excluded from the intraprocedural iteration.

Our algorithm has been implemented in a Fortran 90 research prototype. As Fortran 90 programs become available we plan to measure the efficiency and precision of our algorithm.

References

[ABC+87] 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.

[ACK87] 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.

[Ban79] John Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. *Conference Record of the Sixth ACM Symposium on Principles of Programming Languages*, pages 29–41, January 1979.

[CBC93] 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.

[CF94] Jong-Deok Choi and Jeanne Ferrante. Static slicing in the presence of GOTO statements. *ACM Transactions on Programming Languages and Systems*, 1994. To appear.

[CK89] Keith D. Cooper and Ken Kennedy. Fast interprocedural alias analysis. *Sixteenth ACM Symposium on Principles of Programming Languages*, pages 49–59., January 11-13 1989. Austin, Texas.

[GGL89] 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.

[GW76] 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.

[KMT91] 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.

[KU76] John B. Kam and Jeffrey D. Ullman. Global data flow analysis and iterative algorithms. *JACM*, 23,1:158–171, January 1976.

[Lan92] William Landi. *Interprocedural Aliasing in the Presence of Pointers.* PhD thesis, Rutgers, the State University of New Jersey, 1992.

[Lar89] James Richard Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, University of California, 1989. Technical Report No. UCB/CSD 89/502.

[LH88] 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.

[LR92] 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.

[MLR+93] 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.

[MRB94] T.J. Marlowe, B.G. Ryder, and M.G. Burke. Defining flow-sensitivity in data flow problems. *In Preparation*, 1994.

[Wei80] 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.