Laboratory of Computer Science Research Technical Report LCSR-TR-259, March 1996

Program Decomposition for Pointer-induced Aliasing Analysis*

Sean Zhang Barbara G. Ryder
Department of Computer Science
Rutgers University
New Brunswick, NJ 08903

William Landi
Siemens Corporate Research Inc
755 College Rd. East
Princeton, NJ 08540
email: {xxzhang,ryder}@cs.rutgers.edu, blandi@scr.siemens.com

Abstract

For compile-time pointer aliasing analysis, a program written in the C language can be considered as a sequence of pointer-related assignments. In this paper, we present a technique that decomposes these assignments into unrelated sets in terms of their effects on pointer-induced aliasing. This decomposition will allow different pointer aliasing analysis methods to be applied to individual sets of assignments so that end users of pointer aliasing information can get the efficiency/precision tradeoff desirable for their applications. We show the feasibility of this approach by using both a flow-sensitive and a flow-insensitive aliasing analysis algorithm on a same program. We use the aliasing solutions of the resulting analysis to resolve locations modified or referenced through names containing pointer dereferences (thru-deref MOD/REF); we empirically show that for a number of programs, the resulting analysis is much faster than the *complete* flow-sensitive analysis and yields a thru-deref MOD/REF solution of similar precision.

1 Introduction

Compile-time analysis of pointer-induced aliases is critical for optimization, parallelization, program transformation and many other applications. Over the past few years, many techniques for the analysis have been proposed in the literature [1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 16, 19, 22, 24, 26, 28, 30]. Some of them are more appropriate for aliases involving accesses to heap locations (i.e., heap-based aliases), e.g., [4, 6, 12]. Some are more appropriate for aliases involving accesses to stack locations (i.e., stack-based aliases), e.g., [7, 8]. Others handle both in a similar fashion, e.g., [5, 16, 22, 30]. It has been proposed in [8] that completely different analysis methods have to be considered for stack-based and heap-based aliases. However, it is not clear how different approaches can be combined in a reasonable way to solve the problem for programs that may have both kinds of aliases.

In this paper, we present a program decomposition technique for C programs that enables the combination of different pointer aliasing analysis methods. Basically, for the purpose of compile-time aliasing analysis, a program is considered as a sequence of assignments that have effects on pointer aliasing. We call these assignments *pointer-related assignments*, each of which consists of

*This research was supported, in part, by NSF grants CCR-92-08632, CCR-95-01761 and GER-90-23628.

two object names. From these assignments, we calculate an equivalence relation on object names such that each pointer-related assignment is associated with an equivalence class of the relation. The equivalence classes of the relation can be represented as a labeled, directed multi-graph, where equivalence classes are nodes and prefix relation between object names are edges between nodes. The weakly connected components in the graph decompose the pointer-related assignments into independent sets so that assignments in different sets will not interact with each other in terms of their effects on pointer aliasing. Therefore, different aliasing analysis methods can be applied to individual sets of assignments of the decomposition.

We show the feasibility of combining different aliasing analysis techniques by an analysis that, for a program, uses a flow-insensitive[1] aliasing algorithm for pointer-related assignments involving recursive data structures and uses a flow-sensitive aliasing algorithm for other pointer-related assignments. We use the the aliasing solutions of the resulting analysis to determine locations modified or referenced through names with pointer dereferences (thru-deref MOD/REF). We empirically show that for a number of programs, the analysis is much faster than the *complete* flow-sensitive analysis and yet yields a thru-deref MOD/REF solution of similar precision.

The paper is organized as follows. Section 2 is about our program representation; Section 3 is about the sets and relations of object names in which we are interested. In Section 4 and 5, we present the definition and calculation of the PE relation for program decomposition and the FA relation as flow-insensitive alias information respectively. In Section 6, we show the empirical results of our program decomposition technique and of our experiment of applying both a flow-sensitive and a flow-insensitive aliasing analysis to a same program. We conclude in Section 8 with some future work that we have in mind. In Appendix A, we have a complete example of the PE and the FA relation for a simple program; we also give the aliasing and thru-deref MOD/REF solutions for the program.

2 Program Representation

We consider C programs that do not have type casting, except in calls to system-defined memory allocation routines such as **malloc**() and **calloc**(). Call-by-value parameter passing is assumed. We represent a program in an intermediate form, whose syntax is given in Figure 1. Basically a program consists of a number of procedures. One of these procedures is **main**() and another is a special procedure **_init_**(), which initializes all global variables and calls **main**(). A procedure is a sequence of statements; the first one has to be the entry and the last the exit. Call and return statements are used to represent procedure calls. There are a number of kinds of assignment statements including non-pointer assignment with a basic arithmetic or relational operation, three kinds of pointer assignments, structure assignment with both sides being of structure types. Because of our assumption that there is no type casting, both sides of a pointer assignment or a structure assignment have the same type. Other statements allowed are: heap allocation, heap deallocation, conditional goto and goto. Statements have IDs associated with them, where an ID is in the range of 1..(# of statements). Conditional and goto statements use statement IDs for their destinations. This is a quadruple representation and it can be depicted in a graphical form such as the ICFG[16].

[1] In this paper, we will use flow-sensitive and flow-insensitive to mean algorithm-flow-sensitive and algorithm-flow-insensitive[20].

Program ::=	(Procedure)$^+$	
Procedure ::=	(Statement)$^+$	
Statement ::=	entry of P(FmlName_1,...,FmlName_m)	(procedure entry)
	| exit of P	(procedure exit)
	| call P(ArgName_1,...,ArgName_m)	(call statement)
	| return from P	(return statement)
	| PrmName = PrmName_1 Op PrmName_2	(non pointer assignment)
	| PtrName = NULL	(pointer assignment with NULL RHS)
	| PtrName = &Name	(pointer assignment with & RHS)
	| PtrName = PtrName_1	(other pointer assignment)
	| StrtName = StrtName_1	(structure assignment)
	| HeapAlloc(PtrName)	(heap allocation)
	| HeapDealloc(PtrName)	(heap deallocation)
	| if (PrmName) (goto L_1) (goto L_2)	(conditional goto)
	| goto L	(goto statement)

FmlName_1, ..., FmlName_m: formal variable names
PrmName, PrmName_1, PrmName_2: object names of primitive types
PtrName, PtrName_1: object names of pointer types
StrtName, StrtName_1: object names of structure types
Name, ArgName_1, ..., ArgName_m: object names of any type
L, L_1, L_2: IDs for statement
Op: primitive operators (e.g., arithmetic, relational)
NULL: nil pointer

Figure 1: Intermediate Representation

Object names, which are used extensively in the representation, will be the topic of the next section.

3 Object Names

In the intermediate representation, we refer to memory locations and addresses of these locations through what we call **object names**. An object name for a memory location starts with either a variable name or a heap name, followed by a sequence of applications of structure field accesses (.field) or pointer dereferences ($*$). Variable names are declared in the source program; heap names are created explicitly for locations dynamically allocated in the program and they are of the form $heap_{id}$, where id is the statement ID of the corresponding heap allocation. Statically, some object names always refer to the same memory locations (e.g., p, x.g) and others can refer to different

ObjAux ::=	VarName	(variable name)
	| HeapName	(heap name)
	| ObjAux.field	(structure field)
	| $*$ObjAux	(pointer dereference)
ObjName ::=	ObjAux	
	| &ObjAux	(address operator)

FixedLocName ::= VarName | HeapName | FixedLocName.field

precedence: $* >$ .field $>$ &

associativity:	$*$	right-associative	(i.e., $**$p $=$ $*(*$p))
	.field	left-associative	(i.e., s.f_1.f_2 = (s.f_1).f_2)

Figure 2: Object Names and Fixed Location Names

locations (e.g., $*$p, p$\rightarrow$g) depending on program execution[2]. We call the former **fixed location names**. The address operator (&) can be applied to object names to get addresses of memory locations. Object names without & can be used as either *l-values* or *r-values* in a program, but object names with & can only be used as *r-values*. The syntax of object names and fixed location names is given in Figure 2. Note that $p \rightarrow f$ is same as $(*p).f$ in C.

Each object name has a type associated with it so that we can talk about object names of pointer types or structure types. Only well-typed object names will be considered, that is, a field access can only be applied to a name of structure type with that field name and $*$ can only be applied to a name of pointer type. The address operator can be applied to any name. We assume each structure field in a program has a unique name; you can think of each field name associated with the structure type that it belongs to.

We call structure field accesses (.field) and pointer dereference ($*$) **accessors**. We define three useful functions in Figure 3. Given an object name and an accessor, *apply* returns the object name obtained after application of the accessor; *apply** applies a sequence of accessors to an object name and returns the resulting name. The function NumOfDerefs($a_1 a_2 ... a_n$) is defined to be the number of pointer dereferences ($*$) in the sequence of accessors $a_1 a_2 ... a_n$. For example, we have:

apply(&(p$\rightarrow$f), $*$) = p$\rightarrow$f
apply($*$p,f) = ($*$p).f

apply*(p, $*$ .f) = ($*$p).f

NumOfDerefs($*$ .f) = 1

[2] One of the purposes of pointer aliasing analysis is to determine the locations to which these names may refer.

$$
\begin{aligned}
&apply(o, *\,) &&= *o \\
&apply(\&o, *\,) &&= o \\
&apply(o,field) &&= o.field
\end{aligned}
$$

$$
\begin{aligned}
&apply^*(o,\epsilon) = o \\
&apply^*(o,a_1 a_2 ... a_n) = \quad apply^*(apply(o,a_1),a_2 ... a_n)
\end{aligned}
$$

NumOfDerefs($a_1 a_2 ... a_n$) = the number of $*$ in the sequence of accessors, $a_1 a_2 ... a_n$.

Figure 3: Definition of *apply*, *apply**and NumOfDerefs

We now define sets of object names and relations on sets of object names, in which we are interested.

Definition 3.1 A set of object names, S, is *closed with respect to prefixes* if the following are true:

- If $o \in$ S and o is of the form $\&o_1$, then $o_1 \in$ S.
- If $o \in$ S and o is of the form $o_1.field$, then $o_1 \in$ S.
- If $o \in$ S and o is of the form $*o_1$, then $o_1 \in$ S.

□

For example, the set { p→g, &x } is not closed with respect to prefixes. The set { p, *p, p→g, &x, x } is.

Definition 3.2 A relation R on a set of object names is said to be *type consistent* if for any (o_1,o_2) $\in$ R, o_1 and o_2 have the same type.
□

We will be interested in type consistent relations on sets of object names closed with respect to prefixes.

Definition 3.3 Let S be a set of object names closed with respect to prefixes and R be a relation on S. R is a *weakly right-regular* relation on S if the following are true:

- If $(o_1,o_2) \in$ R, both $o_1^{'} = apply(o_1, *\,)$ and $o_2^{'} = apply(o_2, *\,)$ are in S, then $(o_1^{'},o_2^{'}) \in$ R.
- If $(o_1,o_2) \in$ R, both $o_1^{'} = apply(o_1,field)$ and $o_2^{'} = apply(o_2,field)$ are in S, then $(o_1^{'},o_2^{'}) \in$ R.

□

For example, let S be the set { p, *p, p→g, &x, x, x.g }. The relation { (p,&x) } on S is not weakly right-regular because (*p,x) and (p→g,x.g) are not in S. The relation { (p,&x), (*p,x), (p→g,x.g) } is a weakly right-regular relation on S.

Lemma 3.1 Let S be a set of object names closed with respect to prefixes and R be a weakly right-regular relation on S. If there are a tuple $(o_1,o_2) \in$ R and a sequence of accessors $A = a_1a_2...a_n$ $(n \geq 1)$ such that both $o_1^{'} = apply^*(o_1,A)$ and $o_2^{'} = apply^*(o_2,A)$, are in S, then $(o_1^{'},o_2^{'}) \in$ R.
□

We give a brief proof here.

Consider any prefix of A, $a_1a_2...a_j$, where $1 \leq j \leq n$. Because both $o_1^{'}$ and $o_2^{'}$ are in S, and S is closed with respect to prefixes, it can be proved by induction on j that the object names, $apply^*(o_1,a_1a_2...a_j)$ and $apply^*(o_2,a_1a_2...a_j)$, are in S. Because R is a weakly right-regular relation on S, it can be proved by induction on j that $(apply^*(o_1,a_1a_2...a_j),apply^*(o_2,a_1a_2...a_j)) \in$ R.
□

Definition 3.4 Let S be a set of object names closed with respect to prefixes and R be a relation on S. R^e is the smallest[3] equivalence relation on S containing R.
□

In another words, R^e is the reflexive, symmetric and transitive closure of R.

Definition 3.5 Let S be a set of object names closed with respect to prefixes and R be a relation on S. R^{wr} is the smallest weakly right-regular equivalence relation on S containing R.
□

If R is type consistent, both R^e and R^{wr} are type consistent.

Since both R^e and R^{wr} are equivalence relations, they can be represented as sets of equivalence classes. For example, let S be the set { p, ∗p, p→g, &x, x, x.g } and R = { (p,&x) } be a relation on S. R^e consists of five equivalence classes:

{ p, &x }
{ ∗p }
{ p→g }
{ x }
{ x.g }

And R^{wr} consists of three equivalence classes:

{ p, &x }
{ ∗p, x }
{ p→g, x.g }

4 The PE relation

4.1 Definition of the PE relation

First, we define the concept of **pointer-related assignment** in a program; a statement will be of interest for compile-time aliasing analysis if it contains one or more pointer-related assignments.

[3] By *smallest*, we mean the least number of tuples.

```
struct st {
   int *f;
   int g;
} x, *p, *tt;

main()
{
    int z, u, w, i, *r, *y, **q;
    z = 0;
    p = &x;
    p→f = &z;
    p→g = 0;
    tt = p;

    q = &y;
    *q = &w;
    r = &u;
    *r = 1;
    *q = r;
    i = *(p→f) + **q;
}
```

Figure 4: An Example Program

Definition 4.1.1 A pointer-related assignment in a program is one of the following:

- a pointer assignment
- a structure assignment such that the structure type contains pointer fields
- a formal-actual pair at a call statement such that the two are either of pointer type or of structure type with pointer fields
- a heap allocation, HeapAlloc(p), which has the same effect as the pointer assignment: p = &$heap_{id}$, where id is the statement ID of the heap allocation.
- a heap deallocation, HeapDealloc(p), which, besides the deallocation, has the same effect as the pointer assignment: p = NULL[4].

□

Throughout this section, we will use the program in Figure 4 as an example. The following are the pointer-related assignments in the program:

p = &x
p→f = &z
tt = p
q = &y
*q = &w

[4]This is only true for legal C programs.

r = &u
$*$q = r

For the purpose of aliasing analysis, a program can be considered as a sequence of pointer-related assignments of the form: $lhs = rhs$, where lhs is an object name and rhs is either an object name or NULL.

Definition 4.1.2 The set of object names used in a program, B_0, is defined inductively below:

- If an object name o syntactically appears anywhere in the program[5], then $o \in B_0$.
- If $o \in B_0$ is of structure type, for each field $field$ of the structure, $apply(o,field) \in B_0$.
- If $o \in B_0$ and o is of the form $\&o_1$, then $o_1 \in B_0$.
- If $o \in B_0$ and o is of the form $o_1.field$, then $o_1 \in B_0$.
- If $o \in B_0$ and o is of the form $*o_1$, then $o_1 \in B_0$.

□

By definition, B_0 is closed with respect to prefixes.

For the example program,

B_0 = { p, $*$p, p→f, $*$(p→f), p→g, &x, x, x.f, x.g, tt, q, $*$q, $**$q, &y, y, r, $*$r, &z, z, &w, w, &u, u }

Definition 4.1.3 Given a program and the set B_0 for the program, let R_0 be a relation on B_0 defined below:

$$\mathrm{R}_0 = \left\{ \; (lhs \; , \; rhs) \;\middle|\; \begin{array}{l} lhs = rhs \text{ is a pointer-related assignment} \\ \text{in the program and } rhs \text{ is not NULL} \end{array} \right\}$$

We call ${\mathrm{R}_0}^{wr}$ the PE (Pointer-related-assignment-induced-Equality) relation.
□

Because we only consider programs without type casting, the relation R_0 in the above definition is type consistent. So the PE relation is also type consistent.

For the example program, the PE relation have eight equivalence classes, shown in Figure 5. For each pointer-related assignment in the program, $lhs = rhs$, lhs and rhs (if it is not NULL) are in the same equivalence class of the PE relation. So each pointer-related assignment can be considered associated with a unique equivalence class of the PE relation. In Figure 5, we show the set of pointer-related assignments in the example program for each equivalence class of the PE relation.

[5]Heap names are considered appearing in dynamic allocation statements.

equivalence classes:	pointer-related assignments:
{ p, tt, &x }	{ p = &x, tt = p }
{ ∗p, x }	{ }
{ p→f, x.f, &z }	{ p→f = &z }
{ p→g, x.g }	{ }
{ ∗(p→f), z }	{ }
{ q, &y }	{ q = &y }
{ ∗q, y, r, &w, &u }	{ ∗q = &w, r = &u, ∗q = r }
{ w, u }	{ }

Figure 5: The PE Relation for the Example Program

4.2 Calculation of the PE relation

We assume the following routines are available for initializing and maintaining equivalence classes:

- INIT-EQUIV-CLASS(o), where o is an object name, initializes an equivalence class with one object name, o.
- FIND(o), where o is an object name, returns the equivalence class for o.
- UNION(e_1,e_2), where e_1 and e_2 are two equivalence classes, returns an equivalence class that consists of the object names in both e_1 and e_2.

We further assume that the cost of each call to INIT-EQUIV-CLASS() is a constant time and the cost of each call to FIND() or UNION() depends on the data structure chosen to represent equivalence classes.

Assuming the set B_0 is available, the algorithm calculating the PE relation is given in Figure 6. The algorithm has two phases. In Phase 1, an equivalence class is created for each object name in B_0. Each class maintains a prefix relation between the object names in itself and those in other classes. The initial prefix relation for an equivalence class with object name o, is one of the following cases:

- PREFIX(FIND(o)) = $\{ \; (\mathit{field} \; , \mathit{apply}(o,\mathit{field}) \mid \mathit{apply}(o,\mathit{field}) \in B_0 \; \}$
- PREFIX(FIND(o)) = $\{ \; (* \; , \mathit{apply}(o, * \;) \mid \mathit{apply}(o, * \;) \in B_0 \; \}$
- PREFIX(FIND(o)) = { }, if neither $\mathit{apply}(o, * \;) \in B_0$ nor $\mathit{apply}(o,\mathit{field}) \in B_0$.

Intuitively, if there is a tuple $(a \; , \, o) \in$ PREFIX(e), where a is an accessor, o is an object name and e is an equivalence class, then there are an object name o_1 in e and an object name o_2 in FIND(o) such that $o_2 = \mathit{apply}(o_1,a)$. If equivalence classes are represented by nodes in a graph, a tuple $(a \; , \, o) \in$ PREFIX(e) represents an edge from the node for e to the node for FIND(o).

In Phase 2, we go through all pointer-related assignments in the program and union equivalence classes. When two classes are unioned, their prefix relations are examined; any two equivalence classes having the same prefix relation with the two classes will be unioned; this is done by the recursive calls to the MERGE() routine. This makes sure the weakly right-regular property is satisfied. For example, suppose equivalence class e_1 and e_2 with the following prefix relations, are unioned.

$$\text{PREFIX}(e_1) = \{\ (a_1,b_1)\ ,\ (a_2,b_2)\ \}$$
$$\text{PREFIX}(e_2) = \{\ (a_1,c_1)\ ,\ (a_3,c_3)\ \}$$

As a result, the two equivalence classes, FIND(b_1) and FIND(c_1), will be unioned and a new equivalence class e is created such that e has the following prefix relation and consists of object names in both e_1 and e_2.

$$\text{PREFIX}(e) = \{\ (a_1,b_1)\ ,\ (a_2,b_2)\ ,\ (a_3,c_3)\ \}$$

The algorithm can be thought as a graph algorithm, in which equivalence classes are nodes of a graph and tuples in the PREFIX relations of equivalence classes are edges of the graph. When two equivalence classes are unioned, their nodes are replaced by a new node; all incoming edges to the two nodes and all outing edges from the two nodes become incoming edges and outgoing edges of the new node respectively.

4.3 Complexity

We assume the program is well-typed (i.e., there is no casting except in calls to memory allocation routines); so all object names in an equivalence class will have the same type. We also assume that the maximum number of fields for any structure type is a small constant compared to the number of object names in the program. By this assumption, the size of the PREFIX relation for any equivalence class is a small constant.

Let N_0 be number of object names used in the program, that is, $N_0 = |\ B_0|$.

Given a program, the set B_0 is computed by going through each statement in the intermediate representation of the program and examining object names appearing in the statement. This takes time linear in the size of the intermediate representation and the number of object names in B_0.

Phase 1 of the calculation of the PE relation will take $\mathcal{O}(N_0)$ time. The cost of Phase 2 is dominated by calls to routine MERGE(). Below we will estimate the number of calls to UNION() and FIND() in Phase 2. Besides the recursive calls to itself, each call of MERGE() will incur one call to UNION(), a constant number of calls to FIND() by our assumption that the number of fields of any structure type is a small constant, and a constant cost for other operations in MERGE(). Since each call to UNION() will reduce the number of equivalence classes by one and there are initially N_0 classes, there are no more than N_0 calls to UNION() in Phase 2. Therefore there are no more than N_0 calls to MERGE() and the number of calls to FIND() will be $\mathcal{O}(N_0)$. If we use a fast union/find algorithm such as the one in [27], the cost of all calls to UNION() and FIND() in Phase 2 is $\mathcal{O}(N_0 \times \alpha(N_0,N_0))$, where α is the inverse of the Ackermann's function; the cost of calls to MERGE() in Phase 2 is $(\mathcal{O}(N_0 \times \alpha(N_0,N_0)) + \mathcal{O}(N_0))$. The complexity of the algorithm is $\mathcal{O}(N_0 \times \alpha(N_0,N_0))$.

```
calculate-PE-relation()
{
   /* Phase 1 */
   for each o ∈ B0
   {
      INIT-EQUIV-CLASS(o);
      PREFIX(FIND(o)) = { };
   }
   for each o ∈ B0
   {
      if (o == &o1)
        add ( * , o1) to PREFIX(FIND(o));
      else if (o == *o1)
        add ( * , o) to PREFIX(FIND(o1));
      else if (o == o1.field)
        add (field , o) to PREFIX(FIND(o1));
   }
   /* Phase 2 */
   for each pointer-related assignment, lhs = rhs, where rhs ≠ NULL
      if (FIND(lhs) ≠ FIND(rhs))
        MERGE(FIND(lhs) , FIND(rhs));
}

MERGE(e1,e2)
{
     e = UNION(e1 , e2); /* union the two classes */
     /* calculate the new prefix relation */
     new-prefix = PREFIX(e1);
     for each (a,o) ∈ PREFIX(e2)
         if there is (a1,o1) ∈ new-prefix such that a == a1†
         {
           if (FIND(o) ≠ FIND(o1))
              MERGE(FIND(o) , FIND(o1));
         }
         else
           new-prefix = new-prefix  ∪ { (a,o) };
     PREFIX(e) = new-prefix; /* set the prefix relation */
}
```

†That is, the two accessors are either $*$ or a same field name. We assume each structure field has a unique name.

Figure 6: Calculation of the PE relation

```
struct s_1 { int i , *p; };

struct s_2 { struct s_1     f_21 , f_22 };

struct s_3 { struct s_2     f_31 , f_32 };

...

struct s_n { struct s_n-1   f_n,1 , f_n,2 } x;
```

Figure 7: Exponential Number of Object Names

Note that the number of object names used in a program could be exponential in the maximum nesting depth of structures in the program. For instance, the program segment in Figure 7 will cause exponential number of object names to be created. Although this rarely, if ever, happens in real programs, we are investigating ways to eliminate this exponential worst-case effect.

4.4 Program Decomposition

The result of the algorithm in Figure 6 can be thought as a labeled, directed multi-graph, where nodes are equivalence classes and edges are represented by tuples in the PREFIX relations for equivalence classes; specifically, if there is a tuple (a,o) in PREFIX(e), where a is an accessor, o is an object name and e is an equivalence class, then there is an edge from the node for e to the node for FIND(o), labeled with the accessor a. We call the graph $G_{\rm PE}$. In Figure 8, we show the $G_{\rm PE}$ for the example program, where each node is annotated with the set of object names in the equivalence class that it represents.

Since the PE relation is type consistent, object names in each equivalence class of the relation have the same type. In $G_{\rm PE}$, a node for an equivalence class with object names of a pointer type may [6] have an outgoing edge annotated with $*$; a node for an equivalence class with object names of a structure type has a number of outgoing edges, each of which is annotated with a field name of the structure type.

$G_{\rm PE}$ can be decomposed into weakly connected components. For instance, the $G_{\rm PE}$ for the example program shown in Figure 8 has two weakly connected components. There are two sets that are unique to each component:

- a set of pointer-related assignments, which is the union of the assignments for the nodes in the component, and
- a set of object names, which is the union of the object names for the nodes in the component.

[6] There may not be such an edge. For example, assume that a variable name v in a program is of pointer type and is in an equivalence class by itself. If the name $*v$ is never used in the program and $*v$ is not aliased to any fixed location, then there is no outgoing from the node for v, annotated with $*$.

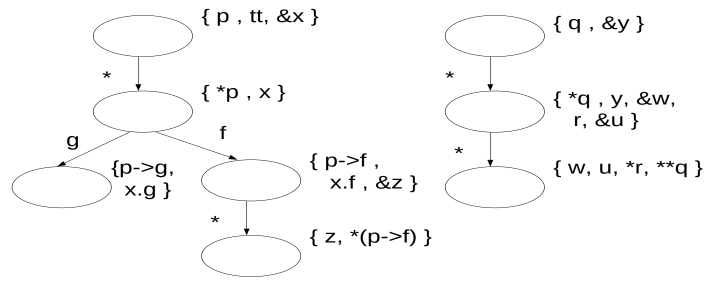


Figure 8: $G_{\rm PE}$ for the example program

The pointer-related assignments for each weakly connected component will only create run-time aliases that involve variable names and heap names in the set of object names affiliated with the component. In another words, the sets of pointer-related assignments for weakly connected components in $G_{\rm PE}$, are independent of each other in terms of their aliasing effects. Therefore, they constitute a program decomposition for pointer aliasing analysis.

To formally show that these weakly connected components are a decomposition for pointer aliasing analysis, we will define a new relation called in next section. The relation can be calculated separately for each weakly connected component of $G_{\rm PE}$, and can be proved to be a safe estimate of the run-time aliases.

5 The FA relation

5.1 Definition of the FA relation

The following are the notations that will be used in this section.

- B_1 is the subset of B_0 that excludes any object name with &, that is,

  $$B_1 = B_0 - \{ \ o \mid o \in B_0 \text{ and } o \text{ is of the form } \&o_1 \ \}$$

  B_1 is closed with respect to prefixes.

- $B_{\rm PE}(n)$ is the set of object names associated with a node n in $G_{\rm PE}$.

  We have $B_{\rm PE}(n) \subseteq B_0$. Any object name in B_1 is in some $B_{\rm PE}(n)$, where n is a node in $G_{\rm PE}$.

- $(B_{\rm PE}(n) \cap B_1)$ is the is the set of object names associated with a node n in $G_{\rm PE}$, which do not contain &.

We assume that a path in $G_{\rm PE}$ consisting of only one node is annotated with an empty sequence of accessors, ϵ. First, We define a set of object names based on paths in $G_{\rm PE}$.

Definition 5.1.1 Given the G_{PE} for a program, B is the set of object names defined below:

$$B = \left\{ o \;\middle|\; \begin{array}{l} \text{there is a path from a node } n \text{ to a node } m \text{ in } G_{\mathrm{PE}} \text{ such that the} \\ \text{path is annotated with a sequence of accessors } a_1 a_2 \ldots a_j \ (j \geq 0) \\ \text{and } o = \mathit{apply}^*(o_1, a_1 a_2 \ldots a_j) \text{, where } o_1 \in (B_{\mathrm{PE}}(n) \cap B_1) \end{array} \right\}$$

□

By definition, B does not contain any object names with & and B is closed with respect to prefixes. Since paths consisting of one node are annotated with ϵ, any object name in $(B_{\mathrm{PE}}(n) \cap B_1)$, where n is a node in G_{PE}, is in B. In another words, $B_1 \subseteq B$.

For the example program in Figure 4, we have:

$$B = \left\{ \begin{array}{l} \mathrm{p}, *\mathrm{p}, \mathrm{p}{\rightarrow}\mathrm{f}, *(\mathrm{p}{\rightarrow}\mathrm{f}), \mathrm{p}{\rightarrow}\mathrm{g}, \mathrm{tt}, *\mathrm{tt}, \mathrm{tt}{\rightarrow}\mathrm{f}, *(\mathrm{tt}{\rightarrow}\mathrm{f}), \mathrm{tt}{\rightarrow}\mathrm{g}, \\ \mathrm{x}, \mathrm{x.f}, *(\mathrm{x.f}), \mathrm{x.g}, \mathrm{q}, *\mathrm{q}, **\mathrm{q}, \mathrm{y}, *\mathrm{y}, \mathrm{r}, *\mathrm{r}, \mathrm{z}, \mathrm{w}, \mathrm{u} \end{array} \right\}$$

Note that some of the names (e.g., $*(\mathrm{tt}{\rightarrow}\mathrm{f})$, $*\mathrm{y}$) in B above are not in the set B_0 for the example program.

By definition, for each object name $o \in B$, there is a path from a node n to a node m in G_{PE} such that the path is annotated with a sequence of accessors A and $o = \mathit{apply}^*(o_1,A)$, where $o_1 \in (B_{\mathrm{PE}}(n) \cap B_1)$. There may be more than one such path in G_{PE}. It is easy to show that all these paths for o will end at the node m in G_{PE}. Here is a sketch of the proof. Suppose there is a path from n_1 annotated with A_1 such that $o = \mathit{apply}^*(o_1,A_1)$, where $o_1 \in (B_{\mathrm{PE}}(n_1) \cap B_1)$ and there is another path from n_2 annotated with A_2 such that $o = \mathit{apply}^*(o_2,A_2)$, where $o_2 \in (B_{\mathrm{PE}}(n_2) \cap B_1)$. Since $\mathit{apply}^*(o_1,A_1) = \mathit{apply}^*(o_2,A_2)$, without loss of generality, we assume $A_1 = A_2'.A_2$; in another word, $o_2 = \mathit{apply}^*(o_1,A_2')$. By the algorithm in Figure 6, there is a path in G_{PE} from n_1 to n_2 such that the path is annotated with A_2'. Therefore, the two paths for o will end at the same node.

Next we define a relation on B.

Definition 5.1.2 Given the graph G_{PE} for a program, let R_1 be a relation on B defined below:

$$\mathrm{R}_1 = \left\{ (o_1',o_2') \;\middle|\; \begin{array}{l} \text{there is a path from a node } n \text{ to a node } m \text{ in } G_{\mathrm{PE}} \text{ such that} \\ \text{the path is annotated with a sequence of accessors } a_1 a_2 \ldots a_j, \\ \mathrm{NumOfDerefs}(a_1 a_2 \ldots a_j) \geq 1,\ o_1' = \mathit{apply}^*(o_1, a_1 a_2 \ldots a_j) \text{ and} \\ o_2' = \mathit{apply}^*(o_2, a_1 a_2 \ldots a_j) \text{, where } o_1 \in B_{\mathrm{PE}}(n) \text{ and } o_2 \in B_{\mathrm{PE}}(n) \end{array} \right\}$$

We call ${\mathrm{R}_1}^e$ the FA (Flow-insensitive Alias) relation.

□

By definition, for each $(o_1',o_2') \in \mathrm{R}_1$, there is a path from a node n in G_{PE} such that the path is annotated with a sequence of accessors A, $\mathrm{NumOfDerefs}(A) \geq 1$, $o_1' = \mathit{apply}^*(o_1,A)$ and $o_2' = \mathit{apply}^*(o_2,A)$, where $o_1 \in B_{\mathrm{PE}}(n)$ and $o_2 \in B_{\mathrm{PE}}(n)$. If o_1 does not contain &, then $o_1' \in B$. If o_1 does contain &, by the algorithm in Figure 6, node n has an outgoing edge labeled with $*$ in G_{PE}; since $\mathrm{NumOfDerefs}(A) \geq 1$, $o_1' \in B$. Similarly, $o_2' \in B$. Therefore, R_1 defines a relation on B.

For the example program in Figure 4, the FA relation has ten equivalence classes:

{ p }
{ tt }
{ ∗p, ∗tt, x }
{ p→f, tt→f, x.f }
{ p→g, tt→g, x.g }
{ ∗(p→f), ∗(tt→f), ∗(x.f), z }

{ q }
{ ∗q, y }
{ r }
{ ∗∗q, ∗y, ∗r, w, u }

Lemma 5.1.1 The FA relation is a weakly right-regular equivalence relation on B.
□

We sketch a proof of the lemma here. Let $(o_1^{'},o_2^{'})$ be a tuple in R_1. By definition, there is a path from a node n to a node m in G_{PE} such that the path is annotated with a sequence of accessors A, NumOfDerefs(A) ≥ 1, $o_1^{'} = apply^*(o_1,A)$ and $o_2^{'} = apply^*(o_2,A)$, where $o_1 \in B_{\mathrm{PE}}(n)$ and $o_2 \in B_{\mathrm{PE}}(n)$.

Suppose the object name $apply(o_1^{'},a) \in B$, where a is an accessor. By definition, there is a path for G_{PE} for $apply(o_1^{'},a)$. Since all paths for $o_1^{'}$ in G_{PE} end at node m, there must be an outgoing edge from node m such that the edge is annotated with a. Therefore, $apply(o_2^{'},a) \in B$; by definition, $(apply(o_1^{'},a),apply(o_2^{'},a)) \in \mathrm{R}_1$.

By the above argument, R_1 is a weakly right-regular relation on B. Furthermore, by the argument, we show that if $(o_1^{'},o_2^{'}) \in \mathrm{R}_1$ and $apply(o_1^{'},a) \in B$, then $apply(o_1^{'},a) \in B$. Note that by definition, R_1 is symmetric; so it is also the case that if $(o_1^{'},o_2^{'}) \in \mathrm{R}_1$ and $apply(o_2^{'},a) \in B$, then $apply(o_1^{'},a) \in B$. Because of these two results, we claim that the ${\mathrm{R}_1}^e$ is a weakly right-regular relation on B. A formal proof will involve induction on the calculation of the reflexive, symmetric and transitive closure of R_1.
□

By definition, the FA relation can be partitioned according to weakly connected components of G_{PE}, that is, there is a subrelation of the FA relation for each component, which is independent of other components in G_{PE}. For instance, for the example program in Figure 4, the FA relation can be partitioned into two subrelations.

We prove in [31] that the FA relation for a program contains the run-time aliases at any program point on an execution path of the program. The subrelation of the FA relation for each weakly connected component of G_{PE}, is a safe estimate of the run-time aliases that can be induced by the pointer-related assignments associated with the component. Therefore, sets of pointer-related assignments affiliated with weakly connected components in G_{PE} form a program decomposition for pointer aliasing analysis.

5.2 Calculation of the FA relation

```
calculate-FA-relation()
{
   for each o ∈ B_1
   {
      INIT-EQUIV-CLASS(o);
      PREFIX(FIND(o)) = { };
   }
   /* Phase 1 */
   for each edge in G_PE from n to m annotated with accessor field
   {
      for each o ∈ (B_PE(n) ∩ B_1)
         add (field, apply(o,field)) to PREFIX(FIND(o));
   }
   /* Phase 2 */
   for each edge in G_PE from n to m annotated with accessor *
   {
      obj-set = { o_1 | o_1 ∈ B_PE(m) and o_1 = apply(o, *), where o ∈ B_PE(n) };
      let o_1 be an arbitrary object name in obj-set
      for each o_2 ∈ obj-set
         if (FIND(o_1) ≠ FIND(o_2))
           MERGE(FIND(o_1),FIND(o_2));
      for each o ∈ (B_PE(n) ∩ B_1)
         if there is not a ( * , o_2) in PREFIX(FIND(o)) such that FIND(o_2) == FIND(o_1)
           add ( * , o_1) to PREFIX(FIND(o))
   }
}
```

Figure 9: Calculation of the FA relation

Since the number of object names in B may be infinite if there is a cycle in $G_{\rm PE}$ (an example of such a $G_{\rm PE}$ is given in Appendix A), we can not always directly calculate the set B. However, by definition of B, names in B may be represented as paths in graphs, and graphs with cycles can represent infinite number of object names. As a matter of fact, $G_{\rm PE}$ is such a graph, where each name in B is represented by one or more paths in the graph.

So the idea is to use a directed graph to represent the FA relation. The graph will have a structure similar to $G_{\rm PE}$, that is, nodes are annotated with a set of object names; edges of the graph are annotated with either $*$ or a field name of a structure type. We would like the following to be true:

(1) Each name in B is represented by one or more paths in the graph.

(2) Each path in the graph represents a set of object names in B.

(3) $(o_1,o_2) \in$ FA *if and only if* the path for o_1 and the path for o_2 end at the same node in the graph.

Given $G_{\rm PE}$, the calculation of the FA relation, shown in Figure 9, is tantamount to the construction of such a graph. The algorithm represents nodes by equivalence classes and edges by tuples in the PREFIX relations of equivalence classes.

Initially, there is one node for each object name in B_1 and there is no edge.

In Phase 1, each edge in $G_{\rm PE}$ annotated with a field name *field*, is examined. Assume the edge is coming out of node n in $G_{\rm PE}$. For each object name $o \in (B_{\rm PE}(n) \cap B_1)$ (i.e., excluding object names with &), an edge is added from the node for FIND(o) to the node for FIND($apply(o,field)$). Note that by the definition of the set B_0, for any object name o of structure type and any field *field* of the type, if $o \in B_0$, then $apply(o,field) \in B_0$.

In Phase 2, each edge in $G_{\rm PE}$ annotated with $*$, is examined. Assume the edge is from node n to node m in $G_{\rm PE}$. First, all the object names $o_1 \in B_{\rm PE}(m)$ such that $o_1 = apply(o,*)$ for some $o \in B_{\rm PE}(n)$, are collected; because of the presence of the edge in $G_{\rm PE}$, there is at least one such object name. Then, nodes for equivalence classes containing these object names are unioned, that is, these names will be in a same equivalence class. Other nodes may also be unioned if they can be reached through same sequences of accessors from the nodes for these names; the MERGE() routine takes care of all these. After the necessary unions, there is a node representing the equivalence class containing all the object names collected; let us call it node x. For each object name $o \in (B_{\rm PE}(n) \cap B_1)$, an edge is added from the node for FIND(o) to the node x.

The result of the calculation, is a labeled, directed multi-graph such that

- Each node in the graph represents an equivalence class; the node is annotated with a set of object names (a subset of B_1) that are in the equivalence class.

  We will use $B_{\rm FA}(x)$ for the set of object names for a node x in $G_{\rm FA}$; $B_{\rm FA}(x) \subseteq B_1$.

- Each edge in the graph from the node for equivalence class e_1 to the node for equivalence class e_2, corresponds to a tuple (a,o) in PREFIX(e_1), where a is an accessor, o is an object name and e_2 = FIND(o); the edge is annotated with the accessor a.

We call the graph $G_{\rm FA}$. In Figure 10, we show the $G_{\rm FA}$ for the example program in Figure 4. The next few lemmas show that $G_{\rm FA}$ is the graph we want. We provide their proofs in [31].

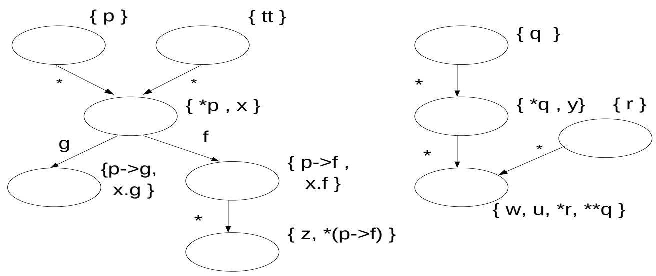


Figure 10: G_{FA} for the example program

Lemma 5.1 For any object name o_1 in B, there is a path from a node x in G_{FA} such that the path is annotated with a sequence of accessors $A = a_1a_2...a_j$ $(j \geq 0)$ and $o_1 = \mathit{apply}^*(o,A)$, where $o \in B_{\mathrm{FA}}(x)$.
□

This lemma corresponds to item (1) on page 17.

Lemma 5.2 For each path from a node x in G_{FA} such that the path is annotated with a sequence of accessors $A = a_1a_2...a_j$ $(j \geq 0)$, and for any object name $o \in B_{\mathrm{FA}}(x)$, the object name $\mathit{apply}^*(o,A)$ is in B.
□

This lemma corresponds to item (2) on page 17.

Lemma 5.3 For each node x in G_{FA}, we define the following set of object names:

$$O_{\mathrm{FA}}(x) = \left\{ o \;\middle|\; \begin{array}{l} \text{there is a path from a node } y \text{ to a node } x \text{ in } G_{\mathrm{FA}} \text{ such that the} \\ \text{path is annotated with a sequence of accessors } a_1a_2...a_j \ (j \geq 0) \\ \text{and } o = \mathit{apply}^*(o_1,a_1a_2...a_j)\text{, where } o_1 \in B_{\mathrm{FA}}(y) \end{array} \right\}$$

These sets constitute a partition of the set B and the equivalence relation induced by these sets is the FA relation.
□

Object names in $O_{\mathrm{FA}}(x)$, where x is a node in G_{FA}, are represented by paths in G_{FA} ending at node x. By this lemma, $(o_1,o_2) \in \mathrm{FA}$ *if and only if* $(o_1,o_2) \in O_{\mathrm{FA}}(x)$, where x is a node in G_{FA}. So this lemma corresponds to item (3) on page 17.

We assume a path in G_{FA} consisting of one node is annotated with ϵ; so by definition of $O_{\mathrm{FA}}(x)$, $B_{\mathrm{FA}}(x) \subseteq O_{\mathrm{FA}}(x)$, for any node x in G_{FA}.

Definition 5.2.1 For each node x in G_{FA}, $B_{\mathrm{FA}}(x)$ is the set of object names associated with x. These sets constitute a partition of the set B_1. We call the equivalence relation induced by these sets *the partial FA relation.*
□

The partial FA relation is the projection of the FA relation on the set B_1. Since the FA relation for a program is a safe estimate of the run-time aliases for the program[31], we can use the partial FA relation as safe alias relation involving *only* object names in B_1. Since we have all fixed location names of the program in B_1, the partial FA relation contains enough information about the fixed locations to which any object name in B_1 with pointer dereferences may be aliased. We will make use of this in our empirical study. For the example program, the partial FA relation has the following equivalence classes:

{ p }
{ tt }
{ ∗p, x }
{ p→f, x.f }
{ p→g, x.g }
{ ∗(p→f), z }

{ q }
{ ∗q, y }
{ r }
{ ∗∗q, ∗r, w, u }

It is worth noting that in the full FA relation for the program shown in Section 5.1, we have an equivalence class { ∗(p→f), ∗(x.f), z }. Here the corresponding equivalence class is { z }. This is the case because the object names, ∗(p→f) and ∗(x.f), are not used in the program; aliases involving any of these names may not be necessary for some applications that utilize aliasing information.

The partial FA relation can be partitioned according to weakly connected components of G_{PE}. The subrelation of the partial FA relation for a weakly connected component in G_{PE}, can be calculated by an algorithm similar to the one shown in Figure 9, which starts with *only* object names for that component (excluding names with &) and examines *only* edges in the component; this subrelation can be used as a safe alias relation for object names associated with the component. The partial FA relation for the example program, can be partitioned into two subrelations.

In our empirical study, we will calculate the subrelations of the partial FA relation for individual connected components and use them as aliasing solutions to resolve fixed locations that are either modified or referenced by object names affiliated with those components.

5.3 Complexity

By a similar argument as the one in Section 4.3, the complexity of the algorithm in Figure 9 is $\mathcal{O}(N_1 \times \alpha(N_1,N_1))$, where N_1 is the number of object names in B_1, that is, $N_1 = |\, B_1|$.

index	program	lines of code	number of statements	number of connected components					number of pointer-related assignments				
				k=1	k=2	2 <k≠ ∞	k=∞	total	k=1	k=2	2<k≠ ∞	k=∞	total
1	allroots	215	420	1	0	0	0	1	14	0	0	0	14
2	fixoutput	401	615	2	0	0	0	2	18	0	0	0	18
3	diffh	268	644	1	1	1	0	3	4	8	86	0	98
4	travel	862	696	0	1	0	0	1	0	44	0	0	44
5	ul	541	1026	5	1	0	0	6	110	11	0	0	121
6	plot2fig	1435	1079	6	1	0	0	7	46	24	0	0	70
7	lex315	719	1300	3	0	0	0	3	30	0	0	0	30
8	loader	1220	1563	9	3	0	2	14	81	73	0	61	215
9	mway	700	1576	14	1	0	0	15	63	7	0	0	70
10	stanford	887	1769	9	4	0	1	14	21	16	0	19	56
11	pokerd	1120	1915	5	1	0	1	7	9	137	0	33	179
12	learn	1461	2622	9	5	0	0	14	104	200	0	0	304
13	xmodem	1705	2686	11	1	0	0	12	53	76	0	0	129
14	compiler	2232	3006	1	3	0	0	4	8	24	0	0	32
15	sim	1422	3019	17	3	0	1	21	146	54	0	32	232
16	assembler	2693	3602	16	2	0	2	20	208	334	0	145	687
17	gnugo	2901	3651	18	0	0	0	18	108	0	0	0	108
18	simulator	3735	5574	12	5	0	4	21	117	132	0	43	292
19	triangle	1925	6117	26	3	2	0	31	90	28	170	0	288
20	football	2222	7313	7	1	1	0	9	306	11	2	0	319

Figure 11: Number of Weakly Connected Components and Pointer-related Assignments

6 Empirical Work

Implementation We have implemented the algorithms that calculate the PE relation and the FA relation. Our implementation is written in C and compiled by *cc* with optimization turned on (*-O2*). The timings are collected on a Sun SPARCStation 10 running SunOS 4.1.3 with 210 megabyte swap space.

The twenty programs we have used are in Figure 11, ordered by the number of statements in our intermediate representation. In the same figure, we give the total numbers of weakly connected components and pointer-related assignments resulted from the program decomposition of each program; the numbers are also broken down by the value of k, which is the maximum number of pointer dereferences ($*$) on any path in a weakly connected component of G_{PE}. For example, the program *ul* has 5 component with $k = 1$ and 1 component with $k = 2$; there are 110 pointer-related assignments for the components with $k = 1$ and 11 for the component with $k = 2$. If k is ∞ for a component, then there is a cycle in the component, which means there may be a recursive data structure. If k is an integer value for a component, there are at most k number of dereferences on any path in the weakly connected component,

The values of k represent certain characteristics of these weakly connected components and the pointer-related assignments associated with them. For instance, if k is 1 for a component, then there are only single-level pointers in the pointer-related assignments affiliated with the component; it is known that the may-aliasing problem for single-level pointers can be solved precisely in polynomial time[15]. If k is ∞ for a component, then the pointer-related assignments for the component are involved with some recursive data structures. The may-aliasing problem for recursive data structures has been proved to be NP-hard[15]; our experiences with the algorithm by Landi and Ryder[16] have shown that it is time-consuming to analyze large programs with recursive data structures.

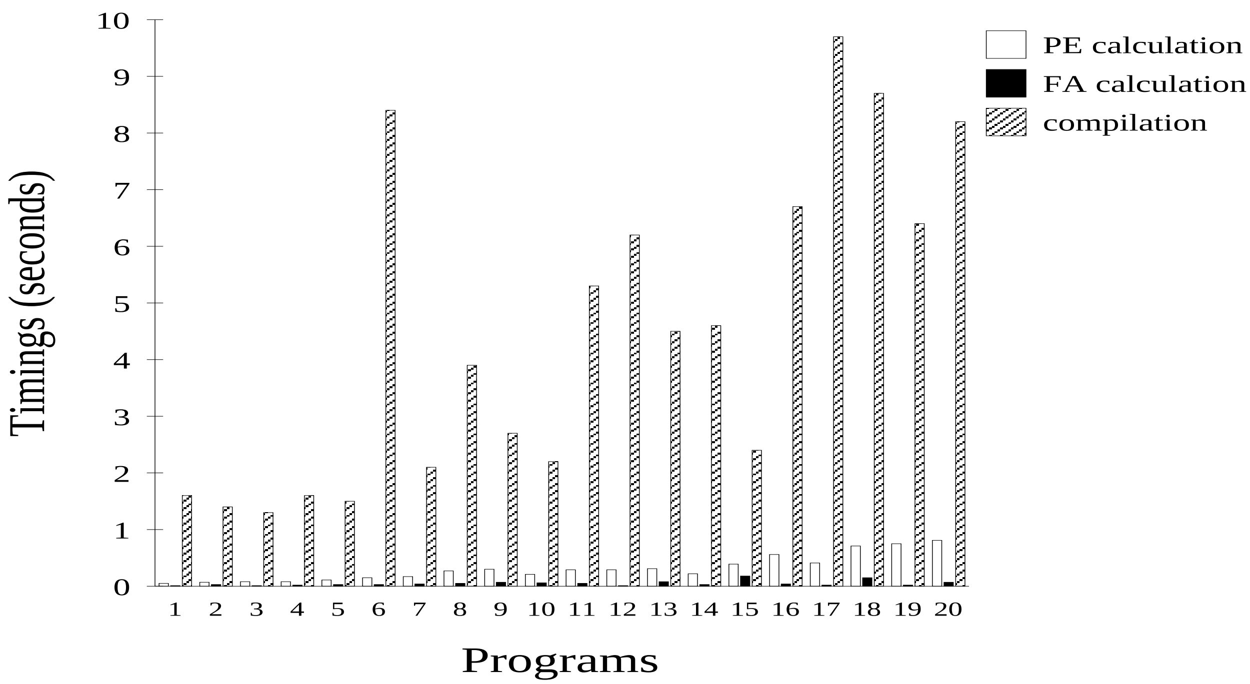
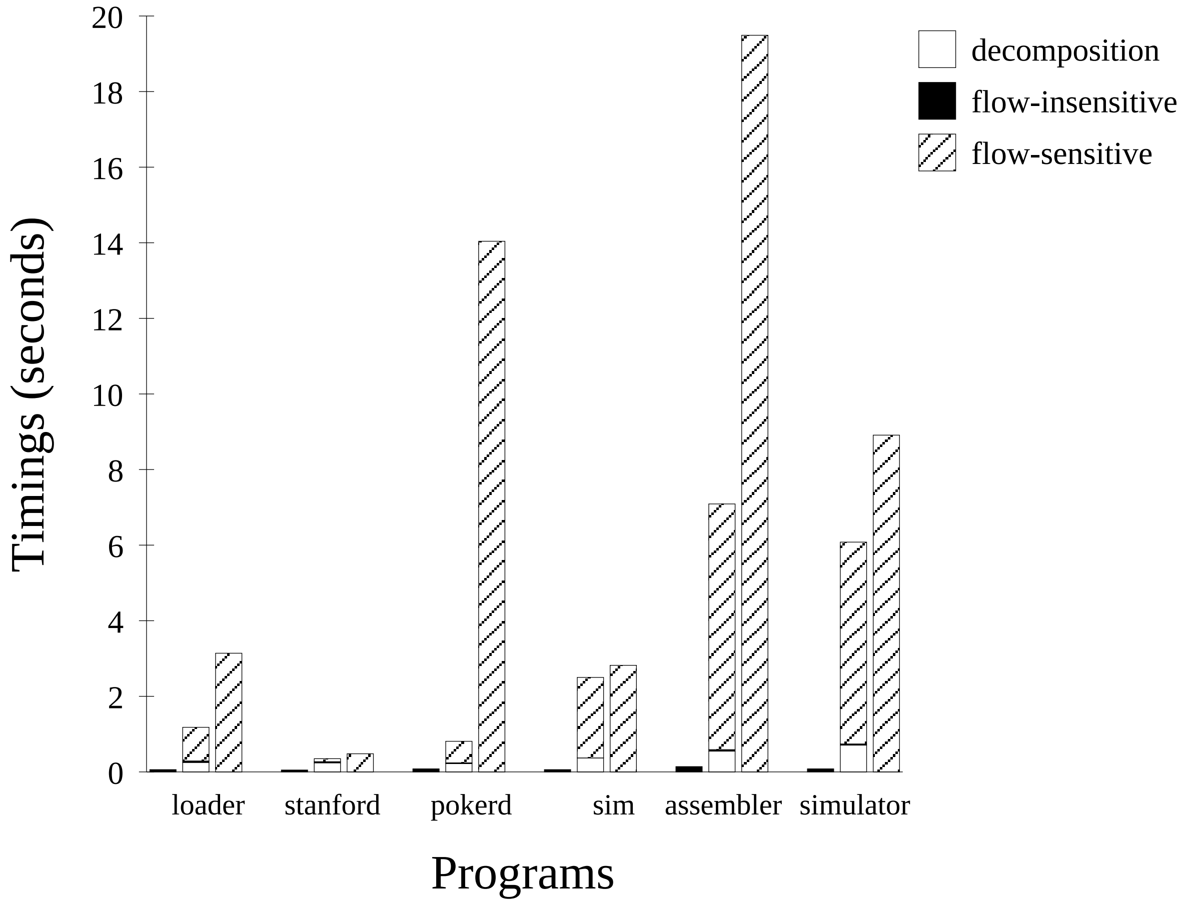
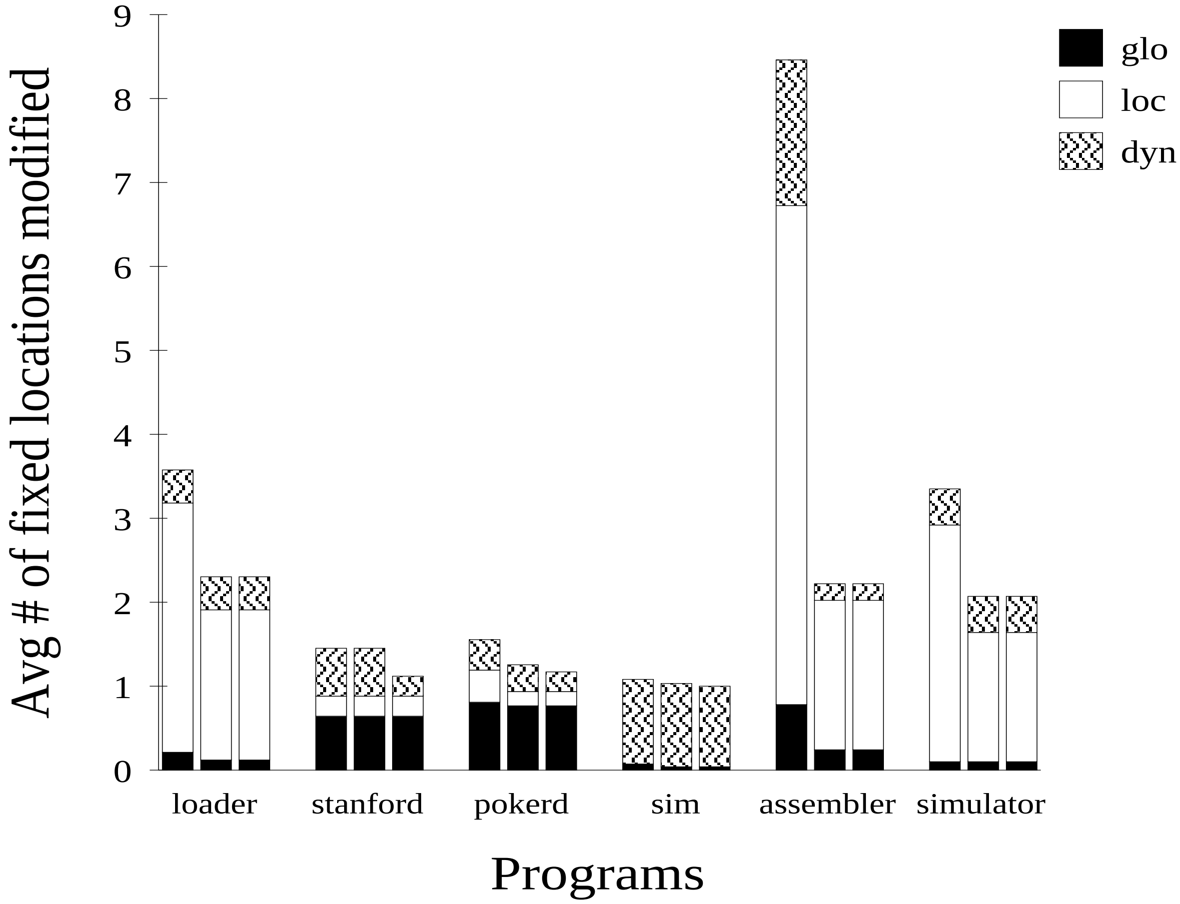


Figure 12: Timings of Calculations of PE and FA relations



left: flow-insensitive **center**: combined **right**: flow-sensitive

Figure 13: Timings of the Three Aliasing Analyses



left: flow-insensitive **center**: combined **right**: flow-sensitive

Figure 14: Thru-deref MOD Using the Three Alias Analyses

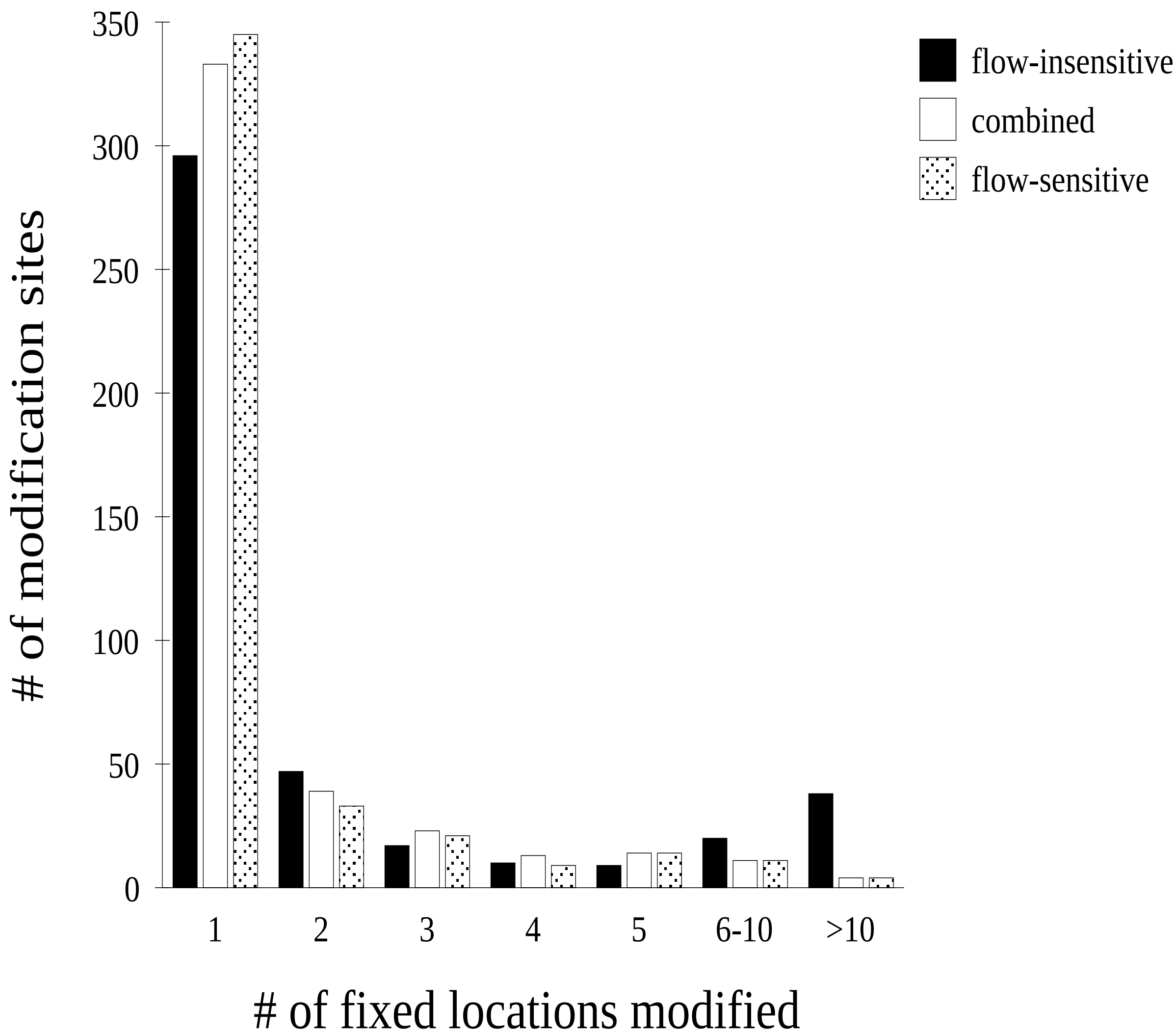
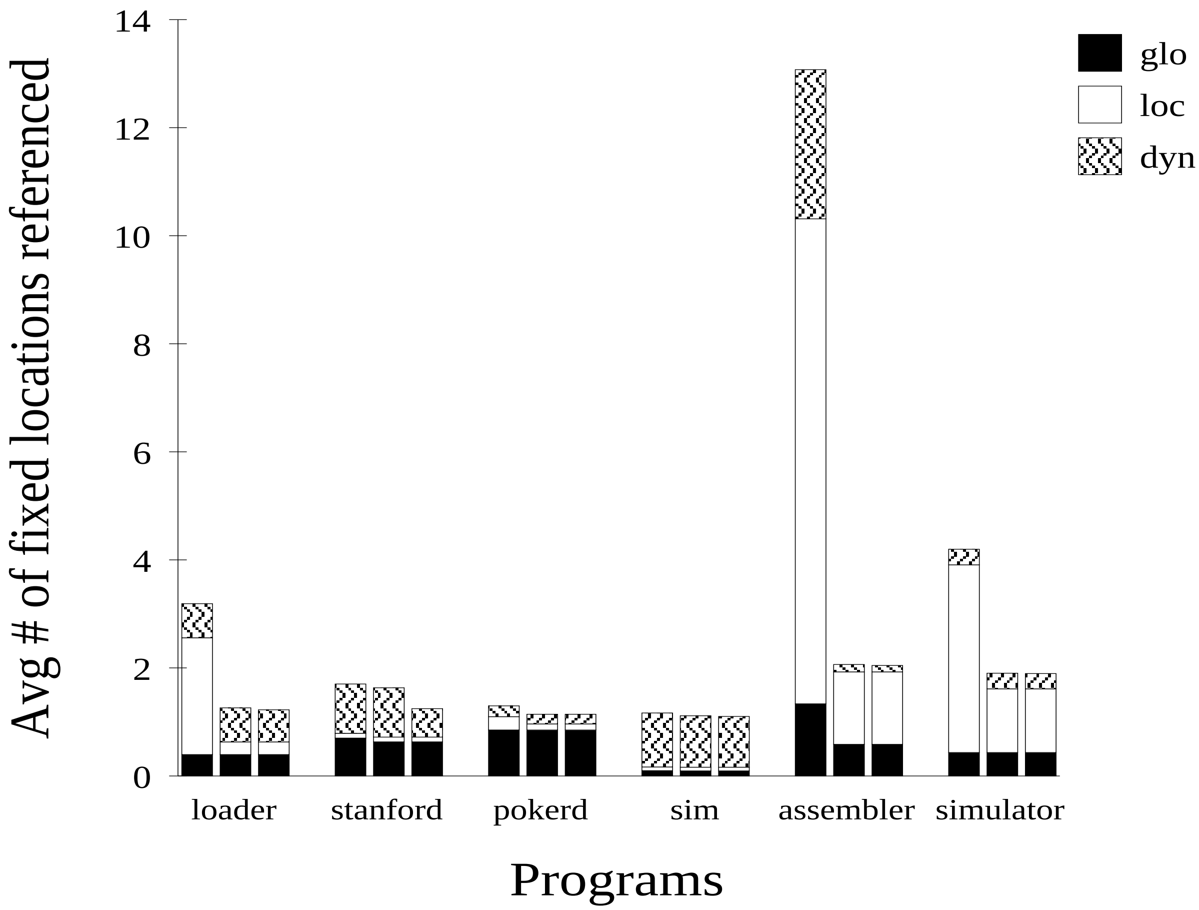


Figure 15: Distribution of Thru-deref MOD for All Six Programs



left: flow-insensitive **center**: combined **right**: flow-sensitive

Figure 16: Thru-deref REF Using the Three Alias Analyses

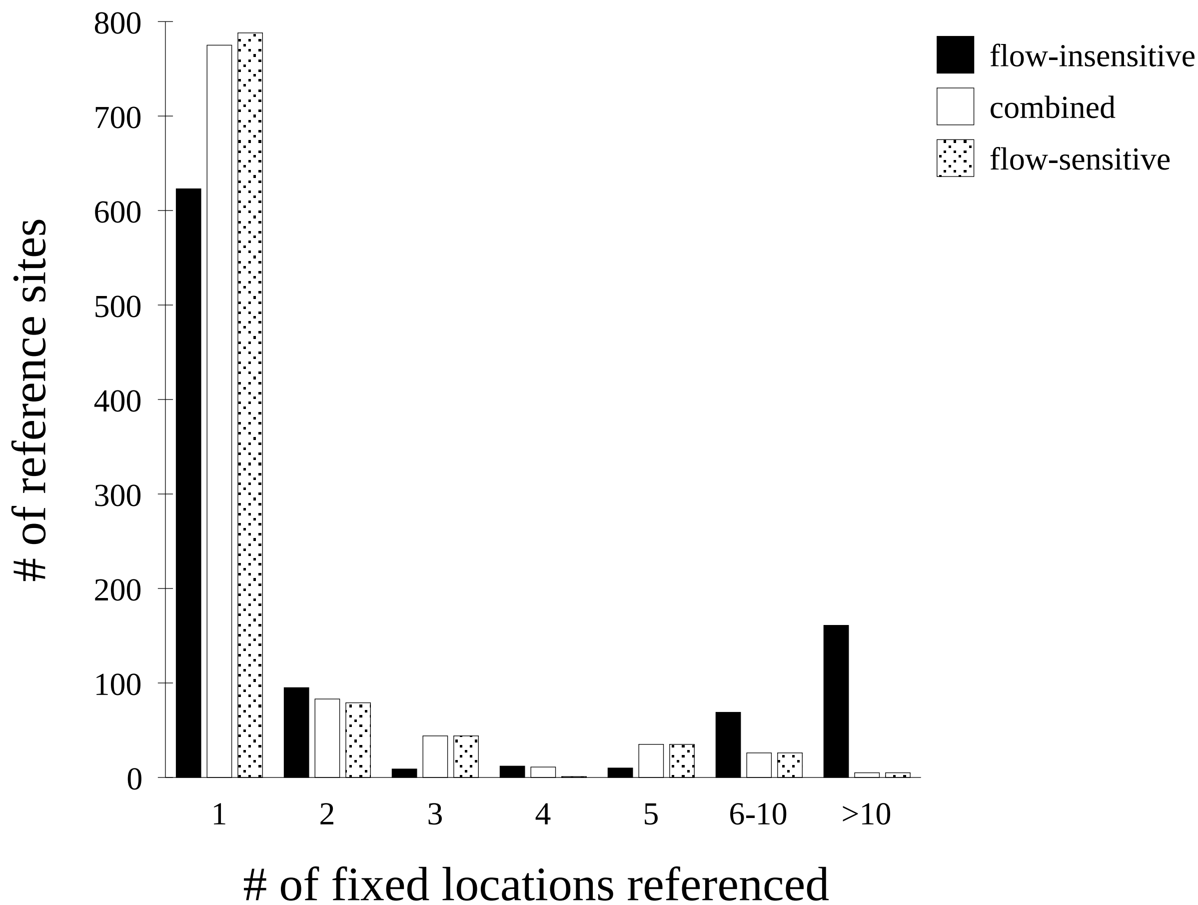


Figure 17: Distribution of Thru-deref REF for All Six Programs

In Figure 12, we present the time for calculating the PE relation and the FA relation of each program; in the same figure, we also show the time for a simple compilation of each program with *no* optimizations enabled. For each program, the calculations of the PE and FA relations take a very small fraction of the time to compile the program. For all twenty programs, calculation of the PE relation requires less than a second; for sixteen programs, it takes less than half a second. For all programs, calculation of the FA relation requires one fifth of a second or less. This shows that both calculations are efficient and practical.

Combined Analysis We have also experimented with application of different pointer aliasing analysis algorithms to independent sets of pointer-related assignments determined by our program decomposition. We used two pointer aliasing analysis algorithms: one is Landi and Ryder's flow-sensitive algorithm[16] and another is the calculation of the partial FA relation given in Section 5.2, which gives us safe flow-insensitive aliasing information. The application of pointer aliasing information that we choose, is to resolve the fixed locations modified or referenced through each object name with pointer dereferences (thru-deref MOD/REF problem). Specifically, we want to do the following:

- for each object name with pointer dereferences (e.g., $*$p, p$\rightarrow$f) as the left hand side of an assignment statement (i.e., a thru-deref modification site), the fixed locations whose values may be modified by this assignment due to aliasing is determined;
- for each object name with pointer dereferences (e.g., $*$p, p$\rightarrow$f) used anywhere else in the program (i.e., a thru-deref reference site), the fixed locations whose values may be referenced through the object name due to aliasing is determined.

The criterion we used to decide which method is applied for the pointer-related assignments associated with a weakly connected component of G_{PE}, is the k value of the component. The flow-sensitive algorithm is applied to those assignments affiliated with components with $k \neq \infty$ and the flow-insensitive algorithm is applied to those affiliated with components with $k = \infty$. Note that assignments in components with $k = \infty$ are related to recursive data structures. We call this approach *combined analysis*. Because of the criterion we have chosen, there are only six programs out of the twenty that have both kinds of components (i.e., components with $k \neq \infty$ and $k = \infty$). For each of these six programs, we applied a *full* flow-sensitive analysis, a *full* flow-insensitive analysis and the combined analysis. In Figure 13, we present the times that the three analysis algorithms took for the six programs. The time for constructing the intermediate program representation is not included. The total time for the combined analysis consists the timings of three steps: the program decomposition, the flow-insensitive analysis and the flow-sensitive analysis. From the figure, we can see:

- The flow-insensitive algorithm is the fastest and the flow-sensitive algorithm is the slowest.
- The combined analysis is in between; it takes less than half the time of the flow-sensitive algorithm for three programs (*loader*, *pokerd* and *assembler*).

For the thru-deref MOD/REF problems, we report the numbers of fixed locations that may be modified or referenced by using the aliasing solution of each analysis. The fixed locations are classified into three types according to object names representing them:

- **glo**: fixed location names involving global variables
- **loc**: fixed location names involving local variables
- **dyn**: fixed location names involving heap names

In Figure 14, we show the *average* number of fixed locations that may be modified through object names with pointer dereferences as *lhs*, by using each of the three analyses. Figure 15 shows the number of thru-deref modification sites in *all* six programs that modify certain numbers of fixed locations (e.g., 1, 2, 3, etc.) for each analysis. From these two figures, we can see that the combined analysis is more precise than the flow-insensitive analysis for 5 of the 6 programs and is close to the flow-sensitive analysis in precision.

In Figure 16 and 17, we present similar results of fixed locations referenced at program points with object names with pointer dereferences. Again we can see the combined analysis is almost as precise as the flow-sensitive analysis.

Although the above results are only for these six programs, they are quite encouraging. We are planning more empirical work to validate these preliminary findings.

Explanation We think the fact that the combined analysis is doing almost as well as the flow-sensitive analysis in terms of the thru-deref MOD/REF solutions, has something to do with our representation of heap locations, the particular problem we are solving (the thru-deref MOD/REF problems) and the flow-sensitive analysis[16] we are using. We use one heap name for each call of system-defined memory allocation routines in a program; the flow-sensitive algorithm also use this approach. Therefore, for the thru-deref MOD/REF problems, more precise alias solution involving heap locations does not necessarily mean more precise solutions; this might be the case with aliases involving data structures. Our empirical study confirmed that for the six programs, the aliasing solutions calculated by the full flow-sensitive analysis do not yields much better MOD/REF solutions than by the combined analysis.

It is our conjecture that the flow-insensitive analysis is not doing as well as either the flow-sensitive analysis or the combined analysis because it calculates a *symmetric* alias relation.

7 Related Work

Many pointer aliasing analysis algorithms have been proposed in the literature [1, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 16, 19, 22, 24, 26, 28, 30]. Any of these algorithms can be employed for individual weakly connected components in our program decomposition. In addition, our decomposition enables a sparse program representation for each component and therefore will allow any analysis algorithm to run faster. The existing analysis algorithms can be classified into flow-sensitive/context-sensitive [5, 6, 7, 8, 9, 10, 11, 12, 16, 30], flow-insensitive/context-insensitive [3, 24, 26, 28], flow-sensitive/context-insensitive[4, 22], and flow-insensitive/context-sensitive[1]. They can also be organized into stack-based aliasing analysis[7, 8], heap-based aliasing analysis[4, 6, 9, 10, 11], and both[5, 16, 22, 24, 26, 30].

The work by the research group at McGill University[7, 8, 9, 10] is particularly related to our work. Their approach is to decouple the stack-based aliasing analysis and heap-based aliasing analysis. They first perform a stack-base analysis[7, 8], which identifies pointers to the heap, and then they apply a heap-based analysis[9, 10] for these heap-directed pointers. Our approach of program decomposition is more general than their approach of decoupling the two problems. First, not all pointers to heap require sophisticated analysis; our approach can identify statements related to recursive data structures, on which a heap-based aliasing analysis may be focused. Second, we do not require these recursive data structures involve only heap-directed pointers; they may involve pointers from or to stack locations.

Theoretical classification of the compile-time pointer aliasing problem in the literature supports the use of different analysis methods. Landi and Ryder[15] first proved that the may aliasing problem is polynomial for single-level pointers and is NP-hard for multiple-level pointers (including recursive data structures). Later Landi[14] proved that for finite-level pointer dereferences (≥ 2), the may aliasing problem is P-space hard and for recursive data structures, where the number of dereferences is not known at compile time, the problem is undecidable. A similar result was reported in [21].

The FA relation is similar to the *points-to relation* [24, 26]. The approach in [24, 26] is based on a non-standard type inference technique and is inspired by the work on using type inference for binding-time analysis[13]. The algorithm handles type casting and indirect calls through function pointers, but do not allow structure types as in C. We handle structures and plan to consider function pointers and type casting in the future. The points-to relation calculated by the algorithms is used to fragment stores for the VDG representation[25, 29].

The FA relation is also similar to the *cheap alias relation* in the work done by Altucher and Landi at Siemens Corporate Research. Their approach is to calculate a program-wise alias relation, which is reflexive, symmetrical, transitive and right-regular. It handles both function pointers and type casting, but relies on type information for structure assignments in C. The idea of program decomposition for pointer aliasing analysis was motivated by their work on constructing call graphs for programs with indirect calls through function pointers.

The $G_{\rm FA}$ for a program, similar to the graph that may result from the analysis in [24, 26], can be perceived as a storage shape graph[4] although it may be quite approximate when there are recursive data structures. Our program decomposition can identify the sets of statements related to recursive data structures and therefore allow shape analysis techniques[4, 10, 23] to be applied to these statements to extract more precise information.

The modification side effect analysis for FORTRAN was given in [2]; the analysis for C was first presented in [17, 18]. Empirical results of modifications/references through pointer indirections were also reported in [8].

8 Conclusion and Future Work

We have presented a program decomposition technique for point-induced aliasing analysis, which works for well-typed C programs. We also provide an algorithm that calculate the flow-insensitive aliases based on the decomposition. We have empirically shown the practicality of the program decomposition technique and the flow-insensitive aliasing calculation.

One of the applications of the program decomposition is to allow different analysis methods to be applied for independent sets of pointer-related assignments in a program. By doing this, end users of pointer aliasing information can get the efficiency/precision trade-off that is desirable for their applications. We have shown the possibility of applying both a flow-sensitive and a flow-insensitive analysis algorithm to a same program. The resulting analysis yields an aliasing solution comparable to the one by a complete flow-sensitive analysis in solving the thru-deref MOD/REF problems, but is much faster than the full flow-sensitive analysis.

Future work includes extending the program decomposition technique to handle other features of the C language such as type casting, function pointers and pointer arithmetic. We also plan to investigate other applications of the technique such as incremental pointer aliasing analysis.

Acknowledgements We would like to thank members of the programming language research group at Rutgers, Tom Marlowe, Phil Stocks, Hemant Pande and Jyh-shiarn Yur, for their helpful comments on this paper.

References

[1] Lars Ole Andersen. *Program analysis and specialization for the C programming language.* PhD thesis, Department of Computer Science, University of Copenhagen, may 1994.

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

[3] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Lecture Notes in Computer Science*, number No. 892, pages 234–250. Springer-Verlag, 1995. Proceedings from the 7th International Workshop on Languages and Compilers for Parallel Computing.

[4] David R. Chase, Mark Wegman, and F. Kenneth Zadeck. Analysis of pointers and structures. In *Proceedings of SIGPLAN'90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990.

[5] J. Choi, M. Burke, and P. Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of te 20th ACM Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[6] Alain Deutsch. Interprocedural may-alias analysis for pointers: Beyond k-limiting. In *Proceedings of SIGPLAN'94 Conference on Programming Language Design and Implementation*, pages 230–241, June 1994.

[7] Maryam Emami. A practical interprocedural alias analysis for an optimizing/parallelizing c compiler. Master's thesis, McGill University, Montreal, Canada, July 1993.

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

[9] Rakesh Ghiya. Practical techniques for interprocedural heap analysis. Master's thesis, McGill University, Montreal, Canada, march 1995.

[10] Rakesh Ghiya. Is it a tree, a dag, or a cyclic graph? a shape analysis for heap-directed pointers in c. In *Conference Record of te 23rd ACM Symposium on Principles of Programming Languages*, pages 1–15, Jan. 1996.

[11] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. In *International Conference on Parallel Processing*, pages II49–56, 1989.

[12] Laurie J. Hendren and Alexandru Nicolau. Parallelizing programs with recursive data structures. *IEEE Transaction on Parallel and Distributed Systems*, 1(1):35–47, 1990.

[13] Fritz Henglein. Efficient type inference for higher-order binding-time analysis. In *Functional Programming and Computer Architecture*, pages 448–472, 1991.

[14] William Landi. Undecidability of static analysis. *ACM letters on programming languages and systems*, 1:323–337, 1992.

[15] William Landi and Barbara G. Ryder. Pointer-induced aliasing: a problem classification. In *Conference Record of the 18th ACM Symposium on Principles of Programming Languages*, pages 93–103, Jan. 1991.

[16] William Landi and Barbara G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *Proceedings of 1992 ACM Symposium on Programming Language Design and Implementation*, June 1992.

[17] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *Proceedings of SIGPLAN'93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[18] William Landi, Barbara G. Ryder, and Sean Zhang. Interprocedural modification side effect analysis with pointer aliasing. Technical Report LCSR-TR-201, Laboratory for Computer Science Research, Rutgers University, March 1993.

[19] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. D. Choi, M. G. Burke, and P. Carini. Pointer-induced aliasing: a clarification. *ACM SIGPLAN Notices*, 28(9):67–70, 1993.

[20] Thomas J. Marlowe, Barbara G. Ryder, and Michael Burke. Defining flow sensitivity for data flow problems. Technical Report Number LCSR-TR-249, Laboratory for Computer Science Research, Rutgers University, July 1995.

[21] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, September 1994.

[22] Erik Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 13–22, june 1995.

[23] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. In *Conference Record of te 23rd ACM Symposium on Principles of Programming Languages*, pages 16–31, Jan. 1996.

[24] Bjarne Steensgaard. Points-to analysis in almost linear time. Technical Report MSR-TR-95-08, Microsoft Research, March 1995. to appear in POPL'96.

[25] Bjarne Steensgaard. Sparse functional stores for imperative programs. In *ACM SIGPLAN Workshop on Intermediate Representation*, pages 62–70, Jan. 1995.

[26] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of te 23rd ACM Symposium on Principles of Programming Languages*, pages 32–41, January 1996.

[27] Robert E. Tarjan. *Data Structures and Network Algorithms.* Society for Industrial and Applied Mathematics, 1983.

[28] W. Weihl. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. In *Conference Record of the 7th ACM Symposium on Principles of Programming Languages*, pages 83–94, Jan. 1980.

[29] Daniel Weise, Roger Crew, Michael Ernst, and Bjarne Steensgaard. Value dependence graphs: Representation without taxation. In *Conference Record of te 21st ACM Symposium on Principles of Programming Languages*, pages 297–310, January 1994.

[30] Robert Wilson and Monica Lam. Efficient context-sensitive pointer analysis for c programs. In *Proceedings of SIGPLAN'95 Conference on Programming Language Design and Implementation*, pages 1–12, june 1995.

[31] Sean Zhang, Barbara G. Ryder, and Willam Landi. Program decomposition for pointer-induced aliasing analysis: Appendices. Technical report, Laboratory for Computer Science Research, Rutgers University, April 1996. In preparation.

A An Example

In this section, we show a complete toy program, the PE relation and the FA relation for the program, the compile-time aliases resulting from the three analyses (flow-sensitive, flow-insensitive and combined), the thru-deref MOD and the thru-deref REF solutions for the program using the aliasing solutions of the three analyses.

The program is given in Figure 18. It is an extension of the example program in Section 4. The PE relation and the FA relation for the program, represented by G_{PE} and G_{FA}, are shown in Figure 19 and 20 respectively. Both G_{PE} and G_{FA} have three weakly connected components with k being 2, 2 and ∞ respectively. Sets of object names for nodes in G_{PE} and G_{FA} are the equivalence classes of the PE relation and the partial FA relation. In particular, sets of object names in G_{FA} represent flow-insensitive aliases.

In Figure 21, we present some of the aliases computed by the three analyses. The flow-sensitive analysis calculates program-point specific aliases; the flow-insensitive analysis, on the other hand, calculates program-wide aliases, represented as equivalence classes of the partial FA relation; the combined analysis has both program-point and program-wide aliases. Due to space limitation, not all program-point aliases are given; at any program point, only aliases involving fixed location name and variables that will referenced afterwards, are shown. Equivalence classes of the partial FA relation consisting of only one object name, are not shown either.

The alias information given in Figure 21 is enough for solving the thru-deref MOD/REF problems; the solutions are given in Figure 22 and 23 respectively.

```
struct table {
   char *name;
   struct table *next;
};

struct st {
   int *f;
   int g;
} x, *p, *tt;

void insert(t,n)
struct table **t;
char *n;
{
   struct table *entry, *s;
   entry = (struct table *) malloc(sizeof(struct table));
   entry→next = *t;
   entry→name = n;
   *t = entry;
   s = (*t)→next;
}

struct table *tab;
char *name, *ss;
int z, u, w, *r, *y, **q, i;

main()
{
   tab = (struct table *) malloc(sizeof(struct table));
   tab→name = NULL;
   name = (char *) malloc(10);
   strcpy(name,"init");
   insert(&tab,name);
   ss = tab→name;

   z = 0;
   p = &x;
   p→f = &z;
   p→g = 0;
   tt = p;

   q = &y;
   *q = &w;
   r = &u;
   *r = 1;
   *q = r;
   i = *(p→f) + **q;
}
```

Figure 18: An Example Program

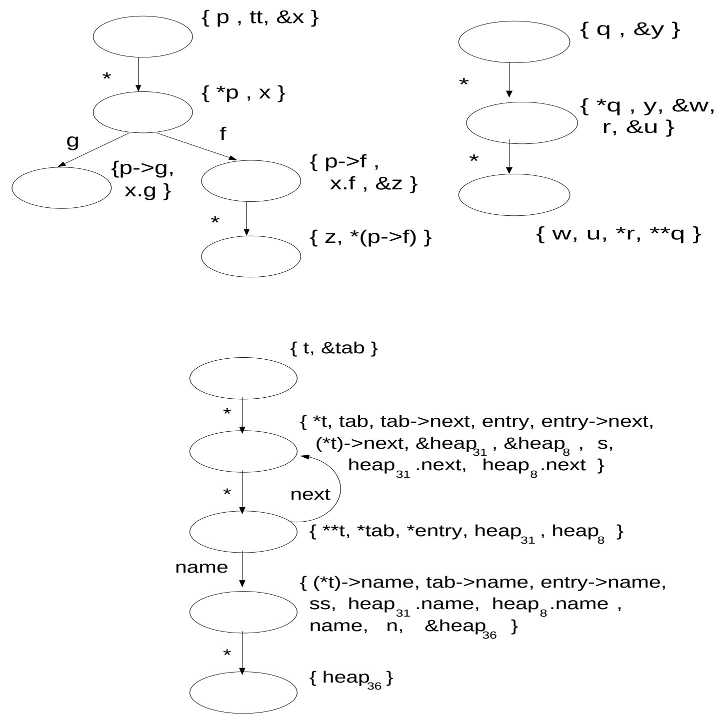


Figure 19: G_{PE} for the Example Program

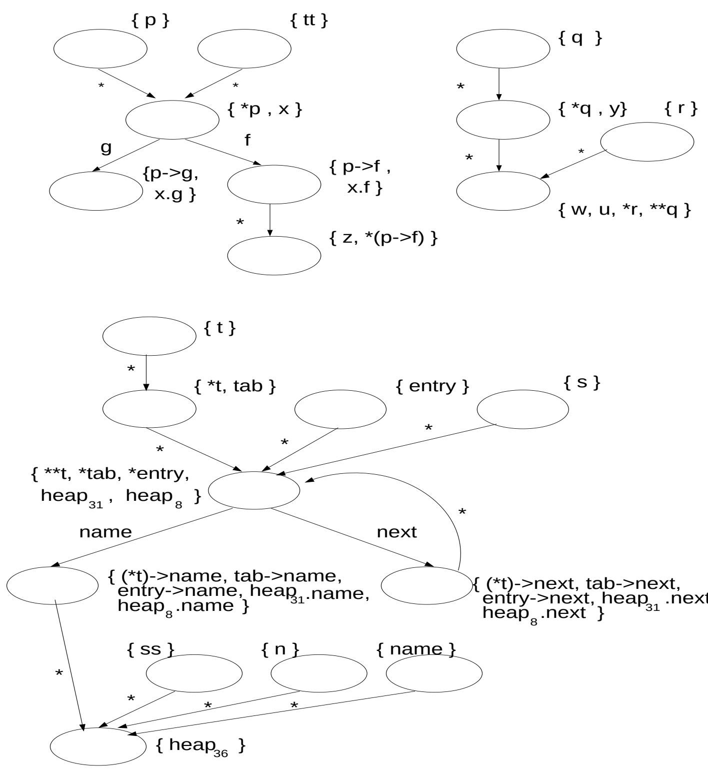


Figure 20: G_{FA} for the Example Program

statement	flow-sensitive analysis (program-point aliases)	combined analysis (program-point aliases & program-wide aliases†)	flow-insensitive analysis (program-wide aliases†)
entry = &$heap_8$	<*t,tab> <*tab,$heap_{31}$> <**t,$heap_{31}$> <*n,$heap_{36}$>	{ *t,tab } { **t, *tab, *entry, $heap_8$, $heap_{31}$ } { (*t)→name, tab→name, entry→name, $heap_8$.name, $heap_{31}$.name } { (*t)→next, tab→next, entry→next, $heap_8$.next, $heap_{31}$.next }	{ *t,tab } { **t, *tab, *entry, $heap_8$, $heap_{31}$ } { (*t)→name, tab→name, entry→name, $heap_8$.name, $heap_{31}$.name } { (*t)→next, tab→next, entry→next, $heap_8$.next, $heap_{31}$.next } { *p, x } { p→f, x.f } { p→g, x.g } { *(p→f), z } { *q, y } { **q, *r, w, u }
entry→name = n	<*t,tab> <*tab,$heap_{31}$> <**t,$heap_{31}$> <*n,$heap_{36}$> <*entry,$heap_8$>		
entry→next = *t	<*t,tab> <*tab,$heap_{31}$> <**t,$heap_{31}$> <*entry,$heap_8$> <*(entry→name),$heap_{36}$>		
*t = entry	<*t,tab> <*tab,$heap_{31}$> <**t,$heap_{31}$> <*entry,$heap_8$> <*(entry→name),$heap_{36}$> <*(entry→next),$heap_{31}$>		
s = (*t)→next	<*t,tab> <*tab,$heap_{31}$>‡ <**t,$heap_8$> <*tab,$heap_8$> <*((*t)→name),$heap_{36}$> <*(tab→name),$heap_{36}$> <*((*t)→next),$heap_{31}$> <*(tab→next),$heap_{31}$>		
tab = &$heap_{31}$			
tab→name = NULL	<*tab,$heap_{31}$>		
name = &$heap_{36}$	<*tab,$heap_{31}$>		
ss = tab→name	<*tab,$heap_{31}$> <*tab,$heap_8$> <*(tab→name),$heap_{36}$> <*(tab→next),$heap_{31}$>		
p = &x			
p→f = &z	<*p,x>	<*p,x>	
p→g = 0	<*p,x> <*(p→f),z>	<*p,x> <*(p→f),z>	
tt = p	<*p,x> <*(p→f),z>	<*p,x> <*(p→f),z>	
q = &y	<*p,x> <*(p→f),z>	<*p,x> <*(p→f),z>	
*q = &w	<*p,x> <*(p→f),z> <*q,y>	<*p,x> <*(p→f),z> <*q,y>	
r = &u	<*p,x> <*(p→f),z> <*q,y> <**q,w>	<*p,x> <*(p→f),z> <*q,y> <**q,w>	
*r = 1	<*p,x> <*(p→f),z> <*q,y> <**q,w> <*r,u>	<*p,x> <*(p→f),z> <*q,y> <**q,w> <*r,u>	
*q = r	<*p,x> <*(p→f),z> <*q,y> <**q,w> <*r,u>	<*p,x> <*(p→f),z> <*q,y> <**q,w> <*r,u>	
i = *(p→f) + **q	<*p,x> <*(p→f),z> <*q,y> <**q,u>	<*p,x> <*(p→f),z> <*q,y> <**q,u>	

†Equivalence classes with only one object name are not shown here.
‡Because of the approximation in the flow-sensitive aliaising algorithm, this alias is not killed by the previous statement.

Figure 21: Compile-time Aliases for the Example Program

thru-deref name	statement	thru-deref MOD		
		flow-sensitive analysis	combined analysis	flow-insensitive analysis
entry→next	entry→next = ∗t	$heap_8$.next	$heap_8$.next $heap_{31}$.next	$heap_8$.next $heap_{31}$.next
entry→name	entry→name = n	$heap_8$.name	$heap_8$.name $heap_{31}$.name	$heap_8$.name $heap_{31}$.name
∗t	∗t = entry	tab	tab	tab
tab→name	tab→name = NULL	$heap_{31}$.name $heap_{31}$.name	$heap_8$.name $heap_{31}$.name	$heap_8$.name
p→f	p→f = &z	x.f	x.f	x.f
p→g	p→g = 0	x.g	x.g	x.g
∗q	∗q = &w	y	y	y
∗r	∗r = 1	u	u	u w
∗q	∗q = r	y	y	y

Figure 22: Thru-deref MOD for the Example Program

thru-deref name	statement	thru-deref REF		
		flow-sensitive analysis	combined analysis	flow-insensitive analysis
∗t	entry→next = ∗t	tab	tab	tab
(∗t)→next	s = (∗t)→next	$heap_8$.next	$heap_8$.next $heap_{31}$.next	$heap_8$.next $heap_{31}$.next
tab→name	ss = tab→name	$heap_8$.name $heap_{31}$.name	$heap_8$.name $heap_{31}$.name	$heap_8$.name $heap_{31}$.name
∗(p→f)	i = ∗(p→f) + ∗∗q	z	z	z
∗∗q	i = ∗(p→f) + ∗∗q	u	u	u w

Figure 23: Thru-deref REF for the Example Program