Program Decomposition for Pointer Aliasing: A Step toward Practical Analyses*

Sean Zhang Barbara G. Ryder William Landi

Department of Computer Science
Rutgers University
Hill Center, Busch Campus
Piscataway, NJ 08855
{xxzhang,ryder}@cs.rutgers.edu

Siemens Corporate Research Inc
755 College Rd. East
Princeton, NJ 08540
wlandi@scr.siemens.com

Abstract

Pointer aliasing analysis is crucial to compile-time analyses for languages with general-purpose pointer usage (such as C), but many aliasing methods have proven quite costly. We present a technique that partitions the statements of a program to allow separate, and therefore possibly different, pointer aliasing analysis methods to be used on independent parts of the program. This decomposition enables exploration of tradeoff between algorithm efficiency and precision. We also present a new, efficient flow-insensitive pointer aliasing algorithm, which is used together with an existing flow-sensitive aliasing algorithm in our experiments. We demonstrate our technique in the context of determining side effects and variable fetches through names containing pointer dereferences (Thru-deref MOD/REF). Initial empirical results using a combination of a flow-sensitive and a flow-insensitive aliasing analysis on the same program, demonstrate that the resulting analysis is much faster than solely using the flow-sensitive method, and obtains similar precision for the Thru-deref MOD/REF problems.

1 Introduction

Many phases of the software development life-cycle require the understanding of large, complex programs. It is not possible to debug, test, modify, maintain, transform or re-engineer a program without knowledge of the side effects and patterns of data usage in the program. Compile-time analysis provides potentially useful information which can be utilized to insure the safety of program optimizations and transformations. Analysis information is also essential for data-flow-based testing [10, 12, 13, 18, 29, 31], semantic change analysis [5, 28, 33, 34, 35, 42, 43], integration of program versions [16, 17, 41] and various other optimizations, such as run-time check elision [1] (e.g., verifying that variables are initialized before they are used).

In languages with general-purpose pointer usage, two names may access the same location during execution; they are called *aliases*. For example, the assignment p = &x makes *p and x aliases. For these languages, the previously mentioned applications require aliasing information to ensure their *safety*, because when aliasing occurs, variables which do not explicitly appear in the code may be affected by value-setting statements. Similarly, aliasing information is necessary to achieve program understanding, particularly for programmers reading code which they did not write. Our focus in this paper is on the *efficient* and *practical* solution of the pointer aliasing problem to facilitate these applications.

Many techniques for compile-time pointer aliasing analysis have been proposed [3, 4, 6, 7, 8, 9, 11, 14, 15, 22, 26, 27, 32, 36, 37, 39, 40]. Some of them are more appropriate for aliases involving accesses to heap locations [6, 8, 11, 14, 15, 26, 36]; others for aliases involving accesses to stack locations [9, 39]. Still others handle both in a similar fashion [3, 4, 7, 22, 27, 32, 37, 40]. All of these methods vary in the precision of the aliasing information calculated and their cost.

For compile-time pointer aliasing analysis, a program can be considered a sequence of assignments having effects on pointer aliasing; we call them *pointer-related assignments*. In this paper, we present a technique that partitions these assignments with respect to pointer aliasing, so that we can utilize different pointer aliasing analysis methods on independent sets of pointer-related assignments, to achieve specific goals for analysis cost and precision. We also present a new, efficient flow-insensitive pointer aliasing algorithm, which is used in our experiments of the tradeoff between cost and precision. Specifically, we use the flow-insensitive aliasing algorithm for pointer-related assignments involving recursive data structures in the program and a more precise, but more costly, flow-sensitive aliasing algorithm [22] for other pointer-related assignments. We are interested in determining which variables experience side effects indirectly through a pointer dereference (Thru-deref MOD) [23] and which values are fetched indirectly (Thru-deref REF). Our initial empirical experiments show that the resulting analysis is faster than a completely flow-sensitive analysis, but demonstrates similar accuracy with respect to Thru-deref MOD/REF solutions. Thus, we have achieved acceptable precision at reasonable cost.

Our program decomposition is achieved by developing an equivalence relation (PE) on the names appearing in a program. The PE relation uniquely partitions the pointer-related assignments in the program so that each of them corresponds to a unique class in the PE relation. Thus, the pointer-related assignments are divided into independent

*This research was supported, in part, by NSF grants CCR95-01761 and GER90-23628.

To appear in the Fourth Symposium on the Foundations of Software Engineering (FSE4), 16-18 October, 1996, San Francisco, California, USA

Program ::=	$(\text{Procedure})^+$
Procedure ::=	$(\text{Statement})^+$
Statement ::=	entry of Proc(FmlName_1,...,FmlName_m)
	| exit of Proc
	| call Proc(ArgName_1,...,ArgName_m)
	| return from Proc
	| PrmName = PrmExp
	| PtrName = NULL
	| PtrName = &Name
	| PtrName = PtrName_1
	| StrtName = StrtName_1
	| HeapAlloc(PtrName)
	| HeapDealloc(PtrName)
	| if (PrmName) (goto ID_1) (goto ID_2)
	| goto ID

Figure 1: Intermediate Representation

ObjAux ::=	VarName	(variable name)
	| HeapName	(heap name)
	| ObjAux.field	(structure field)
	| *ObjAux	(pointer dereference)
ObjName ::=	ObjAux	(w/o address operator)
	| &ObjAux	(w/ address operator)
FixedLocName ::=	VarName	
	| HeapName	
	| FixedLocName.field	
precedence:	$* >$ .field $>$ &	
associativity:	$*$	right-associative
	.field	left-associative

Figure 2: Object Names and Fixed Location Names

sets that will not interact in terms of their effect on pointer aliasing. The PE relation can be used to derive another equivalence relation (FA), which provides flow-insensitive aliasing information for the program.

The paper is organized as follows. Section 2 discusses the program representation, object names and interesting relations on sets of object names. In Section 3 and 4, we present, respectively, the definition and calculation of the PE relation for program decomposition and the FA relation as flow-insensitive alias information. In Section 5, we show the empirical results of the program decomposition and our experiments of tradeoff between cost and precision. We discuss related work in Section 6 and conclude in Section 7 with future work.

2 Basic Concepts

Program Representation We consider C programs that do not have **setjmp**(), **longjmp**(), function pointers, unions, interrupts, or type casting, except casting of calls to system-defined memory allocation routines such as **malloc**(). In our implementation, we handle pointer arithmetic and arrays in a naive manner, but we will ignore these constructs in the discussion of our techniques. We represent a C program in an intermediate form; its syntax is given in Figure 1. A program consists of a number of procedures. A procedure is a sequence of statements; the first one is the entry and the last is the exit. Call and return statements are used to represent procedure calls. There are various kinds of assignment statements including non-pointer assignments, pointer assignments, and structure assignments. Each pointer assignment or structure assignment has one object name as its *lhs* (i.e., left hand side) and another as its *rhs* (i.e., right hand side). Other statements allowed are heap allocation, heap deallocation, if and goto. Statements have IDs associated with them, where an ID is in the range of 1..(# of statements). Goto statements use statement IDs for their destinations. This representation can be depicted in a graphical form such as the *interprocedural control flow graph* (ICFG) [22].

Object Names In the intermediate representation, we refer to memory locations and addresses of these locations through *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 ($*$). Heap names are created explicitly for dynamically allocated memory locations and are of the form $heap_{id}$, where id is the ID of the allocation statement. Some object names always refer to the same memory locations (e.g., p, x.g); we call them *fixed location names*. Others can refer to different locations (e.g., $*$p, p$\rightarrow$g) during program execution. 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.

Each object name has a type associated with it. 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 regardless of type. We assume each structure field in a program has a unique name; intuitively each field name is associated with the structure type to which it belongs.

We call a structure field access (.field) or a pointer dereference ($*$) an *accessor*. Given an object name and an accessor, the function *apply* returns the object name obtained after application of the accessor. The function *apply** applies a sequence of accessors to an object name and returns the resulting name. Their complete definitions are given in Figure 3, where ϵ represents an empty sequence of accessors. Another function $numofderefs(a_1a_2...a_n)$ is defined to be the number of pointer dereferences ($*$) in the sequence of accessors $a_1a_2...a_n$. Examples of these three functions are given below:

$$apply(*\text{p},\text{f}) = \text{p}\rightarrow\text{f}\ [1]$$
$$apply(\&(\text{p}\rightarrow\text{f}), *) = \text{p}\rightarrow\text{f}$$
$$apply^*(\text{p}, *.\text{f}) = \text{p}\rightarrow\text{f}$$
$$numofderefs(*.\text{f}) = 1$$

We say an object name o_1 is a *prefix* of another object name o_2 if there exists a sequence of accessors $a_1a_2...a_n$ such that $o_2 = apply^*(o_1,a_1a_2...a_n)$. For example, both p and $*$p are prefixes of p$\rightarrow$f.

We now define some interesting relations on sets of object names.

[1]Note that $p\rightarrow f$ is same as $(*p).f$ in C.

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

$$
\begin{aligned}
apply^*(o,\epsilon) &= o \\
apply^*(o,a_1a_2...a_n) &= apply^*(apply(o,a_1),a_2...a_n)
\end{aligned}
$$

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

Definition 2.1 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 $o = apply(o_1, field)$, then $o_1 \in$ S.
- If $o \in$ S and $o = apply(o_1, *\,)$, then $o_1 \in$ S.

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

Definition 2.2 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, $o_1^{'} = apply(o_1,*) \in$S and $o_2^{'} = apply(\mathrm{o}_2,*) \in$ S, then $(o_1^{'},o_2^{'}) \in$ R.
- If $(o_1,o_2) \in$ R, $o_1^{'} = apply(\mathrm{o}_1, field) \in$ S and $o_2^{'} = apply(\mathrm{o}_2, field) \in$ S, then $(o_1^{'},o_2^{'}) \in$ R.

This property is characteristic of relations such as the alias relation [22]. Suppose two object names are aliased. If both are of pointer type, their dereferences will be aliased; if both are of structure type, their corresponding fields will be aliased.

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 the relation. The relation { (p,&x), (*p,x), (p→g,x.g) } is a weakly right-regular relation on S.

Definition 2.3 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[2] equivalence relation on S containing R.

Definition 2.4 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.

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. The equivalence classes of R^e and R^{wr} are given below:

<u>R^e:</u>	<u>R^{wr}:</u>
{ p, &x }	{ p, &x }
{ *p }	{ *p, x }
{ p→g }	{ p→g, x.g }
{ x }	
{ x.g }	

[2]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

3 The PE relation

In this section, we define the PE relation for a program, which partitions its statements into independent sets with respect to pointer aliasing analysis. First, we define the concept of a pointer-related assignment in a program.

Definition 3.1 A *pointer-related assignment* in a program is one of the following:

- a pointer assignment
- a structure assignment such that the structure type of its lhs and rhs contains fields of pointer type
- a formal-actual pair at a call statement such that the two are either of pointer type or of structure type with fields of pointer type
- a heap allocation
- a heap deallocation

A heap allocation has the same aliasing effect as the assignment: $p = \&heap_{id}$, where p is an object name of pointer type and *id* is the statement ID of the heap allocation. A heap deallocation is considered to have the same aliasing effect as the assignment p = NULL, where p is an object name of pointer type.

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.

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

p = &x	q = &y
p→f = &z	*q = &w
tt = p	r = &u
	*q = r

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 }	{ }
{ z, *(p→f) }	{ }
{ q, &y }	{ q = &y }
{ *q, y, &w, r, &u }	{ *q = &w, r = &u, *q = r }
{ w, u *r, **q }	{ }

Figure 5: The PE Relation for the Example Program

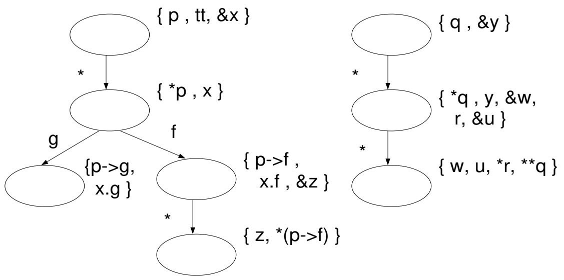


Figure 6: G_{PE} for the example program

Definition 3.2 Given a program, a set of object names, B_0, is defined inductively as follows:

- If o syntactically appears in any statement of the intermediate representation of the program[3], 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 $o = apply(o_1,\ field)$, then $o_1 \in B_0$.
- If $o \in B_0$ and $o = apply(o_1,\ *\ )$, then $o_1 \in B_0$.

Intuitively, B_0 contains object names that syntactically appear in the program and the prefixes of those names that do not have the address operator. It contains *all* fixed location names in the program. By definition, B_0 is closed with respect to prefixes. For the example program, B_0 is the following set:

$$\left\{ \begin{array}{lllllllll} \text{p}, & \text{q}, & \text{r}, & \text{tt}, & \text{u}, & \text{w}, & \text{x}, & \text{y}, & \text{z}, \\ *\text{p}, & *\text{q}, & *\text{r}, & & \&\text{u}, & \&\text{w}, & \&\text{x}, & \&\text{y}, & \&\text{z}, \\ \text{p}\rightarrow\text{f}, & **\text{q}, & & & & & \text{x.f}, & & \\ *(\text{p}\rightarrow\text{f}), & & & & & & \text{x.g}, & & \\ \text{p}\rightarrow\text{g} & & & & & & & & \end{array} \right\}$$

Definition 3.3 Given a program and its B_0 set, let R_0 be a relation on B_0 defined below:

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

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

Intuitively, a tuple $(o_1{,}o_2)$ is in R_0 if the two object names have the same value at a program point due to one pointer-related assignment. A tuple $(o_1{,}o_2)$ is in the PE relation if the two object names have the same value due to a sequence of pointer-related assignments; the PE relation also contains other tuples as pointer-related assignments are considered symmetric[4] and the control flow in the program is not taken into account.

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

[4]This is same as saying that for each pointer-related assignment, $lhs = rhs$, there is an assignment $rhs = lhs$.

The PE relation is a value-equality relation. Suppose two object names have the same value. If both are of pointer type, their dereferences will contain the same value; if both are of structure type, their corresponding fields have the same values. This is why the PE relation needs to be weakly right-regular.

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 is associated with a unique equivalence class of the PE relation.

In Figure 5, we show the PE relation for the example program in Figure 4. The left column in the figure is a list of equivalence classes of the PE relation; to the right of each equivalence class is the set of pointer-related assignments associated with the class. For some classes, there are no affiliated pointer-related assignments.

3.1 Program Decomposition

The PE relation can be depicted as a labeled, directed multigraph, where nodes represent equivalence classes and edges represent the prefix relation of object names in equivalence classes. Specifically, if an object name o is in equivalence class e and $apply(o{,}a)$ is in equivalence class e_1, where a is an accessor, then there is an edge from the node for e to the node for e_1 and the edge is labeled with the accessor a. We will use G_{PE} to refer to this graph. In Figure 6, we show G_{PE} for the example program in Figure 4; in the graph, each node is annotated with the set of object names in the equivalence class it represents.

G_{PE} can be decomposed into weakly connected components. For instance, G_{PE} in Figure 6 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 associated with the nodes in the component, and
- a set of object names, which is the union of the object names associated with the nodes in the component.

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 associated with the component. In other words, the sets of pointer-related assignments for weakly connected components in G_{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 this, we will define a new relation in Section 4, which can be calculated separately for each weakly connected component of G_{PE}, and can be proved a safe estimate of the possible run-time aliases.

3.2 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 object names in both e_1 and e_2.

The cost of each call to INIT-EQUIV-CLASS() is a small constant; the cost of N calls to FIND() or UNION() is almost linear in N if a fast union/find algorithm is used [38].

Given a program and its B_0 set, the algorithm for calculating the PE relation is given in Figure 7. The algorithm has two phases. In Phase 1, an equivalence class is created for each object name in B_0. Besides object names, each equivalence class e maintains a *prefix* set; a tuple (a,o) is in *prefix*[e], where a is an accessor and o is an object name, *if and only if* there is an object name o_1 in e such that $apply(o_1,a)$ is in the equivalence class FIND(o). Intuitively, tuples in *prefix* sets represent edges in G_{PE} for the PE relation being calculated. The initial *prefix* set for an equivalence class e with one object name, o, is one of the following cases:

- $\{\ (f_1, apply(o,f_1)),\ \ldots,\ (f_i, apply(o,f_i))\ \}$, if o is of a structure type with fields, f_1, ..., f_i.
- $\{\ (*, apply(o,*))\ \}$, if $apply(o,*) \in B_0$.
- { }, otherwise.

Here are some of the initial *prefix* sets for the example program in Figure 4:

prefix[FIND(∗p)] = { (g,p→g), (f,p→f) }
prefix[FIND(p)] = { (∗ ,∗p) }
prefix[FIND(tt)] = { }

In Phase 2, each pointer-related assignment in the program is considered and equivalence classes are merged if needed. When two classes are merged, they are unioned and their *prefix* sets are examined; any two equivalence classes having the same prefix relation with the two classes will be recursively merged. This is done in the MERGE() routine and is necessary for the weakly right-regular property.

As an example, suppose the assignment p = &x of the program in Figure 4 is the *first* being considered. The equivalence classes, FIND(p) and FIND(&x), are merged. Their *prefix* sets are:

prefix[FIND(p)] = { (∗ ,∗p) }
prefix[FIND(&x)] = { (∗ ,x) }

```
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 == apply(o1, ∗))
         add ( ∗ , o) to prefix[FIND(o1)];
      else if (o == apply(o1, field))
         add (field , o) to prefix[FIND(o1)];
   /* Phase 2 */
   for each pointer-related assignment, lhs = rhs
      if (rhs ≠ NULL && 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.

Figure 7: Calculation of the PE relation

Thus the equivalence classes, FIND(∗p) and FIND(x), are merged. These two classes have the following *prefix* sets:

prefix[FIND(∗p)] = { (g,p→g), (f,p→f) }
prefix[FIND(x)] = { (g,x.g), (f,x.f) }

Therefore, FIND(p→g) and FIND(x.g) will be merged; so will FIND(p→f) and FIND(x.f).

The result of the algorithm is G_{PE}, where nodes of the graph are represented by the equivalence classes and edges of the graph are represented by the tuples in the *prefix* sets of equivalence classes. Specifically, a tuple (a,o) in *prefix*[e] represents an edge from the node for e to the node for FIND(o) in G_{PE}.

In G_{PE} for the example program (Figure 6), there are two outgoing edges from the node for the equivalence class with object names, ∗p and x; one is to the node for class FIND(p→g) and another to the node for class FIND(p→f).

3.3 Complexity

As assumed earlier, we consider well-typed programs; so all object names in an equivalence class of the PE relation have the same type. We further assume that the maximum number of fields for any structure type is a small constant compared to the number of object names in B_0. By this assumption, the size of the *prefix* set for any equivalence class is a small constant. Let $N_0 = |\ B_0|$.

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 the MERGE() routine. Each call to MERGE() incurs one call to UNION(), a constant number of calls to FIND(), a number of recursive calls to itself, and a constant cost for other operations. 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 [38], 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() is $(\mathcal{O}(N_0 \times \alpha(N_0,N_0)) + \mathcal{O}(N_0))$. Therefore the complexity of the algorithm is $\mathcal{O}(N_0 \times \alpha(N_0,N_0))$.

4 The FA relation

In this section, we define the FA relation based on $G_{\rm PE}$ for a program; this relation provides inexpensive alias information for the program. The following notation will be used in this section.

- B_1 is the subset of B_0 that excludes any object name with &; B_1 is closed with respect to prefixes.
- $B_{\rm PE}(n)$ is the set of object names associated with a node n in $G_{\rm PE}$; it is an equivalence class of the PE relation.

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

Definition 4.1 Given $G_{\rm PE}$ for a program, B is a set of object names such that $o \in B$ *if and only if* the following are true:

- There is a path from a node n in $G_{\rm PE}$ that is labeled with a sequence of accessors $a_1a_2...a_j$.
- $o = apply^*(o_1,a_1a_2...a_j)$, where $o_1 \in (B_{\rm PE}(n) \cap B_1)$.

If $G_{\rm PE}$ has cycles, B has an infinite number of object names. For each object name in B, there may be more than one path in $G_{\rm PE}$ representing the name. One of these paths starts from the node for the variable or the heap name of the object name; all paths for the name are subpaths of that path and end at the same node.

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 $(B_{\rm PE}(n) \cap B_1)$, where n is a node in $G_{\rm PE}$, is in B. Therefore $B_1 \subseteq B$.

For the example program in Figure 4, B is the following set:

$$\left\{\begin{array}{lllllllll}
\text{p}, & \text{q}, & \text{r}, & \text{tt}, & \text{u}, & \text{w}, & \text{x}, & \text{y}, & \text{z}, \\
{*}\text{p}, & {*}\text{q}, & {*}\text{r}, & {*}\text{tt}, & & & \text{x.f}, & {*}\text{y}, & \\
\text{p}\to\text{f}, & {**}\text{q}, & & \text{tt}\to\text{f}, & & & {*}(\text{x.f}), & & \\
{*}(\text{p}\to\text{f}), & & & {*}(\text{tt}\to\text{f}), & & & \text{x.g}, & & \\
\text{p}\to\text{g}, & & & \text{tt}\to\text{g} & & & & &
\end{array}\right\}$$

Next we define a relation on B.

Definition 4.2 Given $G_{\rm PE}$ for a program, R is a relation on B such that $(o_1',o_2') \in \mathrm{R}$ *if and only if* the following are true:

- There is a path from a node n in $G_{\rm PE}$ that is labeled with a sequence of accessors $a_1a_2...a_j$.
- $numofderefs(a_1a_2...a_j) \geq 1$.
- $o_1' = apply^*(o_1,a_1a_2...a_j)$, where $o_1 \in B_{\rm PE}(n)$.
- $o_2' = apply^*(o_2,a_1a_2...a_j)$, where $o_2 \in B_{\rm PE}(n)$.

We call R^e the FA (Flow-insensitive Alias) relation.

Intuitively, a tuple (o_1',o_2') is in the FA relation if the two object names are aliased, assuming pointer-related assignments are considered symmetric and the control flow in the program is not taken into account. We prove in [44] that the FA relation defined above is weakly right-regular.

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

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

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

We prove in [44] 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 $G_{\rm PE}$ 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 $G_{\rm PE}$ form a program decomposition for pointer aliasing analysis.

4.1 Calculation of the FA relation

Since the number of object names in B may be infinite, we can not always directly calculate the set B. However, names in B may be represented by paths in graphs and graphs with cycles can represent an 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 ending at the same node in the graph.

So the idea is to use a labeled, directed graph to represent the FA relation. The graph will have a structure similar to

```
calculate-FA-relation()
{
  for each o ∈ B1
  {
    INIT-EQUIV-CLASS(o);
    prefix[FIND(o)] = { };
  }
  /* Phase 1 */
  for each edge in G_PE from n to m labeled with field
  {
    for each o ∈ (B_PE(n) ∩ B1)
      add (field, apply(o,field)) to prefix[FIND(o)];
  }
  /* Phase 2 */
  for each edge in G_PE from n to m labeled with *
  {
    s = { o1 | o1 = apply(o, *), where o ∈ B_PE(n) };
    s = s ∩ B_PE(m);
    let o1 be an arbitrary object name in s;
    for each o2 ∈ s
      if (FIND(o1) ≠ FIND(o2))
        MERGE(FIND(o1),FIND(o2));
    for each o ∈ (B_PE(n) ∩ B1)
      if there is not a ( * , o2) in prefix[FIND(o)]
      such that FIND(o2) == FIND(o1)
        add ( * , o1) to prefix[FIND(o)]
  }
}
```

Figure 8: Calculation of the FA relation

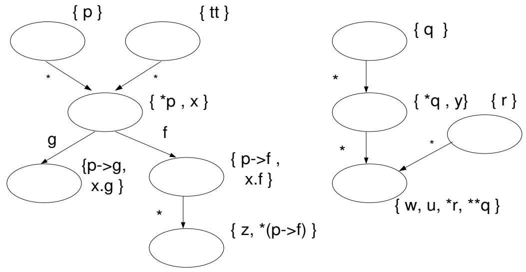


Figure 9: G_{FA} for the example program

G_{PE}, that is, nodes are annotated with a set of object names and edges of the graph are labeled with either $*$ or a field name. We require that:

(1) Each name in B is represented by one or more paths ending at the same node 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 paths for o_1 and the paths for o_2 end at the same node.

Given G_{PE} for a program, the calculation of the FA relation, shown in Figure 8, is tantamount to the construction of such a graph. The algorithm represents nodes by equivalence classes and edges by tuples in the *prefix* sets 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_{PE} labeled with accessor $field$ is examined. Assume the edge is coming out of node n in G_{PE}. For each object name $o \in (B_{\mathrm{PE}}(n) \cap B_1)$, an edge labeled with $field$ is added from the node for FIND(o) to the node for FIND($apply(o,field)$). By 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 $apply(o,field) \in B_0$.

As an example, consider G_{PE} in Figure 6. After the edges labeled with f and g are examined, we have:

prefix[FIND(∗p)] = { (g,p→g), (f,p→f) }
prefix[FIND(x)] = { (g,x.g), (f,x.f) }

In other words, there are two outgoing edges from the node for FIND(∗p), one to the node for FIND(p→g) and another to the node for FIND(p→f). Similarly there are two edges from the node for FIND(x).

In Phase 2, each edge in G_{PE} labeled with $*$ is examined. Assume the edge is from node n to node m in G_{PE}. First, all the object names o_1 such that $o_1 \in B_{\mathrm{PE}}(m)$ and $o_1 = apply(o, *)$ for some o in $B_{\mathrm{PE}}(n)$, are collected; because of the presence of the edge in G_{PE}, there is at least one such object name. Then, equivalence classes containing these object names are merged, that is, these names will be in the same equivalence class. Other equivalence classes may be recursively merged if they have same prefix relation with these classes; this is taken care of by the MERGE() routine. After the necessary merges, there is one node representing the equivalence class containing all the object names collected; let us call it node m_1. For each object name $o \in (B_{\mathrm{PE}}(n) \cap B_1)$, an edge labeled with $*$ is added from the node for FIND(o) to node m_1.

Again, consider G_{PE} in Figure 6. When examining the edge from the equivalence class with object name, p, tt and &x, two names, ∗p and x, will be collected. The equivalence classes, FIND(∗p) and FIND(x), are merged. As a result of the merge, the equivalence classes, FIND(p→g) and FIND(x.g), are merged; so are FIND(p→f) and FIND(x.f). Finally, edges labeled with $*$ are added for the equivalence classes, FIND(p) and FIND(tt); their *prefix* sets are:

prefix[FIND(p)] = { (∗ ,∗p) }
prefix[FIND(tt)] = { (∗ ,∗p) }

The result of the calculation, is a labeled, directed multigraph such that

- Each node in the graph is annotated with a set of object names. We will use $B_{\mathrm{FA}}(x)$ for the set of object names associated with a node x in G_{FA}; $B_{\mathrm{FA}}(x) \subseteq B_1$.
- Each edge in the graph corresponds to a tuple in the *prefix* set of an equivalence class. If (a,o) is a tuple in *prefix*[e_1], where a is an accessor, o is an object name and e_1 is an equivalence class, then there is an edge from the node for e_1 to the node for FIND(o) and the edge is labeled with a.

We call the graph G_{FA}. In Figure 9, we show G_{FA} for the example program in Figure 4. We prove in [44] that G_{FA} satisfies conditions (1), (2) and (3) proposed earlier; that is, G_{FA} represents the FA relation.

As defined, the FA relation can be partitioned according to weakly connected components of G_{PE}. The subrelation

index	program	lines of code	number of statements	number of weakly connected components					number of pointer-related assignments				
				k=1	k=2	2 <k$\neq\infty$	k=∞	total	k=1	k=2	2<k$\neq\infty$	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 10: Weakly Connected Components and Pointer-related Assignments

of the FA relation for a component in G_{PE}, can be calculated by an algorithm similar to the one in Figure 8, which starts with *only* object names for that component (excluding names with &) and examines *only* edges in the component. In our empirical study, we calculated the subrelations of the FA relation for individual weakly connected components.

It is possible to define and calculate the FA relation directly from a program without the PE relation. But the PE relation provides a program decomposition so that subrelations of the FA relation for parts of the programs can be calculated independently.

4.2 Complexity

By a similar argument to the one in Section 3.3, the complexity of the algorithm in Figure 8 is $\mathcal{O}(N_1 \times \alpha(N_1,N_1))$, where $N_1 = \mid B_1 \mid$.

4.3 Flow-insensitive Aliases

Definition 4.3.1 Given G_{FA} for a program, the sets, $B_{\mathrm{FA}}(x)$, where x is a node in G_{FA}, constitute a partition of the set B_1. We call the equivalence relation induced by these sets *the partial FA relation.*

For the example program, the sets, $B_{\mathrm{FA}}(x)$, where x is a node in G_{FA}, are shown in Figure 9.

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, 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 complete information about all fixed locations, to which any object name in B_1 with pointer dereferences may be aliased.

The partial FA relation can also be partioned according to weakly connected components in G_{PE}. The subrelation of the partial FA relation for each component can be used as safe aliasing information involving object names associated with the component.

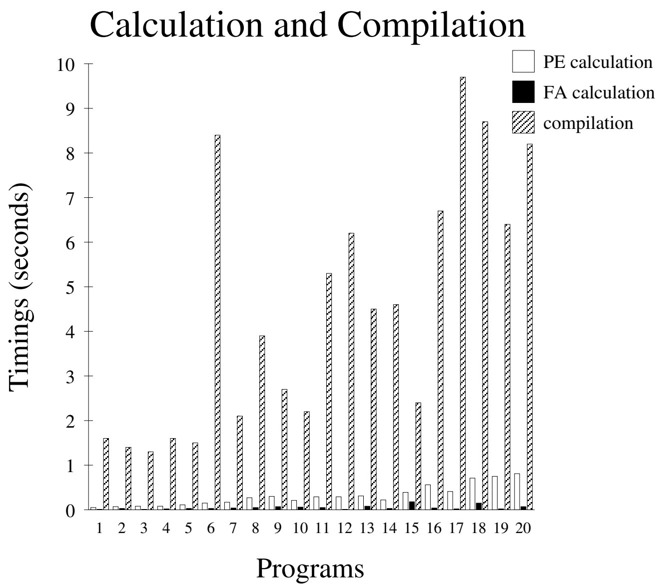
5 Experiments

Implementation Our empirical experiments consisted of forming the PE and FA relations on several C programs and then using the program decomposition they provide to experiment with the idea of applying different pointer aliasing algorithms to independent sets of pointer-related assignments. Our prototype implementation is written in C and compiled by *cc* with optimization turned on (*-O2*). The timings were collected on a Sun SPARCStation 10 running SunOS 4.1.3 with 100 megabytes of physical memory and 210 megabytes of swap space.

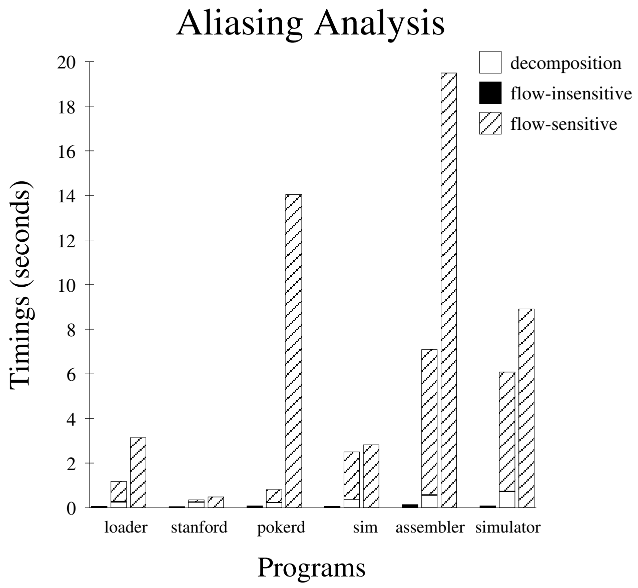
The twenty programs we have used are described in Figure 10, ordered by the number of statements in our intermediate representation. In the same figure, we give the total number of weakly connected components and pointer-related assignments resulting from our program decomposition. The numbers are also 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 components 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 that component. Each weakly connected component corresponds to an independent set of pointer-related assignments. For half of the programs, there are more than 10 such sets. Only for two small programs (*allroot* and *travel*), there is just one set. For the other programs, the decompositions do provide opportunities for using different analysis algorithms.

The values of k for weakly connected components capture certain characteristics of 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 for the component; if k is ∞ for a component, then some of the pointer-related assignments for the component involve recursive data structures.

In Figure 11(a), we present the time for calculating the PE relation and the additional time for calculating the FA relation of each program. In the same figure, we also show the time for a simple compilation of each program with *no*

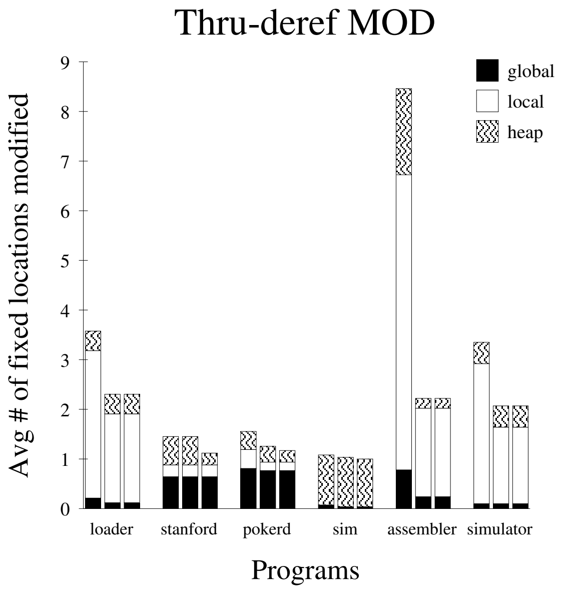


(a)



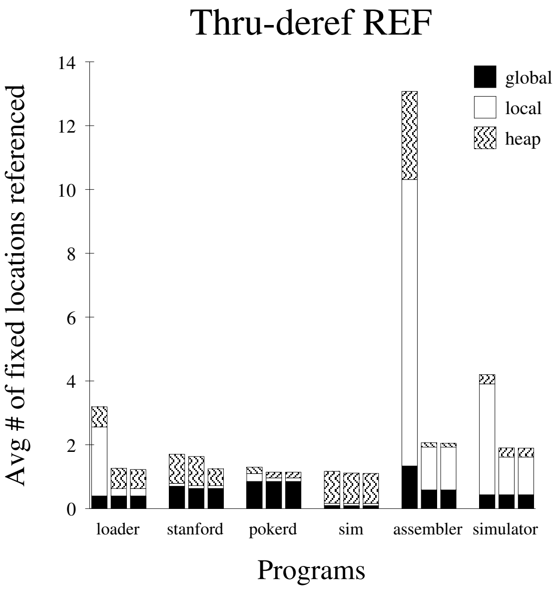
left: flow-insensitive
center: combined
right: flow-sensitive

(b)



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

(c)



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

(d)

Figure 11: Empirical Results

optimizations enabled. For each program, the calculations of the PE and the 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 One application of the program decomposition is to use different pointer aliasing analysis algorithms on independent sets of pointer-related assignments to tradeoff analysis cost and precision. We call this *combined analysis*. To show the potential of this idea, we have experimented with *one* flow-sensitive algorithm, *one* flow-insensitive algorithm, and *one* particular criterion that is used to decide which algorithm (the flow-sensitive one or the flow-insensitive one) is applied to each set of assignments. The decomposition technique, however, is generic in the sense that other aliasing analysis algorithms and other criteria can be used.

For our experiment, we used two pointer aliasing analysis algorithms: one is Landi and Ryder's flow-sensitive algorithm [22] and another is the calculation of the partial FA relation (Section 4), which gives us safe flow-insensitive aliasing information. Our targeted application of pointer aliasing information is to determine the fixed locations modified or referenced through each object name with pointer dereferences in a program, specifically:

- For each object name with pointer dereferences (e.g., $*$p, p$\rightarrow$f) as the left hand side of an assignment statement, the fixed locations whose values may be modified by this assignment due to aliasing is determined (Thru-deref MOD problem).
- For each object name with pointer dereferences used in a non-lhs context in the program, the fixed locations whose values may be referenced through the object name due to aliasing is determined (Thru-deref REF problem).

The particular criterion we used to decide which algorithm to apply 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 was applied to those assignments for components with $k \neq \infty$ and the flow-insensitive algorithm to those for components with $k = \infty$. We call this approach *k-based combined analysis*. Given the criterion we have chosen, only six programs out of the twenty contain both kinds of components. The names of these programs are highlighted in Figure 10. All these programs have recursive data structures; this is a characteristic of the programs for which the k-based combined analysis might be effective. For all other programs, the k-based combined analysis is same as the flow-sensitive analysis by our criterion.

For each of these six programs, we applied three separate analyses: a *completely* flow-sensitive analysis, a *completely* flow-insensitive analysis and a k-based combined analysis. In Figure 11(b), 11(c) and 11(d), we present the timings of the aliasing analyses, the Thru-deref MOD and Thru-deref REF solutions for the six programs respectively. For each program, the left bar represents the result of the flow-insensitive analysis, the center bar the k-based combined analysis and the right bar the flow-sensitive analysis.

In Figure 11(b), we show 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 k-based combined analysis consists of the timings of three steps: the program decomposition, the flow-insensitive analysis and the flow-sensitive analysis. From the figure, we can see that the flow-insensitive analysis is the fastest, the flow-sensitive analysis is the slowest and the k-based combined analysis is in between. For three programs (*loader*, *pokerd* and *assembler*), the k-based combined analysis takes less than half the time of the flow-sensitive analysis.

For the thru-deref MOD/REF problems, we report in Figure 11(c) and 11(d) the *average* numbers of fixed locations that may be modified or referenced through object names with pointer dereferences calculated by using the aliasing solution of each analysis. The fixed locations are classified as **global**, **local**[5] and **heap**. From these two diagrams, we can see that the k-based combined analysis is more precise than the flow-insensitive analysis and is close to the flow-sensitive analysis in precision.

Although the above results are only for six programs, they are quite encouraging. We are planning more empirical work to validate these preliminary findings. Our results that the k-based combined analysis achieves similar precision to the flow-sensitive analysis at less cost and better precision than the flow-insensitive analysis, may reflect several factors. First, since both algorithms use a very simple representation of heap storage, essentially identifying all the cells of a recursive data structure created at a **malloc**() site by the same name, and we are only looking for fixed locations in the solution, it is not surprising that these algorithms would have similar precision for heap aliases. Second, it may be that differences in precision of the aliasing solution will not be exposed by the solutions to the Thru-deref MOD or Thru-deref REF problems. Third, our assumption that pointer-related assignments are considered symmetric may lead to a gross overestimate of aliases in the flow-insensitive algorithm. Further experimentation is needed to verify these possible explanations for our results.

6 Related Work

Many pointer aliasing analysis algorithms have been proposed [3, 4, 6, 7, 8, 9, 11, 14, 15, 22, 26, 27, 32, 36, 37, 39, 40]. Any of these algorithms can be employed for individual weakly connected components in our program decomposition as our decomposition enables a sparse program representation for each component with only the pointer-related assignments associated with that component.

The work by the research group at McGill University [9, 11] is particularly related. Their approach is to decouple the stack-based aliasing analysis and heap-based aliasing analysis. They first perform a stack-base analysis [9], which identifies pointers to the heap, and then they apply a heap-based analysis [11] for these heap-directed pointers. Our approach of program decomposition is more general than their decoupling. First, not all pointers to the 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 to involve only heap-directed pointers; they may involve pointers from or to stack locations.

[5]**loc** here is the sum of *loc* and *nv* in [23].

Theoretical classification of the compile-time pointer aliasing problem supports, to some extent, the use of different analysis methods. The may aliasing problem is proved polynomial for single-level pointers [21] and NP-hard for multiple-level pointers [21, 25]. The problem is also proved P-space hard for finite-level (≥ 2) pointer dereferences [19] and undecidable for recursive data structures [20, 30].

The FA relation is similar to the *points-to* relation [37] and the *PWA* relation [2, 24]. The approach in [37] is based on a non-standard type inference technique; it handles type casting and indirect calls through function pointers, but does not allow structure types as in C. The approach in [2, 24] is to calculate a program-wide alias relation, which is reflexive, symmetric, transitive and weakly right-regular. It handles programs with function pointers and type casting, but relies on type information for structure assignments. Both the points-to relation and the PWA relation are calculated for the whole program while we can compute the FA relation for parts of a program. The idea of program decomposition for pointer aliasing analysis was motivated by the work on constructing call graphs for programs with calls through function pointers [2].

G_{FA} for a program can be perceived as a storage shape graph [6, 26] although it may be quite approximate when there are recursive data structures.

7 Conclusions and Future Work

We have presented a program decomposition technique for pointer aliasing analysis on well-typed C programs. We also provide an algorithm for calculating safe flow-insensitive aliasing solution for a program. The program decomposition allows different analysis methods to be applied for independent sets of pointer-related assignments in a program. We have experimented with applying both a flow-sensitive and a flow-insensitive analysis algorithm to different parts of the same program. The resulting analysis yields an aliasing solution comparable to the one obtained by a completely flow-sensitive analysis for solving the Thru-deref MOD/REF problems, but runs much faster. Our empirical results indicate that the program decomposition technique and the flow-insensitive aliasing calculation are efficient and practical.

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

Acknowledgments We 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. We also wish to thank Rita Altucher, Steve Masticola and Bjarne Steensgaard for their suggestions on earlier drafts of this paper.

References

[1] A. V. Aho, J. E. Hopcroft, and J. D. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[2] Rita Altucher and William Landi. Personal communication. Feb. 1994.

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

[4] 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.

[5] M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute updates. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 274–284, January 1988.

[6] 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.

[7] 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.

[8] 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.

[9] 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.

[10] P. Frankl and E. Weyuker. An applicable family of data flow testing criteria. *IEEE Transactions on Software Engineering*, 14(10):1483–1498, October 1988.

[11] Rakesh Ghiya and Laurie J. Hendren. 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.

[12] M. J. Harrold and M. L. Soffa. Interprocedural data flow testing. In *Proceedings of the Third Testing, Analysis, and Verification Symposium*, pages 158–167, December 1989.

[13] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

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

[15] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of SIGPLAN'89 Conference on Programming Language Design and Implementation*, pages 28–40, June 1989.

[16] S. Horwitz, J. Prins, and T Reps. Integrating non-interfering versions of programs. *ACM Transactions on Programming Languages and Systems*, 1989.

[17] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), January 1990.

[18] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the Sixteenth International Conference on Software Engineering*, 1994.

[19] William Landi. *Interprocedural aliasing in the presence of pointers*. PhD thesis, Rutgers University, Jan. 1992.

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

[21] 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.

[22] 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.

[23] 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.

[24] William Landi, Barbara G. Ryder, Sean Zhang, Phil Stocks, and Rita Altucher. Interprocedural modification side effect analysis for c programs. Technical report, Laboratory for Computer Science Research, Rutgers University, July 1996. In preparation.

[25] J. R. Larus. *Restructuring Symbolic Programs for Concurrent Execution on Multiprocessors*. PhD thesis, University of California Berkeley, May 1989.

[26] James R. Larus and Paul N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of SIGPLAN'88 Conference on Programming Language Design and Implementation*, pages 21–34, June 1988.

[27] 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.

[28] T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–196, January 1990.

[29] Thomas J. Ostrand. Data-flow testing with pointers and function calls. In *Proceedings of the Pacific Northwest Software Quality Conference*, October 1990.

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

[31] S. Rapps and E. Weyuker. Selecting software test data using data flow information. *IEEE Transactions on Software Engineering*, SE-11(4):367–375, April 1985.

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

[33] B. G. Ryder. Ismm: Incremental software maintenance manager. In *Proceedings of the IEEE Computer Society Conference on Software Maintenance*, pages 142–164, October 1989.

[34] B. G. Ryder, W. Landi, and H. Pande. Profiling an incremental data flow analysis algorithm. *IEEE Transactions on Software Engineering*, 16(2):129–140, February 1990.

[35] B. G. Ryder and M. C. Paull. Incremental data flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, January 1988.

[36] 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.

[37] 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.

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

[39] 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.

[40] 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.

[41] W. Yang, S. Horwitz, and T. Reps. A program integration algorithm that accomodates semantics preserving transformations. In *Proceedings of the Fourth ACM SIGSOFT Symposium on Software Development Environments*, pages 133–143, December 1990. also available as SIGSOFT Notes, vol 15, no 6, December 1990.

[42] J. Yur and B.G. Ryder. Incremental anlaysis of the mod problem for c. Laboratory for Computer Science Research Technical Report LCSR-TR-254, Department of Computer Science, Rutgers University, August 1995.

[43] F. K. Zadeck. Incremental data flow analysis in a structured program editor. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 132–143, June 1984. SIGPLAN Notices, Vol 19, No 6.

[44] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer-induced aliasing analysis. Technical report, Laboratory for Computer Science Research, Rutgers University, July 1996. In preparation. An early report of the work is in LCSR-TR-259.