Discovering Context Information for Efficient and Accurate Program Analyses

Donglin Liang and Mary Jean Harrold
College of Computing
Georgia Institute of Technology
Atlanta, GA 30332, USA
{dliang,harrold}@cc.gatech.edu

Abstract

A program analysis may compute spurious information for a program with pointer variables, when the analysis uses alias information provided by efficient flow-insensitive alias analysis algorithms, because the program analysis cannot distinguish the alias relations that hold in a procedure when the procedure is invoked at a specific callsite. This paper presents a technique that determines whether a memory location might be accessed through a pointer dereference at a statement in a procedure when the procedure is invoked at a specific callsite. Empirical studies show that our technique may effectively improve the precision and efficiency of program analyses.

1 Introduction

Many software engineering tools require program analyses to extract information about programs. To compute safe program information for programs written in languages such as C that make extensive use of pointer variables, a program analysis requires alias[1] information that determines the memory locations that may be accessed through pointer dereferences. For example, a program analysis will incorrectly conclude that statement 24 in Figure 1 does not modify `h_11`, the memory allocated by `malloc()` at statement 11 (we refer to the memory returned by `malloc()` at statement s as `h_`s), if aliasing effects are ignored.

Alias information can be computed using an alias-analysis algorithm. To facilitate program analyses on large programs, an alias-analysis algorithm must be efficient [2]. Previous research has resulted in flow-sensitive and flow-insensitive alias analysis algorithms. Flow-sensitive alias-analysis algorithms (e.g., [4, 7]) compute very precise alias information for a program. However, because the flow-sensitive alias-analysis algorithms consider the execution order of the statements, these algorithms must iteratively propagate information throughout the program using the control-flow graph. Studies indicate that these algorithms may be too expensive to be applied to large programs [9, 12]. Flow-insensitive alias-analysis algorithms (e.g., [1, 11]) compute less precise alias information for a program. These algorithms avoid iteratively propagating information throughout the program by ignoring the execution order of the statements. Studies indicate that these algorithms are more efficient and thus, scale better to large programs, than the flow-sensitive algorithms [9, 12].

```
1  char *buf1, *buf2;
2  main() {
3    char in[100];
4    read(in);
5    init1(in);
6    read(in);
7    init2(in);
8  }
9  void init1(char *p1){
10   char *t1;
11   buf1=malloc(100);
12   t1=cpys(buf1,p1);
13   cpys(t1,"buf1.");
14 }
15 void init2(char *p2){
16   char *t2;
17   buf2=malloc(200);
18   t2=cpys(buf2,p2);
19   *t2 = 0xFF;
20 }
21 char *cpys(char *dst,
          char *src) {
22   while(*src)
23     *(dst++)=*(src++);
24   *dst=0;
25   return dst;
26 }
```

Figure 1: Example program.

A program analysis might compute imprecise program information when the analysis uses alias information provided by a flow-insensitive alias-analysis algorithm. One way that this imprecision occurs is that, to resolve pointer dereferences in a program, the program analysis may use the large amount of spurious alias relations that are reported by the flow-insensitive alias analysis algorithm but that might never hold when the program is executed. For example, Steensgaard's algorithm [11] reports that `*t2` may be aliased to both `h_11` and `h_17` in Figure 1. Using this information, a program analysis would report that statement 19 may modify memory locations `h_11` and `h_17`. However, `*t2` is never aliased to `h_11` when the program is executed. Thus,

[1] *Aliasing* occurs when two different names can access the same memory location at a program point.

statement 19 does not modify h_11. There has been significant effort to develop techniques that improve the precision of flow-insensitive alias-analysis algorithms. This effort has resulted in algorithms (e.g., [9]) that report alias relations that are close in precision to flow-sensitive algorithms but that run in a reasonable amount of time for large programs.

Another way that a program analysis might compute imprecise program information is that, when the program analysis analyzes a procedure to compute the information for a specific callsite, it might use alias relations that hold under the callsites other than this specific callsite. This imprecision occurs because many efficient flow-insensitive alias analysis algorithms do not report the callsites under which an alias relation holds. For example, using alias information provided by the flow-insensitive, context-sensitive points-to analysis algorithm (FICS) [9], a program analysis discovers that, in the program in Figure 1, both h_11 and h_17 may be modified at statement 24 in cpys() because both memory locations may be aliased to *dst. Thus, the program analysis reports that both h_11 and h_17 may be modified after statement 18 is executed. However, h_11 is not aliased to *dst when cpys() is invoked at statement 18. Therefore, h_11 is not modified after statement 18 is executed.

To reduce the imprecision introduced into a program analysis in the latter way due to the lack of calling context information in the alias information, we proposed *light-weight context recovery* [10]. This technique efficiently identifies, in a procedure P, each memory location l that is accessed exclusively through the same dereference of a formal parameter f. If the values of the memory locations needed to resolve this dereference do not change in P, then l is accessed in P when P is invoked at a callsite c only if l is aliased to the dereference of the corresponding actual parameter of f. A program analysis then can use this information to reduce the spurious information propagated across the procedure boundaries: the program analysis propagates, from a callsite to the called procedure or from the called procedure to the callsite, only information for the memory locations that may be accessed by a procedure under the context of the callsite. Empirical studies showed that, on many programs, light-weight context recovery can effectively reduce the spurious information computed by program analyses. However, because light-weight context recovery considers only memory locations accessed through dereferences of formal parameters, this technique may not be able to improve a program analysis in some important cases.

To consider the memory locations accessed through dereferences of pointers that are not formal parameters, we developed a new technique. The technique analyzes the ways in which the address value v that is returned at a statement s by language constructs such as the "&" operator or malloc() is propagated and used in a program. To distinguish v from the same address value returned at other statements, we annotate v with the statement number of s. For convenience, we call such an annotated address value a *reference*, and we say this reference is created at s. Let r be a reference that is used to access memory location l at a statement s' in a procedure P. If r is created at a statement outside P and passed into P through formal parameters, then l can be accessed at s' under the context of a callsite only if l may be aliased to dereferences of actual parameters at the callsite. For example, at statement 24 in Figure 1, h_11 is accessed using a reference that is created at statement 11 and passed into cpys() through formal parameter dst. We can conclude that h_11 may be accessed at statement 24 when cpys() is invoked at statement 12 because h_11 may be aliased to *buf1, a dereference of actual parameter buf1. We can also conclude that h_11 is not accessed at statement 24 when cpys() is invoked at statement 18 because h_11 is not aliased to any dereference of actual parameters buf2 and p2.

This paper presents our technique that extends FICS to identify, in a procedure, the memory locations accessed using references that are passed into the procedure through formal parameters. The technique also extends FICS to compute the memory locations that may be aliased to dereferences of actual parameters at each callsite. The paper then illustrates, using interprocedural modification side-effect analysis, an approach that uses the additional information provided by the extended FICS to improve the precision and efficiency of program analyses. The main benefit of our technique is that it is efficient: it will not increase the complexity of FICS in theory; it adds little extra cost to FICS in practice. Another benefit of our technique is that it is effective: it identifies memory locations accessed in a procedure at a higher precision level than the light-weight context recovery. Thus, our technique may significantly improve the precision and efficiency of many program analyses.

The paper also presents a set of empirical studies in which we investigate the effectiveness of our technique. These studies show that:

- For many programs we studied, our technique identifies more than 40% of nonlocal memory locations in a procedure as being modified using references passed into the procedure through formal parameters.
- For many programs we studied, our technique significantly improves the precision of modification side-effect analysis.
- For several programs we studied, our technique computes modification side-effect information close in precision to that computed using Landi, Ryder, and Zhang's algorithm [8], which requires conditional alias information that can be provided only by expensive alias analysis algorithms (e.g., [7]).
- For many programs we studied, our technique computes more precise modification side-effect information than light-weight context recovery.

These results suggest that our technique might effectively improve the precision and efficiency of program analyses.

2 Discovery of Context Information

This section presents an algorithm that computes ontext information that can be used to determine whether a memory location can be accessed at a statement in a procedure under the context of a callsite. First, we introduce some definitions. Then, we will give some intuition about context discovery. Finally, we present the algorithm. (** I will have to distinguish flow-sensitive or flow-insensitive algorithm **)

2.1 Definitions

Memory locations in a program are accessed through *object names*, each of which consists of a variable and a possibly empty sequence of dereferences and field accesses [7]. A *direct* object name contains no dereference whereas an *indirect* object name contains at least one dereference. A memory location is *directly accessed* if it is accessed through a direct object name; otherwise, it is *indirectly accessed*. Given an indirect object name N, the *alias set* of N is the set of memory locations that may be aliased to N.

Object name N_1 is *extended* from object name N_2 if N_1 can be constructed by applying a possibly empty sequence of dereferences and field accesses ω to N_2; in this case, we denote N_1 as $\mathcal{E}_\omega\langle N_2\rangle$. For example, suppose that p is a pointer that points to a **struct** with field a (in the C language). Then $\mathcal{E}_*\langle p\rangle$ is $*p$, $\mathcal{E}_*\langle *p\rangle$ is $**p$, and $\mathcal{E}_{*.a}\langle p\rangle$ is $(*p).a$. Given an object name N, the *reachable set* of N is the set of memory locations that are aliased the object names extended from N with at least one dereference.

2.2 Intuition

Using alias information, a program analysis can identify memory locations that are indirectly accessed through pointer dereferences at a statement in a procedure. Additional information can help the program analysis to further determine whether a memory location is accessed when the procedure is invoked under a specific callsite.

Without global pointers. To simplify the discussion, we first assume that the program has no global pointers. Under this assumption, if memory location l is indirectly accessed at a statement s in procedure P, then l must be accessed using a reference that can be created in three ways.

1. l is accessed using a reference that is created at a statement in P. For example, in Figure 2(a), `x` is accessed at statement 4 using the reference created at statement 3.
2. l is accessed using a reference that is created at a statement in a procedure Q that is called by P, and this reference can be returned to P by Q. For example, in Figure 2(b), `x` is accessed at statement 4 using the reference created at statement 7 and returned from `g()`.
3. l is accessed using a reference that is created before P is invoked, and this reference can be passed into P through formal parameters. For example, in Figure 2(b), `p` is accessed at statement 7 using the reference created at statement 3 and this reference is passed through formal parameter `pp` into `g()`.

Knowledge of the way in which a reference is created, if the reference may be used to access l indirectly at s in P, can help a program analysis estimate whether l may be accessed at s when P is invoked at a specific callsite c. If the reference used to access l at s is created in the first or second way, then, without knowledge of the the dynamic behavior, a program analysis must assume that l may be accessed at s regardless of the callsite under which P is invoked. If the reference used to access l at s is created in the third way, then a program analysis can estimate whether l may be accessed at s under the calling context c by examining the alias information at c. Let A_c be the set of memory locations that may be aliased to indirect object names extended from the actual parameters at c. If a reference of l is created before P is invoked at c, then such reference can be passed into P through formal parameters under c only if $l{\in}A_c$. Thus, if l is accessed using a reference created in the third way, then l may be accessed at s under c only if $l{\in}A_c$.

Let l be a memory location that may be indirectly accessed at statement s in procedure P. Let r be the reference that is used to access l at s. A program analysis can statically estimate, using a flow-insensitive approach, the possible ways in which a reference r may be created. If a statement in P creates a reference of l, then without tracing the propagation of the created reference, the program analysis assumes that r might be created at this statement. Thus, the program analysis concludes that r may be created in the first way. Further more, if a statement in a procedure Q, which is called by P, creates a reference of l and such reference may be returned to P after Q is executed, then the program analysis also assumes that r may be created at this statement. Thus, the program analysis concludes that r may be created in the second way. If either case occurs, the program analysis assumes that l might be accessed at s under each calling context. However, if neither of these cases occurs, then r must be created before P is invoked and r must be passed into P through formal parameters. In this case, the program analysis determines that l might be accessed at s under a specific callsite c only if $l{\in}A_c$.

Because the program analysis does not trace the propagation of the references, it might report spurious information. However, this flow-insensitive nature makes the technique more efficient. In addition, this technique can be used with alias information provided by any algorithm. This generality further makes our technique to be practical.

With global pointers. When a program contains global pointers, memory locations may be accessed through dereferences of such pointers. A global pointer can be assigned at any statement in the program. Without tracing the propa-

1 f() {
2 int *p;
3 p=&x;
4 *p=0;
5 }
	1 f() {
	2 int *p;
	3 g(&p);
	4 *p=0;
	5 }
	6 g(int **pp) {
	7 *pp=&x;
	8 }
		1 f() {
		2 q=&x;
		3 g();
		4 }
		5 g() {
		6 int **t;
		7 t=&q;
		8 **t=0;
		9 }

(a)	(b)	(c)

Figure 2: Example program fragments that use pointers.

gation of the value of a global pointer g, it is difficult to determine the value of g at a statement s in procedure P when P is invoked under a specific callsite. Thus, the program analysis assumes that the same set of memory locations can be accessed through the dereferences of global pointers at a statement in a procedure under the context of different callsites. If a memory location l is indirectly accessed through an object name extended from a global variable at s in P, then the program analysis assumes that l is accessed at s under each calling context.

The presence of global pointers in a program also affects the technique that estimates whether a memory location may be accessed under a specific callsite through dereferences of local pointers and formal parameters. Let s be a pointer-related assignment in procedure P and $rhs\langle s\rangle$, the right-hand side of s, is an object name extended from a global variable. If $lhs\langle s\rangle$, the left-hand side of s, may be aliased to an object name N that is extended from a local pointer variable, then after the execution of s, the memory location in the reachable set of $rhs\langle s\rangle$ can be accessed through proper extensions of N. Therefore, given any memory location l in the reachable set of $rhs\langle s\rangle$, if l is indirectly accessed through an object name extended from a local variable at another statement s_1 in P, then the program analysis assumes that l is accessed at s_1 under each calling context. For example, in the program fragment in Figure 2(c), statement 7 assigns global pointer q to local pointer t. Therefore, after the executation of statement 7, x, the memory location in the reachable set of q, can be accessed in g() through dereference of t. Thus, the program analysis concludes x might be accessed by g() under each calling context.

Let s be a pointer-related assignment in procedure P and $rhs\langle s\rangle$ takes the address of global pointer g. Then, based on arguement similar to that in the previous paragraph, the program analysis assumes that memory locations in g's reachable set might be accessed by P under any calling context.

Summary. To summarize the above discussion, we introduce several definitions.

Leaked Set of Callsite c: The set of memory locations that may be aliased to object names that are extended from actual parameters at c at the program point before c. This set includes the memory locations whose references may be passed into the called procedure through formal parameters.

Context-Independent Set of Procedure P: The set of memory locations l that meet one of the following conditions: (a) a reference of l is created in P; (b) a reference of l is created in procedures called by P and this reference may be returned to P through actual parameters or the return value; or (c) a reference of l can be passed into P through global pointers.

Context-Dependent Set of Procedure P: The set of memory locations l that meet both of the following conditions: (a) a reference of l can be passed into P through formal parameters; and (b) l is not in the context-independent set of P.

We use $LEAK[c]$ to denote the leaked set of callsite c, $CI[P]$ to denote the context-independent set of procedure P, and $CD[P]$ to denote the context-dependent set of procedure P. If l is indirectly accessed through a dereference of nonglobal pointers at a statement in P, then $l \in CI[P] \cup CD[P]$.

Let l be the memory location that is identified as being accessed at a statement s in procedure P. (1) If l is directly accessed, then l is assumed to be accessed at s under each calling context. (2) If l is indirectly accessed through an object name extended from global pointers, then l is assumed to be accessed at s under each calling context. (3) If l is indirectly accessed through an object name extended from non-global pointers, then: if $l \in CI[P]$, then l is assumed to be accessed at s under each calling context; otherwise, l may be accessed at s when P is invoked at a callsite c only if $l \in LEAK[c]$. Therefore, if we can compute the leak set for each callsite and the context-indenpendent set for each procedure, we can determine whether a memory location can be accessed in a procedure under a specific calling context.

Given a callsite c, the leak set $LEAK[c]$ can be computed by merging the reachable sets of the the actual parameters at c. In next subsection, we discuss an algorithm that computes the context-indenpendent sets for the procedures in a program.

2.3 Computation of the Context-Indenpendent Sets

Algorithm ComputeCI in Figure 3 processes the procedures in a reverse topological (bottom-up) order to compute the context-indenpendent set for each procedure. ComputeCI computes two sets of memory locations for each procedure P. The first set of memory locations, $ThruGlobal_P$, contains all the memory locations whose references may be passed into P through global variables. The second set of memory locations, $Heap_P$, contains the memory locations that are allocated from heap in P or in procedures called by P. $Heap_P$ excludes the memory locations that are allocated from heap in the procedures called by P but whose addresses cannot be returned to P because such memory locations cannot be accessed in P. The context-independent set for P is

algorithm ComputeCI($\mathcal{P}$)
input $\mathcal{P}$: a program
global $ThruGlobal_P$: the memory locations whose refereces may be passed into P through global variables
$Heap_P$: the memory locations allocated from heap in P or in procedures called by P
declare W: list of procedures sorted in reverse topological order
begin ComputeCI
1. **foreach** procedure P **do** /*Intraprocedural phase*/
2. **foreach** statement s in P **do**
3. **case** s **do**
4. pointer assignment or memory allocation statement:
5. processExp(s, $rhs\langle s\rangle$)
6. call statement:
7. **foreach** actual parameter a of pointer type **do**
8. processExp(s, a)
9. **endfor**
10. **endcase**
11. **endfor**
12. add P to W
13. **endfor**
14. **while** $W \neq \phi$ **do** /*Interprocedural phase*/
15. take the first procedure P from W
16. **foreach** callsite c to R in P **do**
17. **foreach** memory location loc in $ThruGlobal_R$ **do**
18. **if** nonlocal(loc, c, R) **then**
19. add loc to $ThruGlobal_P$
20. **endif**
21. **endfor**
22. **foreach** memory location loc in $Heap_R$ **do**
23. **if** returned(loc,R) **then**
24. add loc to $Heap_P$
25. **endif**
26. **endfor**
27. **endfor**
28. **if** $ThruGlobal_P$ or $Heap_P$ updated **then** add P's callers to W
29. **endwhile**
end ComputeCI

procedure processExp(s, e)
input s: a statement
e: a pointer expression or a call to memory allocation function
declare P_s: the procedure containing s
begin processExp
30. **case** e **do**
31. &g where g is a global:
32. add g and memory locations in g's reachable set to $ThruGlobal_{P_s}$
33. an object name N extended from a global:
34. add memory locations in N's reachable set to $ThruGlobal_{P_s}$
35. a call to memory allocation function:
36. add the memory location returned by e to $Heap_{P_s}$
37. **endcase**
end processExp

Figure 3: Algorithm ComputeCI.

the union of $ThruGlobal_P$ and $Heap_P$. (*** Why do we do it in this way? ***)

ComputeCI consists of two phases: intraprocedural phase and interprocedural phase. In the intraprocedural phase, ComputeCI processes the statements in each procedure to collect the information from the statements. In the interprocedural phase, ComputeCI propagates information from the called procedures to the calling procedures to further compute the context-independent sets. In the presence of recursion, the interprocedural phase may iterate through procedures involved in recursion and compute a fixed point.

In the intraprocedural phase (lines 1–13), ComputeCI processes each statement s in each procedure P. If s is a pointer assignment or a memory allocation statement (line 4), ComputeCI invokes processExp() to process the right side of the statement (line 5). Otherwise, if s is a call statement (line 6), ComputeCI invokes processExp() to process each actual parameter of pointer type (line 7).

processExp() takes a statement s and a pointer expression e as inputs. If e takes the address of a global variable g, then processExp() adds g and the memory locations in g's reachable set to the $ThruGlobal$ set of P_s, the procedure that contains s. If e is an object name N extended from a global variable, then processExp() adds the memory locations in N's reachable set to the $ThruGlobal$ set of P_s. Otherwise, if e is a call to a memory allocation function, then processExp() adds the memory location returned by e to $Heap$ set of P_s.

In the interprocedural phase, ComputeCI processes each callsite c to procedure R in each procedure P using a worklist W. There are two major steps. In the first step, ComputeCI processes memory locations in $ThruGlobal_R$. For each memory location loc in $ThruGlobal_R$, if loc is nonlocal to R, then loc might also be accessed in P. ComputeCI adds loc to $ThruGlobal_P$. Otherwise, loc is local to R, and thus, cannot be accessed in P. ComputeCI does not add loc to $ThruGlobal_P$.

ComputeCI calls nonlocal() function to determine whether a memory location loc is nonlocal to a procedure R when R is invoked under callsite c. In the absence of recursion, the memory represented by loc is local to R if loc is a variable declared in R. In the presence of recursion, a variable declared in a procedure might represent memory allocated in the activate records of different invocations of the same procedure. Thus, even if loc is a variable declared in R, the memory represented by loc might not be local to R. nonlocal() considers the following two cases. In the first case, c is not a recursive call. (*** Might need to define what is a recursive call ***) nonlocal() returns false if loc is a variable declared in R. Otherwise, nonlocal() returns true. In the second case, c is a recursive call. nonlocal() returns false if (a) loc is a variable declared in P, (b) loc is not in the reachable set of any global variable at P's entry, and (c) loc is not in the leaked set of c. Otherwise, nonlocal() returns true.

In the second step of the interprocedural phase, ComputeCI processes $Heap$ sets. For each memory location loc in $Heap_R$, if loc may be returned to R's callers, then loc may be accessed in P. ComputeCI adds loc to $Heap_P$. Otherwise, loc cannot be accessed in P. ComputeCI does not add loc to $Heap_P$

ComputeCI calls function returned() to determine

whether a memory location may be returned to the callers of a procedure. Let loc be a memory location allocated in procedure P or in procedures called by P. `returned`(loc,P) is true if (a) loc is in the reachable set of a global variable at the exit of P, or (b) loc is in the reachable set of P's return value or P's formal parameters at the exit of P.

3 Empirical Studies

We performed several empirical studies to evaluate the effectiveness of our technique for improving the precision of program analyses. We implemented the extended FICS algorithm and the algorithms that compute the modification side-effects using PROLANGS Analysis Framework (PAF) [5]. In the studies, we also computed the modification side-effects using Landi, Ryder, and Zhang's algorithm (LRZ) [8]. We used the implementation of LRZ provided with PAF. We gathered the data for the studies on a Sun Ultra 10 workstation with 256MB physical memory and 640MB virtual memory. The left side (2nd–4th columns) of Table 1 shows the number of nodes in the control flow graph (*Nodes*), the number of lines of code including comments (*LOC*), and the number of procedures (*Procs*) in each of the subject programs. The subject programs are sorted on *Nodes*.

	Subject Size			*Time for alias analysis*		
program	*Nodes*	*LOC*	*Procs*	T_0	T_E	*Incr(%)*
loader	819	1132	32	0.10	0.11	10.00
pokerd	956	1153	30	0.09	0.09	0.00
ansitape	1087	1596	37	0.13	0.14	7.69
dixie	1357	2100	52	0.14	0.15	7.14
learn	1596	1600	50	0.14	0.16	14.29
unzip	1892	4075	42	0.10	0.12	20.00
diff	1932	1730	44	0.15	0.17	13.33
assembler	1993	2510	58	0.31	0.34	9.68
smail	2430	3212	59	0.33	0.36	9.09
patch	2695	2648	57	0.21	0.22	4.76
simulator	2992	3558	114	0.24	0.26	8.33
flex	3762	6902	93	0.27	0.30	11.11
arc	3955	7325	116	0.34	0.36	5.88
bc	4347	6654	101	0.43	0.44	2.33
space	5601	11474	137	1.00	1.06	6.00
bison	6533	7893	134	0.53	0.56	5.66
larn	11796	9966	295	0.78	0.81	3.85
mpeg_play	11864	17263	135	1.42	1.57	10.56
espresso	15351	12864	306	3.44	3.59	4.36
moria	20316	25002	482	3.07	3.07	0
twmc	22167	23922	247	3.13	3.42	9.27

Table 1: Left: information about the subjects, and Right: time in seconds for FICS (T_0) and extended FICS (T_E).

3.1 Study 1

The purpose of this study is to evaluate the efficiency of the extended FICS algorithm. We investigated the extra time introduced in FICS by our extension that computes the leaked sets and the context-dependent sets. We compared the time required to run FICS without extension (T_0) and the time required to run FICS with extension (T_E). We also measured in percentage the extra time introduced by the extension ($Incr$). $Incr$ is defined as $(T_E - T_0) * 100/T_0$.

The right hand side (5th–7th columns) in Table 1 shows T_0, T_E, and $Incr$. The table shows that extending FICS to compute the leaked sets and the context-dependent sets adds an average of 8% extra cost to the original algorithm. Out of 21 programs shown in the table, only 5 programs have more than 10% extra cost. Thus, the extended FICS can efficiently compute the leaked sets and the context-dependent sets.

3.2 Study 2

The purpose of this study is to evaluate the effectiveness of our technique in improving the precision of computing modification side-effects for the callsites (MOD at callsites). We compared the average size of MOD at callsites computed using our technique (`CD`) with the average size of MOD at callsites computed using three other methods. The first method computes MOD at callsites using the traditional algorithm (`CI`) that collects memory locations modified in a procedure and then propagates the nonlocal memory locations on the procedure call graph. Because the technique presented in this paper improves `CI`, comparing the results of `CI` with the results of `CD` shows the effectiveness of our technique in improving the precision of computing MOD at callsites. The second method computes MOD at callsites using information computed by *context recovery* (`CR`) [10]. We want to compare the effectiveness of the technique presented in this paper with the effectiveness of context recovery in improving program analyses. The third method computes MOD at callsites using Landi, Ryder, and Zhang's algorithm (`LRZ`) [8]. `LRZ` takes advantages of the conditional alias information provided by Landi and Ryder's alias analysis algorithm [7], and computes very precise MOD at callsites for a program. The results computed by `LRZ` serve as a lower bound to that computed using our technique.

The graph in Figure 4 shows the results of this study. From left to right in each group, the dark bar, the grey bar, the white bar, and the bar filled with slanted lines represent the data computed using `LRZ`, `CD`, `CR`, and `CI`, respectively. The results for `CD`, `CR`, and `CI` are computed with alias information provided by FICS. The results for `LRZ` on some programs are not reported because Landi and Ryder's algorithm does not terminate on these programs within 10 hours, the time limit we set for analyses.

From the graph, we can see that using context-dependent sets and leaked sets can significantly improve the precision of modification side-effects computed for the callsites. In fact, for `loader`, `pokerd`, `assembler`, `simulator`, and `space`, the results computed by `CD` are very close to those computed by `LRZ`. This indicates that the technique presented in this paper may effectively improve the precision of many program analyses. Note that because, in some cases, Landi and Ryder's algorithm treats a structure in the same

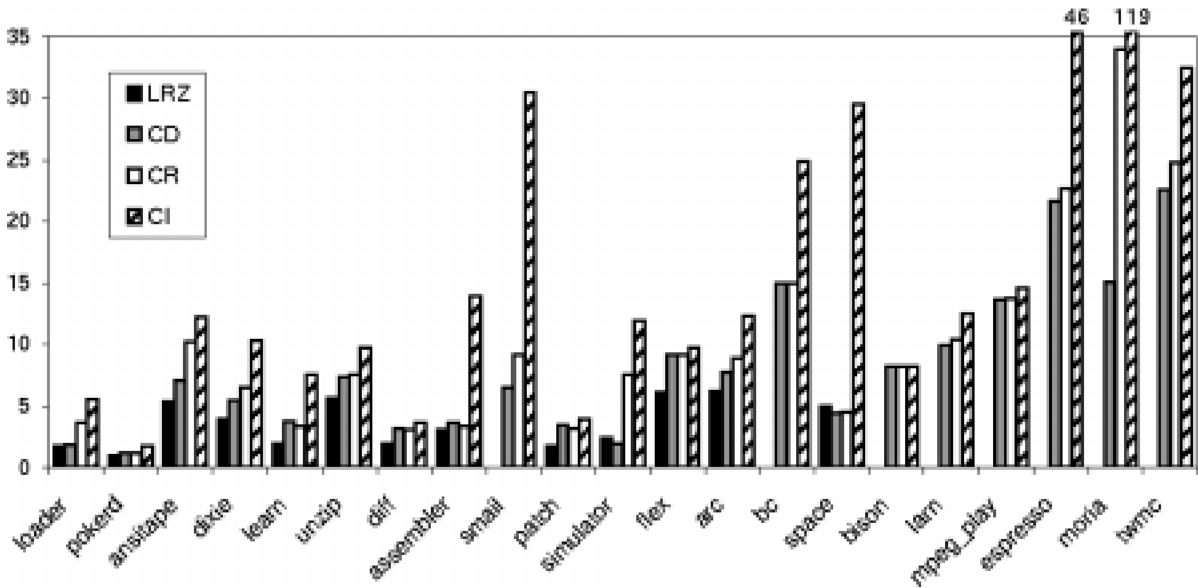

Figure 4: Average Size of MOD at callsites.

way as its fields in alias analysis, LRZ computes larger MOD at callsites than CD for simulator and space.

From the graph, we can also see that, for many of the programs we studied (e.g., ansitape, smail, and moria), CD computes a much smaller MOD at callsites than CR. This result indicates that, generally, the technique presented in this paper is more effective than context recovery in improving the precision of program analyses. However, we also see that, for some programs (e.g., assembler and learn), CD computes a slightly larger MOD at callsites than CR. This suggests that, to get the most precise information, we might have to combine the results computed by both techniques. Our future work includes combining both techniques to further improve program analyses.

In this study, we also compared the time required to compute the MOD at callsites by CD, CI, and CR. We do not compare the time required to compute MOD at callsites by LRZ because the differences in the implementation might significantly affect the accuracy of our comparison. Table 2 shows the results. From the table, we can see that, for most

program	Time(seconds) CD	CI	CR	program	Time(seconds) CD	CI	CR
loader	0.03	0.04	0.07	pokerd	0.05	0.04	0.08
ansitape	0.07	0.07	0.11	dixie	0.06	0.07	0.12
learn	0.07	0.07	0.11	unzip	0.10	0.09	0.15
diff	0.09	0.08	0.13	assembler	0.12	0.15	0.24
smail	0.17	0.20	0.36	patch	0.12	0.10	0.16
simulator	0.15	0.25	0.39	flex	0.22	0.20	0.36
arc	0.26	0.25	0.52	bc	0.25	0.27	0.50
space	0.48	0.79	1.17	bison	0.37	0.34	0.56
larn	1.05	1.14	2.01	mpeg_play	0.68	0.63	2.70
espresso	2.23	2.99	6.36	moria	7.73	25.23	28.86
twmc	1.84	1.76	3.23				

Table 2: Time to compute MOD at callsites.

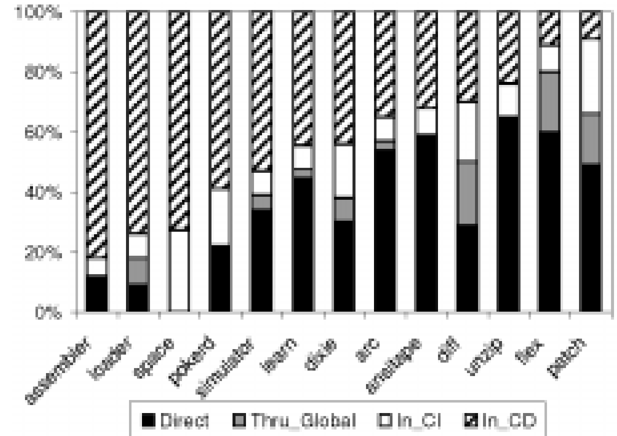

Figure 5: Frequencies of access classifications.

programs we studied, CD runs faster than CI and CR because CD propagates less information across the procedure boundaries than CI and CR. For example, it takes less than 8 seconds for CD to compute MOD at callsites for moria. However, it takes more than 25 seconds for CI and CR to compute this information. The result indicates that using the context-dependent sets and leaked sets might also improve the efficiency of program analyses.

3.3 Study 3

The purpose of this study is to evaluate the potential effectiveness of our technique on improving the precision and efficiency of program analyses. We investigate the frequencies of classifying a nonlocal memory location as directly accessed (Direct), indirectly accessed through global pointer dereferences (Thru_Glob), indirectly accessed through nonglobal pointer dereferences and in the context-independent set (In_CI), and indirectly accessed through nonglobal pointer

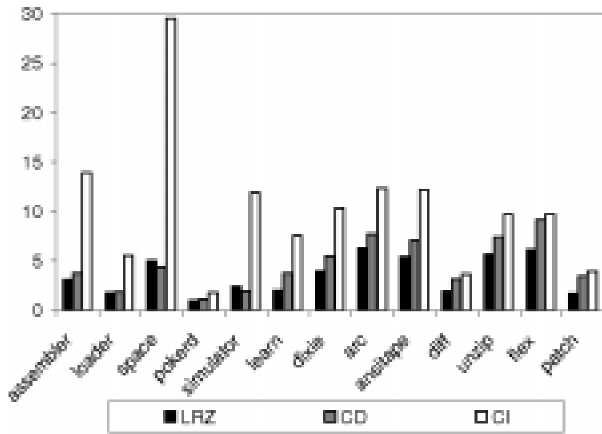

Figure 6: Average size of MOD at callsites.

dereferences and in the context-dependent set (In_CD), when the memory location is identified accessed at a statement in a procedure P. Because our technique improves the precision and efficiency of a program analysis on a program only if there are memory locations classified as In_CD, we expect that the frequency of classifying a memory location as In_CD indicates the effectiveness of using our technique to improve the analysis on this program. To verify this hypothesis, we also investigate the correlation between the frequency of classifying a memory location as In_CD and the improvement on modification side-effect analysis.

Figure 5 shows the average frequencies with which a nonlocal memory location accessed at a statement is classified into each of the four classes; Figure 6 shows the average size of MOD at callsites computed using the LRZ algorithm (LRZ), our algorithm (CD), and the traditional context-insensitive algorithm (CI). In Figure 5, from the bottom to the top on each bar, the segments represent the results for Direct, Thru_Global, In_CI, and CI respectively. In Figure 6, from left to right, the bars in each group represent the results for LRZ, CD, and CI. The programs shown in the graphs are ordered according to the frequency of classifying a memory location as In_CD. Note that some of the programs are not shown in the graphs because the results for LRZ are not available for these programs.

The graphs in Figure 5 and Figure 6 suggest that there is a strong correlation between the frequency of classifying a memory location as In_CD and the effectiveness of our technique in improving the precision of computing MOD at callsites. For example, for programs such as assembler, loader, and space in which a nonlocal memory location accessed at a statement is frequently classified as In_CD, CD computes MOD at callsites almost as precise as LRZ. However, for programs such as flex and patch in which a nonlocal memory location accessed at a statement is rarely classified as In_CD, CD does not make much improvement over CI. These results support our hypothesis, which states that the frequency of classifying a memory location as In_CD indicates the effectiveness of using our technique to improve program analyses.

The graph in Figure 5 also shows that, in the 11 programs, on average, more than 40% of nonlocal memory locations accessed at a statement are classified as In_CD. This result indicates that our technique might be very effective in improving program analyses.

4 Related Work

Many flow-sensitive alias analyses compute *polyvariant* alias information that identifies different alias relations for a procedure under different callsites (e.g. [4, 7]). Using the *conditional* alias information (a kind of polyvariant alias information) provided by Landi and Ryder's algorithm [7], Landi, Ryder, and Zhang [8] proposed *conditional analysis* that computes information for a callsite using alias relations that hold in the called procedure when the procedure is invoked under this callsite. Although this technique can improve the precision of program analyses, it may be too expensive for large program because (1) it requires information provided by expensive alias analysis algorithms; (2) it can increase the complexity of the program analyses.

A flow-insensitive alias analysis algorithm may be extended to compute polyvariant alias information using a technique similar to Reference [6]. This allows a program analysis to compute more precise program information. However, computing polyvariant alias information would require a procedure to be analyzed multiple times, each under a specific calling context. This would make the alias analysis inefficient. Furthermore, to take advantage of the polyvariant alias information, a program analysis might have to analyze a procedure at least once for each calling context, using the alias relations that hold specifically under this calling context. This would increase the cost of the program analysis.

Although the context-dependent sets and the leaked sets are computed using the extended FICS algorithm, these sets can be used together with alias information provided by other algorithms, such as algorithms in [1, 3], to improve program analyses. Our future work includes investigating the effectiveness of our technique when the alias information is provided by various algorithms.

5 Conclusions and Future Work

In this paper, we presented a technique that extends the FICS algorithm to compute additional context information that can be used to improve the precision and efficiency of program analyses. We also demonstrated an approach that uses the context information to improve the precision of computing modification side-effects for the statement in a program. Our empirical results show that our technique can efficiently compute context information. Our empirical results also show that using the context information may significantly improve the precision and efficiency of program analyses.

In our future work, we will further investigate techniques that improve the precision and efficiency of program analyses using the context information. We will also conduct more empirical studies to investigate the effectiveness of using the context information in improving precision and efficiency of various program analyses.

Acknowledgement

This work was supported by grants from Boeing Commercial Airplane and by NSF under grants CCR-9696157 and CCR-9707792 to Ohio State University.

References

[1] L.O. Andersen. Program analysis and specialization for the C programming language. Technical Report 94-19, University of Copenhagen, 1994.

[2] D. Atkinson and W. Griswold. Effective whole-program analysis in the presence of pointers. In *The 6th ACM SIGSOFT Symposium on the Foundation of Software Engineering*, pages 46–55, November 1998.

[3] M. Burke, P. Carini, J. D. Choi, and M. Hind. Flow-insensitive interprocedrual alias analysis in the presence of pointers. In *Language and Compilers for Parallel Computing: Proceedings of the 7th International Workshop*, pages 234–250, 1994.

[4] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–256, June 1994.

[5] Programming Languages Research Group. PROLANGS Analysis Framework. Rutgers Univ., http://www.prolangs.rutgers.edu/, 1998.

[6] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph construction in object-oriented languages. In *Proceedings of ACM SIGPLAN Conference on Object-Oriented Programming Systems, Languages and Applications*, pages 108–124, October 1997.

[7] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural pointer aliasing. In *1992 ACM Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[8] W. Landi, B. G. Ryder, and S. Zhang. Interprocedural modification side effect analysis with pointer aliasing. In *SIGPLAN '93 Conference on Programming Language Design and Implementation*, pages 56–67, June 1993.

[9] D. Liang and M. J. Harrold. Efficient points-to analysis for whole-program analysis. In *7th ACM Symposium on Foundations of Software Engineering*, pages 199–215, September 1999.

[10] D. Liang and M. J. Harrold. Towards efficient and acurate program analysis using light-weight context recovery. In *22nd International Conference on Software Engineering (to Appear)*, June 2000.

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

[12] S. Zhang, B. G. Ryder, and W. Landi. Experiments with combined analysis for pointer aliasing. In *Program Analysis for Software Tools and Engineering '98*, pages 11–18, 1998.