Flow-Sensitive Interprocedural Pointer Induced Alias Analysis *

Lin-Pheng Ni and Nai-Wei Lin
Department of Computer Science and Information Engineering
National Chung Cheng University
MinHsiung, Chiayi 621, Taiwan, ROC

Abstract

Optimizing and parallelizing compilers depend on ambitious data flow analyses and optimizations to generate efficient code. The precision of alias information is crucial to the precision of ambitious data flow analyses and the effectiveness of optimizations, especially for languages with pointer data types. This paper describes a flow-sensitive interprocedural analysis algorithm for the precise computation of aliases induced via pointers. The algorithm improves the precision of the analysis by achieving higher flow-sensitivity. We have implemented a prototype of the analysis for C language and have applied the alias information in interprocedural reaching definition analysis and interprocedural constant propagation. A set of preliminary experiments has been performed for these analyses and optimization.

1 Introduction

Two object names are called *aliases* of each other if they refer to the same memory location. An *intraprocedural alias analysis* computes all possible aliases within a procedure. An *interprocedural alias analysis* computes all possible aliases in a program using information passed across procedure boundaries. A *flow-sensitive* interprocedural analysis will interleave intraprocedural and interprocedural analyses. More specifically, the control flow information is used for each application of intraprocedural analysis. This paper will describe a flow-sensitive interprocedural alias analysis algorithm that improve the precision of the analysis by achieving higher flow-sensitivity, namely, by maintaining more control flow information.

Much work on alias analysis has been done for Fortran language [1, 2, 4, 5, 9]. In Fortran programs, aliases are induced mainly by the binding of reference parameters. In languages such as C, aliases may also be induced by pointers. These pointer induced aliases make the alias analysis much more complicated. Work on pointer induced alias analysis has be described only very recently [8, 3, 6]. This paper will describe a flow-sensitive interprocedural analysis algorithm for the precise computation of aliases induced via pointers.

*This work was supported in part by the National Science Council of ROC under grant number NSC-86-2213-E-194-005.

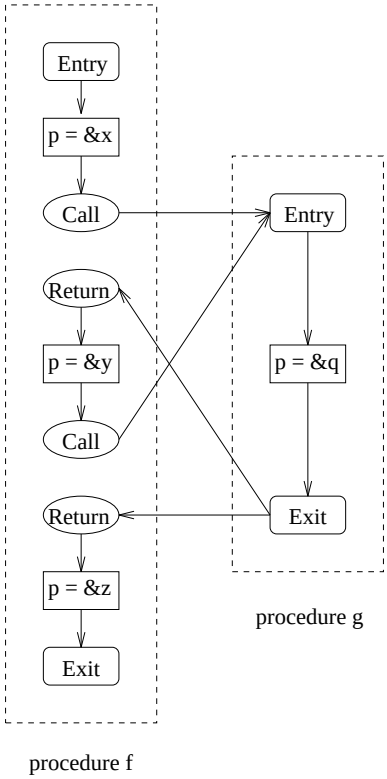

Figure 1: An example of an ICFG

2 Preliminaries

This section summaries the preliminary definitions and notions used in this paper.

2.1 Interprocedural Control Flow Graph

We will represent each program by a directed graph called *Interprocedural Control Flow Graph* (ICFG) [7]. An ICFG is the collection of control flow graphs for the procedures in the program augmented by edges specifying the flow of control between the procedures. Each procedure invocation statement, or call site, in the program is represented by two nodes in the ICFG: a *call node* and a *return node*. For each call site, there is an edge from its call node to the unique *entry node* of the invocated procedure, and there is an edge from the unique *exit node* of the invocated procedure to its return node. To facilitate the manipulation of pointer induced alias information, each pointer assignment in the program is represented by a dedicated node, called *pointer-assign node*. An example of an ICFG is shown in Figure 1.

Every path in the control flow graph of a procedure represents a possible execution sequence of the procedure in a program execution. But this is not true in the case of ICFG. For example, in Figure 1, there are two incoming edges at the entry node of g and two outgoing edges at the exit node of g. The path coming into the entry node of g from the upper call node and going out of the exit node of g to the lower return node cannot appear in any execution of the program. If a path going through the body of g wants to be a possible execution sequence,

it must come in from the call node and go out to the return node of the same call site. A path satisfying the condition above is called a *realizable path*. We will only consider realizable paths in our analysis.

2.2 Alias Relations

We use object names to refer to the memory locations where data objects are stored. An object name is a variable name with a (possibly empty) sequence of dereferences and field selections. Two object names are *aliases* of each other at a program point if they refer to the same memory location at that point. In this paper each alias relation between objects names p and q is represented by an unordered pair, $\langle p, q \rangle$, of these object names, called an *alias pair*. To improve the precision of the analysis, we will associate each alias relation with four attributes: QN, GN, CS, and AS, and denote an alias relation between p and q as a five-tuple $[\langle p, q \rangle, QN, GN, CS, AS]$.

The first attribute QN quantifies the alias relation as a may or must alias relation. An alias relation is called a *may alias* relation if it holds for some execution of the program. An alias relation is called a *must alias* relation if it holds for any execution of the program. The information about may alias relations is useful for constructing reaching definitions, dependence analysis, etc., while the information about must alias relations is useful for constant propagation, register allocation, etc. Moreover, the information about must alias relations can be used to improve the precision of the analysis for may alias relations. Therefore, we will compute both may and must alias relations.

The second attribute GN specifies the node at which the alias relation is generated. This information can help us to improve the precision of intraprocedural analysis because it provides more precise control flow information. The third attribute CS records the call site from which the alias relation is passed into. This information is useful to distinguish alias relations passed from distinct call sites. For an alias relation generated independently within a procedure, its CS attribute would have the value `null`. The fourth attribute SA gives the source alias set from which the alias relation are induced. The alias relations in the source alias set are passed from the same call site. They will be used to probably pass alias relations back to the call site. The usage of these four attributes will become more clear in the later discussion.

An alias relation can induce other alias relations if the object names in the relation can be dereferenced or be field selected. For example, if the type of one of the object names in the relation is a structure type, then the new object names consisting of the original object names and a field selection of the structure type are still aliases of each other. However, if the structure type is recursively defined, the number of such induced alias relations is potentially infinite. We will limit the number of this induction step to some constant, k, called k-limiting [8]. Using k-limiting scheme in an alias relation, an object name with $l > k$ field selections would be approximately represented by the object name with k field selections. For example, if $k = 1$, $p\texttt{->}f1\texttt{->}f2$ would be represented by $p\texttt{->}f1$.

We now introduce three useful functions for manipulating object names.

1. *IsPrefix*(o_1, o_2): This function will return true iff object name o_1 can be transformed into o_2 by a (possibly empty) sequence of dereferences and field selections.

2. *IsProperPrefix*(o_1, o_2): This function will return true iff object name o_1 can be transformed into o_2 by a nonempty sequence of dereferences and field selections.

3. $ApplyTrans(o_1, o_2, o_3)$: This function requires $IsPrefix(o_1, o_2)$ to be true; it will apply to o_3 the sequence of dereferences and field selections necessary to transform o_1 into o_2 and return the resulted object name.

3 Iterative Data Flow Analysis

This section gives an overview of our interprocedural alias analysis. We use iterative data flow analysis to compute alias relations in the program. For each node n in the ICFG, we compute the set of *input alias relations*, In_n, that hold at the entry of n, and the set of *output alias relations*, Out_n, that hold at the exit of n. The effect of a node n on the set of alias relations is denoted by a *transfer function*, $Trans_n$, namely,

$$Out_n = Trans_n(In_n). \tag{1}$$

The transfer functions for various nodes will be described in more detail in the following two sections. For a node n with multiple predecessors, its set of input alias relations is obtained by taking the union of the sets of output alias relations of its predecessors, namely,

$$In_n = \bigcup_{p \in pred(n)} Out_p, \tag{2}$$

where $pred(n)$ denotes the set of predecessors of node n. Notice that a must alias relation a in Out_p of a predecessor p will remain a must alias relation in In_n only if it is in Out_p for every predecessor p of n. If a is not in Out_p for every predecessor p of n, it will become a may alias relation in In_n.

The sets of input and output alias relations for each node in the ICFG are computed iteratively until a fixed point is reached. The computation is devided into two phases: an *initialization phase* and an *iteration phase*. In the initialization phase, the sets of input and output alias relations for each node in the ICFG are first initilized to be the empty set. We then compute the sets of input and output alias relations for each node by traversing the nodes in reverse depth-first order. For each node, we first compute the set of input alias relations using formula (2). If this set is changed, (that is, if it is not the empty set) we then compute the set of output alias relations using formula (1). If this set is changed, (again, if it is not the empty set) we put all its successors into a work-list. The nodes in the work-list will be considered again later since their sets of input alias relations may change. In the iteration phase, we recompute the sets of alias relations for the nodes in the work-list. This computation will continue until the work-list becomes empty. When the computation terminates, the sets of input and output alias relations for each node in the ICFG reach a fixed point. We now consider the transfer functions for various nodes in more detail.

4 Intraprocedural Analysis

Within a procedure, only a pointer assignment may have effects on the sets of alias relations. Therefore, we will only consider the transfer function for a pointer-assign node n. The pointer assignment denoted by n may generate some new alias relations, $Gen_n(In_n)$, may kill some input alias relations, $Kill_n(In_n)$, and may change some input

$$
\begin{aligned}
Gen_n(In_n) \;=\; \{\; [AP, QN, n, CS, SA] \mid\; & [\langle p, a\rangle, QN_L, GN_L, CS_L, SA_L] \in In_n, \\
& [\langle y, z\rangle, QN_R, GN_R, CS_R, SA_R] \in In_n, \\
& IsProperPrefix(q, y) = true, \\
& AP = \langle ApplyTrans(q, y, a), z\rangle, \\
& QN = QuantifierOp(QN_L, QN_R), \\
& OnSamePath(GN_L, GN_R) = true, \\
& (CallSiteOp(CS_L, CS_R) = CS) \neq \bot, \\
& SA = SA_L \cup SA_R \;\}
\end{aligned}
$$

Figure 2: Definition of the function Gen_n

must alias relations into may alias relations, $Change_n(In_n)$. Thus the transfer function for a pointer-assign node n can be expressed as

$$Trans_n(In_n) = Gen_n(In_n) \;\cup\; (Change_n(In_n) \perp Kill_n(In_n)). \tag{3}$$

The rest of this section will consider the functions Gen_n, $Kill_n$, and $Change_n$ in turn.

Let's consider the function Gen_n first. Consider the following pointer assignment, '$p = q$', where p and q are object names. After the execution of this assignment, p will point to the memory location where q originally pointed to before the execution. Hence, this assignment will generate a new alias relation between $*p$ and $*q$, namely, $[\langle *p, *q\rangle, \texttt{must}]$. In addition, this assignment may also generate new alias relations between object names aliased to p and object names aliased to q.

The definition for the function $Gen_n(In_n)$ is given in Figure 2. We now illustrate how a pointer assignment may generate new alias relations in general via Figure 3. Assume that an object name a is aliased to p before the execution; namely, $[\langle p, a\rangle, \cdot]$ is in In_n. This alias relation is referred to an *lhs condition alias relation*. Assume also that an object name z is aliased to the object name y to which q can be transformed by applying a nonempty sequence of dereferences and field selections; namely, $[\langle y, z\rangle, \cdot]$ is in In_n and $IsProperPrefix(q, y)$ is true. This alias relation is referred to a *rhs condition alias relation*. Then z will be aliased to $ApplyTrans(q, y, a)$ after the execution. We say alias relations $[\langle p, a\rangle, \cdot]$ and $[\langle y, z\rangle, \cdot]$ induce the alias relation $[\langle ApplyTrans(q, y, a), z\rangle, \cdot]$ through the pointer assignment '$p = q$'. As an example, the alias relation $[\langle *p, *q\rangle, \texttt{must}]$ is induced by the trivial alias relations $[\langle p, p\rangle, \texttt{must}]$ and $[\langle *q, *q\rangle, \texttt{must}]$. Whether the induced alias relation is a must or may alias relation depends on the quantifiers in both the lhs and rhs condition alias relations. The induced alias relation is a must alias relation only if both condition alias relations are must alias relation. We will use the function *QuantifierOp* to compute the quantifier for the induced alias relation. The operation table for the function *QuantifierOp* is given in Table 1.

We should not use two alias relations that cannot hold at the same time to induce a spurious alias relation. Two alias relations cannot hold at the same time if they are not generated on the same realizable path in the ICFG. These alias relations may appear in the same set due to the union of sets at a join node in the ICFG.

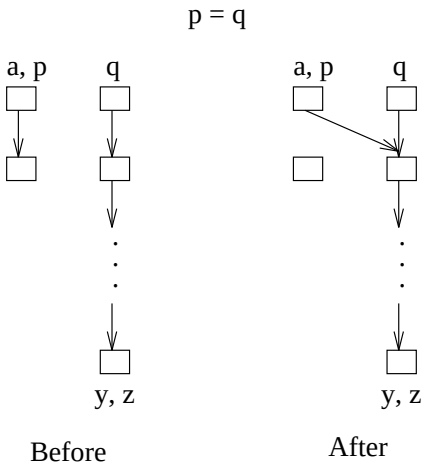

Figure 3: Generation of alias relations via a pointer assignment

QuantifierOp	must	may
must	must	may
may	may	may

Table 1: The operation table for the function *QuantifierOp*

In an alias relation, the information about generating node is used to distinguish alias relations generated at different nodes. We will use the function $OnSamePath(a, b)$ to check if two nodes a and b are on the same path within a procedure. Hence, alias relations generated at nodes a and b can induce a new alias relation only if $OnSamePath(a, b)$ is true. If an alias relation is generated outside a procedure and passed into the procedure, it will be regarded as being generated at the entry node of the procedure. On the other hand, in an alias relation, the information about call site is used to distinguish alias relations passed from different call sites. We will use the function $CallSiteOp(a, b)$ to check if two call sites a and b are conformable. To ensure that we only consider realizable paths, only alias relations induced by alias relations from the same call site can be passed back to that call site. Hence, alias relations passed from call sites a and b can induce a new alias relation only if $CallSiteOp(a, b) \neq \perp$. The operation table for the function *CallSiteOp* is given in Table 2.

Let's consider the function $Kill_n$ next. Consider again the pointer assignment, '$p = q$'. After the execution of this assignment, p will point to the memory location where q pointed to before the execution. Hence, this assignment will kill the input alias relation between $*p$ and an object name x that refer to the memory location

CallSiteOp	null	C_1
null	null	C_1
C_2	C_2	C_1 / $\perp$

Table 2: The operation table for the function *CallSiteOp*

$$Kill_n(In_n) = \{\ [\langle x, z\rangle, \cdot, \cdot, CS, \cdot] \mid [\langle x, z\rangle, \cdot, \cdot, CS, \cdot] \in In_n,\ [\langle p, a\rangle, \texttt{must}, \cdot, CS_L, \cdot] \in In_n,$$
$$IsProperPrefix(a, x) = true,$$
$$CS_L = \texttt{null} \text{ or } CS_L = CS\ \}$$

Figure 4: Definition of the function $Kill_n$

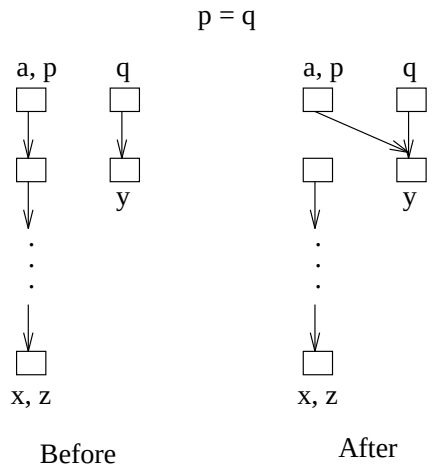

Figure 5: Killing of alias relations via a pointer assignment

where p pointed to before the execution, namely, $[\langle *p, x\rangle, \cdot]$. Moreover, this assignment may also kill input alias relations between object names aliased to p and object names aliased to x.

The definition of the function $Kill_n(In_n)$ is given in Figure 4. We now illustrate how a pointer assignment may kill input alias relations via Figure 5. Assume that an object name a is *must*-aliased to p before the execution; that is, $[\langle p, a\rangle, \texttt{must}]$ is in In_n. Assume also that an object name z is aliased to the object name x to which a can be transformed by applying a nonempty sequence of dereferences and field selections; that is, $[\langle x, z\rangle, \cdot]$ is in In_n and $IsProperPrefix(a, x)$ is true. Then the alias relation $[\langle x, z\rangle, \cdot]$ will be killed after the execution. We say the alias relation $[\langle p, a\rangle, \texttt{must}]$ kills the alias relation $[\langle x, z\rangle, \cdot]$ through the pointer assignment 'p = q'. As an example, the alias relation $[\langle *p, x\rangle, \cdot]$ is killed by the trivial alias relation $[\langle p, p\rangle, \texttt{must}]$.

For the sake of safety, we must be conservative in the computation of kill-information. This means that the set of killed alias relations we computed must be a subset of the set of really killed alias relations. In fact, a must alias relation generated independently within a procedure can kill alias relations passed from any call site. However, a must alias relation passed from a call site can only kill alias relations passed from the same call site, not even alias relations generated independently within the procedure. Furthermore, since a must alias relation must hold on every path from the entry node of the procedure to node n, we can omit the application of the function *OnSamePath*.

Finally, let's consider the function $Change_n$. Consider the pointer assignment, '$p = q$', where p is in the form of $*a$. After the execution of this assignment, $*a$ will point to the memory location where q pointed to before

The set of must alias relations that need to be changed:

$$ChangeMust_n(In_n) = \{\ [< x, z >, \texttt{must}, \cdot, CS, \cdot] \mid [< x, z >, \texttt{must}, \cdot, CS, \cdot] \in In_n,$$
$$[< p, a >, \texttt{may}, \cdot, CS_L, \cdot] \in In_n,$$
$$IsProperPrefix(a, x) = true,$$
$$CS = \texttt{null} \text{ or } CS_L = \texttt{null} \text{ or } CS_L = CS\ \}$$

The set of changed may alias relations:

$$ChangeMay_n(In_n) = \{\ [\cdot, \texttt{may}, \cdot, \cdot, \cdot] \mid [\cdot, \texttt{must}, \cdot, \cdot, \cdot] \in ChangeMust_n(In_n)\ \}$$

The set of changed input alias relations:

$$Change_n(In_n) = (In_n \perp ChangeMust_n(In_n)) \cup ChangeMay_n(In_n).$$

Figure 6: Definition of the function $Change_n$

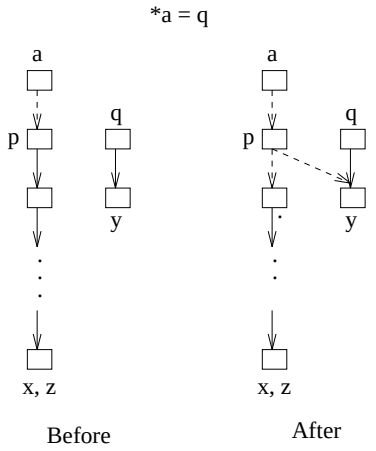

Figure 7: Change of must alias relations into may alias relations via a pointer assignment

the execution. If $*a$ is must-aliased to p, then p will also point to the memory location where q pointed to before the execution. Otherwise, p may still point to the memory location where it pointed to before the execution. In the latter case, however, if $*p$ is must-aliased to an object name x, this assignment will change this must alias relation into a may alias relation. Moreover, this assignment may also change input must alias relations between object names aliased to p and object names aliased to x into may alias relations.

The definition of the function $Change_n$ is given in Figure 6. We now illustrate how a pointer assignment may change input must alias relations into may alias relations via Figure 7. Assume that an object name $*a$ is *may*-aliased to p before the execution; that is, $[\langle *a, p\rangle, \texttt{may}]$ is in In_n. Assume also that an object name z is *must*-aliased to the object name x to which p can be transformed by applying a nonempty sequence of dereferences and field selections; that is, $[\langle x, z\rangle, \texttt{must}]$ is in In_n and $IsProperPrefix(p, x)$ is true. Then the alias relation $[\langle x, z\rangle, \texttt{must}]$ will be changed into $[\langle x, z\rangle, \texttt{may}]$ after the execution. We say the alias relation $[\langle *a, p\rangle, \texttt{may}]$ changes the alias relation $[\langle x, z\rangle, \texttt{must}]$ into $[\langle x, z\rangle, \texttt{may}]$ through the pointer assignment '*a = q'. As an example, the alias relation

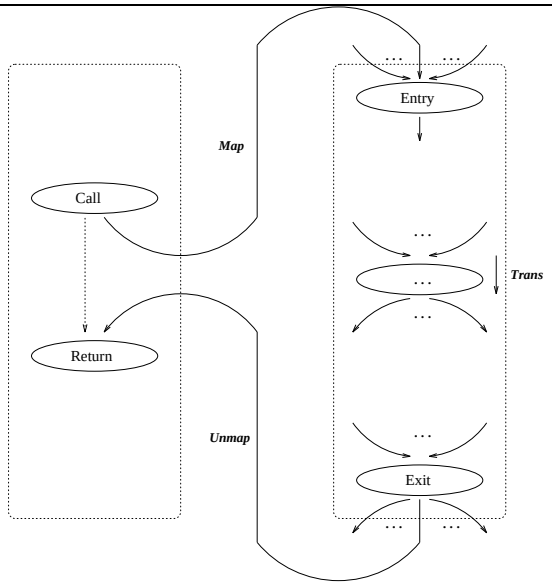

Figure 8: The roles of the functions *Map* and *Unmap*.

$[\langle *p, x\rangle, \texttt{must}]$ is changed into $[\langle *p, x\rangle, \texttt{may}]$ by the alias relation $[\langle *a, p\rangle, \texttt{may}]$. Notice that if $*a$ is must-aliased to p before the execution, then $[\langle *p, x\rangle, \texttt{must}]$ will be killed after the execution as discussed above.

For the sake of safety, we also must be conservative in the computation of change-information. This means that the set of changed alias relations we computed must be a superset of the set of really changed alias relations. It is true that a may alias relation passed from a call site can change a must alias relation passed from the same call site. In addition, a may alias relation generated independently within a procedure can change a must alias relation passed from any call site, and a may alias relation passed from any call site can change a must alias relation generated independently within a procedure. Furthermore, since a must alias relation must hold on every path from the entry node of the procedure to node n, we can also omit the application of the function *OnSamePath* here.

5 Interprocedural Analysis

Across a procedure boundaries, a procedure invocation statement may have effects on the set of alias relations. The effects may be performed at the entry and exit points of a procedure invocation. At the entry point, the effects caused by the parameter passing mechanism of the procedure invocation can be simulated by a sequence of assignments of actual parameters to formal parameters. Hence, the discussion in the previous section can be applied here. In addition to the effects caused by parameter passing, the action at the entry point is done through a function *Map*, and the action at the exit point is done through a function *Unmap*. The roles of functions *Map* and *Unmap* are shown in Figure 8. Function *Map* acts on the set of alias relations passed from the call node of the call site to the entry node of the invocated procedure. Hence, function Map_n denotes the transfer function for an entry node n. Similarly, function *Unmap* acts on the set of alias relations passed from the exit node of the invocated procedure to the return node of the call site. Thus, function $Unmap_n$ denotes the transfer function for

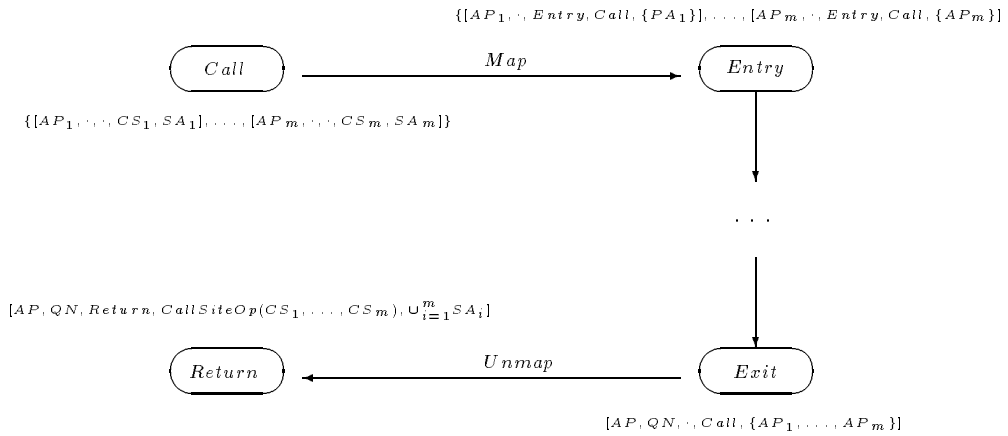

Figure 9: The actions of the functions *Map* and *Unmap*

a return node n. The transfer function for call nodes and exit nodes is the identity function $I(S) = S$.

One of the main functionalities of functions *Map* and *Unmap* is to ensure that we only consider realizable paths in the ICFG. Therefore, these two functions need to maintain suitable information about call site and source alias set for each alias relation passed into and out of the invocated procedure. This functionality is illustrated in Figure 9. We first describe how the function *Map* achieves this functionality. Consider an edge from a call node c to an entry node n in the ICFG. In order to maintain suitable information about call site and source alias set, each of the alias relations in Out_c needs to be transformed before it can be passed into In_n. For each alias relation

$$[AP, QN, GN, CS, SA]$$

in Out_c, we transform it into an alias relation

$$[AP, QN, n, c, \{AP\}].$$

This alias relation says that it is passed from the call site c and it will be regarded as generated at the entry node n within the procedure. The source alias set $\{AP\}$ will be used in the function *Unmap* and will be elaborated shortly. Once In_n is obtained, we apply the transfer function of a pointer assignment to each pointer assignment in the sequence of assignments of actual parameters to formal parameters.

We next describe how the function *Unmap* achieves the functionality mentioned above. Consider an edge from an exit node x to a return node r in the ICFG. Function *Unmap* passes the alias relations in Out_x into In_r according to the information about call site and source alias set. Only alias relations induced by alias relations from the same call site as r or alias relations generated independently inside the invocated procedure are passed into In_r. In order to maintain suitable information about call site and source alias set, each of the alias relations in Out_x needs to be transformed before it is passed into In_r. For each alias relation,

$$[AP, QN, GN, c, \{AP_1, \ldots, AP_m\}],$$

benchmark	without analysis	with analysis	execution time
allroots	720	233	0.90s
fixoutput	1565	988	0.28s
lex315	8654	3786	22.57s
loader	24488	24059	2028.67s
simulator	75875	39799	35656.60s
assembler	93127	58565	74393.61s

Table 3: Experimental results of alias analysis

induced by alias relations from the same call site as r, we will transform it as follows. Assume the alias relations in Out_c that induce the alias pairs $AP_1, \ldots, PA_m$ in the source alias set are, respectively, $[AP_1, \cdot, \cdot, CS_1, SA_1]$, ..., $[AP_m, \cdot, \cdot, CS_m, SA_m]$. Then the transformed alias relation would be

$$[AP, QN, r, CallSiteOp(CS_1, \ldots, CS_m), \bigcup_{i=1}^{m} SA_i].$$

For each alias relation,

$$[AP, QN, GN, \texttt{null}, \{\}],$$

induced by alias relations generated independently inside the invocated procedure, we will transform it into

$$[AP, QN, r, \texttt{null}, \{\}].$$

6 Experimental Results

We have implemented a prototype of our analysis algorithm for the language C and have done an experiment on a suit of benchmarks. We compute the total number of output alias relations at each node of the ICFG for each benchmark. The results of the experiment is listed in Table 3. The second column gives the number of alias relations computed without (or with the most conservative) alias analysis. The third column gives the number of alias relations computed with our alias analysis. The fourth column gives the execution time of the computation using our alias analysis. On the average, the number of alias relations computed is reduced about 40%. This is a significant reduction. However, the execution time required is also very long. The most costly portions are the manipulation of attributes source alias set and generating node.

We also apply the alias information to improve the result of an interprocedural reaching definition analysis and an interprocedural constant propagation. For the interprocedural reaching definition analysis, we compute the average number of reaching definitions at each node of the ICFG. The result for the interprocedural reaching definition analysis is listed in Table 4. The second column gives the number of reaching definitions computed without (or with the most conservative) alias information. The third column gives the number of reaching

program	without alias	with alias
bcmp	4	2
fixoutput	50	40
jacobi	11	11
pattern	8	8
random	67	17
strings	14	9
weather	17	7

Table 4: Experimental results of interprocedural reaching definition analysis

program	without alias	with alias
bcmp	3	3
fixoutput	56	61
jacobi	40	53
pattern	6	6
random	30	38
weather	8	15

Table 5: Experimental results of interprocedural constant propagation

definitions computed with the alias information computed using our alias analysis. On the average, the average number of reaching definitions computed is reduced about 30%.

For the interprocedural constant propagation, we compute the total number of replaced variables in each benchmark. The result for the interprocedural constant propagation is listed in Table 5. The second column gives the total number of replaced variables computed without (or with the most conservative) alias information. The third column gives the total number of replaced variables computed with the alias information computed using our alias analysis. On the average, the total number of replaced variables computed is increased about 25%.

7 Related Work

Our analysis algorithm is most relevant to the three algorithms proposed by Landi and Ryder [8], by Choi, Burke, and Carini [3], and by Emami, Ghiya, and Hendren [6]. A comparison based on the attributes used in an alias relation between these algorithms is given in Table 6.

One of the main differences between these three algorithms is the program representation used in the analysis. Landi and Ryder use an interprocedural control flow graph to represent a program. Thus, both intraprocedural and interprocedural analyses are performed on the ICFG. Choi et al. use control flow graphs to represent procedures and a call graph to represent the interprocedural flow of control. Hence, the interprocedural analysis

	Alias Relation	Must Alias	Source Alias Set	Call Site	Generating Node
LR92	alias pair	no	yes , size $\leq$ 1	no	no
CBC93	alias pair	no	yes	yes	no
EGH94	points-to relation	yes	no	no	no
NL97	alias pair	yes	yes	yes	yes

Table 6: A comparison on the attributes used in an alias relation

is performed on the call graph. Emami et al. use control flow graphs to represent procedures and an invocation graph to represent all the possible procedure invocation instances. Because of this, each procedure invocation instance is analyzed separately. Therefore, they don't need the attributes call site and source alias set associated with an alias relation in their analysis. Choi et al. employ the call site and source alias set attributes to constrain their analysis to consider only realizable paths. Landi and Ryder employ only the source alias set of size one to constrain their analysis to consider only realizable paths. We use the ICFG to represent a program and employ both the call site and source alias set attributes to constrain our analysis to consider only realizable paths.

Another major difference between these three algorithms is the representation of alias relations. Landi and Ryder use exhaustive alias pairs to represent alias relations. Choi et al. use transitive reduction of exhaustive alias pairs as a compact representation of alias relations. Emami et al. actually compute points-to relations instead of alias relations. However, alias relations can be easily computed from points-to relations. Conceptually, points-to relations are rather similar to transitive reduction of exhaustive alias pairs. We use exhaustive alias pairs in our analysis. Landi and Ryder, and Chio et al. do not compute must alias relations or use must alias relations to improve the precision of the analysis, although it is not difficult to incorporate these into their algorithms. In addition, we use the control flow information about the generating node of an alias relation to improve the precision of the intraprocedural analysis.

8 Conclusions

We have described an algorithm for interprocedural pointer induced alias analysis for the language C. This algorithm uses the control flow information about the generating node of an alias relation to improve the precision of the analysis. We have implemented a prototype of the algorithm. We have also applied the alias information to improve the result of an interprocedural reaching definition analysis and an interprocedural constant propagation algorithm.

References

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

[2] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724-736,1978.

[3] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM Symposium on Principles of Programming Languages*, pages 232-245, January 1993.

[4] K. Cooper and K. Kennedy. Fast interprocedural alias analysis. In *Conference Record of the Sixteenth Annual ACM Symposium on Principles of Programming Languages*, pages 49-59, January 1989.

[5] K. Cooper. Analyzing aliases of reference formal parameters. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 281-290, January 1985.

[6] Maryam Emami, Rakesh Ghiya, and Laurie J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the ACM SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242-256, June 1994.

[7] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93-103, January 1991.

[8] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of the ACM SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235-248, June 1992.

[9] E. M. Myers. A precise interprocedural data flow algorithm. In *Conference Record of the Eighth Annual ACM Symposium on Principles of Programming Languages*, pages 219-230, January 1981.