A Hierarchical Approach to Context-Sensitive Interprocedural Alias Analysis

Bixia Zheng Pen-Chung Yew
Dept. of Comp. Sci. and Eng., Univ. of Minnesota, Minneapolis, MN55455

Abstract

In this paper, we present a hierarchical flow-sensitive alias analysis algorithm which parameterizes the context-sensitive level. Our approach groups the pointers in a program by their maximum possible dereference levels. It then orders the analysis of each pointer group by its pointer level, starting from the highest level down to the lowest level. During the analysis of each pointer group, a bottom-up traversal of a program call graph is followed by a top-down traversal with the necessary interprocedural information propagated along the way. The interprocedural information is tagged with call-chains, which are the program call graph paths, to achieve context-sensitivity.

We also provide empirical results to quantify how different context-sensitive levels affect the precision and the efficiency of the algorithm. Our studies show that (1) the precision improvement achieved by increasing the context-sensitive level of the analysis varies significantly depending on the programs analyzed; (2) increasing the maximum call-chain length for a higher degree of context sensitivity may trigger the exponential complexity problem [15, 10, 23]. Thus, it is very desirable for an algorithm to allow users to select an appropriate context-sensitive level that works best for a particular program. By parameterizing the maximum call-chain length used in tagging the interprocedural information, our approach provides this type of flexibility.

Keywords: interprocedural program analysis, alias analysis.

1 Introduction

In languages with general pointer usage, a pointer dereference may potentially access any memory location, thus making it difficult to determine what is defined and used. Pointer alias analysis is a compile-time technique that identifies the potential memory locations each pointer dereference may access. The accuracy of such information directly affects many other analyses and optimizations.

Most of the recent published research works on alias analysis have focused on interprocedural techniques [16, 17, 4, 7, 6, 24, 25, 13, 23, 14], because we may obtain very imprecise results when limiting the analysis within each subroutine. These analysis techniques can be classified into two broad categories: *context-sensitive* and *context-insensitive*. A context-sensitive approach distinguishes a subroutine's effect in different calling contexts while a context-insensitive approach produces a single approximation for all of its calling contexts. The context-sensitive approach, in general, can produce more precise alias information.

To facilitate context-sensitive analysis requires a mechanism to handle a subroutine differently in each of its calling contexts. Emami et al. [7] re-analyzed a subroutine for each of its calling contexts. Wilson and Lam's *partial transfer function* approach [24, 23] groups the calling contexts by their input alias patterns and performs one analysis for each pattern. Another approach tags dataflow values with *sequence token*s and *alias* *assertion*s [21]. It computes a single transfer function for each subroutine by analyzing the subroutine only once. However, all of the above context-sensitive approaches have an exponential time complexity because the invocation graph grows exponentially with the size of a program [23] unless some effort is made to limit the number of contexts in which a subroutine is analyzed.

In this paper, we propose a hierarchical alias analysis algorithm which parameterizes the context-sensitive level. We also provide empirical results to quantify how different context-sensitive levels affect the precision and the efficiency of the analysis. Our algorithm can avoid the exponential complexity problem, especially for large programs, by limiting the context-sensitivity to a low level.

We first divide each alias analysis problem into n sub-problems where n is the maximum *pointer level* of all pointers used in a program. The *level* of a pointer is the maximum level of possible indirect accesses from the pointer, e.g. the pointer level of $p2$ in the definition "*int ** p2*"[1] is two. We then calculate the values of the pointers for each pointer level, starting from pointer level n down to pointer level one. To achieve context-sensitivity, we tag the dataflow values with *call-chain*s which are call graph paths describing how the values are propagated into subroutines. By parameterizing the maximum call-chain length (*max_call_chain_length*), we can control the context-sensitive level of the algorithm. Our approach has the following features:

- It enables the use of a syntax-directed flow-sensitive analysis technique which is generally more efficient than its iterative counterpart [1].
- It facilitates a combined analysis technique [25, 20, 26] which applies different algorithms to analyze different data structure groups in order to improve the precision and the efficiency of the algorithm.
- It can provide a spectrum of context-sensitive algorithms, ranging from a context-insensitive algorithm to a full context-sensitive algorithm, by simply specifying *max_call_chain_length* with different values.
- The algorithm can be extended to analyze non-pointer variables which have a pointer level equal to zero.

In the rest of this paper, we present the hierarchical flow-sensitive alias analysis algorithm and provide empirical results to quantify how different context-sensitive levels affect the precision and efficiency of alias analysis. Section 2 presents the main idea of our hierarchical approach. Section 3 discusses some major concepts regarding memory objects and dataflow value representations. We illustrate the algorithm in Section 5 and present our experimental results in Section 6. Other related works are discussed in Section 7. Finally, Section 8 draws conclusions.

2 The Hierarchical Alias Analysis

As mentioned in the previous section, the *pointer level* of a variable is the maximum level of possible indirect accesses from the variable. For simplicity, we refer to a variable with a pointer level n as an n-level variable. We also refer to a program whose pointer variables have a maximum pointer level of n as an n-level program.

To provide some insight into our hierarchical approach, we first consider an example program in Figure 1. In the example, it is difficult to determine the dataflow values generated or killed by the indirect assignments using pointer dereferences $*p2$, $*p1$ and $*q1$. The side-effect of the indirect assignment "$*p2 = \&c$" is determined by the value of $p2$ when the subroutine is called. On the other hand, the definitions generated by the statements "$*p1 = 1$" and "$*q1 = 1$" are unknown until we determine the side-effect of the subroutine call. Hence, two main issues need to be addressed:

[1] We use the C language to give examples in this paper.

- A subroutine's side-effect may depend on the values of some pointers when the subroutine is called.
- The program segment following a subroutine call may not be analyzed until the side-effect of the subroutine is known.

The first constraint above makes it impossible to use a single bottom-up pass over the call graph to calculate all pointer values while the second one prevents a single top-down pass. Thus, supplementing the interprocedural analysis with the intraprocedural analysis is necessary. Figure 2(a) graphically illustrates this bi-directional dependency in a context-sensitive interprocedural alias analysis.

```
    int a, b, c, *p1, *q1, **p2;
    main() {
s1      p2 = &p1;
s2      sub();
s3      *p1 = 1;
s4      *q1 = 1;
s5  }

    void  sub() {
s6      p1 = &a;
s7      q1 = &b;
s8      *p2 = &c;
s9  }
```

Figure 1: An example program.

However, this dependency cycle can be eliminated if we group the variables by their pointer levels and represent each group with a node in the procedures $main$ and sub. As shown in Figure 2(b), the value for the 2-level variable $p2$ is propagated from $main$ to sub and determines the value of the expression $*p2$. It also determines sub's side-effect on the 1-level variables $p1$ and $q1$. The new values of $p1$ and $q1$ propagate from sub back to $main$ and determine $main$'s side effect on 0-level variables a, b and c.

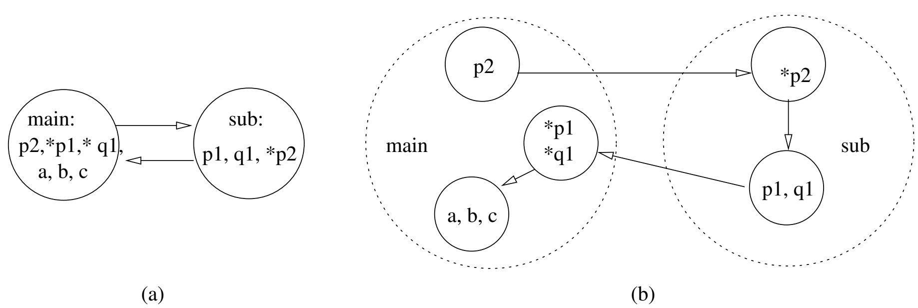

Figure 2: The dependency between $main$ and sub in Figure 1. The arrow pointing from a node A to another node B indicates that the values of the expressions in A determine the values of the expressions in B.

This suggests a two-step alias analysis algorithm to the example program: first, collect the values for the variable $p2$ and show that the statement $s8$ can be treated as "$p1 = \&c$"; then, collect the values for the variables $p1$ and $q1$ so that later analyses can treat "$*p1 = 1$" as "$c = 1$", and "$*q1 = 1$" as "$b = 1$". Our hierarchical alias analysis uses a similar approach. We first divide the alias analysis problem for an n-level program into n subproblems. The ith subproblem analyzes only the assignments to the i-level variables in the program and determines the memory locations accessed by dereferencing these i-level variables. We then solve all subproblems by starting from the nth subproblem down to the first subproblem so that by the time we need to solve the subproblem of a certain pointer level, the higher level pointer values have already been obtained.

3 Major Concepts

This section describes the major concepts used in our hierarchical alias analysis. We first introduce our memory object representation. Then we present our definitions of *pointer level* and *predicate*. Finally, we describe our dataflow value representation.

3.1 Memory Object

We use *Memory Objects (MemObj)* to model run-time memory locations that store information. There are two kinds of MemObjs: *static* and *dynamic*. A static MemObj represents the memory location created for a compile-time variable while a dynamic MemObj represents all the memory objects generated by a memory allocation statement (such as a $malloc$ statement) at run-time. If a memory allocation statement allocates a structure, we create several dynamic MemObjs, one for each field of the structure.

We create one MemObj for a scalar variable, and one MemObj to represent all the elements in an array. We treat a structure or a union variable as an aggregate of MemObjs, each of which corresponds to a field in the structure or the union. The difference between a structure variable and a union variable is that the MemObjs in a union variable are overlapped while those in a structure variable are not. This MemObj representation allows us to distinguish between different fields in a structure or a union and identify the dynamic memory objects by their allocation statements. However, we do not distinguish different elements in an array.

We use MemObjs instead of variables in our dataflow value representation for two main reasons: (1) a dynamically allocated memory object does not correspond to any variable; and (2) a structure or a union variable may correspond to several memory objects.

3.2 Pointer Level

Each MemObj has a *type* attribute. The type for a static MemObj is the type of the corresponding variable, and the type for a dynamic MemObj is derived from the type used in the type-cast operator of the memory allocation statement. For instance, we create an array MemObj of type T for memory object allocated by statement "(T *) malloc(size)".

The *pointer level* of a type corresponds to the maximum level of possible dereferences from a MemObj of that type. The pointer level of the variable *ps* in the definition "*struct S* $\{$*int* $**f;\}$ $*ps$ ", for example, is three because ps can have up to three levels of dereferences ($**((*ps).f)$). The pointer level of a non-recursive type is calculated as follows:

- The *pointer level* of a scalar type is 0.
- The *pointer level* of a type "T *" is $1 + x$ where x is the *pointer level* of the type "T".
- The *pointer level* of a structure or a union equals to the maximum *pointer level* of its field types.

In the absence of recursive data structures, a dereference from an n-level MemObj may access one or more MemObjs with a pointer level less than n. Our hierarchical approach is based on this observation. We will discuss how our algorithm conservatively handles recursive data structures in Section 4.5.

3.3 Predicate

A *predicate* is attached to a dataflow value to represent the condition in which the value is valid. A predicate can be either $null$ or the disjunction of a group of *call-chain*s: $CH_1 \vee CH_2 \vee ... \vee CH_n$. When the predicate is $null$, the corresponding dataflow value is not interprocedurally related. Otherwise, the group of call-chains describe all the calling conditions under which the dataflow value is passed into the current subroutine.

A *call-chain* is an order list of *(s, c)* pairs, represented as $(s_1, c_1)- > (s_2, c_2)...- > (s_n, c_n)$. Each *(s, c)* pair is associated to a subroutine call site, where s is the strongly connected component (SCC) in a program call graph that contains the callee, and c is an ID distinguishing among different calls to the same SCC. Usually, c is a non-negative integer; different call sites to the same SCC have different values of c. However, to avoid an unlimited call-chain length, we set c to -1 for any call site to a recursive SCC.

A call-chain represents how a dataflow value is passed into a subroutine. For instance, a call-chain $(s_1,c_1)- >(s_2,c_2)$ is attached to a pointer value which is first passed into a subroutine in SCC s_1 via a call site with ID c_1, then passed into a subroutine in SCC s_2 via a call site with ID c_2.

3.4 Representing Dataflow Values

A *definition* to a MemObj is a statement that assigns or may assign a value to the MemObj. A definition is represented as $Definition(s, i, m, v, dp)$, where s is the statement that generates the definition, i is a unique ID for the definition, m is the MemObj being defined, v are the *value*s of the object, and dp is the predicate that describes the calling conditions in which the definition is valid. The definition ID is necessary because a statement may generate more than one definition in the presence of pointer dereferences. The predicate dp is also referred to as a *defined-predicate*.

We use $Reach(i, s, rp)$ to represent a definition with an ID i which reaches a statement s under the calling conditions described by a predicate rp. The predicate rp is also referred to as a *reach-predicate*. If we have $Reach(i, s, rp)$, and the definition with an ID i is defined as $Definition(s', i, m, v, dp)$, then we have a dataflow value $ObjectValue(m, v, rp \wedge dp)$ at the statement s. This dataflow value means at the statement s, a MemObj m has the values represented by v under the calling conditions described by $rp \wedge dp$.

A *transfer function TF(sub, n)* describes all the *n-level MemObj*s referenced or modified in the subroutine sub. In the transfer function *TF(sub, n)*, a *MemObj* referenced in sub is represented by a pair (m, p), where m is the MemObj referenced and p is a predicate describing the calling conditions in which the object is referenced. Similarly, a MemObj modified in sub is represented as $(m, (v_1, p), (v_2, p), ...)$. A (v_i, p) pair represents both a possible output value of the MemObj m and the calling conditions in which the value is valid.

4 The Hierarchical Algorithm

In this section, we describe our hierarchical algorithm shown in Figure 3.

4.1 Pre-analysis

In the pre-analysis phase, we traverse the program once to collect the necessary information for later analysis. The collected information includes the maximum pointer level for each subroutine, the maximum pointer level for the whole program, and an indication of whether the program has any recursive data structure or not. This information is used to divide the original problem into subproblems. We also gather the subroutines and the variables whose addresses are assigned to pointers. The addressed variables are used to handle unknown points-to values as described in Section 4.5 while the addressed subroutines are used to determine the potential callees

```
pre-analysis();
//analysis;
handle recursive data structures;
for i = maximum-pointer-level downto 1
    //bottom-up:
    for each SCC scc in a reversed topological order
        if (scc is  not a call graph cycle)
            assume sub is the only subroutine in scc;
            compute the reaching definitions for i-level MemObjs;
            summarize the subroutine's effect on i-level MemObjs in TF(sub,i).
        else //scc is a call graph cycle
            for each subroutine sub in scc
                initialize TF(sub, i) = {};
            change = true;
            while (change)
                change = false;
                for each subroutine sub in scc
                    compute the reaching definitions for i-level MemObjs;
                    summarize the subroutine's effect on i-level MemObjs in TF(sub, i);
                    if the new TF(sub, i) is different from the old one, set change = true.
    //top-down:
    for each SCC scc in topological order
        if (scc is a call graph cycle)  //propagate values for calls within the same SCC
            change = true;
            while (change)
                change = false;
                for each subroutine in scc
                    for each call statement to a callee-subroutine sub which is also in scc
                        for each MemObj referenced in TF(sub, i)
                            propagate the value of the MemObj from the current procedure(caller) to sub(callee);
                            if the new values of MemObj are different  from the old ones
                                change = true;
        for each subroutine in scc
            traverse each statement in the subroutine
                if encounter a call to subroutine sub  and sub is not in scc
                    for each MemObj referenced in TF(sub, i)
                        propagate the values of the MemObj from the current procedure(caller) to sub(callee);
                if encounter a dereference from a i-level MemObj
                    annotate the dereference with the (i-1)-level MemObjs
                    that the i-level MemObj may points-to.
```

Figure 3: An overview of the hierarchical algorithm.

of the indirect calls via function pointers. During the pre-analysis phase, we also construct a program call graph, compute its SCCs, and assign a (s, c) pair to each subroutine call (Section 3.3).

4.2 Bottom-up analysis

To calculate the reaching definitions for all i-level MemObjs and the transfer function $TF(sub, i)$ for each subroutine sub, a bottom-up analysis for pointer level i traverses the SCCs of the program call graph in a reversed topological order (see Figure 3).

4.2.1 Handling Assignments

An assignment assigns a value to a MemObj. There are two kinds of assignments: direct and indirect. An assignment with a variable on its left-hand-side(LHS) is a direct assignment. It generates a definition with $null$ as its defined-predicate, which means the definition is generated regardless of how the subroutine is entered. On the other hand, an assignment with a pointer dereference on its LHS is an indirect assignment. An indirect assignment generates one or more definitions depending on the number of points-to values of the pointer. The defined-predicate for a definition generated by this indirect assignment determines the conditions in which the pointer points to the MemObj being defined.

Generating a new definition affects the current reaching definitions in two ways. First, a definition generated by an assignment reaches its immediate next statement with a reach-predicate $null$. This $null$ reach-predicate means that under the condition that the definition is generated, the definition definitely reaches its immediate next statement. Second, generating a new definition may *kill* other reaching definitions by changing their reach-predicates. In general, if we have $Definition(s', j, m, v, dp)$ and $Reach(i, s', rp)$ at a statement s' where both definition i and definition j define the same MemObj, then we have $Reach(i, s' + 1, rp \wedge (\neg dp))$ at the next statement $s' + 1$. If the value of $rp \wedge (\neg dp)$ is $false$, definition i is completely killed. Otherwise, it is conditionally killed. A flow-sensitive algorithm considers both of the above two effects while a flow-insensitive algorithm does not consider the *killing* effect. Therefore, by turning on or off the above *killing* effect, our algorithm can be either flow-sensitive or flow-insensitive.

4.2.2 Backward Propagating Dataflow Values

In order to update the current reaching definitions during the bottom-up analysis, we also consider the modification side effect of the callee subroutine. Because we traverse the call graph in its reversed call order, when we encounter a call to subroutine sub, its transfer function $TF(sub, i)$ has already been computed and can be applied to determine the call statement's effect on the current dataflow value.

Assume *sub* modifies a MemObj *m* with a value *v* and a predicate *p*, represented by $ObjectValue(m, v, p)$. The operator $Returnable$ compares the predicate *p* with the (s, c) pair associated with the call site to determine whether the dataflow value can be propagated back to the call site or not. If $ObjectValue(m, v, p)$ can be propagated back to a call site with (s, c), the (s, c) pair should be detached from all the call-chains in the predicate p. The operator $Detach(p, (s, c))$ calculates the new predicate for the dataflow value.

When $ObjectValue(m, v, p)$ is propagated back to the call site with (s, c), a definition $Definition\ (s', i, m, v, Detach(p, (s, c)))$ is generated, where s' is the call statement, and i is a new definition ID. The new definition will reach the immediate next statement. This is represented by have $Reach(i, s' + 1, null)$.

4.2.3 Syntax-Directed Method

A syntax-directed flow-sensitive analysis algorithm uses dataflow equations for regular control structures, thus avoiding the need to iteratively analyze control structures with backward jumps to calculate a fix-point solution[1]. To apply this syntax-directed method, two criteria must be met: (1) the definitions generated by any statement (*gen-set*) and the definitions killed by any statement (*kill-set*) are independent of the current dataflow values; (2) the program being analyzed cannot have a control structure other than sequential, branch(if or switch), and loop.

Normally, a syntax-directed method cannot be directly applied to programs with indirect assignments using pointer dereferences because both the *gen-set* and the *kill-set* for any indirect assignment are not constants but rather depend on the current reaching definitions. More specifically, the *gen-set* and the *kill-set* of the i-level MemObjs depend on the reaching definitions of the (i+1)-level MemObjs. However, in our hierarchical approach, by the time we analyze the i-level MemObjs, the reaching definitions of the (i+1)-level MemObjs have already been obtained, and the *gen-set* and the *kill-set* of the i-level MemObjs are independent of the reaching definitions of the i-level MemObjs. This is one of the reasons that we can use a syntax-directed method.

To meet the second criterion, we normalize irregular control-flow where-ever possible and marks the program segments containing irregular control structures which cannot be normalized. We use a syntax-direct method to calculate flow-sensitive dataflow values for regular control structures. On the other hand, we calculate flow-insensitive dataflow values for program segments with irregular control structures by turning off the *killing* effect of any assignments within the segments (Section 4.2.1). Thus, we avoid the need to iteratively analyze each subroutine to calculate a fix-point solution.

4.3 Top-down analysis

The top-down analysis for a pointer level i, as shown in Figure 3, traverses the SCCs in their topological order. It propagates the values of the i-level MemObjs from the callers to the callees and annotates any dereference from an i-level MemObj with the (i-1)-level MemObjs it may potentially access.

4.3.1 Forward Propagating Dataflow Values

We check a subroutine sub's transfer function $TF(sub, i)$ when we encounter a call to it during the top-down analysis. If a MemObj is referenced in the subroutine, we propagate the values of the MemObj from the caller to the callee.

When a dataflow value $ObjectValue(m, v, p)$ is propagated to a subroutine via a call site with (s, c), the (s, c) pair is appended to the end of the call-chains in the predicate p. This is to denote that the information is passed one step further along the call graph. The operator $Append(p, (s, c))$ calculates the new predicate for a dataflow value propagated to a subroutine.

If $ObjectValue(m, v, p)$ is propagated into a subroutine via a call site with a (s, c) pair, we generate $Definition\ (s_e, i, m, v,\ Append(p, (s, c)))$ and $Reach(s_e, i, null)$, assuming s_e is the entry of the subroutine. Because we traverse the procedures according to their call order, by the time we start to analyze a subroutine, the input values of all its referenced i-level MemObjs are available.

4.3.2 Evaluating Pointer Dereferences

The top-down analysis also annotates the memory objects which can be potentially accessed by a dereference from an i-level MemObj. The input values along with the reaching definitions of the i-level MemObjs can determine an i-level MemObj's value at a certain statement. If we assume that we have $ObjectValue(m, Address(x), p)$ at a statement s, then a dereference expression $*m$ at s can be annotated with $Reference(x, p)$. This indicates that the dereference expression $*m$ will access the MemObj x under the predicate p.

4.4 Parameterizing the Context-Sensitive Approach

We set a parameter called $max\_call\_chain\_length$ to limit the length of the call-chains in the predicates tagging the dataflow values. Without any limitation, a call-chain in a predicate can be as long as the depth of the program call graph. With the limit set by this parameter, the predicate used to tag the pointer values can only contain the most recent $max\_call\_chain\_length$ call sites in the call-chain. This may cause a merge of the dataflow information propagated beyond the most recent $max\_call\_chain\_length$ call sites. Thus, controlling the value of $max\_call\_chain\_length$ can control the context-sensitivity of the algorithm.

The parameter $max\_call\_chain\_length$ is also directly related to the complexity of the algorithm. Let l denote the $max\_call\_chain\_length$ used in tagging the dataflow value, and c denote the maximum number of call sites for a subroutine. There can be up to c^l call-chains in a predicate used in tagging the dataflow values for the subroutine. The conjunction operator ($\wedge$) is the most complicated operator among the three predicate operators (negative $\neg$, disjunction $\vee$ and conjunction $\wedge$) used in dataflow value calculation and has the complexity of $O(c^{2l})$, or the square of the number of call-chains. Without any predicate on the dataflow values, the operations to $generate$ or $kill$ a definition have the complexity of $O(1)$. In the presence of predicates, these two operations need to handle predicate calculation and have the complexity of $O(c^{2l})$. The overall complexity of our algorithm is, therefore, $O(ntc^{2l})$, where n is the maximum pointer level of the program and t is the time to compute the reaching definition for a program with single level pointer. Pande et al. [18] showed that t is polynomial. Thus, the exponential complexity of the algorithm lies in the value of $max\_call\_chain\_length$.

There is a trade-off between the complexity and the precision of the algorithm. An algorithm with a larger value of $max\_call\_chain\_length$ may provide more precise information at the cost of more analysis time. Thus, the parameter $max\_call\_chain\_length$ in our approach allows users to select a prudent context-sensitive level that works best for a particular program.

4.5 Handling Complicated Language Features

For simplicity, our discussion above ignores some problematic language features. We now consider some of those features and modify our algorithm accordingly in order to handle real-world programs.

Recursive data structures pose some difficulties to our hierarchical approach because dereferencing a pointer of a recursive data structure may not "lower" its pointer level. To overcome this difficulty, we analyze all recursive data structures before the analysis of other MemObjs as shown in Figure 3. Theoretically, we can incorporate any existing recursive data structure analysis algorithm into our approach since recursive data structures are usually handled separately. However, in our current implementation, we first annotate any indirect write to recursive data structures using pointer dereference, such as "*p->next = a-value*" where p is defined as "*struct S* { *struct S *next;* } **p*", with all the addressed MemObjs which have the same type as the dereference expression (*p->next*). Then, we use the bottom-up and top-down analysis algorithms described in Figure 3 to analyze the recursive data structures.

Type-casting may also be a problem because we order the analysis of MemObjs via their defined types. We identify three categories of type-casting and their corresponding handling strategies:

- Type-casting a pointer value to a lower-level pointer violates our assumption that dereferencing an n-level MemObj may access one or more (n-1)-level MemObjs. As shown in Figure 4(a), dereferencing a 1-level MemObj $p1$ accesses a 2-level MemObj $p2$. The analysis for pointer level two may result in a wrong value for $p2$ because it ignores the fact that the statement $s2$ modifies $p2$. To avoid this error, the pre-analysis phase sets the *type-cast-to-lower-pointer-level* flag of $p2$ when it encounters the type-casting expression in the statement $s2$. Later analyses conservatively assume that a dereference from a MemObj with the *type-cast-to-lower-pointer-level* flag set may access any addressed MemObjs.

- Type-casting a non-pointer value to a pointer, as shown in Figure 4(b), is considered to be constructing a pointer value from scratch. We assume a later dereference from the pointer may potentially access any addressed memory object.

- Figure 4(c) shows that if we first type-cast a pointer value of one structure type to a pointer of another structure type, then dereferencing the pointer (*ps2->c[i]*) may access an unknown portion of a structure ($s1$). In this case, we conservatively assume the dereference may touch all fields in the structure.

User-defined memory-allocation subroutines in a program make it difficult to determine the type of the dynamically allocated memory objects. To overcome this difficulty, we create one *heap object* instead of using the dynamic memory object naming scheme described in Section 3.1 for any program containing user-defined memory allocation subroutines.

The non-local control flow caused by setjmp/longjmp can be handled as follows: treat the longjmp statement as a return statement; then model a general program piece with setjmp in Figure 4(d) as the program piece in Figure 4(e). Our analysis also assumes signal handlers in a program do not affect global pointer values.

```
int *p1, **p2;
s1: p1 = (int *) (&p2);
s2: *p1 = ...; //modify p2
```

(a)

```
int c, *p1;
p1= (int*)c;
```

(b)

```
struct S1 {
   int a, b;
} s1;
struct S2 {
    char c[8];
}*ps2;

ps2 = (struct S2 *) (&s1);
... = ps2->c[i];
```

(c)

```
return_value = setjmp(buffer);
... other statements...
sub(); //sub has a long jmp statement
```

(d)

```
return_value = setjmp(buffer);
L1:
   ... other statements ...
   sub();
   if (a-unknown-condition)
      goto L1;
```

(e)

Figure 4: Handling Some Complicated Language Features

5 Experiments

We implemented the above algorithm in our Agassiz Compiler [11]. The implementation of both the compiler kernel and the alias analysis algorithm requires an object-oriented programming style which avoids dangling pointers and memory leakage. Unlike other approaches which do not take into account such considerations, this approach tends to increase memory usage and slow down the compiler.

To explore the trade-offs between the efficiency and the precision of alias analysis, we studied three flow-sensitive algorithms with different levels of context-sensitivity: (1) context-insensitive ($max\_call\_chain\_length$= 0); (2) context-sensitive with $max\_call\_chain\_length = 1$, which distinguishes the calling contexts by the most recent call site in the call-chains; (3) context-sensitive with an unlimited $max\_call\_chain\_length$.

For each of the above algorithms, we measured its speed and its memory usage. We also calculated the average number of target objects($AvgNumTarget$) for indirect reads and indirect writes to measure the precision of the algorithms. Our results were collected on a 200 MHz Intel Pentium Pro machine with 256MB main memory and 768MB swap space running Linux 2.1.132.

5.1 Benchmark Programs

Our benchmark suite contained a total of 14 programs: four from SPEC95, six from SPEC92, and four other pointer intensive programs [2]. Table 1 describes some important characteristics of the benchmark programs. These program characteristics are collected after the control-flow normalization phase, which may duplicate codes. As can be seen, the third column of the table lists the number of lines in each program while the fourth column reports the number of user-defined functions (including main) used in the program. The number of direct calls to user defined functions and the number of indirect calls via function pointers are shown in the next two columns. The following two columns present the number of recursive functions, and whether the programs contain recursive data structures or not. The maximum pointer level of the programs, which are also the number of subproblems for the analysis of the non-recursive data structures of the programs, are shown in the next-to-last column. The last column reports whether the programs contain irregular control flow or not.

5.2 Results and Discussion

Table 2 shows the speed and the memory usage for the three algorithms. The analysis speed in the third and fourth columns shows the analysis time(in seconds) and the number of lines analyzed by the algorithms per second. The fifth column gives the maximum memory used by the analysis, excluding the memory used by the

Bench-mark	Source	# lines	# u.d.f.	# call to u.d.f.	# call via f.p.	# rec. func	rec. d.s.	max p.l.	irr. cont.
alvinn	spec92 cfp	272	8	7	0	0	no	2	no
anagram	Austin [2]	646	15	22	0	1	no	3	no
ks	Austin [2]	782	13	16	0	0	yes	2	no
026.compress	spec92 cint	1503	14	83	0	0	no	2	no
129.compress	spec95 cint	1924	18	38	0	0	no	2	no
ft	Austin [2]	2157	27	47	0	0	yes	2	no
eqntott	spec92 cint	3457	59	543	11	9	yes	3	no
yacr2	Austin [2]	3979	51	156	0	5	no	2	no
ear	spec92 cfp	5239	68	140	0	1	no	3	no
sc	spec92 cint	8455	147	1008	2	20	yes	3	no
espresso	spec92 cint	14838	314	1621	15	27	yes	3	no
m88ksim	spec95 cint	19915	239	1158	3	3	yes	4	no
go	spec95 cint	29246	372	2099	0	1	no	3	no
ijpeg	spec95 cint	31249	271	391	446	28	yes	5	no

Table 1: Benchmark program characteristics (u.d.f.=user-defined-function; f.p.=function-pointer; rec.=recursive; d.s.=data structure; p.l.=pointer-level; irr.cont=irregular-control-flow;)

program intermediate representation(IR). The last column shows the ratio of the maximum memory used by the analysis to the memory used by IR. Presenting the memory usage this way allows us to distinguish the memory used by the compiler data structures from the memory used by the analysis.

As expected, Table 2 reveals that larger programs do not necessarily require more analysis time. Program go, for instance, is about twice as large as program $espresso$ (Table 1) but required an analysis time less than 1/11 that of program $espresso$. The fact that larger programs do not always require more analysis time can also be seen from the wide range of numbers in the fourth column. For example, the context-insensitive algorithm achieved 121 $\sim$ 1309 lines per second for 11 programs but only 18 lines per second for program $espresso$. The difference in the numbers of lines per second suggests that other factors, such as program structure complexity and analysis precision, may affect analysis time.

Table 2 also reveals that some programs are very sensitive to the exponential complexity problem while the other programs are not. For example,the context-sensitive analysis with $max\_call\_chain\_length = 5$ for program $espresso$ was approximately five times slower than the context-insensitive analysis of the program. Also, the memory used by the context-sensitive analysis with $max\_call\_chain\_length = 5$ for this program was more than three times that used by the context-insensitive analysis of the program. Moreover, the complexity for the analysis of program $espresso$ grew so rapidly that we were not able to complete its unlimited context-sensitive analysis and collect the result within twenty hours. On the other hand, increasing the $max\_call\_chain\_length$ did not increase the time and memory usage for the analysis of programs $alvinn$, $anagram$, ks, $yacr2$, ear, and go. In general, a program with most of its subroutines having than one call site and many pointer values propagated along deep call graph paths tends to be sensitive to the exponential complexity problem. Furthermore, small programs usually do not suffer from the exponential complexity problem.

Table 3 shows the average number of target objects($AvgNumTarget$) for indirect reads and indirect writes, as well as the total number of indirect reads and indirect writes. At run time, a pointer dereference should access at least one memory object. Thus, one is the lower bound for an $AvgNumTarget$. An analysis with an $AvgNumTarget$ close to one means that the algorithm is precise. However, a larger $AvgNumTarget$ may indicate either a reduced precision due to the algorithm, or that the pointer dereferences are actually accessing more than one memory object on average at run time.

As can be seen from Table 3, increasing the level of context-sensitivity resulted in precision improvement on

Bench-mark	Algo-rithm	Speed		Memory Usage(KB)	
		time (sec.)	lines per sec.	ana. max	ana : ir
alvinn	1	0.90	302	1100	0.25
	2	0.90	302	1108	0.25
	3	0.90	302	1108	0.25
anagram	1	1.40	461	1324	0.28
	2	1.41	458	1336	0.28
	3	1.41	458	1340	0.28
ks	1	2.18	359	1588	0.33
	2	2.21	354	1624	0.34
	3	2.21	354	1624	0.34
026.compress	1	2.98	504	1648	0.30
	2	3.21	468	2288	0.42
	3	3.67	410	2288	0.42
129.compress	1	1.47	1309	1328	0.19
	2	1.50	1283	7344	0.19
	3	1.53	1283	8496	0.21
ft	1	2.34	922	1648	0.28
	2	2.35	918	1672	0.29
	3	2.38	906	1680	0.29
eqntott	1	38.72	89	7132	0.74
	2	60.49	57	7688	0.80
	3	67.46	51	8296	0.86
yacr2	1	5.70	698	2432	0.33
	2	5.92	672	2604	0.35
	3	6.19	643	2796	0.37
ear	1	5.26	996	2784	0.36
	2	5.32	983	2900	0.38
	3	5.34	981	2968	0.39
sc	1	69.62	121	11040	0.55
	2	73.69	115	11040	0.55
	3	110.03	77	15880	0.79
espresso	1	840.61	18	67332	2.00
	2	1668.33	9	138292	4.11
	3∗	4179.14	4	216656	6.43
m88ksim	1	295.98	67	44808	0.62
	2	353.30	56	48452	0.67
	3∗	1203.58	17	62036	0.86
go	1	75.11	389	15404	0.36
	2	75.72	386	16252	0.38
	3	76.75	381	16260	0.38
ijpeg	1	145.41	214	15508	0.30
	2	167.31	187	15508	0.30
	3	1018.33	31	25680	0.49

Table 2: The speed and the maximum memory usage for the three algorithms in Section 5 ($3*$: the algorithm with max_call_chain_length=5, we were unable to collect the result for the algorithm with unlimited max_call_chain_length because of the huge invocation graph).

Bench-mark	Algo-rithm	indirect reads		indirect writes	
		AvgNumTarget	tot	AvgNumTarget	tot
alvinn	1	1.00	34	1.00	9
	2	1.00		1.00	
	3	1.00		1.00	
anagram	1	1.39	23	1.00	9
	2	1.39		1.00	
	3	1.39		1.00	
ks	1	2.08	113	1.00	2
	2	2.08		1.00	
	3	2.08		1.00	
026.compress	1	1.00	100	1.00	32
	2	1.00		1.00	
	3	1.00		1.00	
129.compress	1	1.13	30	1.06	32
	2	1.13		1.06	
	3	1.00		1.00	
ft	1	1.00	151	1.00	5
	2	1.00		1.00	
	3	1.00		1.00	
eqntott	1	1.26	1243	1.19	539
	2	1.25		1.18	
	3	1.25		1.18	
yacr2	1	1.25	393	1.89	81
	2	1.23		1.65	
	3	1.23		1.65	
ear	1	4.69	181	2.05	96
	2	3.87		1.68	
	3	3.87		1.68	
sc	1	1.31	1828	1.55	127
	2	1.30		1.38	
	3	1.30		1.38	
espresso	1	22.66	4322	31.00	1242
	2	22.08		28.96	
	3*	21.62		28.28	
m88ksim	1	7.36	1792	5.77	308
	2	6.11		3.15	
	3*	6.10		2.98	
go	1	17.10	78	9.73	38
	2	1.02		1.06	
	3	1.00		1.00	
ijpeg	1	1.43	3235	1.36	1275
	2	1.37		1.30	
	3	1.26		1.22	

Table 3: The average number of indirect read/write targets for the three algorithms in Section 5 and the total number of indirect reads and writes for each program.

Scheme	Algorithm	Time (sec.)	ana. memory (KB)	AvgNumTarget for indirect reads	AvgNumTarget for indirect writes
scheme-1	1	840.61	67332	22.40	29.89
	2	1668.33	138292	21.81	28.96
	3$*$	4179.14	216656	21.37	28.29
scheme-2	1	193.99	24864	2.17	2.10
	2	250.85	29712	2.13	2.02
	3$*$	467.28	45228	2.09	1.97

Table 4: Comparing two schemes in modeling the dynamic memory objects for the analysis of program $espresso$. (scheme-1: the scheme described in Section 3.1. scheme-2: create one heap MemObj.)

nine out of 14 programs. However, the degree of improvement varied significantly depending on the programs analyzed. The two algorithms with $max\_call\_chain\_length >= 1$ achieved an $AvgNumTarget$ close to one for go, a significant improvement over the context-insensitive version. This is because most of the pointer values in the program are passed only one step along the call graph. However, for the other five programs, $alvinn$, $anagram$, ks , $026.compress$ and ft, the three algorithms achieved the same precision. This suggests that either most of the dereferenced pointers are not interprocedurally related, or most of their subroutines are called with the same points-to values, or that the reduced precision is caused by the conservativeness in handling recursive data structures (programs ks and ft use recursive data structures extensively). For the remaining eight programs, increasing the level of context-sensitivity only slightly improved the precision of the analysis.

Thus, considering both the cost, including the analysis time and the peak memory usage, and the precision of the algorithms, the benefit of increasing the $max\_call\_chain\_length$ varies depending on the programs analyzed. For the programs where exponential cost is not likely a problem, increasing the $max\_call\_chain\_length$ to achieve the best precision result is a good choice. On the other hand, limiting the $max\_call\_chain\_length$ to a small number is a practical way to efficiently handle the programs that are sensitive to the exponential growth without sacrificing very much precision. Our approach allows users to select a prudent value of the $max\_call\_chain\_length$ which works best for each particular program. A similar approach has been used in most compilers for varying levels of optimization.

Table 4 compares two schemes in modeling the dynamic memory objects for the analysis of program $espresso$. The first scheme is the one used in our previous algorithm, which creates one or several dynamic MemObjs for each memory allocation statement (Section 3.1). The second scheme, on the other hand, creates one heap object for all the memory allocation statements. As revealed by the table, replacing the first dynamic memory object modeling scheme with the second one speeded up the context-insensitive algorithm by more than four times and the most context-sensitive algorithm by nine times. Memory usage was also significantly reduced in the three algorithms with the second scheme. Because the second scheme collapses all dynamic MemObjs into one heap object, the difference in the AgvNumTarget for the context-insensitive algorithm in both schemes suggests that: (1) approximately 21 out of the 22.4 AgvNumTargets for indirect reads in the first scheme are dynamic memory objects; (2) approximately 28 out of the 29.89 AgvNumTargets for indirect writes in the first scheme are dynamic memory objects. A similar situation was also found in the remaining two context-sensitive algorithms. Our other studies have shown that most of the dynamic objects in program $espresso$ are related to recursive data structures. This suggests that if an algorithm cannot precisely analyze recursive data structures, using the first dynamic memory object modeling scheme may complicate the analysis.

6 Other Related Work

We use a syntax-directed flow-sensitive algorithm to analyze each subroutine while most of the previous flow-sensitive alias analysis methods use an iterative algorithm. A syntax-directed flow-sensitive algorithm is generally considered to be more efficient than its iterative counterpart. However, whether the extra effort results in real saving in time has not been firmly established.

Hind et al. [14] presented another approach to improve the efficiency of an iterative flow-sensitive alias analysis algorithm. They provided empirical results to demonstrate the effectiveness of four techniques (shared alias sets, a working list, a sorted working list, and forward binding filters) in speeding up the iterative algorithm. However, a direct comparison of the speed of their algorithm with the speed of our flow-sensitive context-insensitive algorithm is difficult for four main reasons: (1) Our machine environment is different from theirs. (2) Our context-insensitive algorithm is implemented in a framework which can result in context-sensitive algorithms. The extra data structures and programs to support context-sensitivity may slow down the context-insensitive algorithm. (3) Our pointer analysis is implemented as an integral part of a scalar dataflow analysis. It not only collects points-to information for later non-pointer analysis, but also collects definition-use information for pointer variables. Their algorithm, on the other hand, only calculates points-to information. (4) Their algorithm is based on a sparse evaluation graph [5, 3] with only the pointer-related assignments and function calls in a program. Ignoring the non-pointer-related assignments this way can simplify the control-flow graph, thereby speeding up the analysis.

Both Zhang et al. [25, 26] and Ruf [20] presented a program decomposition alias analysis algorithm. They first divided object names into equivalent classes using pointer-related assignments. They then used the "prefix" relation [2] between the object names to draw dependency edges between equivalent classes. Finally, they constructed the subproblems out of the above dependency graph in different manners: Zhang et al. viewed each weakly connected component of the graph as a subproblem while Ruf turned the graph into a DAG by collapsing each strong connected component into one node and treated each node in the DAG as one subproblem. Compared to our approach, the above two approaches usually result in finer subproblems, thus resulting in a greater reduction in memory usage. However, it remains to be seen that finer subproblems will result in a faster analysis of the original problem.

Wilson and Lam's *partial transfer function (PTF)* approach [24, 23] also provides the flexibility to adjust the precision and the efficiency of their analysis. They provided four different criteria to decide whether to re-analyze a subroutine or to re-use an existing PTF. This leads to four variations of their algorithm with different levels of context-sensitivity. Unlike their approach, our approach controls the analysis via the number of the most recent call sites in the call-chains tagging the data flow values. Our approach can result in a spectrum of algorithms with unified semantics while their algorithm identifies only four variations. This may not be a distinctive advantage but certainly indicates a major difference in our approach.

In his studies of the precision of context-insensitive and context-sensitive algorithms, Ruf concluded that a context-sensitive algorithm did not provide any precision improvement over the context-insensitive version when considering only the relevant points-to information for the pointer dereferences in the programs [19]. He also suggested that his conclusion might be limited to his benchmark suite. On the other hand, our results showed that for nine out of 14 programs, the most context-sensitive algorithm demonstrated precision improvement over the context-insensitive method.

The way we handle context sensitivity is similar to previous approaches which tag the interprocedural dataflow values with *call strings*. However, unlike some previous approaches which use *k-limiting* to handle the potentially unlimited call strings in the presence of recursion[22, 16, 12], we combine the calling contexts

[2] or "pointed-to-by" relation as in [20]

for recursive calls as in [7].

The way we handle recursive data structures is not as precise as the algorithms addressed in [16, 8, 9]. We agree with Emami et al. [7] that the pointer analysis problem can be divided into two distinct subproblems: analyzing pointers to non-recursive data structures and analyzing pointers to recursive data structures. In this paper, we focus on analyzing pointers to non-recursive data structures. An improvement to our approach would be to incorporate a more precise recursive data structure analysis algorithm. This is possible because our hierarchical approach separates the analysis of recursive data structures from the analysis of other pointer levels.

7 Conclusions

We have presented a hierarchical alias analysis approach which not only offers varying levels of context-sensitivity but also enables the use of an efficient syntax-directed dataflow analysis technique. This hierarchical approach is based on the observation that a dereference from a memory object with pointer level n results in an access to one or more memory objects with pointer level $n-1$. Thus, we postpone the analysis of the memory objects for a certain pointer level until we have obtained the points-to information for all the higher pointer level memory objects. This approach can be extended to analyze non-pointer variables.

An implementation of our approach can result in a spectrum of context-sensitive alias analysis algorithms with a context-insensitive algorithm on one end and a full context-sensitive algorithm on the other. Our experimental results show that the precision improvement achieved by increasing the context-sensitive level of the analysis varies significantly depending on the programs analyzed. Furthermore, increasing the maximum call-chain length for a higher degree of context sensitivity may trigger an exponential complexity problem. Thus, it is very important for an algorithm to allow users to select a prudent context-sensitive level which works best for a particular program. By parameterizing the maximum call-chain length used in tagging the dataflow values, our approach is able to provide this type of flexibility.

Acknowledgments

This work was supported in part by the National Science Foundation under grant nos. MIP-9610379 and CDA-9502979; by the U.S. Army Intelligence Center and Fort Huachuca under contract DABT63-95-C-0127 and ARPA order no. D346, and a gift from the Intel Corporation.

References

[1] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley Publishing Company, 1986.

[2] Todd Austin. Pointer intensive programs, http://www.cs.wisc.edu/∼austin/austin.html.

[3] Michael Burke, Paul Carini, Jong-Deok Choi, and Michael Hind. Interprocedural pointer anlias analysis. In *IBM Research Report RC 21055*, 1997.

[4] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Proceedings of the 20th Annual SCM Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[5] Jong-Deok Choi, Ron Cytron, and Jeanne Ferrante. Automatic construction of sparse data flow evaluation graphs. In *Proceedings of the 18th ACM Symposium on the Principles of Programming Languages*, pages 55–66, January 1991.

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

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

[8] Rakesh Ghiya and Laurie J. Hendren. Connection analysis: A practical interprocedural heap analysis for c. In *Proceedings of the e 8th Workshop on Languages and Compilers for Parallel Computing*, 1995.

[9] 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 *Proceedings of the 23rd Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 1–15, January 1996.

[10] G.Ramalingam. The undecidability of aliasing. In *ACM Transaction on Programming Languages and Systems, Vol.16 ,No.5*, pages 1467–1471, September 1994.

[11] The Agassiz Compiler Group. http://www.cs.umn.edu/research/agassiz.

[12] Mary Jean Harrold and Mary Lou Soffa. Efficient computation of interprocedural definition-use chains. In *ACM Transactions on Programming Languages and Systems Vol.16, No.2*, volume 16, pages 175–204, March 1994.

[13] Michael Hind and Anthony Pioli. An empirical comparison of interprocedural pointer alias analyses. In *IBM Research Report RC 21058*, 1997.

[14] Michael Hind and Anthony Pioli. Accessing the effects of flow-sensitivity on pointer alias analyses. In Giorgio Levi, editor, *Lecture Notes in Computer Science, 1503*, pages 57–81. Springer-Verlag, 1998. Proceedings from the *5th International Static Analysis Symposium*.

[15] William Landi and Barbara G. Ryder. Pointer-induced aliasing: A problem classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of rogramming Languages*, pages 93–103, January 1990.

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

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

[18] H. D. Pande, W. Landi, and B. G. Ryder. Interprocedural def-use associations for c systems with single level pointers. In *IEEE Transactions on Software Engineering*, volume 20(5), pages 382–403, May 1994.

[19] Erik Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–31, June 1995.

[20] Erik Ruf. Partitioning dataflow analyses using types. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 15–26, January 1997.

[21] Patrick W. Sathyanathan and Monica S. Lam. Context-sensitive interprocedural analysis in the presence of dynamic aliasing. In *Proceedings of the First SUIF Compiler Workshop*, pages 1–20, 1996.

[22] M. Sharir and A. Pnueli. Two approaches to interprocedural data flow analysis. In S. Muchnick and N. Jones, editors, *Program Flow Analysis : Theory and Applications*, pages 189–236. Prentice Hall, 1981.

[23] Robert P. Wilson. Efficient context-sensitive pointer analysis for c programs. In *PhD Thesis, Stanford University*, December 1997.

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

[25] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step towards practical analyses. In *Proceedings of the 4th Symposium on the Foundations of Software Engineering*, October 1996.

[26] Sean Zhang, Barbara G. Ryder, and William Landi. Experiments with combined analysis for pointer aliasing. In *Proceedings of Workshop on Program Analysis for Software Tools and Engineering*, June 1998.