INCREMENTAL ANALYSIS FOR FLOW- AND CONTEXT-SENSITIVE DATA-FLOW PROBLEMS

BY JYH-SHIARN YUR

A dissertation submitted to the

Graduate School—New Brunswick

Rutgers, The State University of New Jersey

in partial fulfillment of the requirements

for the degree of

Doctor of Philosophy

Graduate Program in Computer Science

Written under the direction of

Barbara Gershon Ryder

and approved by

____

____

____

____

New Brunswick, New Jersey

October, 1999

ABSTRACT OF THE DISSERTATION

Incremental Analysis for Flow- and Context-Sensitive Data-Flow Problems

by JYH-SHIARN YUR

Dissertation Director: Barbara Gershon Ryder

ABSTRACT

Data-flow analysis is widely used in extracting from source programs useful information for program optimization, program understanding, program restructuring, and testing. When a quality solution is demanded, consideration of the control flow information and calling contexts in the data-flow analysis is inevitable, but it usually involves extensive computation. When the data-flow information is used in an application in which users may interactively change the source program and check program properties involving the data-flow information, then keeping the information consistent with the current state of the program and performing the update efficiently will become very important.

Incremental data-flow analysis seeks to update data-flow information after source code changes without recomputation from scratch, based on the knowledge of the former solution and the changes. The data-flow problem we are targeting is the interprocedural modification side effect (*MOD*) problem for C. Landi and Ryder proposed a decomposition for the problem, which consists of two major sub-problems: the pointer aliasing problem computes the set of aliases at each program point, and the *PMOD* problem computes the interprocedural side effect of a procedure. Based on the problem decomposition, we proposed incremental algorithms for these two sub-problems respectively to handle some atomic changes. Then, for the *PMOD* problem, we conducted a feasibility study of our incremental algorithms, which revealed, on a average, a small source change, such as deletion of a single-line statement, causes a change of a small scale (about 3.5%) to the program representation, and affects only 4% of the old *PMOD* solution. This indicates a great opportunity for incremental analysis. As for the pointer aliasing problem, we empirically studied the effectiveness of our prototyped incremental algorithms in terms of performance and precision. In handling a small source change, our incremental algorithm outperforms the exhaustive algorithm by a *6-fold speedup*. In addition, our incremental algorithms compute the same solution as the exhaustive algorithm in 74% of the tests, while simply restarting iteration without reinitialization does so in less than 10% of the tests. Generalization of the approach to handle other flow- and context-sensitive data-flow problems as well as processing multiple changes together is also discussed.

Keywords

Interprocedural data-flow analysis, interprocedural modification side effect analysis, pointer aliasing, incremental analysis, software maintenance

Acknowledgements

I would like to thank my advisor, Dr. Barbara Ryder, the greatest mentor on the earth, for her support over the years. I thank Dr. Bill Landi, Dr. Phil Stocks, Dr. Lori Pollock, Dr. Alex Borgida, and Dr. Apostolos Gerasoulis for their comments and suggestions for my thesis work. I also want to thank all the partners of the PROLANGS group, who together made my study at Rutgers so fruitful and delightful. I also appreciate my two lovely sons, Daniel and Justin, for being well-behaved when their dad was too busy to play with them. Finally, I have to thank my dear wife Chien-chien for her encouragement and patience.

Dedication

To my wife, Chien-chien, my sons, Daniel and Justin,

and the Lord, my Savior.

Table of Contents

Abstract ii

Acknowledgements iv

Dedication v

List of Figures ix

1. INTRODUCTION 1

2. BACKGROUND 5

2.1. Object Names 5

2.2. Why Flow sensitivity and Context sensitivity 6

2.3. Program Representation 9

2.4. *MOD* Problem for C 10

2.5. The *PointerAliasing* problem for C 14

2.6. Correctness and Precision of An Incremental Algorithm 16

3. RELATED WORK 17

3.1. Aliasing Analysis 17

3.2. *MOD* Problem 17

3.3. Incremental Data-Flow Analyses 18

3.4. Marlowe-Ryder Hybrid Algorithm 19

3.5. Calling Contexts and Interprocedural Data-Flow Analysis 20

3.6. Dependence Graph in Data-Flow Analysis 21

4. INCREMENTAL ALIASING ANALYSIS 23

4.1. Exhaustive Aliasing Analysis 23

4.2. Overview of the Incremental Aliasing Algorithm 24
4.3. Naive Falsification 27
4.4. Selective Falsification 33
4.4.1. Deleting A Statement 35
4.4.2. Adding A Statement 38
4.5. Example 41
4.6. Potential Imprecision 42
4.7. Multiple Changes 44
4.8. Demand-Driven Analysis 44

5. INCREMENTAL PMOD ALGORITHM 49
5.1. The Call Multigraph and Call-RA Graph 49
5.2. Reformulation of *PMOD* on the Call-RA Graph 52
5.3. Factorization of *PMOD* 53
5.4. Non-structural Changes 57
5.4.1. Change to *CondIMOD* 57
5.4.2. Change to *nv_backbind* 58
5.5. Structural Changes 60
5.5.1. Deleting an Edge 60
5.5.2. Adding an Edge 61
5.6. Impact of Source Changes 63
5.7. Example 65
5.8. Multiple Changes 66

6. EMPIRICAL EFFECTIVENESS OF INCREMENTAL ANALYSIS 68
6.1. Test Environment 68
6.2. Impact on the *PMOD* problem 70
6.3. Empirical Effectiveness of Incremental Aliasing Analysis 74
6.4. Impact on an Example Application: USE/MOD through Dereferenced Pointers 79

7. EXTENSIONS OF THE INCREMENTAL ALIASING ALGORITHMS 84

7.1. The Incremental Aliasing Algorithm in the Presence of Multiple Changes Reconsidered 84

7.2. Information Optimization and Incrementalization 87

7.3. Flow-Sensitive Data-Flow Analysis 88

7.4. Context-Sensitive Data-Flow Analysis 89

7.5. Partition of the Flow Graph Reconsidered 90

8. SUMMARY AND FUTURE WORK 91

8.1. Summary 91

8.2. Future Work 92

References 94

Vita 103

List of Figures

2.1. Example of flow- and context-insensitive aliasing analysis 6

2.2. Aliasing solution by flow-sensitive and context-insensitive analysis . . . 7

2.3. Aliasing solution by flow- and context-sensitive analysis 8

2.4. An example C program and its ICFG 10

2.5. Decomposition of the *MOD* problem [LRZ93] 11

2.6. Data-flow equations for *MOD* subproblems 13

2.7. Example for *MOD* computation 14

4.1. Exhaustive algorithm for *pointer aliasing* 25

4.2. Incremental aliasing algorithm for handling addition/deletion of a single statement 26

4.3. Naive falsification 29

4.4. Naive alias falsification by following the control flow 30

4.5. Reintroduce aliases for naive falsification 31

4.6. Iterative procedure for the incremental algorithm 32

4.7. Procedures for propagating aliases among functions 34

4.8. Procedures for falsifying aliases which are potentially affected by deleting a pointer assignment 36

4.9. Procedures for falsifying the aliases that may be affected by adding a new statement 40

4.10. An example of adding a recursive call 41

4.11. An example of deleting a pointer assignment 42

4.12. An example of imprecision 43

4.13. Illustration of handling multiple queries on a demand basis 46

5.1. Definitions of the call graph and the call-RA graph 49

5.2. An example of the call-RA graph 50
5.3. Reformulations of *PMOD* equation using *nv_backbind* 53
5.4. Factorization of *PMOD* 54
5.5. Factorization of the *PMOD* problem 55
5.6. Procedure *modify_CondIMOD* for handling modification to *CondIMOD* 56
5.7. Procedure *modify_backbind* for handling modification to *nv_backbind* . 58
5.8. Procedure *delete_an_edge* 59
5.9. Procedure *add_an_edge* 61
5.10. Procedure *merge_regions* for merging a set of regions 62
5.11. An example of source change 66
6.1. Experiment dataset 69
6.2. Proportions of four interesting statement types 70
6.3. Percentages of tests having impact on the call-RA graph and the *PMOD* solution 71
6.4. Impact on the call-RA graph by test categories 72
6.5. Impact on the *PMOD* solution of a positive test for each experiment program 73
6.6. Impact on the *PMOD* solution of a positive test for each test category . 73
6.7. Percentage of tests that are active 74
6.8. Impact on the aliasing solution in an active test 75
6.9. Execution time of the incremental algorithms for handling deletion of one interesting statement 76
6.10. Execution time for handling deletion of different kinds of interesting statements 77
6.11. Percentage of active tests which are hits 78
6.12. Hit rates for handling deletion of different kinds of interesting statements 78
6.13. Numbers of static USE/MOD through dereferenced pointers 79
6.14. Average numbers of fixed locations used/modified through dereferenced pointers at a USE/MOD-thru-deref site 80

6.15. Percentages for inactive tests, tests which affect USE-thru-deref only, tests which affect MOD-thru-deref only, and tests which affect both USE/MOD-thru-dere 80
6.16. Average percentages of USE/MOD-thru-deref sites whose solutions are affected by the source change 81
6.17. Percentage of tests which have are hits for the USE/MOD-thru-Deref problems 82
6.18. Percentage of USE- or MOD-thru-deref sites which have different solutions from that obtained by the exhaustive approach 83
7.1. Identify the unnecessary falsification 86

Chapter 1

INTRODUCTION

Data-flow analysis collects facts about the definition and use of data in a program by statically simulating the execution of the program; this is also called compile-time analysis. Data-flow analysis has been widely utilized in many applications such as program optimization, program understanding, program restructuring, and testing. Usually, the fact collection in data-flow analysis is done through information propagation in two levels: intraprocedural and interprocedural propagation. Intuitively, a data-flow algorithm is *flow-sensitive* if the intraprocedural propagation reflects the sequencing (i.e., control flow) of statements within a procedure, and *context-sensitive* if the interprocedural propagation employs calling contexts to guide the information to be propagated from a called procedure back to relevant call sites [LRS$^+$98, RHS95, EGH94, LR92, JM82b, SP81]. If a quality solution is demanded, the consideration of the control flow information and calling contexts in data-flow analysis (i.e., flow- and context-sensitive data-flow analysis) will be inevitable, but it usually involves extensive computation.

Interprocedural Modification Side Effect Analysis

The data-flow problem this thesis is targeting is the interprocedural modification side effect (MOD) problem for languages with general-purpose pointers, such as C [LRZ93, YRLS97, LRS$^+$98], which basically computes *the set of variables which may be modified by a procedure in any execution of the program.* The MOD information is essential to other compile-time analyses (e.g., definition-use association [PLR94, HS94, GH98, CR99, Cha99]) and in many software applications such as data-flow-based testers [HS91, BH93, HFGO94, Ost90, Wey94, CR99, FW93, FI98], semantic program browsers, debuggers, and compilers [ASU86]. Data-flow-based testers utilize definition-use information, needing aliases at uses of values obtained through dereferenced pointers. Program slicing [GL91, HRB90, OO84, RR95, GS96, LH96, HC98, SHR99, TCFR96, Tip96, Ven91, Wei84, TAFM97, AG96, AG98] tools determine the set of statements relevant to a criteria by tracing the definition-use information. An intelligent debugger may suggest the set of variables that need to be monitored at a statement with dereferenced pointers with the *MOD* information. For a semantic program browser, this kind of data-flow information is definitely what its information engine is seeking.

There have been several algorithms [Ban79, Bur90, Coo85, CK88, CK87] proposed for solving the *MOD* problem for FORTRAN-like languages, in which the aliasing can be caused through parameter-binding, and won't change during the procedure call. However, those techniques cannot be applied to a language with general-purpose pointers (e.g., C). For the *MOD* problem in the presence of general-purpose pointers, Landi & Ryder [LRZ93] proposed a problem decomposition, which consists of two major sub-problems: the pointer aliasing problem computes the set of aliases at each program point, and the *PMOD* problem computes the interprocedural side effect of a procedure. For the pointer aliasing subproblem, Landi and Ryder [LR92] proposed a flow- and context-sensitive algorithm, which was later incorporated in the general algorithm [LRZ93] for solving the *MOD* problem. The problem decomposition also provides an algorithm schema [LRS+98] for the *MOD* problem, which can be parameterized by the aliasing method used.

Incremental Data-Flow Analysis

When the data-flow information is used in an application (e.g., a software development/maintenance tool) in which users may interactively change the source program and check program properties involving the data-flow information, then keeping the information consistent with the current state of the program and performing the update efficiently will become very important. *Incremental data-flow analysis* seeks to update data flow information after a program change rather than recomputing it *from scratch*, with the belief that the change impact will be limited and that the update cost will be proportional to the size of the impact. Our incremental analysis is based on Landi and Ryder's problem decomposition for *MOD* [LRZ93], and their flow- and context-sensitive algorithm [LR92] for the pointer aliasing problem.

For the aliasing problem, there is approximation inherent in Landi-Ryder's problem formulation. Reuse of the old solution in our incremental algorithm will allow the approximation to accumulate over the course of source changes; that is, when a source change is made, the solution computed by our incremental algorithm may be less precise than the one computed by an exhaustive approach, but both the solutions are approximate. Granted that, we will relax the usual precision requirement on an incremental algorithm, and two incremental versions of the Landi-Ryder exhaustive pointer aliasing algorithm [LR92], which is flow- and context-sensitive, are presented. For the *PMOD* problem, we defined a program representation to encode the relation of the calling contexts among procedures, and then showed how to update the required information in computing *PMOD* when changes are made that affect program representation.

Outline

The rest of this thesis is organized as follows. In Chapter 2 we define the problem we are targeting (i.e., the MOD problem for C). Chapter 3 gives an overview of related research. There are two major subproblems - the pointer aliasing problem ($ALIAS$) and the interprocedural side effect problem (*PMOD*). Incrementalization of the MOD problem requires the incrementalization of these two subproblems. Chapters 4 and 5 explain incremental analysis for these two subproblems respectively. In Chapter 6 we present some empirical data from the feasibility study of the incremental *PMOD* analysis, and the effectiveness study of incremental pointer aliasing analysis. In Chapter 7, we discuss possible generalization of the incremental algorithms for the *PMOD* and pointer aliasing problems to handle other flow- and context-sensitive data flow problems. In this chapter, we also explain extension of the incremental aliasing algorithm to support query-based analysis, and possible performance improvement of the algorithm. Finally, we summarize and preview future work in Chapter 8.

The main contributions of this thesis are as follows:

- incrementalization of the data-flow algorithms for two major data-flow sub-problems, *ALIAS* and *PMOD*, of the MOD problem for C. The incremental pointer aliasing algorithms are the first which attempt to incrementalize a flow- and context-sensitive data flow algorithm in the presence of a *small source change* (i.e., at a program point).

- a new program representation, the *call-RA graph*, for representing the relation of calling contexts among procedures, as expressed by reaching aliases.

- empirical studies for the above algorithms. We have studied 28 real C programs in the feasibility study for *PMOD*, and among them, 12 relatively large programs have been studied for the effectiveness of the incremental aliasing analyses. We summarize the findings as follows:

  - for incremental *PMOD*, over tests which demonstrated change in the call-RA graph, on average only 5% of the graph nodes were affected.
  - for incremental aliasing, over tests which demonstrated change in the aliasing solution, on average only 3.2% of all aliases changed, and our incremental algorithm calculated the same aliasing solution as the exhaustive algorithm would have on average in 74% of these tests. Besides, on average, our incremental algorithm outperformed the exhaustive algorithm by a 6-fold speedup.
  - for statement-level MOD/USE through pointer dereference, on average over tests which demonstrated changes to these problem solutions, the size of the change was small (5-8%) and our incremental algorithm calculated the same solution as the exhaustive algorithm in about 90% of these tests.

- extension of the incremental aliasing algorithm for efficiently handling multiple changes and supporting query-based analysis

- discussion of generalizations of the incremental algorithm for other flow- and context-sensitive data-flow problems.

Chapter 2

BACKGROUND

In this chapter, we will first show how flow and context sensitivity can affect the precision of the computed solution in data-flow analysis by examples. Then, for the target problem (*MOD* for C), we will briefly explain the program representation, the problem definition, the problem decomposition proposed by Landi and Ryder [LRZ93, LRS$^+$98], and some terminology we use.

The subset of C considered in our *MOD* problem includes sophisticated pointer usage and heap data structures, but not pointer arithmetic and function pointers, and arrays are treated as aggregates.

2.1 Object Names

Memory locations are referred to through *object names.* An object name is a variable name with a (possibly empty) sequence of pointer dereferenced (*) value and/or field accesses (.field). The dereference (*) of a pointer refers to the location it points to, which may vary during the execution of the program. Thus, an object name is called a *fixed location* if it is not a dereferenced pointer (e.g., object name `*p` is not a fixed location, but `p` is a fixed location [LRZ93].) If a data structure contains a self-referential pointer (e.g., linked lists, or trees) then it may imply an unbounded number of object names. Landi [LR92] limited the potential object names to a bounded number by limiting the number of dereferences to some constant, k, which is called $k - limiting$ [JM82a]. In addition to the variable names defined in declarations, *heap names* are explicitly created for locations dynamically allocated in the program. A heap name $heap_n$ refers to the heap locations that are allocated at a program point (or statement) n.

```
int x,y,z;
int *p,*q;
foo()                bar(int *w)
{                    {
    p=&x;             c: q=w;
    p=&y;            }
 a: bar(p);

 d: *q=…;
 u: …=x+z;

 b: bar(&z);
}
```

(a) An example C program

```
<*p,x>
<*p,y>

<*w,x>
<*w,y>

<*q,*w>
<*q,x>
<*q,y>

<*w,z>
<*q,z>
```

(b) Flow- and context-insensitive aliasing solution

Figure 2.1: Example of flow- and context-insensitive aliasing analysis

Two object names, say o_1 and o_2, are *aliased* to each other, if they both may refer to the same memory location some time during execution; an unordered pair (o_1,o_2) is used to represent an alias, as in [LR91]. An alias can occur by assigning the address of a variable (or memory location) to a pointer (such an assignment statement is called *a pointer assignment*), and can be propagated, killed or can generate other aliases at other statements within the same procedure, or even across procedure boundaries through calls. Besides, parameter binding involving pointers will also be treated as a kind of pointer assignment, and thus may cause aliasing effects at the entry of the called procedure. Pointer-aliasing analysis basically computes all the possible aliases in the program. Next we will demonstrate how flow and context sensitivity may affect the pointer-aliasing analysis using an example.

2.2 Why Flow sensitivity and Context sensitivity

In flow- and context-insensitive analysis, the sequencing and control-flow of a program is ignored, that is, the program is regarded as a set of statements, and a data-flow algorithm will examine the statements one by one and process the aliasing effect it may contribute [Cou86, Gua88, BCCH94, And94, SH97b, Ste96, Wei80, ZRL96, HP98].

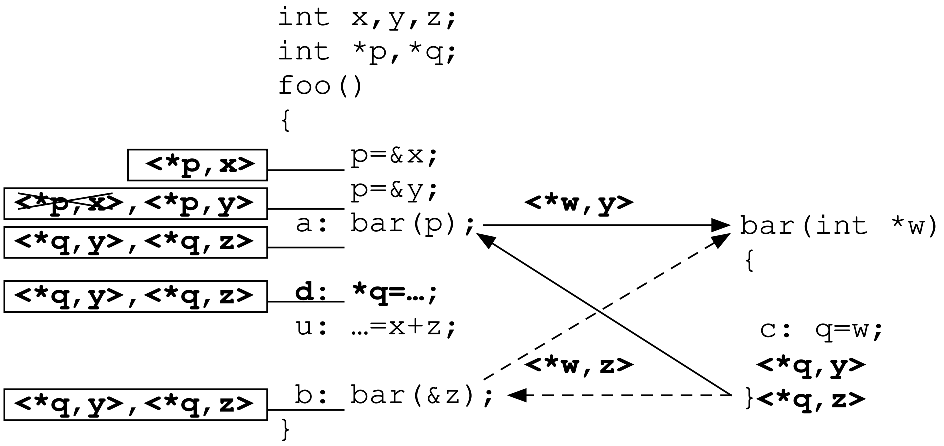


Figure 2.2: Aliasing solution by flow-sensitive and context-insensitive analysis

Usually a single set of aliasing information which holds for the whole program is computed. Figure 2.1(a) shows a simple C program, and (b) gives its final aliasing solution[1] computed by flow- and context-insensitive aliasing analysis. The first two pointer assignment statements in the program contribute the first two aliases, <*p,x> and <*p,y>, in the solution respectively, and with these two aliases, the parameter binding of function call `a:bar(p)` then generates the next two aliases, <*w,x> and <*w,y>. Then, the statement `c:q=w` in function `bar` basically makes pointer `q` point to the same location as pointer `w`, thus creates alias <*q,*w>, and further generates the next two aliases, <*q,x> and <*q,y>. Finally, another call of function `bar` at statement `b` results in alias <*w,z>, which combined with alias <*q,*w>, results in alias <*q,z>.

Then, with the aliasing solution, if we want to know what variables may be modified at statement `d:*q=`... (i.e., *MOD* at statement `d`), variables `x`, `y`, and `z` will be reported, since they all may be aliased to `*q`. If the *MOD* information is later used in definition-use analysis, an association between statements `d` and `u` will be reported, even though it is impossible in this program (explained later).

The precision of the aliasing solution can be improved if we consider flow sensitivity in the aliasing analysis. A flow-sensitive algorithm basically distinguishes the aliasing information by program points, that is, each program point will keep a set of data-flow

[1]For clarity, the solution has been simplified by excluding some irrelevant aliases, such as <*p,*w>.

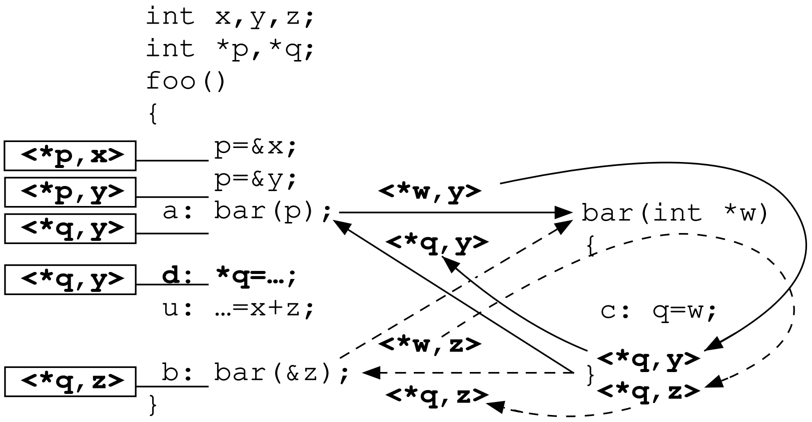


Figure 2.3: Aliasing solution by flow- and context-sensitive analysis

information which holds only at that point. Shown in Figure 2.2, the computation of such a data-flow algorithm starts with the first statement `p=&x`, where alias `<*p,x>` is generated and propagated to the next statement `p=&y`, which in turn kills the incoming alias and generates alias `<*p,y>`. The aliasing information will also be propagated interprocedurally through function calls. A data-flow algorithm is context-sensitive, if it distinguishes the data-flow effect of a procedure by the calling context reaching its entry, that is, when propagating data-flow information from a called procedure, the algorithm will use the calling context to determine the relevant call sites. In the program, there are two call sites to function `bar` (statements `a` and `b`), which respectively introduce aliases `<*w,y>` and `<*w,z>` at the entry of function `bar`, and with those aliases, the statement `c:q=w` in turn generates aliases `<*q,y>` and `<*q,z>` at the exit respectively. Thus, a flow-sensitive, but context-insensitive algorithm will propagate both aliases `<*q,y>` and `<*q,z>` to both call sites of function `bar`, and both aliases will then reach statement `d`. Then, with this aliasing solution, since both variables `y` and `z` may be aliased to `*q` at statement `d`, they are the variables that may get modified at the statement. From this example, we can see the solution quality of the aliasing analysis will affect the solution quality of other data-flow analysis which depends on the aliasing analysis. Though the precision of the MOD solution at that statement is improved, the spurious definition-use association between statements `d` and `u` still exists.

Carefully examining the alias propagation in Figure 2.3, we can find that alias <*q,y> at the exit of function bar depends on the calling context (alias <*w,y>) reaching the entry, which in turn is introduced at the first call site a only. Similarly, alias <*q,z> depends on the alias <*w,z>, which is introduced at the second call site b only. If a flow- and context-sensitive algorithm is employed in the aliasing analysis, instead of propagating all the aliases generated in function bar to both call sites, it will propagate an alias to a call site which provides the required calling context for that alias to happen. That is, alias <*q,y> will be propagated to call site a only, and <*q,z> to call site b only. Then, we will find *q at statement d can be aliased to variable y only, and the statement may modify variable y only. Thus, there should not be any definition-use association between statements d and u.

The above example has demonstrated that to maximize the precision of the obtained data-flow solution requires the consideration of flow and context sensitivity in the data-flow analysis. However such an approach involves extensive computation. Just because computing the whole solution is expensive, when a small change is made to the source programs, an effective incremental algorithm which can update the solution efficiently, and at the same time maintain its precision will be attractive in applications.

2.3 Program Representation

A program is represented as an *Interprocedural Control Flow Graph* or *ICFG* [LR91] which is the union of statement-level control flow graphs for each function in the program; each function has a unique *entry* and *exit* node. Each *call site* is split into *call* and *return* nodes. Edges are added to connect a call node to the entry node of the called function, and the exit node to the corresponding return node. Figure 2.4 shows the ICFG of an example C program, where there are two pairs of call/return edges corresponding to those two call sites to procedure F.

```
int x,y;
int *p,*q;

main()
{
 c1: F(&x);
 s1: p=&y;
 c2: F(p);
 s2: *q=…;
}

F(int *r)
{
 s3: q=r;
}
```

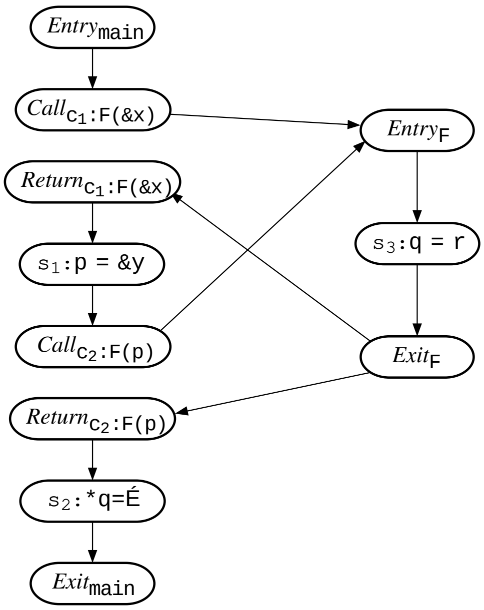


Figure 2.4: An example C program and its ICFG

2.4 *MOD* Problem for C

The *MOD* problem ascertains how program execution may affect the values of fixed locations (variables) in that program. Statement execution can change the value of a variable either by direct assignment or by indirect assignment through an alias of the lefthand-side variable. In C, a variable is aliased to the dereference (i.e., operator `*`) of a pointer which points to it (e.g., statement `p=&x` will make `*p` and `x` aliased to each other.) To obtain a precise solution to the *MOD* problem, we will need precise pointer aliasing information. Figure 2.5 shows the relevant parts of the problem decomposition of *MOD* for C programs with general-purpose pointers [LRZ93]. The following briefly explains each subproblem:

DIRMOD

$DIRMOD(n)$ contains the set of variables which may experience direct side effects through assignments at program point n. This can be calculated while parsing the program during compilation.

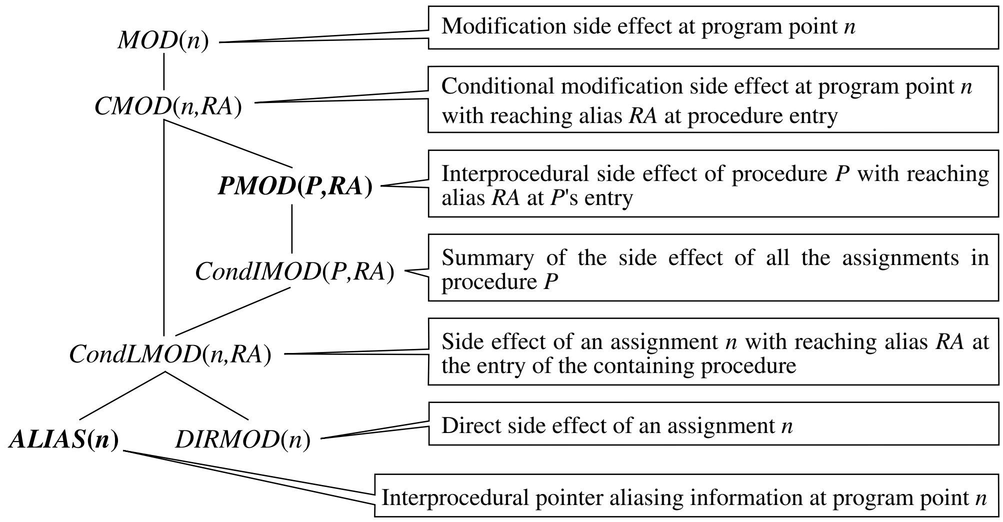


Figure 2.5: Decomposition of the MOD problem [LRZ93]

ALIAS

$ALIAS(n)$ is the pointer aliasing problem at program point n. Even though a flow- and context-sensitive aliasing analysis [LR92] is utilized, the problem decomposition also allows plugging a different type of aliasing analysis (e.g., a flow-insensitive one [Cou86, Gua88, BCCH94, And94, SH97b, Ste96, Wei80, ZRL96, HP98], which is more efficient, but less precise [SRLZ98]) in the computation. An alias, called *a reaching alias* at the entry of the procedure containing program point n, is used to approximate the calling context under which we are performing analysis at statement n,[2] and $ALIAS(n, RA)$ denotes such a conditional aliasing analysis. By "tagging" the aliasing information with reaching aliases, the computation is now *sensitive* to the calling contexts and will then propagate the aliasing information between procedures only when the calling context matches. Further definition and discussion of the interprocedural flow-sensitive aliasing analysis is given in the next section.

CondLMOD

By widening of direct side effects to include indirect effects through aliases, we obtain $CondLMOD(n, RA)$ from $DIRMOD(n)$ and $ALIAS(n, RA)$. $CondLMOD(n, RA)$

[2]We will refer to a specific reaching alias or calling context interchangeably.

only contains fixed locations.

CondIMOD

For a procedure P and a reaching alias RA, $CondIMOD(P, RA)$ (Equation (2.1) of Figure 2.6) is the union of $CondLMOD(n, RA)$ sets for all statements n in procedure P. *CondIMOD* summarizes all side effects in P except those due to call statements.

PMOD

$PMOD(P, RA)$ is the entire set of fixed locations modified by a procedure P in calling context RA, including the effects of calls from P. The *PMOD* solution (Equation 2.2) is the union of the local *CondIMOD* solutions and, at call sites, *PMOD* information propagated from procedures called by P. In the equation, at a call site $call_Q$ in procedure P with a calling context RA, $Reach(call_Q, RA)$ represents the set of aliases reaching the entry of Q implied by the parameter bindings or aliases that hold at $call_Q$. The function b_{call_Q} maps names from the scope of the called procedure Q to that of the calling procedure P.

MOD

$MOD(n)$ computes the MOD information (i.e., the set of fixed locations that can be modified by the execution of a statement n) which can be an assignment statement, or a function call. The $MOD(n)$ solution can be obtained by summarizing possible side effects over all executions of n (i.e., for all calling contexts which occur at the entry of the procedure containing n).

In this decomposition, there are only two data-flow problems: $ALIAS$ and *PMOD*. All others shown are just set combinations of their subproblems. Figure 2.7 gives the solutions to various MOD subproblems of an example program. For an assignment statement through dereference of a pointer, such as `a:*q=`... in function `F` and `b:*u=`... in function `H`, its side effect requires the aliasing information at the statement. Figure 2.7(b) gives the per program-point aliasing information. Each alias is tagged with the

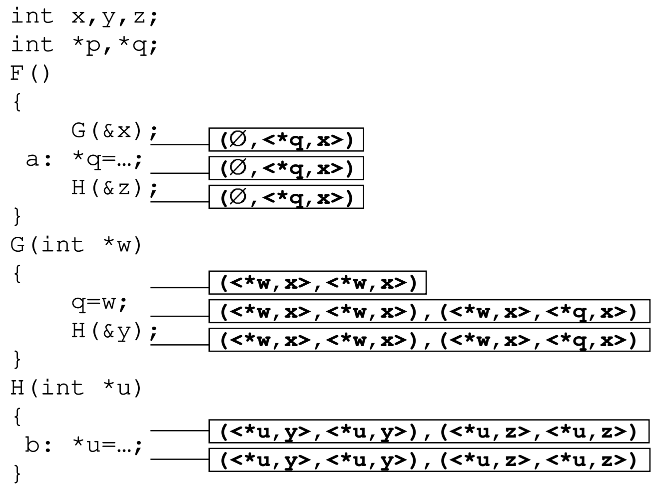
$$CondIMOD(P, RA) = \bigcup_{\substack{n \text{ is an assign-} \\ \text{ment in } P}} CondLMOD(n, RA) \quad (2.1)$$

$$PMOD(P, RA) = CondIMOD(P, RA) \cup \bigcup_{\substack{call_Q \text{ in } P \text{ and } RA' \in \\ Reach(call_Q, RA)}} b_{call_Q}(PMOD(Q, RA')) \quad (2.2)$$

$$CMOD(n, RA) = \begin{cases} CondLMOD(n, RA) & \text{if } n \text{ is an assignment;} \\ \bigcup_{RA' \in Reach(n,RA)} b_n(PMOD(Q, RA')) & \text{if } n \text{ is a call of } Q; \\ \emptyset & \text{otherwise.} \end{cases} \quad (2.3)$$

Figure 2.6: Data-flow equations for *MOD* subproblems

reaching alias at the procedure entry which is required to generate that alias at a specific point, and $\emptyset$ indicates no alias is required. With the pointer aliasing information, the side effect of statement `a` (i.e., $CondLMOD(\mathtt{a}, \emptyset)$, and $CMOD(\mathtt{a}, \emptyset)$ also) is $\{\mathtt{x}\}$. As for statement `b:*u=...` in function `H`, `*u` is aliased to variable `y` with reaching alias `<*u,y>`, and variable `z` with reaching alias `<*u,z>`. Thus, with respect to those two reaching aliases, $CondLMOD(\mathtt{a}, <*\mathtt{u}, \mathtt{y}>)$ is $\{\mathtt{y}\}$, and $CondLMOD(\mathtt{a}, <*\mathtt{u}, \mathtt{z}>)$ is $\{\mathtt{z}\}$. $CondIMOD(P, RA)$ is the union of the *CondIMOD* of all the assignment statements in procedure P with respect to reaching alias RA, not including the side effect of the procedures called by procedure P. Figure 2.7(c) gives the *CondIMOD*, *PMOD*, *MOD* solutions. $PMOD(P, RA)$ then computes the interprocedural side effect of procedure P with respect to reaching alias RA by additionally including the *PMOD* of the procedures called by procedure P with respect to the relevant reaching aliases. For example, $PMOD(\mathtt{G}, \emptyset)$ will include $PMOD(\mathtt{H}, <*\mathtt{u}, \mathtt{y}>)$, but not $PMOD(\mathtt{H}, <*\mathtt{u}, \mathtt{z}>)$ which is relevant only to the function call `H(&z)` in function `F`. At last, $MOD(P)$ is the union of $PMOD(P, RA)$ over all reaching aliases RA's.



(a) example program (b) aliasing solution

	CondIMOD	PMOD
(F,∅)	{x}	{x,y,z}
(G,∅)	{q}	{q,y}
(G,<*w,x>)	{q}	{q,y}
(H,∅)	{ }	{ }
(H,<*u,y>)	{y}	{y}
(H,<*u,z>)	{z}	{z}

MOD(F)={x,y,z}
MOD(G)={q,y}
MOD(H)={y,z}

(c) *CondIMOD*, *PMOD*, and *MOD* solutions

Figure 2.7: Example for *MOD* computation

2.5 The *PointerAliasing* problem for C

Aliases generated at a program point can be propagated to other program points intraprocedurally and interprocedurally. Thus, in computing the set of aliases which may hold at a program point, the aliasing analysis should determine what aliases may reach that point and the point's aliasing effect on the incoming aliases. Next, we will give a formal definition of the pointer aliasing problem.

Interprocedural May Alias

We have seen how aliases can be introduced by pointer assignments and function calls. If we are interested in the aliasing information at a specific program point, we have to define the aliasing solution we are computing. Landi defined the *Interprocedural May Alias* problem which computes the *MOP* (*meet over all paths*) solution as follows:

$\{< n, (o_1, o_2) > \mid \exists$ an interprocedural executable path, $\rho n_1 n_2 \cdots n_{i-1} n$, in the *ICFG* on which o_1 and o_2 may be aliased at node $n\}$, where ρ is the entry of the program.

From the formulation, we can see that the aliasing analysis is sensitive to control flow. In addition, the formulation also specifies the most precise solution that a static data-flow analysis can compute, however, there may be infinitely many static paths due to loops and recursions. There may also be infinitely many object names due to recursive data structures. Therefore, the MOP solution may not be computable. Landi and Ryder [LR91] have proved that the intraprocedural aliasing problem is *NP*-hard in the presence of multi-level pointers, and Landi [Lan92, Ram94] later showed that the precise intraprocedural aliasing problem in the presence of recursive data structures is *undecidable*. Thus only an approximate solution can be expected from a polynomial-time algorithm for the pointer aliasing problem.

Approximate Interprocedural May Alias

Landi and Ryder [LR92] then defined the *Conditional May Alias* problem which conceptually divides an interprocedural path into two paths, $\rho \cdots entry(n)$ and $entry(n) \cdots n$, where $entry(n)$ is the entry node of the function containing n. By doing this, the algorithm can solve separately for aliases at n under different calling contexts; thus, it obtains a context-sensitive solution by keeping calling context information with each alias and matching calling contexts during alias propagation. A calling context is approximated by a single *reaching alias*, RA, at $entry(n)$.[3] A tuple (n, RA, PA) is in the solution of the *Conditional May Alias*, if there exists a path in the $ICFG$ from ρ to $entry(n)$, which introduces reaching alias RA at $entry(n)$ **and** there also exists a path from $entry(n)$ to n which introduces alias PA at node n. A tuple (n, ϕ, PA) denotes that the alias PA is generated at node n independently of the reaching alias. Based on the conditional alias problem, Landi and Ryder [LR92] proposed a polynomial-time algorithm for the interprocedural aliasing problem. The algorithm computes a function *may_hold* at each node of the $ICFG$. If (n, RA, PA) is in the solution, $may\_hold(n, RA, PA)$ is YES; otherwise it is NO.

After the solution for the conditional alias problem is computed, the solution for the Interprocedural May Alias problem at certain program point n can be computed as the following set:

$\{PA \mid (may\_hold(n, \phi, PA) = YES) \ \vee \ (\exists$ a reaching alias RA such that $may\_hold(n, RA, PA) = YES)\}$

[3] This approximation is *precise* for programs only containing single level aliases [LR91].

2.6 Correctness and Precision of An Incremental Algorithm

For a static data-flow analysis, the *precise solution* is the solution obtained by summarizing data-flow information through symbolic execution over all static paths [Bar78]. Since the precise solution may be expensive to compute, or even undecidable, we say that *a solution is safe* if it is more *conservative* (or *over-estimating*) than the precise solution; the closer to the precise solution a solution is, the more precise it is. A data-flow algorithm is *correct* if it always computes a *safe solution*. For the Interprocedural May Alias problem, a *safe solution* is one which contains all the possible aliases that may occur at execution time and possibly some spurious aliases. Similarly, a safe solution for the MOD problem should be a superset of the set of variables that may actually be modified at execution time.

In the context of incremental analysis, when a change is made to the original program, the solution obtained by an incremental algorithm is *incrementally precise* if it is the same as the one computed by the corresponding exhaustive algorithm applied to the changed program, and an incremental algorithm is incrementally precise if it always *incrementally precise*-ly updates the solution. However, for simplicity, in the chapters discussing the incremental algorithms, we use the term *precise solutions* (or *algorithms*) to refer to *incrementally precise* solutions (or algorithms). Reiteration is a common technique for incremental data-flow analysis, which usually needs reinitialization of the data-flow value to a *safe initial estimate*. An initial estimate is *safe*, if we can be sure that it should be more conservative than any safe solutions. Reiteration from such a safe initial estimate will then converge to *a least conservative fixed point* [KU76, KU77], which is the most precise solution computable by iteration.

We have seen how the MOD problem can be factored into two data-flow subproblems - $ALIAS$ and $PMOD$; an incremental MOD algorithm for C programs needs to solve those two problems incrementally. In the following chapters, we will explain our approaches for incrementalizing the $ALIAS$ and $PMOD$ problems.

Chapter 3

RELATED WORK

3.1 Aliasing Analysis

Programming languages with different constructs for pointers will have aliasing problems of different degrees of difficulty. For the reference parameter aliasing problem of Fortran-like languages, aliases can only be created at procedure calls, and will not change throughout the execution of the called procedure. This makes the aliasing problem insensitive to the control flow. Cooper [Coo85] proposed an efficient flow-insensitive algorithm for the reference parameter aliasing problem.

As for a language with general pointers like C, the aliasing problem with the presence of multi-level pointers and recursive data structures is much more difficult than its counterpart in Fortran [LR91, Lan92, Ram94]. Many flow- and context-sensitive pointer aliasing analyses for C have been proposed [Coo89, LR92, CBC93, MLR$^+$93, EGH94, GH98, WL95, Ruf95, HA90, SFRW90, CRL99]. Recently, there have been many investigations of pointer aliasing algorithms which vary in cost and precision. Several concentrate on aliases in heap storage [HPR89, CWZ90, Deu94, HN90, EGH94, GH96a, GH96b, JM82a, LH88, SRW98]. Others calculate program-wide (flow-insensitive) aliases [Cou86, Gua88, BCCH94, And94, SH97b, Ste96, Wei80, ZRL96]. Still other work concentrated on aliases in higher order functional languages [Deu90, NPD87].

3.2 *MOD* Problem

For the *MOD* problem for Fortran-like languages, Banning [Ban79] proposed a decomposition, which separates the aliasing problem from the *MOD* problem; Cooper and Kennedy [Coo85, CK88, CK87] further decomposed the problem into side effects on global variables and side effects accomplished through parameter passing, and proposed an efficient flow-insensitive algorithms. The MOD problem for C is more complicated than its counterpart for Fortran mainly because of the inherent difficulty of the required pointer aliasing problem. Thus, an algorithm that attempts to achieve the best precision will have to make its computation sensitive to the control flow and calling context information. Landi and Ryder's problem decomposition and flow- and context-sensitive algorithms [LRZ93] are the first to solve the modification side effect problem for C. For a C-like language, Choi *et al.* mention an interprocedural modification side effect algorithm based on their flow- and context-sensitive pointer aliasing analysis [CBC93, MLR$^+$93], where they use both the call site and reaching aliases as the context. Further studies of the interprocedural side effect analysis using various aliasing methods can be found in [LRS$^+$98, SRLZ98].

Another approach to side effect analysis is to perform an interprocedural pointer aliasing analysis and then identify all variables that get modified through dereference of a pointer using the found aliases [SH97a, ZRL96, ZRL98, YRL98].

3.3 Incremental Data-Flow Analyses

Many incremental algorithms have been developed for data-flow analysis, which are useful, especially in a programming environment [Zad84]. Some incremental analyses use incremental elimination methods [Bur90, CR88, RP88]; some are based on the technique of restarting iteration [CK84, PS89, YRL99], and some are a combination of these two techniques [MR90]. An incremental elimination method first partitions the flow graph, and then establishes the data-flow function from the head node of a region to each interior node; when the external information reaching a region changes, the solution at an interior node can be updated by one computation. There are several ways of performing region partitioning. Interval analysis is an elimination method, which uses the *natural loops* of the programs [AC76, Bur84, SS78] as the partition elements, or *Tarjan intervals* [ASU86, SS78]. Some elimination methods use the immediate dominator in region decomposition [Car88, CR88].

When a change is made, an incremental iterative method updates the data-flow solution by restarting the iteration from the old solution to reach a new fixed point. Restarting iteration provides a mechanism for reusing the previously computed information. However, a precise solution for the updated problem instance will not be obtained unless some sufficient conditions are met [RMP88]. Pollock and Soffa [PS89] present precise incremental iterative algorithms, using change classification and reinitialization, for bitvector problems. A comparison of these incremental iterative algorithms is found in [BR90].

For the reference parameter aliasing problem of Fortran-like languages, Marlowe and Ryder [MR91] incrementalized Cooper's exhaustive algorithm [Coo85] by applying their incremental framework, which will be briefly explained later.

3.4 Marlowe-Ryder Hybrid Algorithm

The hybrid data-flow analysis algorithm [MR90] is based on a graph decomposition (usually by strongly connected components) which divides the flow graph into single entry regions. Local variants of the data-flow problem are defined and solved on each region. These problems make it possible to calculate the data-flow solution within the region as a function of incoming information from the region exits (for backward data-flow problems such as MOD) or the region entry (for forward problems such as reaching definitions). Thus, we can perform a propagation of global data-flow information forwards or backwards in topological order on the reduced graph, followed by a propagation within each region to solve the data-flow problem. The benefit of a hybrid approach is that it enforces locality on a data-flow problem and thus contains the extent of iteration. Thus, it is also very suitable for adapting to an incremental approach.

The solution procedure can be made incremental in the following manner for backward problems. Given a source code change, map it into the corresponding changes to the flow graph. For changes that do not invalidate the current graph decomposition or topological order, first update the solutions to the local problems, where necessary, in the regions corresponding to the source code change. Secondly, if some region entry node solution has changed, then propagate this change backwards on the reduced graph to region ancestors, as far as necessary. For each region whose exit node(s) have a changed solution, recalculate the global solutions for nodes within that region. If a source code change affects the region decomposition or topological order, then these must be adjusted before these other steps and recalculation occurs from change points [MR90].

The hybrid algorithm must be specialized to specific instances of data-flow problems, such as *PMOD*. The key step in specializing a problem is factoring the global data-flow problem into a set of local problems that are solved separately on the region. No constructive definition of factorization is given in [MR90, Mar89].

3.5 Calling Contexts and Interprocedural Data-Flow Analysis

There are many ways of distinguishing call contexts in data-flow analysis. Sharir and Pnueli [SP81] suggested the *call-strings approach*, which tags the data-flow information with the encoded call history in which it was obtained. A similar approach was used in Hendren and Nicolau's points-to algorithm [HN90, EGH94], where every procedure activation is analyzed separately; Emami [Ema93] later improved the approach by reusing the results with similar calling contexts. With this model, Landi and Ryder [LR91, LR92] basically used the *reaching aliases* as the encoding of the calling contexts in their aliasing problem; Choi *et al.* [MLR$^+$93, MLR$^+$93] used the immediate past call site and the *source alias sets*. Other studies of control flow analyses (CFA) in higher order functional languages can be found in [Shi91, NN97], in which a suffix of the call stack is used to approximate the calling context (e.g., 0-CFA (without distinguishing call sites), 1-CFA (last call site), etc.).

As for a call graph which incorporates calling contexts in it, such as the call-RA graph, Grove *et al.* [GDDC97] proposed a parameterized algorithmic framework for call graph construction in object-oriented languages. They discuss several domains of calling contexts, such as call chains and the class information about the actual parameters.

Basically, the richer the calling context considered in the call graph construction, the more precise the constructed call graph, and the more precise the data-flow analysis based on the computed call graph, but the more expensive and less scalable the computation. Thus, our call-RA graph can be regarded as an instantiation of *context-sensitive call graph* [GDDC97].

Tagging the calling contexts to data-flow facts, such as Landi's formulation of interprocedural may alias problem [LR92], has also been used in separate computation for information reuse. Harrold and Rothermel [HR96] used the technique for separate analysis of modules for the interprocedural may alias problem. By module, they meant a group of interacting procedures that has a single entry point, and does not contain any *holes* (i.e., calls to a non-library function not in the group). Their module analysis first determines all the possible calling contexts (i.e., reaching aliases) from the definition of the module (which is similar to the idea of representative problems in Marlowe and Ryder's hybrid algorithm [MR90]), and then analyzes the effect of the module on those contexts by propagating them throughout the module. The set of possible calling contexts (referred to as the *potential alias set (PASet)* in [HR96]) for a module M is obtained by considering all the possible aliases that can happen among global variables accessed in M, parameters to M, and variables that do not appear in M but can be accessed through their aliases (which are similar to *non-visibles* in [LRZ93]). After a module is analyzed, the next time when analyzing a program calling that module, the system will use the calling contexts that actually happen in a calling program as *indices* for retrieving the relevant data-flow information. Their empirical data showed that the size of *PASet* is reasonable for those modules which are well-designed for reusability; otherwise, it is large.

3.6 Dependence Graph in Data-Flow Analysis

Reps *et al.* [RHS95] proposed a framework which transforms an interprocedural data-flow problem into a graph reachability problem. The applicability of their framework is restricted to the class interprocedural data-flow problems with finite data-flow fact sets and distributive data-flow functions. The transformation is done by representing (or *encoding*) the data-flow function at a program node as a relation between the data-flow facts before and after the node; composition of two relations is also defined. For a call site, the effect of the called procedure is then summarized as a relation between the data-flow facts before and after the call site. The original interprocedural flow graph is then extended to a supergraph by graphically representing the relation; each node is a pair of a program node and one possible data-flow fact. After the transformation, a data-flow fact d holds at a certain program node n, iff there is a "realizable path" in the supergraph from the start of the procedure `main` to the *supernode* that represents fact d at node n.

Chapter 4

INCREMENTAL ALIASING ANALYSIS

In this chapter, we will explain our approach for incrementalizing the pointer aliasing problem for C. In C, the pointer aliasing effect at a statement can be propagated to, and thus affect aliasing at its intraprocedural and interprocedural successors. An algorithm which attempts to maximize the precision of the obtained aliasing solution should consider control flow information and procedure calling contexts in the computation. The algorithm on which our incremental pointer aliasing analysis is based is the Landi and Ryder approximate pointer aliasing algorithm [LR92], which is flow- and context-sensitive. We will first briefly explain the Landi and Ryder algorithm, and then incrementalize it for handling source changes.

4.1 Exhaustive Aliasing Analysis

We have mentioned that what the Landi-Ryder approximate algorithm [LR92] computes is a function *may_hold* for an alias PA at node n with a reaching alias RA at the entry of the containing procedure (i.e., *Conditional May Alias* problem.) Figure 4.1 shows the algorithm as having three main steps.[1] Firstly, the function *may_hold* is initialized with a default value, NO, and an empty *worklist* for iteration is created. Secondly, new aliases introduced through pointer assignments and parameter bindings at a call site are handled by invoking (step 2) procedures *alias_intro_by_assignment* and *aliases_intro_by_call* respectively, where the *may_hold* of each newly found alias is set to YES, and the alias then is placed on the worklist. Finally, a fixed point iteration using a worklist is performed in step 3 to find more aliases that may be generated intraprocedurally at assignments based on the introduced aliases, or interprocedurally

[1] For a more complete description of the Landi-Ryder algorithm, see [LR92].

through function calls/returns. During one step in the iteration, an alias tuple is removed from the worklist, and propagated to successor nodes by invoking appropriate handling procedures[2], which correspond to applying the semantics of a respective node to an alias to obtain the set of aliases it implies at successor nodes. We can see (in step 3.4) that aliases are propagated within a procedure by following the control flow. When a call site is encountered, procedures *alias_at_call_implies* and *alias_at_exit_implies* handle the matching of the calling contexts and the mapping of the aliases between the calling and called functions, which can avoid aliases being propagated along some non-executable paths (i.e., *unrealizable* paths) [LRZ93]. When a new alias is generated at a node, function *make_true* (in step 3.4.2) will set its *may_hold* value to YES, and put it on the worklist. After the iteration terminates, $may\_hold(n, RA, PA) = YES$ indicates that (n, RA, PA) is in the solution of the $Conditional\ May\ Alias$ problem, and thus PA is in the solution of the $Approximate\ Interprocedural\ May\ Alias$ problem.

4.2 Overview of the Incremental Aliasing Algorithm

When a change is made to the program, an incremental algorithm based on the iterative technique usually has the following two phases in order to obtain the *incrementally precise* solution [BR90, RMP88]

1. Reinitialize the solution at potentially affected program points,

2. Restart iteration from those potentially affected program nodes.

That is, before restarting iteration, a reinitialization of the region of the program to which the effects of a source change can extend, typically called the *affected region*, is taken to retain a *safe initial estimate* The effectiveness of an incremental algorithm relies on the hypothesis that the affected region rarely covers the whole program, and is frequently limited to a small part of the program. In practice, since the actual affected region is not known *a priori*, an incremental algorithm will have to approximate a

[2]These procedures have been augmented with one extra parameter which is the value of the *may_hold* function to be set for the implied aliases. With YES as the argument (as in Figure 4.1), these handling functions will *generate* aliases by setting their *may_hold* to YES.

procedure *exhaustive_aliasing*(G)
  G: a program representation, interprocedural control flow graph $(ICFG)$;
begin
  /* 1. only performed implicitly */
  1. initialize *may_hold* with a default value NO;
     create an empty *worklist*;
  2. **for** each node n in G
     2.1 **if** n is a pointer assignment
           aliases_intro_by_assignment(n, YES);
     2.2 **else if** n is a call node
           aliases_intro_by_call(n, YES);
  3. **while** *worklist* is not empty
     3.1 remove (n, RA, PA) from *worklist*;
     3.2 **if** n is a call node
           alias_at_call_implies(n, RA, PA, YES);
     3.3 **else if** n is an exit node
           alias_at_exit_implies(n, RA, PA, YES);
     3.4 **else** /* intraprocedural propagation */
           for each $m \in successor(n)$
           3.4.1 **if** m is a pointer assignment
                   alias_implies_thru_assign(m, RA, PA, YES);
           3.4.2 **else**
                   make_true(m, RA, PA);
end

Figure 4.1: Exhaustive algorithm for *pointer aliasing*

procedure *incremental_aliasing*(G,n)
  G: an $ICFG$;
  n: a statement to be changed;
begin
1. falsify the affected aliases, which are either generated at n, or depend on other affected aliases.
2. update G to reflect the change to statement n;
3. *worklist*=*reintroduce_aliases*(G);
4. *reiterate_worklist*(*worklist*,YES);

end

Figure 4.2: Incremental aliasing algorithm for handling addition/deletion of a single statement

potentially affected region which subsumes the actually affected region. A bad guess could result in unnecessary reinitialization to a large part of the program, and then cause redundant re-computation at those program points which are not actually affected. For example, a trivial incremental algorithm is simply restarting iteration without the reinitialization phase. Although it is quick and *correct*, the obtained solution may be a far-gone over-estimate of the precise solution. The other extreme approach is reinitializing the solution at every node, but this degenerates the incremental algorithm to an exhaustive algorithm. A good incremental algorithm needs to find a reinitializing strategy that balances efficiency and precision. In addition, even if a program point needs to be reinitialized, its old solution may still be reused if the required context holds. An incremental algorithm may achieve a better performance improvement by reducing unnecessary reinitialization, and reusing the old solution as much as possible.

A high level description of our incremental aliasing algorithm for handling a small source change, such as adding or deleting a single statement, is given in Figure 4.2. The algorithm is composed of four main steps. When a change (addition or deletion) to a statement is taken, the first step (step 1 in Figure 4.2), called *alias falsification*, removes (or $falsifies$) the *affected aliases* that are either generated at the changed statement or are dependent on other affected aliases. This step basically performs the task of reinitialization so as to obtain a safe initial point for the subsequent iteration that computes the updated solution. The structure of the $ICFG$ is updated (in step 2) to reflect the change. The third step (step 3 in Figure 4.2), called *alias reintroduction*, adds the relevant aliases on a worklist, and the *reiteration* phase (step 4) will then iterate to the final aliasing solution.

For this framework, the strategy of falsification directly affects the performance of the incremental algorithm. A straight-forward way of falsification is to falsify the entire old aliasing solution, but then the incremental algorithm degenerates to an exhaustive algorithm. In this chapter, we propose two strategies of falsification: *naive falsification* and *selective falsification*. *Naive falsification* falsifies aliases by following control flow; *selective falsification* further uses data dependences to reduce the set of aliases it falsifies. We will discuss these two strategies in the following two sections.

Before explaining how we handle the alias falsification, let me re-state the meaning of an alias tuple (n, RA, PA). The alias tuple basically denotes that with a reaching alias RA (i.e., there is a path from procedure `main` to the procedure which generates alias RA at the procedure entry), there is a path from the procedure entry to node n which makes alias PA hold at node n. Then, a source change may affect an alias in two ways. If the source change causes RA no longer existing at the procedure entry, then (n, RA, PA) will not hold accordingly. In addition, if node n is a successor of the source change, or a call site to a function which directly or indirectly contains the source change, then it may affect the path which generates PA at node n, and thus may affect (n, RA, PA). The alias falsification we am going to explain next basically considers these two aspects of effect a source change may cause.

4.3 Naive Falsification

The idea behind the naive falsification is to falsify all the aliases at the control flow successors (intra- and inter-procedurally) of the changed statement. In procedure $naive\_falsification$ in Figure 4.3, a forward walk in the $ICFG$ is taken from the changed statement. At each node visited, we assume that all the aliases at the visited node may be affected, and just discard them (step 2) and mark the node *TOUCHED* (step 3). When an exit node of a function is encountered, the walk is resumed at the return node of each call to the function (step 4), and we keep discarding aliases until we reach the exit of the main program.

The reaching alias RA in an alias tuple (n, RA, PA) plays the role of a calling context for the function containing n, that is, (n, RA, PA) indicates that given RA at the entry of the function, PA is implied at n. In the falsification walk, such a calling context provides a "fire-wall" for the aliases in the called function from being falsified. In order to "save" some aliases which may be reusable in the later updating phase, when a call is visited in the falsification, instead of discarding the aliases in the called function, function *disable_aliases* in Figure 4.3 just *disables* all the reaching aliases at the entry of the function (step 2), and marks the function *INFLUENCED* (step 3). That is, those aliases that are disabled are still *memoized*. The alias disabling will proceed recursively to other functions called by the current function (step 4).

Figure 4.4 pictorially demonstrates the alias falsification by the naive falsification, where a procedure is symbolized by a bar, and different coloring (black for *TOUCHED*, shaded for *INFLUENCED*, but not *TOUCHED*, and blank for not *TOUCHED* or *INFLUENCED*) denotes how the program nodes are affected by the alias falsification. Starting at a source change at program point n in procedure `P`, the falsification walk "touches" the program nodes (with black coloring) by following the control flow within a procedure and return edges (but not call edges) from called procedures to calling procedures, and all the way to the main procedure of the whole program. If a call site is encountered, the falsification walk marks the called procedure and other procedures it recursively calls *INFLUENCED* (by shading.) After the falsification, those nodes which are not affected by the source change are uncolored.

Since a calling context (i.e., a reaching alias at function entry) may now be disabled, whether or not a tuple (n, RA, PA) is in the aliasing solution can not be determined solely by the value of $may\_hold(n, RA, PA)$, but in addition we need $may\_hold(e, RA, RA)$, where e is the entry of the function containing n. That is, a tuple (n, RA, PA) is in the aliasing solution if both $may\_hold(n, RA, PA)$ and $may\_hold(e, RA, RA)$ are YES. On the other hand, if $may\_hold(e, RA, RA)$ is $FALSIFIED$ (i.e., alias RA is disabled at entry e), then all the aliases with RA as the reaching alias at nodes within

/* Perform alias falsification corresponding to step 1 in Figure 4.2 */
procedure $naive\_falsification(n)$
  n: a statement to be changed;
begin
  1. **if** n is marked *TOUCHED*, **return**;
     /* Falsify aliases at the changed node n */
  2. set all $may\_hold(n, RA, PA)$ to NO;
  3. mark n *TOUCHED*;
  4. **if** n is an exit node
       for each call node c which calls the function containing n;
         $naive\_falsification$(corresponding return of c);
  5. **else if** n is a call node
     5.1 $disable\_aliases$(entry of the function called by n);
     5.2 $naive\_falsification$(corresponding return of n);
  6. **else for** each $m \in successor(n)$
         $naive\_falsification(m)$;
end

procedure $disable\_aliases(e)$
  e: entry of the function whose aliases will be disabled;
begin
  1. **if** e is marked *INFLUENCED*, **return**;
  2. set all $may\_hold(e, RA, RA)$ to $FALSIFIED$;
  3. mark e *INFLUENCED*;
  4. **for** each call node c in function e;
       $disable\_aliases$(entry of the function called by c);
end

Figure 4.3: Naive falsification

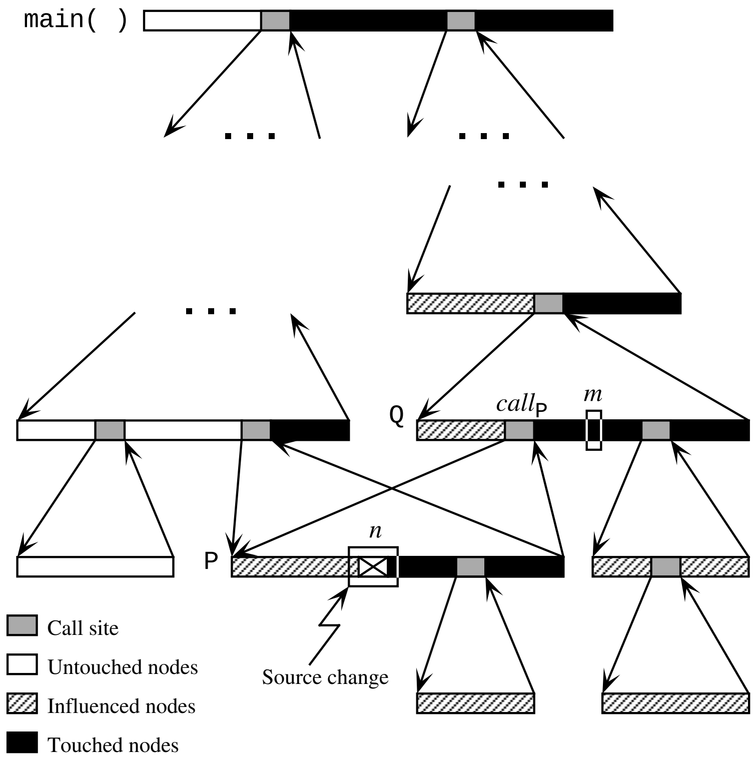


Figure 4.4: Naive alias falsification by following the control flow

the corresponding function are *implicitly* disabled, and will not appear in the aliasing solution.

After the naive falsification, all the aliases at *TOUCHED* nodes are discarded (i.e., falsified), and those reaching the entries of *INFLUENCED* functions are disabled. The *ICFG* then is modified to reflect the change to the source program. Iteration may now be restarted to update the aliasing solution at the *TOUCHED* and *INFLUENCED* nodes. Before reiteration, we have to reintroduce aliases (step 3 in Figure 4.2) to the worklist. Aliases will be reintroduced within functions and/or across functions. It is unnecessary and inefficient, to add all remaining aliases after falsification to the worklist. For reintroducing aliases across functions, procedure *reintroduce_aliases* (in Figure 4.5) examines each call site. If a call site is *TOUCHED* or its called procedure is *INFLUENCED*, the aliases at the call site need to be repropagated; that is, the effect of the call is reintroduced (step 2.1), and the aliases at the call site are repropagated (step 2.2). For reintroducing aliases within functions, the initial effect (i.e., aliasing effect independent of the reaching alias) of a *TOUCHED* pointer assignment is reintroduced

/* Perform alias reintroduction corresponding to step 3 in Figure 4.2 */
procedure *reintroduce_aliases*(G)
  G: an $ICFG$;
return
  a worklist for keeping the reintroduced aliases;
begin
  1. create an empty *worklist*;
     /* Inter-procedural propagation */
  2. **for** each call node c in G which is *TOUCHED* or whose called procedure is *INFLUENCED*,
     2.1 *aliases_intro_by_call*(c, YES);
     2.2 *repropagate_aliases*(c, *worklist*);
     /* Intra-procedural propagation */
  3. **for** each *TOUCHED* node n in G
     3.1 **if** n is a pointer assignment statement,
           aliases_intro_by_assignment(n,YES);
     3.2 **for** each $m \in$ predecessor(n)
           repropagate_aliases(m, *worklist*);
  4. **return** *worklist*;
end

procedure *repropagate_aliases*(n,*worklist*)
  n: a program node in the $ICFG$;
  worklist: a worklist for keeping the reintroduced aliases;
begin
  for each (n, RA, PA)
    add (n, RA, PA) to *worklist*;
end

Figure 4.5: Reintroduce aliases for naive falsification

(step 3.1), and aliases are repropagated to the pointer assignment from its predecessors (step 3.2).

Next, iteration is restarted over the worklist obtained by the alias reintroduction step. Procedure *reiterate_worklist* in Figure 4.6 is quite similar to the iteration (step 3) in Figure 4.1, except it will try to reuse the previously existing but disabled aliases in a *INFLUENCED* function by calling *aliases_propagated_at_call* instead. Since procedure *reiterate_worklist* will also be used for the selective falsification strategy (mentioned later), parameter *value* is set YES (or $FALSIFIED$) to generate (or to falsify) aliases by invoking procedure *make_true* (or *make_false*). Here, *value* is set to YES.

When a call node is encountered in the reiteration, procedure *aliases_propagated_at_call*

/* Perform reiteration corresponding to step 4 in Figure 4.2 */
procedure *reiterate_worklist*(*worklist*,*value*)
    worklist: a worklist for keeping the alias tuples to be processed;
    value: value that will be given to $may\_hold(n, RA, PA)$;
begin
    1. **while** *worklist* is not empty **do**
        1.1 remove (n, RA, PA) from *worklist*;
        1.2 **if** n is a call node
                $aliases\_propagated\_at\_call(n, RA, PA, value)$;
        1.3 **else if** n is an exit node
                $alias\_at\_exit\_implies(n, RA, PA, value)$;
        1.4 **else** /* intraprocedural propagation */
                for each $m \in successor(n)$
                1.4.1 **if** m is a pointer assignment
                        $alias\_implies\_thru\_assign(m, RA, PA, value)$;
                1.4.2 **else if** *value* is YES
                        $make\_true(m, RA, PA)$;
                1.4.3 **else** /* *value* is $FALSIFIED$ */
                        $make\_false(m, RA, PA)$;
end

procedure $make\_false(n, RA, PA)$
    n: a program point;
    RA: reaching alias at the entry of the function containing n;
    PA: alias to be falsified at n;
begin
    1. **if** $may\_hold(n, RA, PA) = FALSIFIED$, **return**;
    2. set $may\_hold(n, RA, PA)$ to $FALSIFIED$;
    3. mark n *TOUCHED*;
    4. add (n, RA, PA) to the *worklist*;
end

Figure 4.6: Iterative procedure for the incremental algorithm

in Figure 4.7 will check whether the implied alias reaching the entry of the called function existed before. If not, it will be handled as a new reaching alias (step 2.1). Otherwise, the alias is "enabled" (step 2.2.1), and this *implicitly* enables all the other aliases at the *unTOUCHED* nodes of the function which depend on that reaching alias. Procedure *inter_proc_propagate* is then called (step 2.2.2) to recursively enable other reaching aliases of those functions called by the function. This will save the incremental algorithm from "recomputing" those aliases which existed before and are still reusable, but the reuse of previously disabled aliases may introduce imprecision in the updated aliasing solution, which will be discussed at the end of this section.

If the procedure where a reaching alias is enabled contains *TOUCHED* nodes, since a *TOUCHED* node may have aliasing effect on those implicitly enabled aliases, for each *TOUCHED* node in the procedure, procedure *inter_proc_propagate* will repropagate aliases implicitly enabled at that node (step 2). Take procedure `Q` in Figure 4.4 as an example, we can see the procedure is *INFLUENCED* and some of its constituent program nodes (which are control flow successors of node $call_{\mathtt{P}}$) are *TOUCHED*. During the reiteration phase, if a reaching alias, say RA, is enabled at `Q`'s entry, for each node, say node m, in the procedure, all the aliases (m, RA, PA) which were implicitly disabled in the falsification phase are now enabled implicitly. We then have to consider if those implicitly enabled aliases can re-generate any previously falsified aliases in the *TOUCHED* nodes, thus, those implicitly enabled aliases will be repropagated to node m.

4.4 Selective Falsification

The naive falsification mentioned above is simple, but it naively falsifies all the aliases at each inter- and intra-procedural successor of a changed node, many of which may have no dependence on the change at all. Next we are going to refine our incremental algorithm by using a better falsification strategy, $selective\ falsification$, which only falsifies aliases that are actually generated by the changed statement and other aliases implied by those aliases. This strategy reduces the set of aliases to be falsified, but at the price of extra computation. The procedures corresponding to steps 2-4 of the

procedure *aliases_propagated_at_call*$(n, RA, PA, value)$
n: a call node;
RA: reaching alias at the entry of the function containing n;
PA: possible alias at n;
$value$: value to set the propagated aliases;
begin
1. **let** e be the entry of the function called by n, and
   r the corresponding return node of n;

   /* aliasing effect propagated to the entry node e */
2. **for** each RA' in $bind(n, e, PA)$
   2.1 **if** $may\_hold(e, RA', RA') = NO$ and $value = YES$
       $make\_true(e, RA', RA')$;
   2.2 **else if** $(may\_hold(e, RA', RA') = FALSIFIED$ and $value = YES)$ or
       $(may\_hold(e, RA', RA') = YES$ and $value = FALSIFIED)$
       2.2.1 set $may\_hold(e, RA', RA')$ to $value$;
             /* Recursively enable (or disable) all the reaching aliases at the entry of
             functions reachable from e */
       2.2.2 $inter\_proc\_propagate(e, RA', value)$;

   /* aliasing effect propagated to the return node r */
3. (Same as what is done for propagating aliases to the return node in procedure
   alias_at_call_implies, except it will make_true or make_false the implied aliases,
   depending on what *value* is)

end

procedure *inter_proc_propagate*$(e, RA, value)$
e: a function entry;
RA: a previously existing reaching alias at e;
$value$: new value for RA at e, and to be propagated to the reaching aliases at entries of functions reachable from function e;
begin
1. **for** each call node c in the function
   1.1 **let** e' be the entry of the function called by c;
   1.2 **for** each (c, RA, PA)
         for each RA' in $bind(c, e', PA)$, such that $may\_hold(e', RA', RA') \neq value$
         1.2.1 set $may\_hold(e', RA', RA')$ to $value$;
         1.2.2 $inter\_proc\_propagate(e', RA', value)$;
   /* When a reaching alias is enabled, aliases depending on that reaching alias are
   repropagated to the $TOUCHED$ nodes */
2. **if** $value = YES$ /* RA is enabled at e */
     for each $TOUCHED$ node n in the function
       for each $m \in predecessor(n)$
         for each (m, RA, PA') with $may\_hold(m, RA, PA') = YES$
           add (m, RA, PA') to the *worklist*;

end

Figure 4.7: Procedures for propagating aliases among functions

incremental algorithm (in Figure 4.2) do not require changes.

In selective falsification, the set of initially affected aliases which are directly generated at the changed node is computed. Then an iteration finds other aliases which may depend on those initial aliases. Finding the set of initially affected aliases is not a trivial task. Including too many irrelevant aliases will cause unnecessary computation not only in the falsification phase, but also in the reiteration phase. The computation of the initially affected aliases depends on what kind of source code change occurs. Next, we consider two kinds of atomic source change respectively: deleting a statement, and adding a statement.

4.4.1 Deleting A Statement

The computation of the initially affected aliases depends on the type of statement being deleted. The alias falsification for deleting a pointer assignment differs from that for deleting a function call, and we will discuss them individually.

Deleting a pointer assignment

If a pointer assignment is to be deleted, procedure $falsify\_for\_deleting\_assign$ (in Figure 4.8) is taken to falsify the aliases that may be affected by the assignment. First, step 2 falsifies the aliases introduced at the pointer assignment (i.e., generated independently of the reaching aliases). With $FALSIFIED$ as the last argument, the procedure *aliases_intro_by_assignment* will falsify (i.e., set the *may_hold* value to $FALSIFIED$) the aliases that the pointer assignment introduces. Step 3 then passes all the reaching aliases to the statement from its predecessors, and falsifies the implied aliases, except those which are preserved. An alias $PA = (o_1, o_2)$ is preserved if the left-hand-side of the pointer assignment statement is not a $prefix$[6] of either o_1 or o_2; this is what the if statement in step 3 does. After steps 2 and 3, the *worklist* will contain the aliases that the pointer assignment affects initially. Step 4 is an iteration step to falsify more aliases at other program nodes which depend on those initially affected aliases.

[6] An object name o_1 is a $prefix$ of o_2 iff o_1 can be transformed to o_2 by a sequence of dereferences and field accesses.

[7] $\emptyset$ indicates that no aliases are assumed to reach at the entry.

/* Perform alias falsification for deleting a pointer assignment corresponding to step 1 in Figure 4.2 */
procedure $falsify\_for\_deleting\_assign(n)$
n: a pointer assignment to be deleted;
begin
1. create an empty $worklist$;
   /* Falsify the aliases introduced at the node. */
2. $aliases\_intro\_by\_assignment(n, FALSIFIED)$;
3. **for** each $m \in predecessor(n)$
   for each $(m, RA, PA = (o_1, o_2))$ with $may\_hold(m, RA, PA) = YES$
   if the left-hand side of n is a $prefix$[3]of either o_1 or o_2, or both
   /* Node n may have aliasing effect on PA. */
   $alias\_implies\_thru\_assign(n, RA, PA,\ FALSIFIED)$;
4. $reiterate\_worklist(worklist, FALSIFIED)$;

end

procedure $falsify\_for\_deleting\_call(n)$
n: a function call to be deleted;
begin
1. create an empty $worklist$;
   /* Falsify the aliases generated at the call site with no reaching aliases. */
2. **let** e and x be the corresponding entry node and exit node of the function called by n respectively;
3. $aliases\_propagated\_at\_call(n, \emptyset^4, \emptyset, FALSIFIED)$;
4. **for** each (n, RA, PA) with $may\_hold(n, RA, PA) = YES$
   /* If the called function may generate new aliases from the aliases implied by PA at the call site */
   if $\exists\ RA' \in (bind(n, e, PA) - bind(n, e, \emptyset))$, such that some PA' ($\neq RA'$) is implied from RA' at x
   $aliases\_propagated\_at\_call(n, RA, PA,\ FALSIFIED)$;
5. (If a function becomes unreachable from the main program after the call node is deleted, steps 3 and 4 are repeated on those calls within each of the unreachable functions.)
6. $reiterate\_worklist(worklist, FALSIFIED)$;

end

Figure 4.8: Procedures for falsifying aliases which are potentially affected by deleting a pointer assignment

With $FALSIFIED$ as the argument for *value*, procedure *reiterate_worklist* in Figure 4.6 will selectively $make\_false$ (step 1.4.3) aliases it finds during its traversal in the $ICFG$. Procedure $make\_false(n, RA, PA)$ not only sets $may\_hold(n, RA, PA)$ to $FALSIFIED$, but also marks node n as $TOUCHED$. If a call node is visited during the traversal (step 1.2 in Figure 4.6), the procedure *aliases_propagated_at_call* (in Figure 4.7) will selectively disable the implied aliases at the entry of the called function and mark the function *INFLUENCED*. The alias-disabling will be performed recursively on other functions called by this function by execution of *inter_proc_propagate* at step 2.2.2. [5] During the selective falsification, a node where aliases are made false is marked *TOUCHED*, and a function where reaching aliases are disabled is marked *INFLUENCED*.

Deleting a function call

If the statement to be deleted is a function call, procedure $falsify\_for\_deleting\_call$ (in Figure 4.8) is used to falsify the aliases that may be affected by the call. Step 3 first falsifies the aliases generated by the call without any reaching alias involved. To determine which aliases may be directly affected by the call and need to be falsified, procedure $falsify\_for\_deleting\_call$ (in Figure 4.8) examines each of the aliases reaching the call (in step 4). If the called function does imply new aliases based on that alias, then procedure *aliases_propagated_at_call* is called to disable the reaching aliases implied at the entry of the called function, and to falsify the implied aliases at the corresponding return node. Since the deletion of a function call may cause some functions which are reachable originally to become unreachable, those calls within these unreachable functions to other reachable functions should also be treated as deleted. Steps 3 and 4 are repeated on those calls (in step 5). Finally, an iteration starting from the initially affected aliases is taken to complete the alias falsification needed for deleting a function call.

[5] The disabling step is the same in both falsification strategies; they only differ in how to determine which reaching aliases to disable.

4.4.2 Adding A Statement

While the alias falsification for deleting a statement is to falsify the aliases that may be generated at the statement, the alias falsification for adding a statement is to falsify the aliases that will not be preserved by the statement and other aliases implied by those initially affected aliases. Next, we will discuss how to falsify aliases that may be affected by adding a pointer assignment and a function call respectively.

Adding a pointer assignment

To be able to falsify the aliases that will not be preserved by the to-be-added pointer assignment, we first need to obtain its aliasing effect. Thus the first part (steps 1 to 5) of procedure $falsify\_for\_adding\_assign$ in Figure 4.9 is to compute the effect of the newly added pointer assignment on the aliases propagated from its predecessor. When a pointer assignment n is to be added between node m and all its successors, in step 1, node n is added as a successor of node m, but left without successors. An empty *worklist* is created at step 2. The aliases which will be introduced at node n and those propagated from its predecessor m are added to *worklist* in steps 3 and 4 respectively. Though step 5 calls for an iteration, it is basically for computing the effect of node n on the aliases that are propagated to it. An alias can be regarded as *killed* by a node, if it is *not* preserved by the node. An alias PA at node m that is not preserved at node n is added to *worklist* (in step 6) for falsification. An iteration is then performed at step 7 to falsify more aliases at other nodes which directly or indirectly depend on those initially affected aliases found at step 6.

Adding a function call

When adding a function call, we first compute the aliases that will be introduced by the call and those that will be generated from the aliases propagated to the call. First, we add a pair of call/return nodes as a successor of node m, and just leave them open-ended (in step 1). Second, the aliases that will be introduced by the call are computed in step 4, and all the aliases at node m are propagated through the call n (in step 5 and 6). Then we compare the aliasing solutions that hold before and after the call. The aliases which hold at the call node (i.e., before the call), but are not preserved at the return node are falsified and added to the worklist (in step 7). Then an iteration is performed to falsify other aliases which depend on the initially falsified aliases. The iteration (in step 8) then falsifies more aliases which depend on the initially falsified aliases.

However, if the called function of the to-be-added call recursively calls the changed function, the aliasing effect of the call and the changed function will be interdependent on each other. That is, those aliases the call preserves will depend on the return effect of the changed function, and which recursively depends on the alias-preserving effect of the call. Such interdependence may make the previously mentioned falsifying procedure miss some aliases that are supposed to be falsified, and thus introduce more spurious aliases in the final solution. Figure 4.10 provides such an example[6], where a function call G() is being added to program point c in function F. For simplicity, at each program point, we list the implied aliases only, but do not include associated reaching alias. Before the addition, nodes m and n (n is equivalent to the exit node of function F from the viewpoint of aliasing) have alias sets {<*p,x>} and {<*p,x>, <*p,y>} respectively. In the alias falsification phase, the algorithm first has to determine what aliases at its future successor, i.e., node n, will not be preserved by the call G() and should be falsified. Since function G recursively calls F, we need the aliasing effect of function F to in computing that of function G. If we use the old alias solution of function F, which is {<*p,x>,<*p,y>}, we will get both <*p,x> and <*p,y> at the return node of the call G(), and thus will not falsify any aliases. That is, the final aliasing solution obtained by the incremental algorithm will include alias <*p,x> at the return node of the call G() and all its successors. However, alias <*p,x> that reaches the entry of function G will never hit the exit, because the *then* branch must eventually be taken to end the recursion; that is, alias <*p,x> at node m should be falsified.

Thus, if the call to be added recursively calls the changed function, we will just use the naive falsification to ensure the precision of the updated aliasing solution.

---

[6] In the example, the set of aliases given at a function for the return node.

procedure $falsify\_for\_adding\_assign(n,m)$
n: a pointer assignment to be added;
m: the statement after which a new statement n is added;
begin
1. make n as a successor of m, and leave n without any successors;
2. create an empty *worklist*;
3. $aliases\_intro\_by\_assignment(n, YES)$;
4. $repropagate\_aliases(m, worklist)$;
5. $reiterate\_worklist(worklist, YES)$;
6. **for** each $(m, RA, PA = (o_1, o_2))$ with $may\_hold(m, RA, PA) = YES$, and $may\_hold(n, RA, PA) = NO$
   /* PA is not preserved by node n */
   add (m, RA, PA) to *worklist*;
7. $reiterate\_worklist(worklist, FALSIFIED)$;

end

procedure $falsify\_for\_adding\_non\_recur\_call(n,m)$
n: a function call to be added;
m: the statement after which a new call n is added;
begin
1. create a call node c, and a return node r for the function call n, and make c as a successor of m, and leave r without any predecessors;
2. **let** e and x be the corresponding entry node and exit node of the function call n respectively;
   /* compute the aliasing effect of the call */
3. create an empty *worklist*;
4. $aliases\_intro\_by\_call(c, YES)$;
5. **for** each (m, RA, PA) with $may\_hold(m, RA, PA) = YES$
   $make\_true(c, RA, PA)$;
6. $reiterate\_worklist(worklist, YES)$;
   /* falsify the aliases that will not be preserved by the call */
7. **for** each (m, RA, PA) with $may\_hold(m, RA, PA) = YES$, and $may\_hold(r, RA, PA) = NO$,
   /* PA will not be preserved in the execution of the call */
   $make\_false(m, RA, PA)$;
8. $reiterate\_worklist(worklist, FALSIFIED)$;

end

Figure 4.9: Procedures for falsifying the aliases that may be affected by adding a new statement

```
Program                                  Original Solution      New Solution
int x,y;
int *p;
main()
{
     p=&x;                               <*p,x>                 <*p,x>
     F();                                <*p,x>,<*p,y>          <*p,y>
}
F()
{                                        <*p,x>                 <*p,x>
     if (...)
          p=&y;  to be added             <*p,y>                 <*p,y>
     else {
 c:       G();                                                  <*p,y>
 m:       *p=1;                          <*p,x>                 <*p,y>
     }
 n:  printf("*p=%d",*p);                 <*p,x>,<*p,y>          <*p,y>
}
G()
{                                                               <*p,x>,<*p,y>
     F();                                                       <*p,y>
}
```

Figure 4.10: An example of adding a recursive call

4.5 Example

We will use a simple program in Figure 4.11 as an example for our incremental algorithm. The original aliasing solution is given in the second column. Let the pointer assignment at program point `c` be the statement to be deleted. If the naive falsification algorithm is used, it basically falsifies all the aliases at the nodes which are control-flow successors of program point `c`. In this case, all but the alias `<*p,x>` at program point `a` are falsified. If the selective falsification algorithm is used, it first computes and falsifies the alias generated at the pointer statement, which is `<*p,y>`. The iteration phase will then falsify more aliases that depend on alias `<*p,y>`, as listed in the figure. When the falsified alias `<*p,y>` reaches the function call `G(&y)`, the algorithm will disable the aliases it implies, which are `<*p,y>` and `<*p,*q>`. Then the aliases (shaded in the figure) at other nodes in function `G` which depend on those disabled aliases (i.e., `<*p,y>` and `<*p,*q>`) are implicitly disabled. In the figure, we also number the aliases with the order of falsification. The falsification of an alias and its derived aliases will stop when these aliases are killed by another pointer assignment, or reach the end of their scope. In Figure 4.11, for example, the falsification of alias `<*p,y>` ends at pointer assignment `b:p=&z` in the `main` function, since it kills the alias `<*p,y>` propagated to it. Another

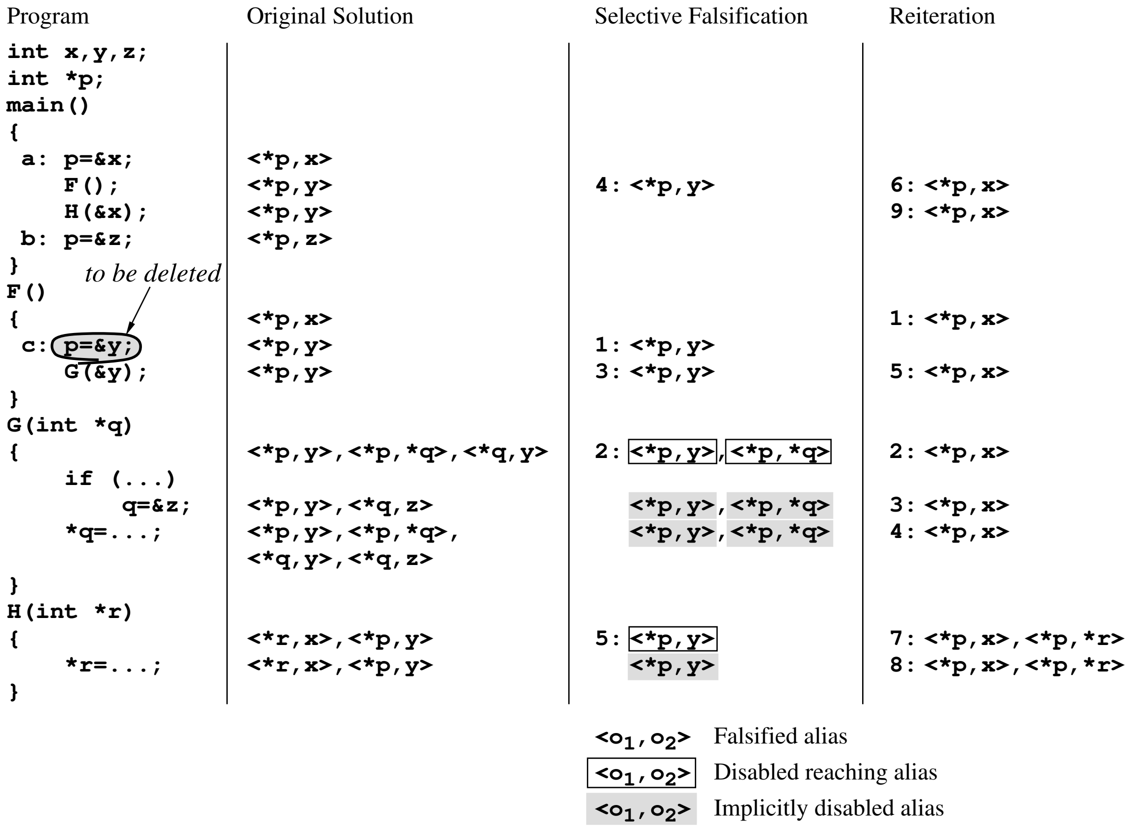


Figure 4.11: An example of deleting a pointer assignment

example is that the falsified alias `<*q,y>` in function `G` will not be propagated to its calling function `H`, since `q` is a local variable in function `G`. From this example, we see that the selective falsification algorithm generally falsifies less aliases than the naive one.

After the alias falsification phase, aliases are introduced for reiteration; alias `<*p,x>` is reintroduced at the entry of procedure `F`, and then propagated to the function call `G(&y)`. The reiteration will generate alias `<*p,x>` and other aliases it implies at other program points. We also number the aliases with the order of creation in the reiteration.

4.6 Potential Imprecision

One source of approximation in the Landi-Ryder algorithm for the Interprocedural May Alias problem is that the *may_hold* function at a node takes one alias, instead of a set, as the reaching alias to the entry of the function containing that node. Thus if an alias

Program	Original Solution	Updated Solution by the Incremental Algo.
`int x;`		
`int **p,*q,*r;`		
`main()`		
`{` *to be deleted*		
`c: p=&q;`	`(ø,<*p,q>)`	
`r=&x;`	`(ø,<*p,q>),(ø,<*r,x>)`	`(ø,<*r,x>)`
`F();`	`(ø,,<*p,q>),(ø,<*r,x>),` `(ø,<**p,*r>),(ø,<*q,*r>),` `(ø,<*q,x>),(ø,<**p,x>)`	`(ø,<*r,x>),(ø,<**p,*r>),` `(ø,<**p,x>),(ø,<*q,x>)`
`}`		
`F()`		
`{`	`(<*p,q>,<*p,q>),(<*r,x>,<*r,x>)`	`(<*r,x>,<*r,x>)`
`d: *p=r;`	`(<*p,q>,<*p,q>),(<*r,x>,<*r,x>),` `(ø,<**p,*r>),(<*p,q>,<*q,*r>),` `(<*r,x>,<*q,x>),(<*r,x>,<**p,x>)`	`(<*r,x>,<*r,x>),` `(ø,<**p,*r>),(<*r,x>,<*q,x>),` `(<*r,x>,<**p,x>)`
`}`		

Figure 4.12: An example of imprecision

is implied from two aliases with different reaching aliases, it is safe to pick either one as the reaching alias for the implied alias [LR92]. Figure 4.12 gives such an example. Two aliases `<*p,q>` and `<*r,x>` reach the entry of procedure `F`, and the pointer assignment `d:*p=r;` generates several aliases. Among these aliases, alias `<**p,*r>` is generated independent of any reaching aliases; aliases `<*p,q>` and `<*r,x>` are just preserved; alias `<*q,*r>` is implied because of alias `<*p,q>`, and alias `<**p,x>` is implied because of alias `<*r,x>`. As for alias `<*q,x>`, shaded in the figure, it is implied by combining two aliases `<*p,q>` and `<*r,x>` at the pointer assignment. In the example, alias `<*r,x>` is chosen as the reaching alias, and thus an alias tuple `(<*r,x>,<*q,x>)` is generated. This approximation is basically in the problem formulation, but since the incremental algorithm tries to reuse as much of the old aliasing solution as possible, it will make the approximation *accumulative*. In the above example, if pointer assignment `c:p=&q` is deleted, alias `<*p,q>` will be falsified in procedure `main`, and the reaching alias `<*p,q>` at the entry of procedure `F` will be disabled accordingly. However since the reaching alias `<*r,x>` still holds at the entry after the source change, the incremental algorithm will regard `<*q,x>` as a holding alias at program point `d`. This is overestimating, but safe.

In the empirical chapter, we will show that the effects overestimation seems very limited.

4.7 Multiple Changes

The incremental algorithms mentioned in this chapter are for handling addition/deletion of one single statement. Although any source change can be handled by applying the relevant algorithm, in the presence of multiple source changes, better efficiency can be achieved by processing multiple changes as a whole. That is, instead of applying the relevant algorithms to handle the changes individually, we can combine the falsification phase of the required algorithms, reintroduce aliases from the affected nodes, and then update the aliasing solution in one single iteration. Since multiple changes usually exhibit locality, the sets of affected nodes for these changes may have a great overlap, and combining corresponding phases may avoid multiple passes of falsification and reiteration of the common affected aliases.

4.8 Demand-Driven Analysis

While the complete data-flow analysis computes the information at all program points, a demand data-flow analysis predicts whether a given data-flow fact holds at a given point [BJ78, Rep94, HRS95, DGS95, DGS97]. The Landi-Ryder aliasing algorithm is a whole-program analysis; we are not seeking to derive a demand-driven version from it, but to handle changes and queries on a demand-driven basis. That is, after a change is made at a node n, if we are interested in the possible impact of the change on the aliasing solution at another particular program node x, the incremental algorithms may meet the need by employing the idea of *lazy evaluation*.

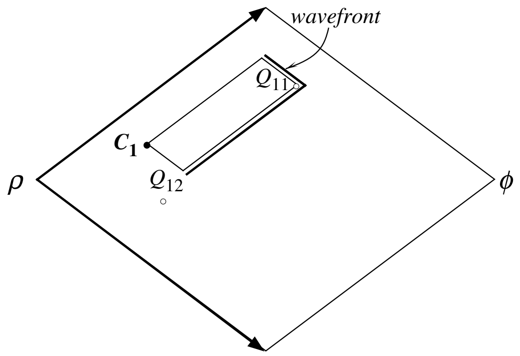
First, a backward control flow edge walk from the program node of interest, i.e., x, is taken. The walk will continue from entry nodes to call nodes, and from return nodes to exit nodes, and each node being visited in mark $RELEVANT$. Then, we check if node n resides in the set of all $RELEVANT$ program nodes. If not, there is no way for the change to affect the aliasing solution at node x, and we will be able to answer the desired query based on the old aliasing solution; the processing of the change is postponed, or batched with future changes.

If the set of $RELEVANT$ nodes does contain node n, it means the change may affect the aliasing solution at the demanded node, and we need to *explore* the change. First, the falsification phase of the incremental algorithm starting at node n is performed. Since only the aliasing solution at node x is required, in the falsification phase (and later reiteration phase also), only the falsified aliases propagated to *RELEVANT* program nodes will be put on the worklist, and those propagated to an *IRRELEVANT* node (from a *RELEVANT* node) will be stored in an auxiliary list, called the *falsifying spread*, at that node, and the node is called a *wavefront node*. We call the set of all wavefront nodes the *wavefront* of the change. Then aliases are reintroduced to the *TOUCHED* nodes. Reiteration is taken to generate and propagate the aliases to *RELEVANT* program nodes only. That is, those aliases propagated to a wavefront node will not be propagated further, but only checked with the aliases in the falsifying spread of the node. *During the reiteration phase, the aliases propagated to a wavefront node will cancel out the same aliases in its falsifying spread.* When the reiteration over the primary worklist terminates, the obtained aliasing solution at the demanded node is updated, but the whole-program solution is only *partially updated*.[7] We call the program region where both the falsification and reiteration phases have been performed the *updated region*; queries regarding nodes in the updated region can be answered without further computation. *Thus, the aliasing solution at a program node m is updated if m is in the latest updated region or the backward walk originated from m does not visit any wavefront nodes.*

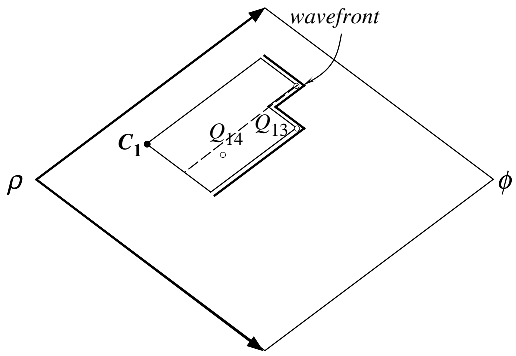
Figure 4.13 pictorially demonstrates the procedure mentioned above, where the whole program is denoted by a diamond, ρ and ϕ are the entry and the exit of the whole program respectively, and the execution proceeds from left toward right. The whole-program aliasing solution is pre-computed, a change C_1 is presented and a query Q_{11} is demanded on the changed program, see Figure 4.13(a). After a cycle of falsification and reiteration on the *RELEVANT* nodes with respect to query Q_{11}, the aliasing solution is partially updated to cover the query. Then another query Q_{12} is demanded. The new set of *RELEVANT* nodes is computed by a backward walk from query Q_{12}. We can see C_1 is not in the set of *RELEVANT* nodes of query Q_{12}, thus the query will not

[7]If all nodes have empty falsifying spread, the whole-program solution is *completely updated*.

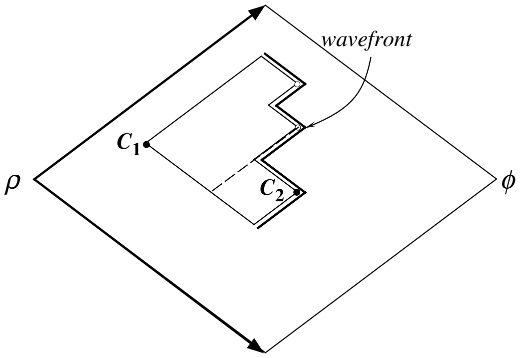
Execution Direction $\longrightarrow$



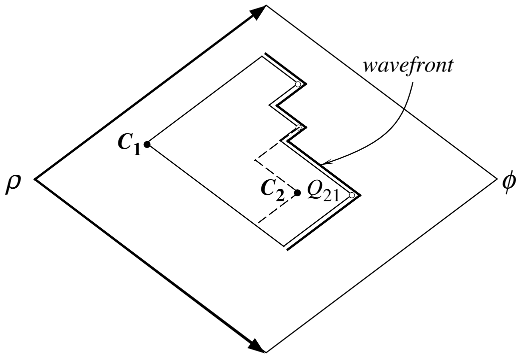
(a) After query Q_{11}



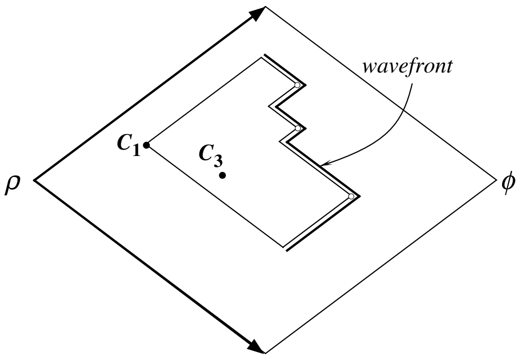
(b) After query Q_{13}



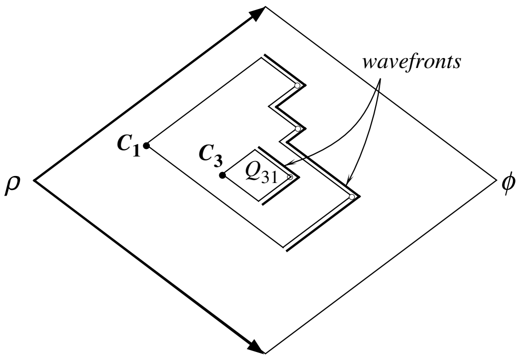
(c) After change C_2



(d) After query Q_{21}



(e) After change C_3



(f) After query Q_{31}

Figure 4.13: Illustration of handling multiple queries on a demand basis

be affected by the change, and can be answered based on the old aliasing solution.

If another demand is made out of the updated region, similarly, a backward walk is taken from the new demanded node to determine the set of *new RELEVANT* program nodes. The aliases in the falsifying spread of a wavefront node visited in the walk are fetched and put on the primary worklist for falsification. An iteration is then taken to falsify more aliases, and those aliases propagated to a *new IRRELEVANT* node will be stored in its falsifying spread. Next, aliases are reintroduced to the *new TOUCHED* nodes, and reiteration will follow to update the aliasing solution at those nodes. After another cycle of falsification and reiteration, the updated region is now extended to cover the new demanded node, and the wavefront has also moved past the new demanded node. Let's continue the example in Figure 4.13. For a query Q_{13} in (b), since its set of *RELEVANT* nodes will include some wavefront nodes, alias falsification is performed starting from the aliases in the falsifying spread of those visited wavefront nodes (i.e., those on the dashed line), and reiteration will follow. After that, the updated region has been extended to cover query Q_{13}, and the wavefront has also moved past the query. If another query is presented within the new updated region, such as query Q_{14}, it can be answered without any further computation.

The interleaving of falsification and reiteration will benefit in two aspects. The computation for updating the solution is amortized to each query, a query can be answered quickly without completely updating the whole-program solution. At a wavefront node, the generated aliases can cancel out the aliases in its falsifying spread, and this will save some unnecessary computation, and speed up the updating.

If the user makes another change at node m of the changed program, then we have to *synchronize* the new change with the current wavefront. First, the updated region has to be extended to cover node m [8], i.e., the aliasing solution at node m should be updated, and the new change will then act as a new source of falsifying spreads. When a query at node y is presented, similarly, a backward walk from node y is first taken, and at an *old* wavefront node visited, the aliases in its falsifying spread are moved into

[8] If node m is not affected by previous changes, nothing needs to be done.

the primary worklist for falsification. If the *unexplored* change at node m is also visited in the walk, the aliases initially affected by the node will be added into the primary worklist; otherwise, the query at node y is not sensitive to the new change at node m. Then, we can just follow the previously mentioned procedures to process the queries. Back to the example in Figure 4.13(c), a new change C_2 is *synchronized* with the old wavefront by extending the region to cover it. If another query Q_{21} is demanded on the changed program, after alias falsification starting from the aliases in the falsifying wave of those *RELEVANT* wavefront nodes (those on the dashed line) and reiteration, the updated region will be extended as shown in (d). A different case is given in (e), where a new change C_3 is made within the update region. Since change C_3 is within the updated region, it is *synchronous* with the previous changes, i.e., its aliasing solution already reflects the previous changes. Then a query Q_{31} is demanded in the old updated region. By a backward walk from query Q_{31}, it can be found that the old wavefront is *IRRELEVANT* to the query, but change C_3 is *RELEVANT*. Thus, after another cycle of falsification and reiteration with respect to query Q_{31}, the aliasing solution at the query is updated, and a new wavefront appears, shown in (f). All queries asked thereafter will be affected by these two wavefronts.

The arguments above for handling queries on a demand basis presumes that all the changes are *non-structural* (i.e., they do not affect the control and calling structure of the program), and may not be able to correctly handle changes which modify the control and calling structure of the program, i.e., *structural changes*.

Chapter 5

INCREMENTAL PMOD ALGORITHM

In this chapter, we will first explain the definition of our program representation for the *PMOD* problem, and then describe incremental *PMOD* in terms of the Marlowe and Ryder hybrid data-flow algorithm schema [MR90] for handling changes to the program representation.

5.1 The Call Multigraph and Call-RA Graph

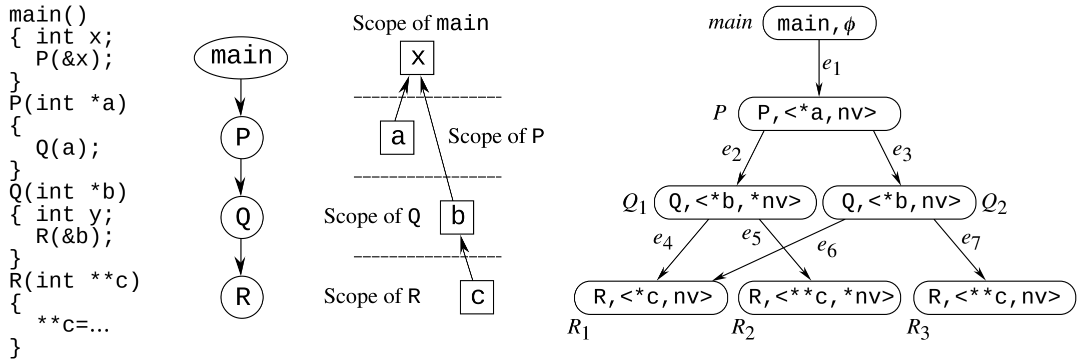
Interprocedural static analyses often use a *call multigraph*(Figure 5.1): a graphical representation of the possible calling relations between procedures in a program. A node represents a procedure and an edge from P to Q represents a call from P to Q. In order to distinguish calling contexts during our *PMOD* analysis, we introduce the *call-RA graph*, a variant of a call multigraph.[1] In a call-RA graph(Figure 5.1), each node corresponds to a procedure, reaching alias pair. Each edge represents information

Call graph $= < V, E >$
$V = \{P \mid \text{ procedure } P \text{ in the program}\}$
$E = \{(P, Q) \mid P \in V \text{ and } Q \in V \text{ and}$
$\qquad\qquad \text{there is a call site of procedure } Q \text{ in procedure } P\}$

Call-RA graph $= < V', E' >$
$V' = \{(P, RA) \mid \text{procedure } P \text{ is called with } RA \text{ reaching its entry}\}$
$E' = \{((P, RA), (Q, RA')) \mid (P, RA) \in V' \text{ and } (Q, RA') \in V' \text{ and}$
$\qquad\qquad P \text{ calls } Q \text{ and } RA' \in \mathit{Reach}(call_Q, RA)\}$

Figure 5.1: Definitions of the call graph and the call-RA graph

[1] The call-RA graph is not a multigraph, however.



(a) Program (b) Call graph (c) Memory map (d) Call-RA graph

Node	*main*	*P*	Q_1	Q_2	R_1	R_2	R_3
PMOD	{x}	{nv}	{ }	{nv}	{ }	{ }	{nv}

Edge	e_1	e_2	e_3	e_4	e_5	e_6	e_7
nv_backbind	{x}	{a}	{nv}	{b}	{nv}	{b}	{nv}

Source,Destination	P,R_1	P,R_2	P,R_3
nv_backbind *	{ }	{a}	{nv}

(e) *PMOD*, *nv_backbind* and *nv_backbind**

Figure 5.2: An example of the call-RA graph

at a specific call site flowing on a path from the entry of the calling procedure to the entry of the called procedure. The size and precision of the *call-RA graph* depend on the precision of the algorithms for computing the call graph and pointer aliases.

Figures 5.2(b) and (d) respectively show the call multigraph and call-RA graph for the simple program in Figure 5.2(a). Figure 5.2(c) shows the memory locations pointed to by the parameters at the entry of procedure `R`. Multiple nodes for a procedure in the call-RA graph indicate that the procedure is called with different reaching aliases. ϕ in node *main* indicates that the `main` procedure is executed with no reaching aliases assumed at its entry. Procedure `main` calls procedure `P` by passing the address of `x` to formal `a` of `P`. The parameter binding creates alias pair `<*a,x>`; this alias is used to represent this calling context. Note how the alias captures the relationship in Figure 5.2(c). Since `x` is a local in `main`, and thus not accessible in `P`, we use a special symbol, `nv`, instead of `x`, as an abstraction of the variables in the calling procedure which are *non_visible*, but still accessible through their aliases in the called procedure [LR92]. Thus, edge e_1 is created to connected node (`main`,ϕ) and node (`P`,`<*a,x>`). Note that `nv` in different nodes can represent different things. Given <*a,nv> at the entry of procedure P, the call of Q creates two alias pairs, <*b,*nv> and <*b,nv> and thus generates two call-RA graph nodes Q_1 and Q_2, with corresponding edges e_2 and e_3. The nv in node Q_1 represents local a in procedure P, and the nv in node Q_2 represents nv in node P, which itself represents x in *main*. Thus use of a formal of the calling procedure as an actual argument can create nv in the *called node*, which corresponds to *another* nv in the *calling node*. Similarly, given <*b,nv> at the entry of procedure Q, the call of R creates two aliases pairs, <*c,nv> and <**c,nv>, which correspond to two call-RA graph nodes R_1 and R_3 respectively, and nv in node R_1 represents local b in procedure Q; nv in node R_3 represents another nv in node Q_2.

In computing *PMOD* for a procedure with a specific reaching alias (which corresponds to a call-RA graph node), we have to include the modification side effect of other procedures (with proper reaching aliases) that the procedure calls directly or indirectly, which will involve *mapping* of nv. Figure 5.2(e) shows the *PMOD* solutions for each call-RA graph node. It is easy to see that node R_3 modifies nv through its alias **c, thus $PMOD(R_3) = \{\texttt{nv}\}$[2]. Computation of *PMOD* at node Q_2 needs to include the *PMOD*'s of R_1 and R_3, since they are connected to Q_2 by call-RA graph edges. That is, $PMOD(Q_2)$ will contains the nv in node R_3, which actually maps to nv in node Q_2 from what is mentioned above, and $PMOD(Q_2) = \{\texttt{nv}\}$. Similarly, we can obtain $PMOD(P) = \{\texttt{nv}\}$. As for $PMOD(main)$, since nv in node P corresponds to local x in procedure main, $PMOD(main) = \{\texttt{x}\}$. From this example, we can see the computation of *PMOD* is sensitive to the calling context (i.e., reaching aliases), and it involves collection of *PMOD* at other reachable call-RA graph nodes *and* variable (including nv) mapping to the target node. In the next section, we will talk about reformulation of the *PMOD* equation on the call-RA graph.

[2] *PMOD* contains fixed locations only, thus both $PMOD(R_1)$ and $PMOD(R_2)$ are $\emptyset$.

5.2 Reformulation of *PMOD* on the Call-RA Graph

Since the mapping of the non-visible variables is needed in computing *PMOD*, we thus embed this binding information for `nv` along the call-RA graph edges. For a call-RA graph edge $e = ((P, RA), (Q, RA'))$, we define $nv\_backbind_e$ as the set of all names in node (P, RA) to which `nv` in node (Q, RA') can bind. For example, $e_1 = ((\mathtt{main}, \phi), (\mathtt{P}, < *\mathtt{a}, \mathtt{nv} >))$, $nv\_backbind_{e_1}$ is the set of all names in node $(\mathtt{main}, \phi)$ to which `nv` in node (`P,<*a,nv>`) can bind, (namely, {`x`}). Figure 5.2(e) shows the *nv_backbind* sets for other edges in the call-RA graph.

With this extension of the call-RA graph, the original *PMOD* equation (Equation (2.2) in Figure 2.6), can be rewritten as Equation (5.1) in Figure 5.3. The figure also shows our equivalent new formulation in Equations (5.2) and (5.3), using $nv\_backbind^*_e$, which makes it easier to apply the hybrid algorithm [MR90]. E' is the edge set of the call-RA graph. The function b maps names between scopes, mapping globals to themselves, discarding locals (and formal parameters), and mapping *non_visibles* to their corresponding names in scope. In Equation (5.1), *PMOD* was calculated as the union of indirect and direct side effects within the procedure and those side effects due to calls, on globals and *non_visible* variables (through their aliases). $nv\_backbind_{((P,RA),(Q,RA'))}$ is used by a binding function from an immediate successor (Q, RA') to (P, RA) to account for *non_visibles*. In Equation (5.3), we use $nv\_backbind^*_{((P,RA),(R,RA''))}$ to account at (P, RA) for *non_visibles* from any node (R, RA'') reachable through a non_visible binding chain from a successor of (P, RA). A *non_visible binding chain* is a path in the call-RA graph along each edge of which *nv_backbind* contains `nv`. $nv\_backbind^*$ keeps track of *non_visibles* through multiple levels in the call-RA graph. For example, in Figure 5.2(d), $nv\_backbind^*_{(P,R_2)}$ is {`a`}, and $nv\_backbind^*_{(P,R_1)}$ is $\emptyset$, because there is a non_visible binding chain (e_5) connecting Q_1 and R_2, but there is no such chain between a successor of P and R_1. Also, (e_3 e_7) form a non_visible binding chain from P to R_3. Thus, $nv\_backbind^*_{(main,R_3)} = \{\mathtt{x}\}$. *PMOD* is formulated in Equation (5.3) as a reachability problem in that $PMOD(P, RA)$ collects $CondIMOD(R, RA'')$ of all reachable (R, RA'') and maps them into the scope of procedure P. Similarly, in Equation (5.2),

$$PMOD(P, RA) = CondIMOD(P, RA) \cup \bigcup_{e=((P,RA),(Q,RA'))\in E'} b(PMOD(Q, RA'), nv\_backbind_e) \tag{5.1}$$

$$nv\_backbind^*_{((P,RA),(R,RA''))} = \bigcup_{\substack{e = ((P,RA),(Q,RA')) \in E', \text{ and} \\ \exists \text{ a non\_visible binding chain con-} \\ \text{necting } (Q,RA') \text{ and } (R,RA'')}} nv\_backbind_e \tag{5.2}$$

$$PMOD(P, RA) = CondIMOD(P, RA) \cup \bigcup_{\substack{(R,RA'') \text{ is} \\ \text{reachable} \\ \text{from } (P,RA)}} b(CondIMOD(R, RA''), nv\_backbind^*_{((P,RA),(R,RA''))}) \tag{5.3}$$

Figure 5.3: Reformulations of *PMOD* equation using *nv_backbind*

nv_backbind* is also defined as a reachability problem in terms of *non_visible* binding.

5.3 Factorization of *PMOD*

For our incremental *PMOD* algorithm, we use a certain partitioning of the call multigraph to induce a partition of the call-RA graph. For a certain partitioning of the call multigraph, we make all the call-RA graph nodes (P, RA) for all nodes P in a call multigraph region into one region in the call-RA graph. That is, for each procedure P, all its related call-RA graph nodes $\{(P,*)\}$ are in the same call-RA graph region, denoted as $Region(P)$, and if procedures P and Q are in the same region of the call multigraph, then two sets of call-RA graph nodes, $\{(P,*)\}$ and $\{(Q,*)\}$, are also in the same region (i.e., $Region(P) = Region(Q)$) of the call-RA graph. A topological order is calculated on the reduced call-RA graph (which is isomorphic to the reduced call multigraph). In this section, we will explain the factorization of the *PMOD* problem, and describe the update algorithms for handling changes to the call-RA graph.

Observing Equation (5.3) in Figure 5.3, we can see that the global *PMOD* solution at a node n in a region depends on the local *PMOD* information of the region, the

$$LOC_1(P,RA) = CondIMOD(P,RA) \cup \bigcup_{\substack{e = ((P,RA),(Q,RA'')) \in \\ E' \text{ and } Q \in Region(P)}} b(LOC_1(Q,RA'), nv\_backbind_e) \tag{5.4}$$

$$LOC_2(P,RA) = \bigcup_{\substack{((P,RA),(Q,RA')) \in E' \\ \text{and } (P,RA) \neq (Q,RA')}} \begin{cases} \{(P,RA)\} & \text{if } Q \notin Region(P) \\ LOC_2(Q,RA') & \text{otherwise} \end{cases} \tag{5.5}$$

$$LOC_3(P,RA) = \bigcup_{\substack{e = ((P,RA),(Q,RA')) \in E' \\ \text{and } (P,RA) \neq (Q,RA')}} \begin{cases} \{(P,RA)\} & \text{if } Q \notin Region(P) \\ LOC_3(Q,RA') & \text{else if } \mathtt{nv} \in nv\_backbind_e \\ \emptyset & \text{otherwise} \end{cases} \tag{5.6}$$

$$nv\_backbind^*_{((P,RA),(X,RA''))} = \bigcup_{\substack{e = ((P,RA),(Q,RA')) \in E' \\ \text{and } (X,RA'') \in LOC_3(Q,RA')}} nv\_backbind_e \tag{5.7}$$

$$\begin{aligned} PMOD(P,RA) \;=\; & LOC_1(P,RA) \cup \\ & \bigcup_{(X,RA') \in LOC_2(P,RA)} b(EXT(X,RA'), nv\_backbind^*_{((P,RA),(X,RA'))}), \\ & \text{where } EXT(X,RA') = \bigcup_{\substack{e = ((X,RA'),(H,RA'')) \in E' \\ \text{and } H \notin Region(P)}} b(PMOD(H,RA''), nv\_backbind_e) \end{aligned} \tag{5.8}$$

Figure 5.4: Factorization of *PMOD*

global *PMOD* information propagated from other regions, and the mapping of the global information to the scope of node n. Thus, we factor the *PMOD* problem into three local problems: LOC_1, LOC_2, and LOC_3, which will be used to capture those three types of information. The global *PMOD* solution at a node in a region can then be recovered from the solutions to these local problems.

In contrast with $PMOD(P,RA)$ in equation (5.1), $LOC_1(P,RA)$ (Figure 5.4, equation (5.4)) at node (P,RA) is a restricted instance of $PMOD(P,RA)$ to $Region(P)$, which summarizes the effects of *CondIMOD* of all reachable nodes from (P,RA) within $Region(P)$. $LOC_2(P,RA)$, defined by equation (5.5), calculates the exit nodes of $Region(P)$ reachable from node (P,RA). Global information arriving at these exit

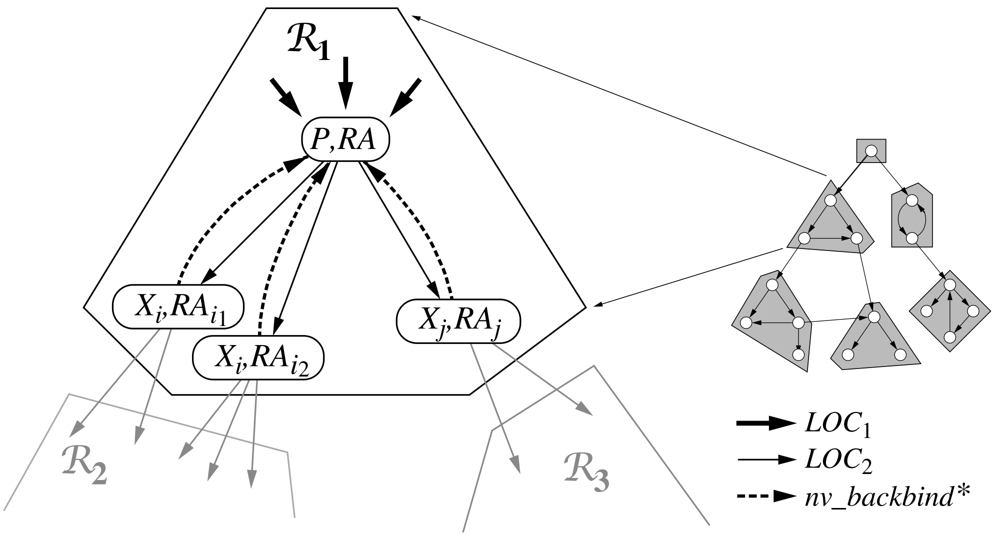


Figure 5.5: Factorization of the *PMOD* problem

nodes will be gathered and mapped back to the scope of (P, RA). In computing *nv_backbind** (Figure 5.3, equation (5.2)), we need to determine reachability through a non_visible binding chain. Thus, we define another local problem $LOC_3(P, RA)$, which computes the the set of exit nodes which are reachable from (P, RA) through a non_visible binding chain, and its data equation is shown as Equation (5.6). Equation (5.7) then gives the definition of $nv\_backbind^*_{((P,RA),(X,RA''))}$ using $LOC_3(P, RA)$.

Figure 5.5 pictorially shows how these local problem solutions are integrated in computing the final solution. For a node in a region, LOC_1 collects the local modification effect; LOC_2 computes the set of reachable exit nodes (which will be *receivers* for the global information in the later propagation phase); and *nv_backbind** (which incorporates LOC_3) provides the mapping information from each reachable exit node to the target node.

Combining the solutions to those local problems and the global information propagated from other regions, we are able to recover the global *PMOD* solution for a node (P, RA) using equation (5.8). In this equation, $LOC_1(P, RA)$ captures the local effect of the region containing (P, RA). For every exit (X, RA') reachable from node (P, RA), the *PMOD* information from a head node (H, RA'') of one of its successor

procedure $modify\_CondIMOD(G,n)$
  G: a call-RA graph;
  n: a call-RA graph node whose $CondIMOD(P, RA)$ has changed;
begin
  1. **let** $worklist = \{n\}$;
  2. **for** those nodes in *worklist* whose *CondIMOD* change is not additive,
     re-initialize LOC_1 of those nodes and all their ancestors in the same region to their respective *CondIMOD*;
  3. **while** *worklist* is not empty
     3.1 remove (P, RA) from *worklist*;
     3.2 re-compute $LOC_1(P, RA)$ using equation (5.4);
     3.3 **if** $LOC_1(P, RA)$ changes,
         add predecessors of (P, RA) in $Region(P)$ to *worklist*;
  4. call $propagate\_PMOD(G, Region(P))$;
end

procedure $propagate\_PMOD(G, \mathcal{R})$
  G: a call-RA graph;
  $\mathcal{R}$: a region in the call-RA graph;
begin
  1. **for** each region S starting from region $\mathcal{R}$ in a reverse topological order of the reduced call-RA graph,
     1.1 compute the *PMOD* solution at the entry node of region $\mathcal{S}$ using equation (5.8);
     1.2 **if** the computed *PMOD* solution changes, propagate the updated solutions to $\mathcal{S}$'s predecessor regions;
  2. **for** each region $\mathcal{S}$ which receives updated global *PMOD* information,
     recalculate the global solutions for nodes within region $\mathcal{S}$;
end

Figure 5.6: Procedure $modify\_CondIMOD$ for handling modification to *CondIMOD*

regions (which is $PMOD(H, RA''))$ is propagated to (X, RA'). All the global information arriving at an exit node (X, RA') is collected and mapped back to node (P, RA) using $nv\_backbind^*_{((P,RA)(X,RA'))}$.

Next, we will consider two classes of changes on the call-RA graph, non-structural and structural changes, and analyze how to update the solutions to reflect these changes. One thing worth mentioning is that a change in the aliases reaching a call may cause structural changes (and/or non-structural changes) to the call-RA graph, whereas this would be considered a non-structural change on the call multigraph (i.e., a call multigraph does not differentiate nodes by the calling aliases).

5.4 Non-structural Changes

Non-structural changes only change local information, like *CondIMOD* and *nv_backbind*; the structure and region decomposition of the call-RA graph remain the same.

5.4.1 Change to *CondIMOD*

When the values of *CondIMOD* at some nodes change, simply restarting fixed point iteration from the old solution does not always yield a precise (i.e., *incrementally precise*) solution. Restarting iteration can get a precise solution only if the old solution is a *safe initial estimate*; otherwise, all or part of the solutions must be re-initialized to a safe initial value [RMP88]. Changes that can be accommodated by restarting iteration are called *additive*.

Figure 5.6 shows the algorithm for handling changes of *CondIMOD*. Only the LOC_1 information can be affected by the change of *CondIMOD*. Step 2 checks which changes are additive and does the necessary re-initialization. A *CondIMOD* change is additive if the set of global and *non_visible* variables of the new *CondIMOD* is a superset of that of the old *CondIMOD*. Thus, for an additive change of *CondIMOD*, the old LOC_1 solution is a safe initial estimate, and restarting iteration will yield a precise solution. Otherwise, a simple and safe new initial estimate can be made by re-initializing LOC_1 at the changed node and its ancestors in the same region to the corresponding *CondIMOD*. Iteration using a worklist is then applied to compute the new local solutions in step 3 of Figure 5.6. Step 4 updates the global *PMOD* solutions in the regions needing to be changed based on the updated local solutions. In step 4.1, starting at the changed region with the largest topological order number and proceeding in a reverse topological order on the reduced graph, the *PMOD* information at entry nodes of the changed regions is computed, and if the *PMOD* information changes, it is propagated to predecessor regions. The inter-region propagation continues until there are no changes at entry nodes. In step 4.2, a region receiving updated global *PMOD* information recalculates the global *PMOD* solution at its nodes by combining the updated global information and the local solutions. One thing worth mentioning is that not every node in a region

procedure $modify\_backbind(G,e)$
G: a call-RA graph;
e: a call-RA graph edge, $((P, RA),(Q, RA'))$;
begin
1. **if** e is an inter-region edge, then
   call $propagate\_PMOD(G,Region(P))$;
2. **if** e is an intra-region edge:
   2.1 **if** the change of $nv\_backbind_e$ is not additive:
      2.1.1 **for** (P, RA) and every ancestor of (P, RA) in the region, say (T, RA''),
         2.1.1.1 re-initialize $LOC_1(T, RA'')$ by retaining only global variables in current LOC_1, and adding $CondIMOD(T, RA'')$ to the new LOC_1;
         2.1.1.2 re-initialize LOC_3 to an empty set;
   2.2 restart iteration on the region to get the solutions to LOC_1 and LOC_3 using equations (5.4) and (5.6) respectively;
   2.3 call $propagate\_PMOD(G,Region(P))$;

end

Figure 5.7: Procedure $modify\_backbind$ for handling modification to *nv_backbind*

with new incoming information needs to re-compute their *PMOD*. Only those nodes whose LOC_3 contains an exit node with new incoming information need to re-compute *PMOD*.

5.4.2 Change to *nv_backbind*

Change of $nv\_backbind_e$ associated with an edge e is another kind of non-structural change, which affects LOC_1 and LOC_3. If the changed edge connects two regions, since it only changes the incoming information to the source region of edge e, the local solutions are still valid. After recalculating the incoming global *PMOD* information to the source region, we can just invoke subroutine $propagate\_PMOD$ (step 4 in Figure 5.6) to update the *PMOD* solutions in the regions with changed global *PMOD* information. If the changed edge is within a region, before this information propagation, we have to update the solutions of LOC_1 and LOC_3 within that region, analogously to steps 1 to 3 in Figure 5.6.

The algorithm $modify\_backbind$ in Figure 5.7 deals with the change of *nv_backbind* case by case. If the edge whose *nv_backbind* changes is inter-region, then we just call procedure $propagate\_PMOD$ to redo inter-region propagation and intra-region propagation to update the *PMOD* solutions in the influenced regions. If the changed edge, say $e = ((P, RA), (Q, RA'))$, is within a region, then we have to consider whether the change of *nv_backbind* is additive. A change of *nv_backbind* is additive if the change does not remove `nv` from *nv_backbind*. If the change is additive, then the old solutions to LOC_1 and LOC_3 are still safe initial estimates, and restarting iteration (in step 2.2) from these old solutions can yield precise solutions. But, if the change removes `nv` from $nv\_backbind_e$, these old solutions are not safe initial estimates. Thus, before restarting iteration, (in step 2.1.1) for (P, RA) and every ancestor of (P, RA), say (T, RA''), $LOC_1(T, RA'')$ is re-initialized by retaining only globals in current LOC_1 and adding $CondIMOD(T, RA'')$ to the new LOC_1. LOC_3 is re-initialized to an empty set.

---

procedure *delete_an_edge*(G,e)
  G: a call-RA graph;
  e: a call-RA graph edge;
begin
  1. **if** e is an inter-region edge $((X, RA), (H, RA'))$:
    1.1 recalculate the *PMOD* information arriving at exit node (X, RA);
    1.2 call *propagate_PMOD*(G,*Region*(X));
    Note:
    • Removal of the only inter-region edge leaving from node (X, RA) would make it no longer an exit node; that is, (X, RA) should be removed from LOC_2 and LOC_3 at all nodes in *Region*(X).
    • Removal of the only incoming edge of *Region*(H) would make it unreachable from the root.
  2. **if** e is an intra-region edge $((X, RA), (H, RA'))$:
    2.1 **for** every node (Q, RA'') in the set of (P, RA) and its ancestors in *Region*(P),
      2.1.1 re-initialize $LOC_1(Q, RA'')$ and $LOC_2(Q, RA'')$ to $CondIMOD(Q, RA'')$ and $\emptyset$, respectively;
      2.1.2 **if** $nv\_backbind_e$ contained `nv`, re-initialize $LOC_3(Q, RA'')$ to $\emptyset$;
    2.2 iterate to the solutions of LOC_1, LOC_2, and LOC_3 on *Region*(P) using equations (5.4), (5.5), and (5.6) respectively;
    2.3 call *propagate_PMOD*(G,*Region*(P));
end

Figure 5.8: Procedure *delete_an_edge*

5.5 Structural Changes

Structural changes are those which can change the shape of the call-RA graph or its decomposition, and thus are harder to handle than non-structural changes. Structural changes may be caused by many ways. An edge may be added or deleted if the aliases reaching the entry of a procedure change. Source changes, like changing a parameter binding, adding or deleting a call, and adding or deleting aliases reaching a call, all may change the aliases reaching the entry of the called procedure, and thus may cause structural changes. Figure 5.11 shows structural changes made by changing the aliases reaching a call (i.e., a dashed line indicating deletion of edge e_5 and a bold line addition of edge e_8).

In the following discussion, we will consider edge deletion and addition. Addition or deletion of nodes can be converted to a sequence of edge changes.

5.5.1 Deleting an Edge

Usually, deleting an inter-region edge is less difficult than deleting an intra-region edge, since it just changes the incoming information to the source region of the edge, whose local solutions are still valid. We need to redo inter-region propagation starting at the source region of the deleted edge. Intra-region propagation is then performed on each region with the changed incoming information.

Case 1 of algorithm *delete_an_edge* in Figure 5.8 deals with deleting an inter-region edge[3], which is similar to changing the *nv_backbind* of an inter-region edge.

Case 2 in Figure 5.8 summarizes what is done for deleting an edge $e = ((P, RA), (Q, RA'))$ within a region[4]. In this case, LOC_1, LOC_2, and LOC_3 at (P, RA) and its ancestors in the same region are invalidated because of the edge deletion, and thus need to be re-initialized as in step 2.1 before restarting iteration. After re-initialization, we update

[3]Edge deletion can sometimes render a region exit node no longer an exit, and then the node should be removed from LOC_2 and LOC_3 for all nodes in the region. In addition, deleting an inter-region edge may also make a region unreachable from the root of the call-RA graph.

[4]Sometimes, edge removal within a region can make the region further decomposable. In parallel data-flow analysis, decomposition of large regions is beneficial to load balancing among processors.

procedure *add_an_edge*(G,e)
G: a call-RA graph;
e: a call-RA graph edge, $((P, RA), (Q, RA'))$, to be added;
begin
1. **if** e is an intra-region edge:
   1.1 iterate LOC_1, LOC_2, and LOC_3 to their new solutions using equation (5.4), (5.5), and (5.6) respectively;
   1.2 call *propagate_PMOD*(G,*Region*(P));
   Note: adding (or connecting) a new node to the call-RA graph will be considered as adding a new edge whose target node is an isolated node

2. **if** e is an inter-region edge:
   2.1 **if** e is a forward edge and Q is the entry of a region
      2.1.1 **if** (P, RA) was not an exit node, then
         iterate LOC_2 and LOC_3 to new solutions using equation (5.5) and (5.6) respectively to account for (P, RA) becoming an exit
      2.1.2 include the $PMOD$ information coming from e in exit node (P, RA);
      2.1.3 call *propagate_PMOD*(G,*Region*(P));
   2.2 **else** /* a new decomposition is needed because e is a back edge or a forward edge into the interior of a region */
      2.2.1 /* determine the minimal set of regions, say $\mathcal{R}_{i_1}, \mathcal{R}_{i_2}, ..., \mathcal{R}_{i_k}$, to be merged in order to make edge e become an edge within the resulting region. */
         2.2.1.1 find the immediate common dominating region of *Region*(P) and *Region*(Q).
         2.2.1.2 the regions between the dominating region and *Region*(P) and *Region*(Q) form the set of regions to be merged.
      2.2.2 $\mathcal{R} = merge\_regions(\{\mathcal{R}_{i_1}, \mathcal{R}_{i_2}, ..., \mathcal{R}_{i_k}\})$;
      2.2.3 edge e is now an edge within region $\mathcal{R}$, and its addition can be handled by applying part 1 of the algorithm;

end

Figure 5.9: Procedure *add_an_edge*

the local solutions, and then the global solution.

5.5.2 Adding an Edge

Adding an edge within a region is an additive change with respect to our local problems, and can be accommodated by restarting iteration from previous solutions, shown in case 1 of algorithm *add_an_edge* in Figure 5.9. The procedure call to *propagate_PMOD* at step 1.2 then updates the global *PMOD* solutions in the region based on the updated local solutions, and performs the inter- and intra-region propagations, if needed.

But, if the edge added connects two regions (which is case 2 in Figure 5.9), then

procedure *merge_regions*($\mathcal{M}$)
$\mathcal{M}$: a set of regions, $\{\mathcal{R}_{i_1}, \mathcal{R}_{i_2}, ..., \mathcal{R}_{i_k}\}$ in the call-RA graph;
return
the resulting region by merging regions in $\mathcal{M}$;
begin
1. create a new region $\mathcal{R} = \mathcal{R}_{i_1} \cup \mathcal{R}_{i_2} \cup ... \cup \mathcal{R}_{i_2}$;
2. determine the exit nodes with respect to region $\mathcal{R}$, denoted as a set $EXIT(\mathcal{R})$;
/* adjust LOC_1, LOC_2, and LOC_3 to make them consistent with the resulting region. */
3. **for** each region $\mathcal{S}$ in $\mathcal{M}$, in a reverse topological order.
/* inter-region propagation among $\mathcal{R}_{i_1}, \mathcal{R}_{i_2}, ..., \mathcal{R}_{i_k}$ */
3.1 **for** each exit node (X, RA) of region $\mathcal{S}$, compute the following

$$LOC'_1(X, RA) = \bigcup_{\substack{e\,=\,((X,RA),(Y,RA'))\,\in\,E' \\ \text{and } Region(X) \neq Region(Y) \\ \text{and } Region(Y) \in \mathcal{M}}} b(LOC_1(Y, RA'), nv\_backbind_e)$$

$$LOC'_2(X, RA) = \bigcup_{\substack{e\,=\,((X,RA),(Y,RA'))\,\in\,E' \\ \text{and } Region(X) \neq Region(Y) \\ \text{and } Region(Y) \in \mathcal{M}}} LOC_2(Y, RA')$$

$$LOC'_3(X, RA) = \bigcup_{\substack{e\,=\,((X,RA),(Y,RA'))\,\in\,E' \\ \text{and } Region(X) \neq Region(Y) \\ \text{and } Region(Y) \in \mathcal{M} \text{ and} \\ \texttt{nv} \in nv\_backbind_e}} LOC_3(Y, RA')$$

/* intra-region propagation within each of $\mathcal{R}_{i_1}, \mathcal{R}_{i_2}, ..., \mathcal{R}_{i_k}$ */
3.2 **for** each node (Z, RA'') in region $\mathcal{S}$, update its LOC_1, LOC_2, and LOC_3 as follows:

$$LOC_1(Z, RA'') = LOC_1(Z, RA'') \cup \bigcup_{\substack{(X,RA)\,\in \\ LOC_2(Z,RA''))}} b(LOC'_1(X, RA), nv\_backbind^*_{((Z,RA''),(X,RA))})$$

$$LOC_2(Z, RA'') = \{LOC_2(Z, RA'') \cup \bigcup_{\substack{(X,RA)\,\in \\ LOC_2(Z,RA''))}} LOC'_2(X, RA)\} \cap EXIT(\mathcal{R})$$

$$LOC_3(Z, RA'') = \{LOC_3(Z, RA'') \cup \bigcup_{\substack{(X,RA)\,\in \\ LOC_3(Z,RA''))}} LOC'_3(X, RA)\} \cap EXIT(\mathcal{R})$$

4. **return** $\mathcal{R}$;
end

Figure 5.10: Procedure *merge_regions* for merging a set of regions

we have to check if it causes a cycle in the reduced call multigraph[5] or invalidates the single-entry property for regions. If the added inter-region edge is a forward edge and its destination node is the entry of a region, both properties above will still hold on the changed graph, and handling addition of such an edge (step 2.1) is similar to edge deletion. If either of these two properties is violated, we have to do a region merge in order to recover the violated properties. After that, the edge to be added can be handled as an intra-region edge (step 2.2.3). Step 2.2.1 first finds the immediate common dominator of the target and destination regions of the newly added edge, and then makes those regions between them the set of regions to be merged. Given such a set of k regions, say $\mathcal{M} = \{\mathcal{R}_{i_1}, \mathcal{R}_{i_2}, ..., \mathcal{R}_{i_k}\}$, procedure *merge_regions* (Figure 5.10) is called at step 2.2.2 to merge them into a large region and updates the LOC_1, LOC_2 and LOC_3 solutions at each node to make them consistent with the resulting region. (step 2.2.3). Since a exit node in the old region partition may not be one in the resulting region, in procedure *merge_regions*, the exit nodes for the composite region $\mathcal{R}$ are recognized (step 2). To update the LOC_1, LOC_2 and LOC_3 solutions at each node in the composite region $\mathcal{R}$, may require the solutions from other constituent regions, and the idea of the inter- and intra-region propagation can be employed to avoid iteration. For each region $\mathcal{S}$ in $\mathcal{M}$ in a reverse topological order, in step 3.1, LOC_1, LOC_2 and LOC_3 from entries of $\mathcal{S}$'s successor regions in $\mathcal{M}$ are collected at $\mathcal{S}$'s exit nodes (as LOC_1', LOC_2', and LOC_3'). For each node in region $\mathcal{S}$, it updates (in step 3.2) its local solutions by combining the old solutions and the collected information at the reachable exit nodes. In updating LOC_2 and LOC_3, $EXIT(\mathcal{R})$ is used to filter out those old exit nodes which no longer are in the resulting region $\mathcal{R}$.

5.6 Impact of Source Changes

We have shown how to handle changes to the call-RA graph. The incremental algorithm is effective when the impact of the source code changes on the call-RA graph and the *PMOD* solution is small. In this section, we discuss how a source code change may affect

[5]The partitioning on the call-RA graph is induced by the partitioning on the call multigraph.

the *PMOD* solution through changes to the call-RA graph. The primary statements of interest are assignment statements and function calls, and both may change the set of aliases reaching their statement exits. A change on the aliasing information may change the indirect side effect of the assignment, and thus change *CondIMOD* for the procedure. In addition, changes reaching a call site may change the aliasing information reaching the entry of the called procedure and the parameter binding information, and thus may cause old edges to be deleted and/or new edges to be added in the call-RA graph. Next we will discuss in details the call-RA graph changes corresponding to a source code change.

Assignment Statements We have mentioned in Chapter 2.5 that an assignment statement which may assign a pointer value (i.e., an address) is called a *pointer assignment*; otherwise, it is called a *non-pointer assignment*. A non-pointer assignment statement cannot affect aliasing. Thus, changes to a non-pointer assignment can cause changes to *CondIMOD* of the containing procedure only, which are non-structural. If the statement altered is a pointer assignment, then it may cause non-structural and/or structural changes to the call-RA graph.

The impact on the *PMOD* solution made by the change of an assignment depends not only on the type of the statement (pointer or non-pointer assignment), but also on what variables are modified and how the containing procedure is used. For example, if we remove a non-pointer statement that modifies only a local variable, then such a source change only will affect the *PMOD* solutions for the containing procedure. However, if the statement we remove modifies a global variable that is not modified elsewhere in the program, then doing so may cause changes to the *PMOD* solutions for all procedures calling it directly or indirectly; the larger the set of such procedures, the greater the impact.

Actual-Formal Bindings Changing actual-formal bindings at a call site may change the set of variables to which a *non_visible* variable in the called procedure can bind, and thus change the values of *nv_backbind* associated with the edges corresponding to the call. Of course, it may also cause changes in the reaching aliases for the called procedure. This could result in edge deletion and/or edge addition in the call-RA graph.

Procedure Calls Insertion of a call may cause new aliases to reach the entry of the called procedure; that is, new edges to new nodes in the call-RA graph may be created. On the other hand, deletion of a call may cause the opposite effect (i.e., edge deletion in the call-RA graph.) Basically, the impact on the call-RA graph and the *PMOD* solution of inserting/deleting a function call depends on how the called function is used in the program. For example, if we remove the only call to a key function in the program, it sometimes disconnects many other functions invoked by that function, and thus causes lots of edges to be deleted. On the other hand, deleting a call to a function that is a leaf node in the call-RA graph causes much less impact on the graph. Any change at a call site may also kill and/or create aliases reaching those statements after the call site, so it may cause all kinds of call-RA graph changes.

5.7 Example

In general, a change of the source program may cause a sequence of changes, structural and/or non-structural, to the call-RA graph. For example in Figure 5.11(a), we add a pointer assignment statement `b=&y` to the source program, and it causes the following call-RA graph changes:

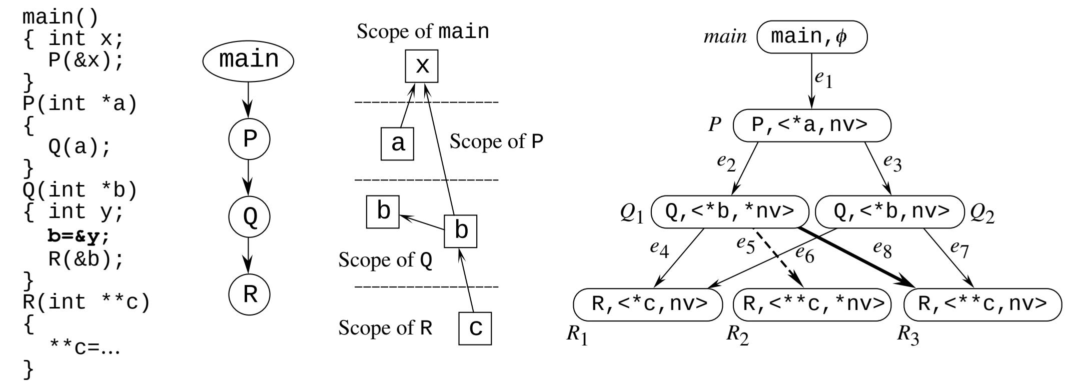
(i) *CondIMOD* changes at nodes Q_1 and Q_2,

(ii) an *nv_backbind* change of edge e_7 (see *nv_backbind* in Figure 5.2 (e)),

(iii) deletion of edge e_5, and

(iv) addition of edge e_8.

The first two changes are non-structural, and the others are structural (with respect to the call-RA graph). Figures 5.2(b) and (d) respectively show the resulting call multigraph and call-RA graph, where the call multigraph remains unchanged. The algorithms mentioned in the previous section can then be invoked to handle these changes of the call-RA graph, one by one. Thus, algorithm $ModifyCondIMOD$ is called to handle changes in *CondIMOD* for Q_1 and Q_2, algorithm $ModifyBackbind$ to handle the *nv_backbind* change of edge e_7, algorithm *DeleteAnEdge* to handle deletion of edge e_5, and algorithm *AddAnEdge* to handle addition of edge e_8. Figure 5.11(e) also gives the *PMOD* solutions before and after the source change. We can see that the source change affects the *PMOD* solutions of four call-RA graph nodes: *main*, P, Q_1, and Q_2, and node R_2 becomes unreachable due to the source change.



(a) Program (b) Call graph (c) Memory map (d) Call-RA graph

Node	*main*	*P*	Q_1	Q_2	R_1	R_2	R_3
Old *PMOD*	{x}	{nv}	{ }	{nv}	{ }	{ }	{nv}
New *PMOD*	{x}	{nv}	{b,y}	{b,y}	{ }	unreachable	{nv}

Edge	e_1	e_2	e_3	e_4	e_6	e_7	e_8
nv_backbind	{x}	{a}	{nv}	{b}	{b}	{y}	{y}

(e) *PMOD* and *nv_backbind*

Figure 5.11: An example of source change

5.8 Multiple Changes

In this section, we have described the updates as separate steps for ease of understanding. By examining the algorithms, we can find that each of our proposed updating algorithms is composed of two major steps: local solution updating and inter-region information propagation. Although any change to the call-RA graph can be handled by applying the relevant algorithm, in the presence of multiple changes to the call-RA graph, we can achieve better efficiency by processing multiple changes as a whole. That is, instead of applying the relevant algorithms to handle the changes individually, for intra-region changes, we can combine the local solution updating steps of the required algorithms, and update the local solutions for each region in one single iteration.

Then, one pass of inter-region information propagation can be applied to handle inter-region changes and propagate updated information among regions. In handling multiple changes, this method avoids multiple iterations within each region and multiple passes of inter-region information propagation.

Predicting Impact As we can see, an arbitrary, even small, change of the source program could result in a wide range of changes to the call-RA graph. The impact of a source change is not solely dependent on the kind of change made. So, to make general statements about kinds of source code changes we need an empirical study of the impact of many source changes. Our study is discussed in the next chapter.

Chapter 6

EMPIRICAL EFFECTIVENESS OF INCREMENTAL ANALYSIS

To demonstrate the effectiveness of the incremental approach, we studied the impact on PMOD and aliasing solution of deleting single source statements.

6.1 Test Environment

Our suite of test programs consists of 28 programs, which are listed in Figure 6.1, ordered by the number of $ICFG$ nodes; among them, 16 programs we used to test the incremental aliasing analysis. All the tests were performed on a 75 MHz Sun SPARCstation-20 running Solaris 2.5.1 with 352 Mbyte physical memory and 250 Mbyte swap space. For each program in our dataset, we first determined the *interesting* source statements of four types:

1. non-pointer assignment (nPtr.);
2. pointer assignment[1] statements (Ptr.);
3. function calls without aliasing effect (Calls w/o Ptr.); and
4. *affecting function calls*, which are calls to functions containing pointer assignments directly or indirectly, which thus can potentially affect aliases.

The figure shows the numbers for these four types of interesting statements respectively. Each test corresponded to deletion of one interesting source statement from the original program. Since only pointer assignments and affecting function calls can

[1]If a library function call can have aliasing side effects, it is encoded as a sequence of assignment statements rather than a function call.

potentially affect aliases, we tested these two types of statements only in the empirical study for the aliasing analysis. In each test, we calculated the aliasing solution for the program exhaustively, recalculated it incrementally after deleting the statement, and then also recalculated it exhaustively on the changed program for performance and precision comparison. As for the *PMOD* problem, we tested all four types of statements, and in each test, we calculated the solutions for the original and changed programs exhaustively, and then measured the impact of the source change on the call-RA graph and the solution, if any.

Program	lines of code	ICFG nodes	# procs	#nPtr assign.	#Ptr assign.	#calls w/o ptr.	#affecting calls	tested in[2]
allroots	215	422	8	46	1	13	3	M
fixoutput	401	617	7	101	8	0	10	M
travel	862	698	16	109	14	2	19	M
ul	541	1011	15	83	38	19	14	M
plot2fig	1435	1075	27	116	5	60	13	M
lex315	719	1311	18	108	8	8	64	M
compress	1490	1316	16	187	41	6	15	M
loader	1219	1560	31	161	29	40	22	A / M
mway	700	1575	22	202	9	33	4	M
stanford	887	1768	48	167	4	51	10	M
dixie	2129	2324	36	239	51	3	57	M
learn	1483	2571	36	324	54	15	50	M
xmodem	1705	2662	28	250	31	9	95	M
compiler	2232	2950	39	254	2	149	195	M
sim	1439	3032	17	333	63	0	26	A / M
cdecl	1015	3132	33	129	46	4	30	A / M
diff	1726	3287	43	323	60	54	62	A / M
unzip	4106	3404	40	415	51	4	32	A / M
assembler	2693	3593	53	353	90	66	118	M
gnugo	2901	3621	29	494	1	37	19	M
lharc	3303	4238	87	453	87	25	137	A / M
patch	2672	4597	56	444	83	24	186	A / M
simulator	3733	5329	100	488	47	222	167	A / M
arc	7507	5498	96	719	104	56	118	A / M
tbl	2511	6105	85	466	120	56	97	A / M
triangle	1930	6118	19	514	75	18	13	A / M
football	2222	7308	59	493	5	57	73	M
moria	24596	38091	445	4203	535	1277	1756	A / M

Figure 6.1: Experiment dataset

[2]M: tested in the incremental PMOD; A: tested in the incremental aliasing

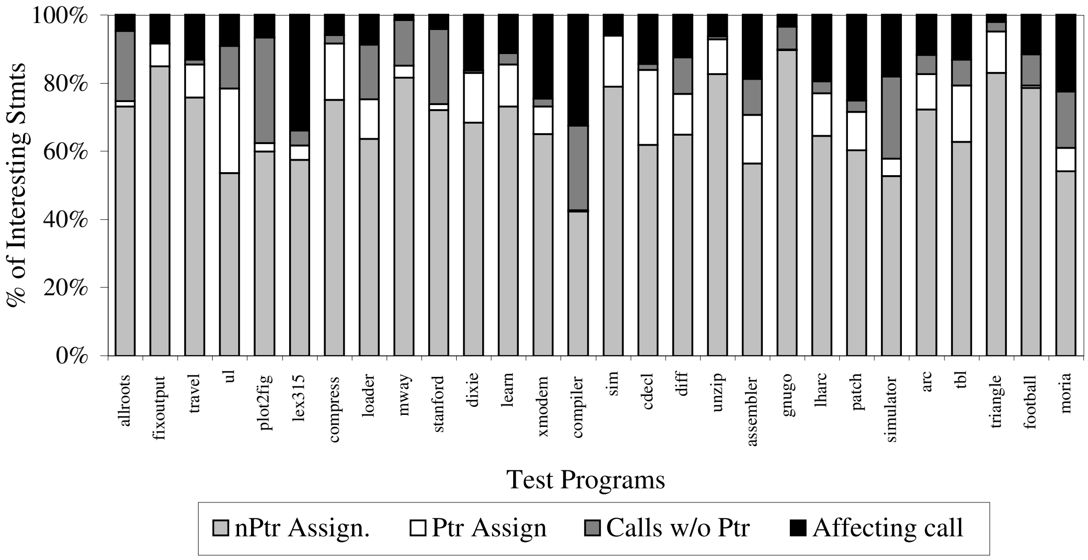


Figure 6.2: Proportions of four interesting statement types

One reason we chose this approach is that it was reasonably simple to implement, which enabled us to collect data on many potential changes across many programs. Ideally, we would like to study change histories of programs, so that the changes we test more accurately reflect modifications likely to be made by programmers. However, the kind of change we can obtain from two check-in's in a version control system usually consists of program-wise (or module-wise) code modification, and does not provide the granularity of source code changes our incremental approach is targeting.

6.2 Impact on the $PMOD$ problem

For the *PMOD* problem, a test is called *positive* if it causes changes to the call-RA graph or information associated with nodes or edges of the call-RA graph, and is called *active* if it causes changes to the *PMOD* solution; all active tests are also positive. Deletion of different types of statements may have varying degrees of impact on the call-RA graph and the *PMOD* solution. Figure 6.3, shows the corresponding percentages of positive and active tests respectively, for each statement category across all our programs (see Figure 6.1), with the height of a bar representing the percentage of all tests for that category. In each program, we calculated the average percentage of tests of each test type (e.g., positive, active, no call-RA graph change) for each interesting statement

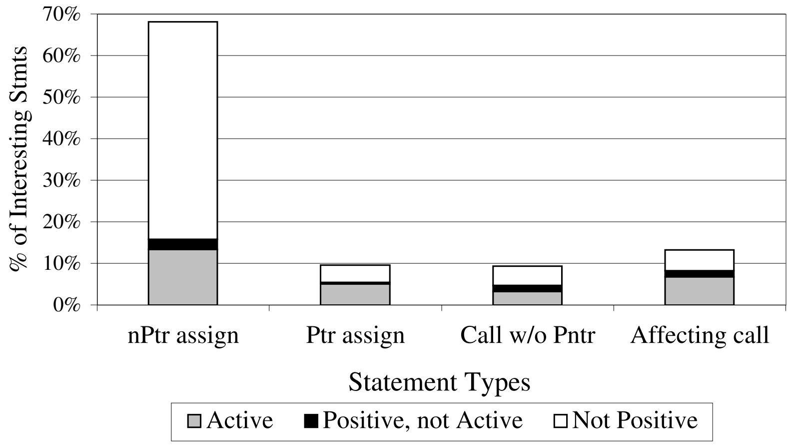


Figure 6.3: Percentages of tests having impact on the call-RA graph and the *PMOD* solution

category; then we averaged these averages across all programs. This calculation of an average of averages, repeated in several of our measurements reported in this section, we will call an *overall average*.

Deleting a non-pointer assignment statement may not cause any changes to the call-RA graph if its lefthand-side variable is assigned by other assignment statements in the same procedure. For similar reason, deleting a call site may not affect the call-RA graph, if the changed procedure has more than one call site to the same called procedure. We expect that deleting a non-pointer assignment statement is less likely to affect the call-RA graph and the *PMOD* solution than removing a statement of another category. In contrast, deleting an affecting function call is most likely to cause changes to the call-RA graph and the *PMOD* solution.

The table in Figure 6.4 gives the overall average of the percentages of call-RA graph nodes with changed *CondIMOD*, call-RA graph edges with changed *nv_backbind*, call-RA graph edges deleted and added in a *positive test*. For each category of interesting statement, the second row in the table is the overall average percentage of call-RA graph nodes whose *CondIMOD* changes because of a statement deletion. Since a non-pointer statement can not affect aliasing information, deleting such a statement may change only the set of variables modified by the containing procedure (i.e., *CondIMOD*). As for deleting a statement of the other three categories, edge deletion is the common

Stmt. type / Rep. changes	Non-pointer assignments	Pointer assignments	Fun. calls w/o ptr. assign.	Affecting calls
% Call-RA graph nodes w/ *CondIMOD* Δ	0.55%	0.52%	0.29%	0.60%
% Call-RA graph edges w/ *nv_backbind* Δ		0.16%	0.03%	0.01%
% Call-RA graph edges deleted		7.95%	2.09%	10.61%
% Call-RA graph edges created		0.16%	0.00%	0.09%

Figure 6.4: Impact on the call-RA graph by test categories

structural change. Since an affecting function call combines the effect of a function call and the pointer assignments its called procedure contains, deleting such a function call causes structural changes of the largest scale on the call-RA graph, as expected.

In a test, a call-RA graph node is *affected* if its *PMOD* solution changes, and a call-RA graph node is *influenced* if one of its successors is affected, but the node itself is not. Figure 6.5 shows the the average percentages of call-RA graph nodes affected and influenced in a positive test for each program. The sum of these two percentages (i.e., the height of each bar) gives an estimate of the minimum percentages of call-RA graph nodes that must be examined by any safe incremental algorithm in order to update the *PMOD* solution. Except for the first few small programs, this value for most programs is below 6%, and the overall average is 4.68%(indicated by a bold line in the figure). This indicates that a minor source code change, such as deleting a statement, generally does not have great impact on the call-RA graph or the *PMOD* solution.

We have further examined the distribution of the number of affected and influenced nodes for each program, and found that it is not a normal distribution, but instead has a steep peak around a small number of nodes (i.e., 2-3). This indicates that a great percentage of the tests only affect a small number of call-RA graph nodes. Furthermore, the values given in Figure 6.5 are obtained by taking averages over positive tests only; thus if we included all tests in the averages, the height of the bars would be reduced by more than 50%. This result reveals great opportunities for our incremental technique.

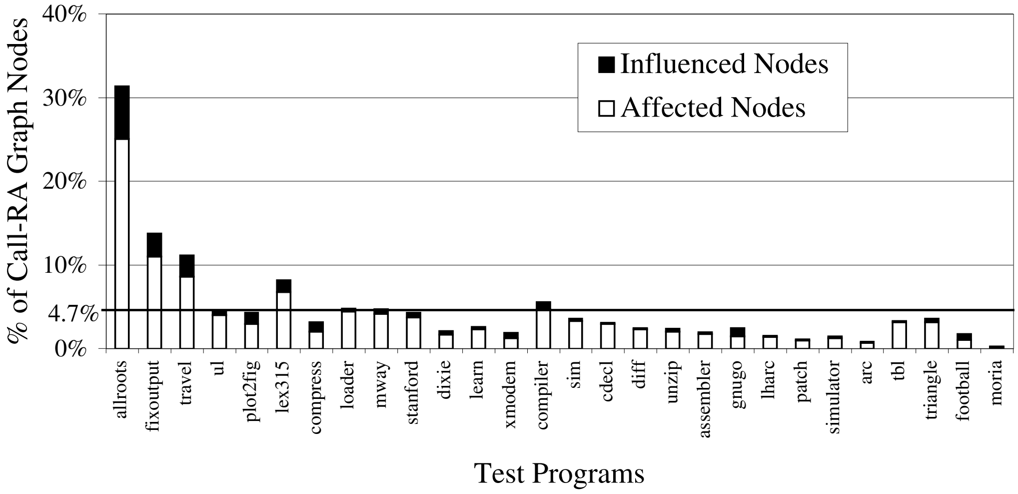


Figure 6.5: Impact on the *PMOD* solution of a positive test for each experiment program

Next, we examine the average impact of a positive test by test category. Figure 6.6 gives the overall average percentages of call-RA graph nodes affected and influenced, respectively, in a positive test for each test category. The impact of deleting a non-pointer statement is quite limited. Actually, if the variable modified by a non-pointer statement is a local variable, then it will just affect the *PMOD* solution of the containing procedure. However, if the modified variable is a global, the effect can be large, depending on the depth of the corresponding call-RA node in the graph. Deleting another category of statement may affect the *PMOD* solution to a larger extent.

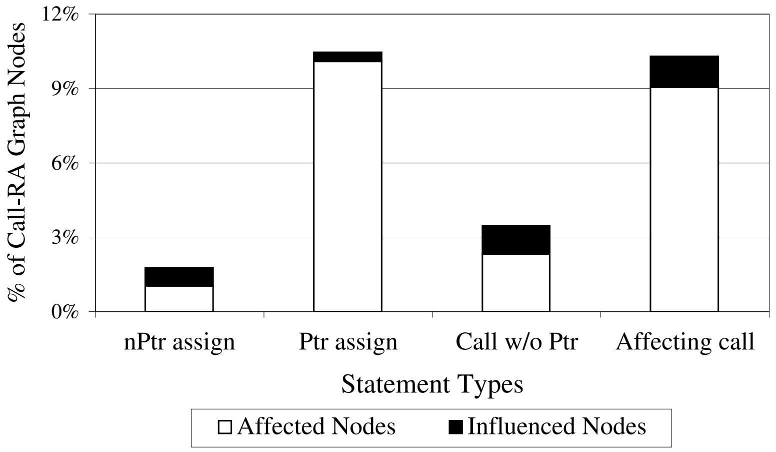


Figure 6.6: Impact on the *PMOD* solution of a positive test for each test category

Comparing the total number of call-RA graph changes in Figures 6.4 and the total impact (affected or influenced nodes) on $PMOD$ in 6.6, we can see that a source change that causes a larger call-RA graph change usually causes a larger impact on $PMOD$. This proportionality may open a door to finding an index that can be used to determine when it is worthwhile to update the solution incrementally, and when this will be tantamount to using an exhaustive approach.

6.3 Empirical Effectiveness of Incremental Aliasing Analysis

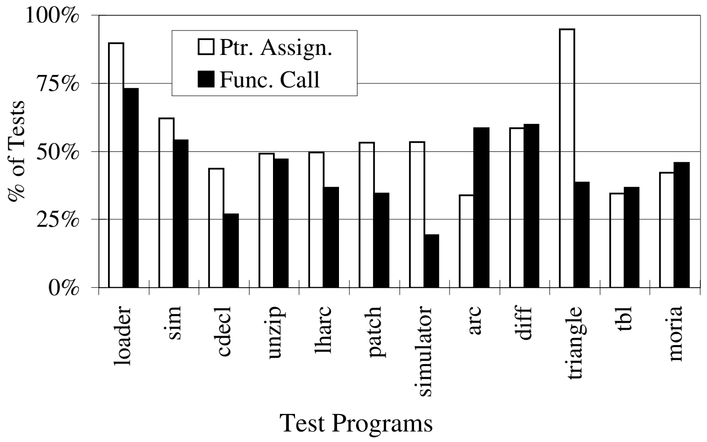


Figure 6.7: Percentage of tests that are active

For the pointer aliasing problem, a test is called *active* if it causes changes to the aliasing solution at some program points other than the deleted statement. Deleting an affecting function call may cause those functions called directly or indirectly to become unreachable. Since the aliasing solution at the unreachable functions can simply be discarded due to their unreachability, a test on an affecting function call is *active* only if it causes changes to the aliasing solution in some reachable function. Sometimes the pointer effect of a statement may be duplicated by other pointer-related statements within the function, or may have been propagated to the statement from the calling function; deleting such a statement may not change the aliasing solution, so this test is *inactive*. Figure 6.7 shows the percentages of tests which are active by deleting a pointer assignment and an affecting function call respectively, with the programs ordered left to right by the number of $ICFG$ nodes. Deletion of an affecting function call does not appear more likely to be active than deletion of a pointer assignment. This may be because the aliasing effect of a called function is *modular*; a function tends to manipulate the aliasing among its local variables, and/or propagate the aliasing information to the functions it calls. Because the aliasing effect is mainly on the local variables, when a call to this type of function is deleted, it will not cause a great impact on aliasing information of the calling function. An inactive test does not imply an easy incremental update, but an active test potentially causes more work for the incremental algorithm than an inactive one. Thus, in the following empirical measurements, the averages were taken over the active tests only, and doing so makes the obtained results more conservative.

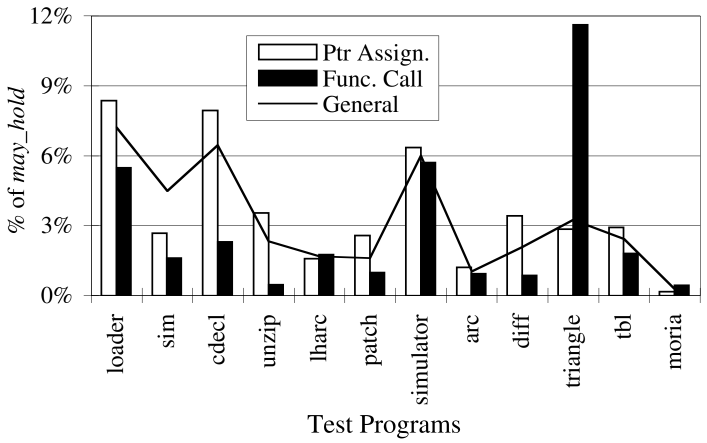


Figure 6.8: Impact on the aliasing solution in an active test

For each program, we measured the average impact on the aliasing solution of deleting one interesting statement by calculating the average number of changed aliases as a percentage of the total number of aliases[3] for the original program. Figure 6.8 gives as bars the percentages for active tests on the two types of interesting statements. It also shows a *general average* across all programs as a polygonal line; each of the 12 points on the line represents the average percentage impact over all interesting tests for a particular program. For example, for `triangle`, 3% of the aliases are affected on average by deletion of an active pointer assignment, whereas 11.5% are affected by deletion of an active affecting function call, and the general average is 3.4%. The *overall average*, or average of the 12 points mentioned above, is about 3.2%; this indicates that a minor

[3] An alias is a tuple (n, RA, PA) such that $may\_hold(n, RA, PA) = YES$.

source code change, such as deleting a statement, generally does not have great impact on the aliasing solution. By averaging over each interesting statement type separately, we find that deleting a pointer assignment usually causes a greater impact (3.6%) on the aliasing solution than deleting an affecting function call (2.4%). Usually, the impact on the aliasing solution of a source change in a small program is greater than that in a large program. However, another factor is sometimes more dominant than the program size, namely how the aliases are created and propagated in the program. In `triangle`, most of the active tests on affecting function calls are calls to some functions which create and manipulate aliases on some global data structures in the `main` function. Thus, deleting such a function call will cause a large impact on the aliasing solution.

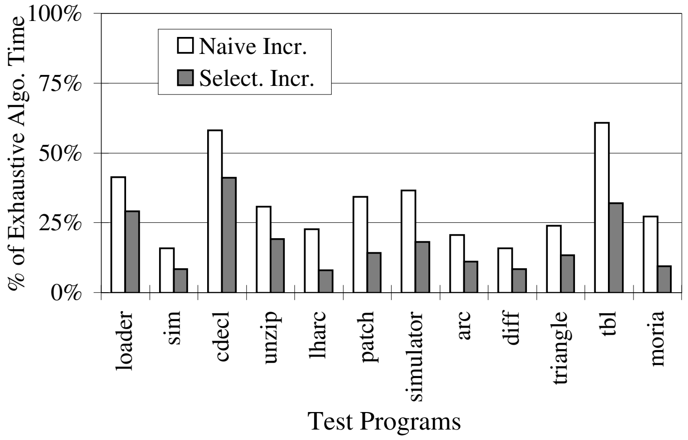
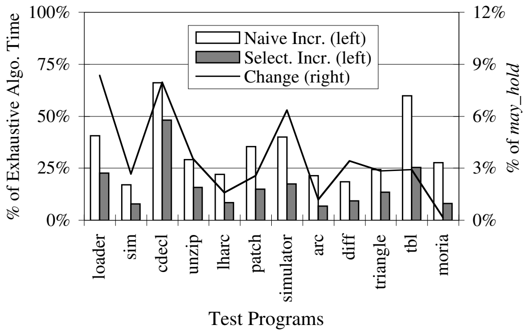
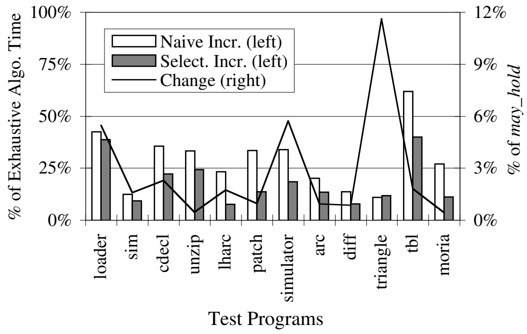


Figure 6.9: Execution time of the incremental algorithms for handling deletion of one interesting statement

To demonstrate the time savings of our incremental algorithm, we calculated the *relative execution time* spent by the incremental algorithm using the two falsification strategies (naive and selective) to update the aliasing solution as a percentage of the execution time for recomputing the updated aliasing solution exhaustively. The bar chart in in Figure 6.9 shows the average relative execution times of the naive and selective incremental algorithms in an active test, respectively. The selective incremental algorithm is superior to the naive one. While the overall average for the naive incremental algorithm is 33%, that for the selective incremental algorithm is about 17%, which is a 6 times speedup over the exhaustive algorithm. For large test programs like `moria`, the average speedup is as high as 11. Comparing the execution time with the size of impact in Figure 6.8, we have observed that the execution time of our incremental algorithms does not show strong correlation with the *size* of impact, but instead, with *how the aliases are used and propagated*. Carefully examining the test cases in which the incremental algorithm is efficient in handling a source change that causes a large impact, we found that if most of the impact caused by a source change is either addition of new aliases or deletion of old aliases (especially by implicit disabling), but not both, the incremental algorithm generally delivers much better performance than the exhaustive algorithm in handling such a change. On the other hand, if the set of aliases falsified by the falsification phase of the incremental algorithm has a large overlap with the set generated by the reiteration phase (which is usually caused by multiple paths for generating the same aliases at a program point), the incremental algorithm will be less efficient.



(a) Pointer-assignments



(b) Affecting function calls

Figure 6.10: Execution time for handling deletion of different kinds of interesting statements

In Figure 6.10, we further show the execution time of the incremental algorithms for handling deletion of pointer assignments and affecting function calls respectively. In the same figure, the average impact on the aliasing solution (shown as a line scaled with the right y-axis) is superimposed for comparison. For the deletion of a pointer assignment, the general average relative execution times of the naive and selective incremental algorithms are 35% and 16% respectively. For the deletion of an affecting function call, the general average relative execution times are 28% and 17% respectively.

In chapter 4, we have explained that approximation may occur while the incremental

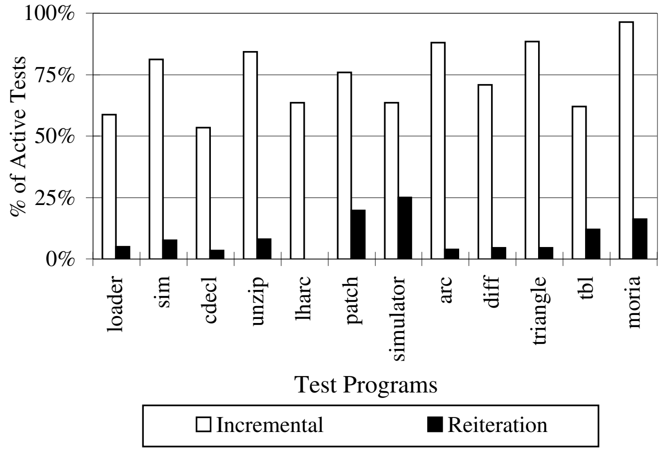
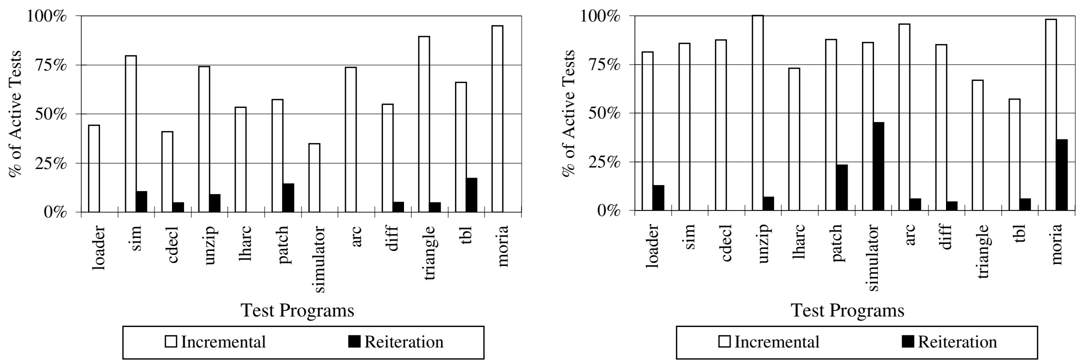


Figure 6.11: Percentage of active tests which are hits

algorithms reuse the old solution. A solution obtained by an incremental algorithm is a *hit*, if it is the same as the solution computed by the exhaustive one; the *hit rate* for a program is the percentage of tests in which the incremental algorithm achieves a hit, a measure of incremental precision. Figure 6.11 shows the hit rates for our proposed incremental algorithm[4] versus reiteration without any reinitialization (which also computes a safe but less precise solution); the average hit rates are 74% and 9% respectively. In most of the tests, the incremental algorithm computes the same solution as the exhaustive one applied to the changed program.



(a) Pointer-assignments (b) Affecting function calls

Figure 6.12: Hit rates for handling deletion of different kinds of interesting statements

We also show the hit rates of the incremental algorithms for handling deletion of pointer assignments and affecting function calls in Figure 6.12 respectively. The

[4]The naive and selective incremental algorithms will demonstrate the same hit rate.

observed fact that most of the aliasing effect of a called function is on its local variables seems to have effect on the hit rates, also. The general averages for deletion of pointer assignments and affecting function calls are 84% and 65% respectively.

The empirical results demonstrate that the incremental algorithm achieves a multifold speedup over the exhaustive algorithm, and the sacrifice of the precision is limited compared with the reiteration without reinitialization.

6.4 Impact on an Example Application: USE/MOD through Dereferenced Pointers

Program	# USE-thru-deref	# MOD-thru-deref
loader	108	78
sim	226	130
cdecl	214	25
diff	137	100
unzip	69	50
lharc	159	123
patch	155	132
simulator	169	107
arc	223	158
tbl	132	277
triangle	1105	241
moria	2421	1382

Figure 6.13: Numbers of static USE/MOD through dereferenced pointers

We have discussed the impact of deleting a source statement with aliasing effect on the aliasing solution. However, for an application using the aliasing information, at a certain program point n, only a subset of the aliases which may hold at n is relevant, thus the impact on the application caused by deleting a source statement may be different than that on the full aliasing solution. The application we studied is *USE/MOD-thru-deref*, which computes the set of fixed locations used/modified through dereferenced pointers (e.g., `x=*p` for USE-thru-deref and `(*q).a=y` for MOD-thru-deref). We compare two versions of USE/MOD-thru-deref: one uses our incremental aliasing algorithm for handling source changes, and the other uses the trivial algorithm (i.e., reiteration without any reinitialization.) Since the computation for the

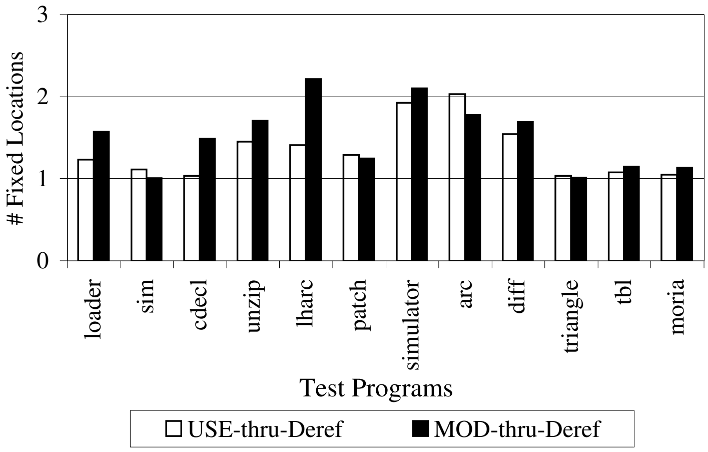


Figure 6.14: Average numbers of fixed locations used/modified through dereferenced pointers at a USE/MOD-thru-deref site

aliasing information occupies a major portion of the execution time for USE/MOD-thru-deref, the speedup delivered by the incremental aliasing will greatly improve the performance for the USE/MOD-thru-deref analysis. Of course, the trivial algorithm will be much faster than our incremental algorithm, but we can anticipate that its imprecision in the aliasing information will definitely affect the precision of the final USE/MOD-thru-deref solutions.

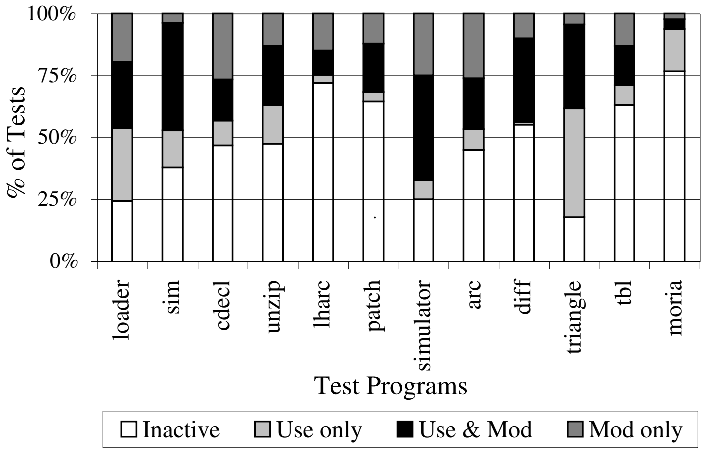
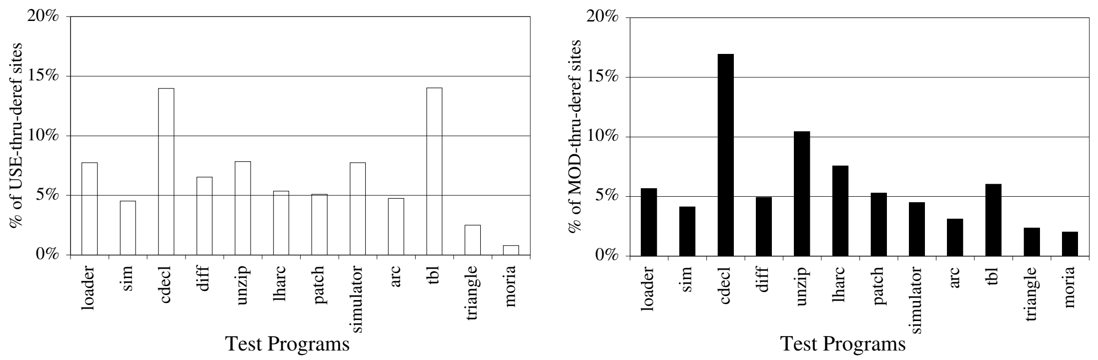


Figure 6.15: Percentages for inactive tests, tests which affect USE-thru-deref only, tests which affect MOD-thru-deref only, and tests which affect both USE/MOD-thru-dere

The table in Figure 6.13 lists the number of USEs/MODs through dereferenced pointers for each test program, and Figure 6.14 further shows the average number of fixed locations[5] referenced at a USE/MOD-thru-deref site by using the aliasing solution obtained by the flow- and context-sensitive analysis. Then for each active test in the incremental aliasing analysis (i.e., a test that impacts the aliasing information), we compare the solution of USE/MOD-thru-deref obtained by the incremental analyses with that from the exhaustive approach. Figure 6.15 gives the respective percentages for those tests of four categories: tests which do not affect USE/MOD-thru-deref (i.e., *inactive tests*), tests which affect USE-thru-deref only, tests which affect MOD-thru-deref only, and tests which affect both USE-thru-deref and MOD-thru-deref. A test is an *active* test for the USE-thru-deref (MOD-thru-deref) problem if it causes impact on USE-thru-deref (MOD-thru-deref). In an active test for USE-thru-deref (MOD-thru-deref), we measured the impact of the source change by computing the *affected region* (i.e., with a changed solution) as a percentage of the USE-thru-deref (MOD-thru-deref) sites whose USE-thru-deref (MOD-thru-deref) solutions change after the source change. A general average over all active tests for each program is then computed, and Figure 6.16 (a) and (b) show the general average percentages for USE-thru-deref and MOD-thru-deref across all programs respectively. The larger the affected region is, the more the programmer uses aliases in manipulating data across the program; we observed that a large program (e.g., `moria`) usually has a smaller affected region.



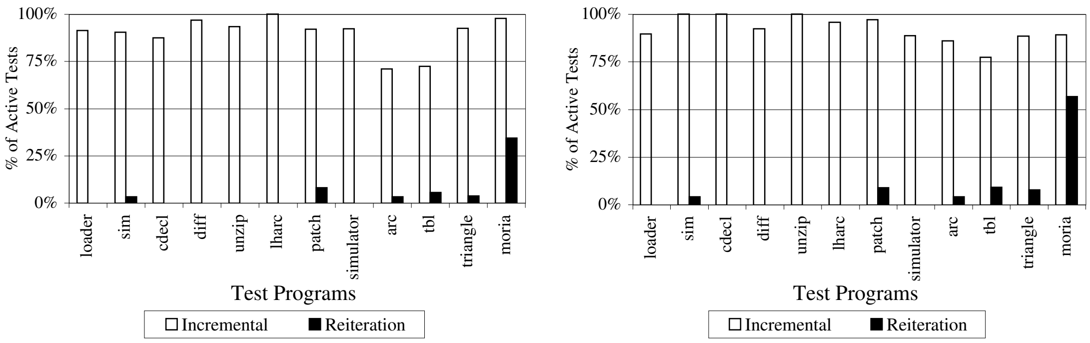
(a) USE-thru-Deref

(b) MOD-thru-Deref

Figure 6.16: Average percentages of USE/MOD-thru-deref sites whose solutions are affected by the source change

A test is a *hit* for the incremental analysis if it computes the same solution as the

[5]The measurement using the aliasing information computed by the flow- and context-insensitive analysis will be less precise [SRLZ98].



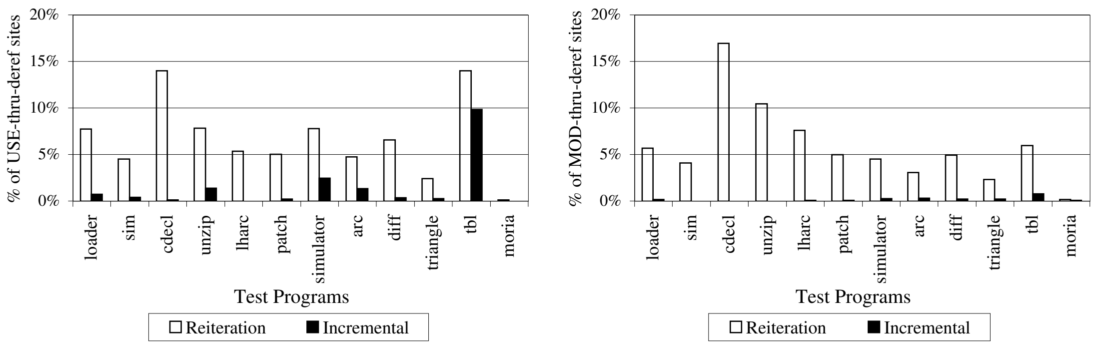
(a) USE-thru-Deref

(b) MOD-thru-Deref

Figure 6.17: Percentage of tests which have are hits for the USE/MOD-thru-Deref problems

exhaustive approach at all the USE/MOD-thru-deref sites. Figure 6.17 (a) and (b) show the respective hit rates on USE- and MOD-thru-deref of all *active* tests for our proposed incremental algorithm versus the trivial algorithm. The average hit rates over all the test programs for our incremental algorithm and the trivial algorithm are 90% and 4.8% for USE-thru-deref, and 92% and 7.5% for MOD-thru-deref, respectively.

The hit rate gives us the percentage of *active* tests in which the incremental approach delivers the same quality of solution as the exhaustive one. However, when a test is a *miss* (that is, the solution that the incremental approach computes is less precise), we then compute the percentage of the USE- or MOD-thru-deref sites which have different solution from the one computed by the exhaustive approach, called the *imprecise region.* Figure 6.18 (a) and (b) give the general averages of the percentage over all active tests for each program on USE- and MOD-thru-deref respectively. For USE-thru-deref, the overall average percentages for our incremental algorithm and the trivial algorithm are 1.4% and 6.7% respectively; for MOD-thru-deref, 0.2% and 5.9% respectively. That is, when using our incremental approach, on average, about 1.4% of the USE-thru-deref sites will have less precise solutions than that obtained by the exhaustive approach, but when using the trivial algorithm, the imprecise region is 6.7% of the USE-thru-deref sites. Therefore, we can see that for an application (such as USE/MOD-thru-deref) of our proposed incremental aliasing analysis will benefit by the speedup in computing the aliasing information, and will compute much more precise solutions than the trivial one.



(a) USE-thru-Deref (b) MOD-thru-Deref

Figure 6.18: Percentage of USE- or MOD-thru-deref sites which have different solutions from that obtained by the exhaustive approach

Chapter 7

EXTENSIONS OF THE INCREMENTAL ALIASING ALGORITHMS

We have discussed the incrementalization of two data-flow problems, where the pointer aliasing problem for C is flow- and context-sensitive, and the *PMOD* problem is context-sensitive. In this chapter, we will further discuss some issues of our incremental approach and how the approach can be extended to handle other flow- and context-sensitive data-flow algorithms.

7.1 The Incremental Aliasing Algorithm in the Presence of Multiple Changes Reconsidered

When a source change is made at a program point n, the naive incremental algorithm basically treats all n's intra-procedural and inter-procedural successors potentially affected, and just "falsifies" the aliasing information at those successors. This strategy looks inefficient in reusing aliases, that is, it may regenerate lots of aliases which are falsified but still reusable, and thus increases the burden of the later reiteration phase. However, the strategy is straightforward and "fast" for the alias falsification phase, and it is much less sensitive to the type of source change. As for the selective incremental algorithm, the extra computation in the falsification reduces the number of aliases falsified and may also reduce the scale of the potentially affected region, and therefore reduces the workload of the reiteration phase. The selective incremental algorithm requires more knowledge of how the source changed may affect the aliasing information propagated to it, and is thus more sensitive to the type of source change than the naive one.

In addition, multiple changes may have different effect on these two incremental algorithms. When multiple source changes are present, in practice they usually demonstrate lexical locality (e.g., within one procedure), which is beneficial to our incremental algorithms. High lexical locality among the multiple source changes will make their potentially affected regions have a great deal of overlap, i.e., *semantic locality*. Obviously, the naive algorithm will benefit more from the overlap than the selective one.

In order to combine the advantages of both algorithms, we will extend our incremental approach by uniting these two algorithms. The *united incremental approach* is based on the demand-driven version of our incremental algorithm. From the selective incremental algorithm, we have seen that to minimize the set of aliases falsified requires the knowledge of what aliases the changed statement may affect, *and* how the effect propagates. That is, when a source change is made at a program point n, to avoid unnecessary falsification on an alias, the algorithm should try to identify the program point as close to n as possible at which the alias will still hold after the change. Take the program in Figure 7.1 as an example, where the pointer assignment `b: p=&x;` is to be deleted. Since the statement generated alias `<*p,x>`, to handle its deletion, the falsification phase of our incremental approach will falsify the alias at b's successors. However, the falsification is unnecessary, since alias `<*p,x>` can also be generated at program point a, and thus the incremental algorithm will regenerate the alias in the reiteration phase. The ideal case is to identify that the deletion of the statement will not affect the aliasing information at other points, but this is difficult; loops and procedure calls (especially recursive calls) make it even more complicated. Though the ideal case may not be easy to achieve, if an algorithm can identify unnecessary falsification as close to the ideal case as possible, it will help speed up the update greatly. For the example, our united incremental approach (explained later) will identify that the falsification of `<*p,x>` at program point `c` is not necessary.

When a change is made at program point n, the idea of the united incremental approach is to divide the potentially affected region (i.e., the set of all n's intra- and inter-procedural successors) obtained by the naive incremental algorithm into two parts: one part (called *strongly affected sub-region*) contains all the program points in the

```
int x,*p;
main()
{
   a:p=&x;
   while (...) {
         :
      b:p=&x;
         :          to be deleted
   }
   c:...
     :
   d:*p=...;
}
```

Figure 7.1: Identify the unnecessary falsification

region which can reach n[1], and the other (called *weakly affected sub-region*) contains all the other program points which can never reach n. We also refer to the set of program points in the weakly affected sub-region which are adjacent to the strongly affected sub-region as the *cut set*[2] for the source change at n. Since there is no way for the program points in the weakly affected sub-region to affect the aliasing information at the program points in the strongly affected sub-region, the bisection of the potentially affected region also provides a way to divide the incremental update into two parts. The cut set will act as a checkpoint for the incremental update. That is, we can update the aliasing solutions in the strongly affected sub-region first; then at each program point in the cut set, we compare the new aliasing solution with the old one and find which aliases will not hold after the source change at the point and should be falsified. Then, we perform incremental update on the weakly affected sub-region. Of course, if the aliasing solutions along the cut set do not change at all, the weakly affected sub-region is updated automatically and we save the computation of falsification and regeneration of aliases. We will use the naive algorithm and the selective algorithm on the strongly affected sub-region and the weakly affected one respectively.

Back to the example in Figure 7.1, when the statement `b:p=&x;` is to be deleted, the

[1]The strongly affected sub-region can be obtained by taking a backward walk from the changed program point n, and a node being visited is in the strongly affected sub-region if it is also an intra- or inter-procedural successor of n. This is similar to the computation of strongly connected components, but on the interprocedural control flow graph.

[2]For most cases, the cut set will be just a singleton.

whole `while` statement and it succeeding statements are potentially affected, that is, their aliasing solutions may be affected by the source change. In the potentially affected region, the program points in the `while` statement that can reach the program point `b` form the strongly affected sub-region; on the other hand, the program points which succeed the `while` statement and cannot reach back to program point `b` form the weakly affected sub-region, and {`c`} is the singleton cut set. For handling such a source change, the united incremental approach will first apply the naive incremental algorithm to the strongly affected sub-region, that is, the aliasing solutions at the program points in the while statement will be falsified, the aliases at program point `a` will be reintroduced, and the reiteration will be performed on the `while` statement. During the reiteration, aliases propagated to program point `c` will be buffered, and when the reiteration is finished, the aliasing solutions in the strongly affected sub-region are updated. Then, by comparing the new aliasing solution and the old one at program point `c`, we will find the aliasing solution does not change, and no further computation is needed on the weakly affected sub-region. The united incremental approach will handle multiple changes in a similar manner. The set of all the successors of the changed nodes form the potentially affected region, and those nodes in the region which are also predecessors of any changed node form the strongly affected sub-region. Then we can just follow the procedure mentioned above for handling a single source change to update the solutions for the strongly and weakly affected sub-regions. The locality of multiple changes makes the strongly affected sub-region of a change to have a great overlap or even to co-reside with those of others. The larger the overlap is, the less proportional the execution time of the naive algorithm will be to the number of changes.

7.2 Information Optimization and Incrementalization

Usually, a data-flow algorithm will try to speed up the computation by preventing data-flow information from being propagated to irrelevant program points. One commonly used method is to approximate the set of possibly used variables in each procedure, and in the data-flow analysis, data flow information irrelevant to a procedure will not be propagated to it. This can save computation for the data-flow analysis, but for the incremental analysis, when a change is made, the information for such an optimization should be updated (exhaustively or even incrementally). Then based on the updated information, if data-flow information which was irrelevant to a procedure now becomes relevant, it should be repropagated to the procedure. On the other hand, some data-flow information which was relevant may now become irrelevant. Though not necessary, for reasons of consistency, it would be better to remove the data-flow information already propagated.

7.3 Flow-Sensitive Data-Flow Analysis

For the pointer aliasing problem, although our incremental algorithm is based on the Landi-Ryder algorithm, the framework can be adapted to incrementalize other flow- and context-sensitive data-flow algorithms, which tag the data-flow information with a representation of calling context. The general approach involves

1. obtaining a safe initial estimate for the to-be-updated solution,
2. using an efficient worklist algorithm for calculating the updated solution and
3. performing delayed updates within called functions, by disabling possibly affected calling contexts and sometimes subsequently reusing memorized values.

The key point for applying our incremental approaches to other flow- and context-sensitive data-flow algorithms is tagging the context which generates a data-flow fact to the data-flow fact. For example, the framework of the Landi and Ryder pointer aliasing algorithm will work on the points-to analysis [EGH94][3], in which case a tagging reaching alias will be a points-to relation. Our incremental approaches (naive and selective) will also work to update the points-to solution.

In addition, since approximation may be needed in the context tagging, just like the underlying aliasing algorithm for our incremental analysis, how the approximation will

[3] Rather than embedding the context in the abstraction being estimated, Emami *et.al* [EGH94] explicitly represented all invocations in an invocation graph

be accumulated over the course of source changes and incremental updates should also be investigated in incrementalizing an algorithm.

7.4 Context-Sensitive Data-Flow Analysis

Our incremental algorithm for the *PMOD* problem is based on the Marlowe-Ryder's hybrid framework, which partitions the flow graph into regions and then factors the computation of *PMOD* for a node in each region into three local problems.

1. collecting the data-flow information restricted to the region at that node
2. determining the set of exit nodes of the region reachable from that node
3. determining the set of exit nodes of the region reachable through a non_visible binding chain, which is later used to compute the mapping function of the data-flow information from a reachable exit node to that node

When handling different context-sensitive algorithms, we first need to define the corresponding *call-context graph* (e.g., the call-RA graph for our problem), where each node is a procedure, calling context pair (P, C), and an edge, say $((P, C),(Q, D))$, denotes that there is a call to procedure Q in procedure P, and with a calling context C at procedure P, the call causes the calling context D to occur at procedure Q. The call-context graph will then be partitioned into regions with single entry (which will be further discussed later), and the reduced graph should be acyclic. Then a context-sensitive data-flow problem usually involves collection of data-flow information from procedures with matched contexts. With the region partition, part of the data-flow information needed to compute the final solution at a node will be from its containing region, and the other will be from other regions.

Among those three local problems for our incremental algorithm, the first one characterizes the collection of the intra-region data flow information; the second one basically computes the set of exit nodes of the region reachable from the node. These two local problems are usually needed and insensitive to the target data-flow problem. What is left is to map the data-flow information from other regions gathered at the exit nodes back to the scope of the target node, and this is data-flow problem dependent. If the local problem for back mapping can be formulated as a reachability problem (like the third local problem for incremental *PMOD*) then our incremental algorithm can be easily adapted to handle a different data-flow problem.

7.5 Partition of the Flow Graph Reconsidered

The representative problem for region $\mathcal{R}$ in the Marlowe and Ryder's hybrid algorithm serves to abstract data-flow information from other regions and to compute the region's effect on the abstract information. The reason why the Marlowe and Ryder's hybrid algorithm requires each region to have only one entry is because the representative problem for a region does not include the entry as a parameter. A region can have multiple entries if the formulation of the representative problem is parameterized with the entry; that is, the abstract data-flow information will have to be parameterized with the entry.

Instead of explicitly performing the region's effect on the abstract external information, the representative problem in our problem factorization computes the reachable entries (local problem #2) and then summarizes the effect from a reachable entry to an interior node local problem #3). Then, in the global propagation phase, we will know which part of external information propagated to a region should be collected, and how to map the collected information to a target node. The formulation of the backmapping is specific to the entry receiving the external data-flow information. Thus as long as the reduced graph is acyclic, the single-entry requirement on region partition can be lifted for our incremental *PMOD* algorithm. There are many ways to partition the flow graph, such as acyclic regions, cut point decomposition, dominator decomposition, etc. [Mar89]. For the incremental analysis, since the quality (e.g., locality) of multiple source changes is strongly related to the semantics of a program and may affect the performance of the incremental analysis, the program's semantics should also be considered in choosing a region partition.

Chapter 8

SUMMARY AND FUTURE WORK

8.1 Summary

Basically, the work in this thesis is dealing with the incrementalization of the exhaustive algorithms for solving two major subproblems, pointer aliasing and *PMOD*, of the interprocedural modification side effect problem for C.

For the pointer aliasing problem, we have incrementalized Landi & Ryder's flow- and context-sensitive exhaustive aliasing algorithm to handle deletion and addition of single statements in a C program. Based on the general incremental approach, two versions of incremental aliasing algorithms (naive and selective) were proposed. The studies of deleting pointer assignments and affecting function calls using 12 C programs, show the promise of the incremental algorithm. On average, a 6-fold speedup of incremental over exhaustive analysis was observed. The relatively small impact (overall average 3.2% of all aliases in unchanged program) of a single source statement deletion was documented. Acceptable precision (on average solution agreement on 75% of tests) was demonstrated. We have extended the base incremental algorithm to support query-based analysis.

For the *PMOD* problem, we proposed a new program representation, the call-RA graph, to describe the relation of the calling contexts between procedures. With Marlowe and Ryder's hybrid framework, we incrementalized the computation of *PMOD* to handle non-structural and structural changes to the call-RA graph. We conducted a feasibility study of 28 programs by measuring the impact of deleting one interesting statement on the structure of call-RA graph and the *PMOD* solution. We categorized the interesting statements into four types: non-pointer assignments, pointer assignments, affecting function calls, and function calls without pointer effect. Deletion of an interesting statement without pointer effect cause much less impact on the call-RA graph and the *PMOD* solution than deletion of an interesting statement with pointer effect. On average over the active tests, only 4.7% of the call-RA graph nodes are either affected or influenced by deletion of an interesting statement. This result reveals great opportunities for our incremental technique.

8.2 Future Work

With the promising results from our first attempt to incrementalize the exhaustive flow- and context-sensitive data-flow algorithm, we are considering extending the work along several dimensions:

- optimize the prototype for better time- and space-efficiency.
- study the sensitivity of performance of the incremental algorithm to the number of multiple source changes.
- extend the algorithm to handle a broader class of source changes, such as changes of control flow (e.g., addition/deletion of control flow edges) and coarse-grain source changes (e.g., changes to a block, or a function).
- apply this by consolidating several fine-grain source changes with great locality to a single coarse-grain source change. (For example, if a function has been changed greatly, it can be treated as deletion of the function, followed by addition of a "new" function which is basically the changed function. Then, instead of handling many statement-level source changes, the incremental analysis can treat them as one procedure-level source change.)
- Since the incremental algorithm does not always outperform the exhaustive algorithm, seek a factor which can approximately "predict" the impact of a source change and the effectiveness of the incremental algorithm for handling the change. Such a predictive factor can help determine when to use the incremental algorithm to handle the change, and when not to.
- prototype the query-based incremental algorithm, and empirically study its performance.

- A direction which seems orthogonal to our current incremental analysis, but will strongly affect the design of the incremental algorithm and its application in a software development tool is to study the behavior of the real-world source changes that a user will make with a software tool. It might include the lexical and semantic relation (e.g., locality) of the source changes, and when and where the data flow information is used.

References

[AC76] F. E. Allen and J. Cocke. A program data flow analysis procedure. *Communications of the ACM*, 19(3):137–147, 1976.

[AG96] D. Atkinson and W. Griswold. The design of whole-program analysis tools. In *Proceedings of the 18th International Conference on Software Engineering*, pages 16–27, 1996.

[AG98] D. Atkinson and W. Griswold. Effective whole-program analysis in the presence of pointers. In *Proceedings of the ACM SIGSOFT '98 Symposium on the Foundations of Software Engineering*, pages 46–55, November 1998.

[And94] L. O. Andersen. *Program analysis and specialization for the C programming language*. PhD thesis, DIKU, University of Copenhagen, 1994. Also available as DIKU report 94/19.

[ASU86] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers: Principles, Techniques, and Tools*. Addison-Wesley, 1986.

[Ban79] J. Banning. An efficient way to find the side effects of procedure calls and the aliases of variables. In *Conference Record of the Sixth Annual ACM Symposium on Principles of Programming Languages*, pages 29–41, January 1979.

[Bar78] J. M. Barth. A practical interprocedural data flow analysis algorithm. *Communications of the ACM*, 21(9):724–736, 1978.

[BCCH94] Michael Burke, Paul Carini, Jong-Doek Choi, and Michael Hind. Flow-insensitive interprocedural alias analysis in the presence of pointers. In *Proceedings of the 7th International Workshop on Languages and Compilers for Parallel Computing*, pages 234–250. Springer-Verlag, August 1994.

[BH93] S. Bates and S. Horwitz. Incremental program testing using dependence graphs. In *Conference Record of the Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 384–396, January 1993.

[BJ78] W. A. Babich and M. Jazayeri. The method of attributes for data flow analysis, part II: Demand analysis. *Acta Informatica*, 10:265–272, 1978.

[BR90] M. Burke and B. G. Ryder. A critical analysis of incremental iterative data flow analysis algorithms. *IEEE Transactions on Software Engineering*, 16(7), July 1990.

[Bur84] M. Burke. An interval analysis approach toward interprocedural data flow analysis. Computer Science Technical Report RC 10640, IBM Thomas J. Watson Research Center, Yorktown Heights, New York, July 1984.

[Bur90] M. Burke. An interval-based approach to exhaustive and incremental interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 12(3):341–395, July 1990.

[Car88] M. D. Carroll. *Dataflow Update via Attribute and Dominator Update*. PhD thesis, Department of Computer Science, Rutgers University, May 1988.

[CBC93] Jong-Deok Choi, Michael Burke, and Paul Carini. Efficient flow-sensitive interprocedural computation of pointer-induced aliases and side effects. In *Conference Record of the Twentieth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 232–245, January 1993.

[Cha99] R. Chatterjee. *Modular Data-flow Analysis of Statically Typed Object-oriented Programming Languages*. PhD thesis, Department of Computer Science, Rutgers University, October 1999. in preparation.

[CK84] K. Cooper and K. Kennedy. Efficient computation of flow insensitive interprocedural summary information. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 247–258, June 1984. SIGPLAN Notices, Vol 19, No 6.

[CK87] K. Cooper and K. Kennedy. Complexity of interprocedural side-effect analysis. Computer Science Department Technical Report TR87-61, Rice University, October 1987.

[CK88] K. Cooper and K. Kennedy. Interprocedural side-effect analysis in linear time. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 57–66, June 1988.

[Coo85] K. Cooper. Analyzing aliases of reference formal parameters. In *Conference Record of the Twelfth Annual ACM Symposium on Principles of Programming Languages*, pages 281–290, January 1985.

[Coo89] B. G. Cooper. *Ambitious Data Flow Analysis of Procedural Programs*. Master's thesis, University of Minnesota, May 1989.

[Cou86] D. S. Coutant. Retargetable high-level alias analysis. In *Conference Record of the Thirteenth Annual ACM Symposium on Principles of Programming Languages*, pages 110–118, January 1986.

[CR88] M. D. Carroll and B. G. Ryder. Incremental data flow analysis via dominator and attribute updates. In *Conference Record of the Fifteenth Annual ACM Symposium on Principles of Programming Languages*, pages 274–284, January 1988.

[CR99] Ramkrishna Chatterjee and Barbara G Ryder. Data-flow-based testing of object-oriented libraries. Department of Computer Science Technical Report DCS-TR-382, Rutgers University, March 1999.

[CRL99] Ramkrishna Chatterjee, Barbara G. Ryder, and William. A Landi. Relevant context inference. In *Conference Record of the Twenty-sixth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, January 1999.

[CWZ90] D. R. Chase, M. Wegman, and F. K. Zadeck. Analysis of pointers and structures. In *Proceedings of the SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 296–310, June 1990. SIGPLAN Notices, Vol 25, No 6.

[Deu90] A. Deutsch. On determining lifetime and aliasing of dynamically allocated data in higher-order functional specifications. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 157–168, January 1990.

[Deu94] A. Deutsch. Interprocedural may alias for pointers: Beyond k-limiting. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 230–241, June 1994.

[DGS95] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. Demand-driven computation of interprocedural data flow. In *Conference Record of the Twenty-second Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, January 1995.

[DGS97] Evelyn Duesterwald, Rajiv Gupta, and Mary Lou Soffa. A practical framework for demand-driven interprocedural data flow analysis. *ACM Transactions on Programming Languages and Systems*, 19(6):359–370, 1997.

[EGH94] M. Emami, R. Ghiya, and L. J. Hendren. Context-sensitive interprocedural points-to analysis in the presence of function pointers. In *Proceedings of the SIGPLAN '94 Conference on Programming Language Design and Implementation*, pages 242–257, June 1994. Published as SIGPLAN Notices, 29 (6).

[Ema93] Maryam Emami. A practical interprocedural alias analysis for an optimizing/parallelizing C compiler. Master's thesis, McGill University, Montreal, Canada, July 1993.

[FI98] P. Frankel and O. Iakounenko. Further empirical studies of test effectiveness. In *ACM SIGSOFT '98 Sixth International Symposium on the Foundations of Software Engineering*, pages 153–162, November 1998.

[FW93] P. Frankel and S. Weiss. An experimental comparison of the effectiveness of branch testing with data-flow testing. *IEEE Transactions on Software Engineering*, 19(8):774–787, August 1993.

[GDDC97] D. Grove, G. DeFouw, J. Dean, and C. Chambers. Call graph constrction in object-oriented languages. In *Proceedings of ACM SIGPLAN Conference on Object-oriented Programing Systems, Languages and Applications (OOPSLA'97)*, pages 108–124, October 1997.

[GH96a] R. Ghiya and L. Hendren. Connection analysis: A practical interprocedural heap analysis for c. *International Journal of Parallel Programming*, 1996.

[GH96b] R. Ghiya and L. Hendren. Is it a tree, a dag or a cyclic graph? A shape analysis for heap-directed pointers in C. In *Conference Record of the Twenty-third Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 1–15, January 1996.

[GH98] R. Ghiya and L. Hendren. Putting pointer analysis to work. In *Conference Record of the Twenty-fifth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 121–133, 1998.

[GL91] K. Gallagher and J Lyle. Using program slices in software maintenance. *IEEE Transactions on Software Engineering*, 17(8):751–761, August 1991.

[GS96] R. Gupta and M. L. Soffa. Hybrid slicing: An approach for refining static slices using dynamic information. In *Proceedings of Third ACM SIGSOFT Symposium on Foundations of Software Engineering*, pages 29–40, 1996.

[Gua88] C. A. Guarna. A technique for analyzing pointer and structure references in parallel restructuring compilers. In *Proceedings of the International Conference on Parallel Processing*, pages 212–220, 1988.

[HA90] W. L. Harrison III and Z. Ammarguellat. Parcel and Miprac: parallelizers for symbolic and numeric programs. In *Proceedings of International Workshop on Compilers for Parallel Computers*, pages 329–346. Ecole des Mines de Paris - CAI, UPMC - Laboratoire MASI, December 1990. Paris, France.

[HC98] M. J. Harrold and N. Ci. Reuse-driven interprocedural slicing. In *Proceedings of the 20th International Conference on Software Engineering*, pages 74–83, April 1998.

[HFGO94] M. Hutchins, H. Foster, T. Goradia, and T. Ostrand. Experiments on the effectiveness of dataflow- and controlflow-based test adequacy criteria. In *Proceedings of the Sixteenth International Conference on Software Engineering*, pages 191–200, May 1994.

[HN90] L. Hendren and A. Nicolau. Parallelizing programs with recursive data structures. *IEEE Transaction on Parallel and Distributed Systems*, 1990.

[HP98] M. Hind and A. Pioli. Assessing the effects of flow sensitivity on pointer alias analysis. In *Proceedings of International Static Analysis Symposiusm (SAS'98)*, pages 57–81. Springer-Verlag, September 1998.

[HPR89] S. Horwitz, P. Pfeiffer, and T. Reps. Dependence analysis for pointer variables. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 28–40, June 1989.

[HR96] Mary Jean Harrold and Greg Rothermel. Separate computation of alias information for reuse. *IEEE Transactions on Software Engineering*, 22(7), July 1996.

[HRB90] S. Horwitz, T. Reps, and D. Binkley. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12(1), January 1990.

[HRS95] S. Horwitz, T. Reps, and M. Sagiv. Demand interprocedural dataflow analysis. In *Proceedings of the Third ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 104–115, October 1995.

[HS91] M. J. Harrold and M. L. Soffa. Selecting and using data for integration testing. *IEEE Software*, 8(2):58–65, March 1991.

[HS94] M. J. Harrold and M. L. Soffa. Efficient computation of interprocedural definition-use chains. *ACM Transactions on Programming Languages and Systems*, 16(2):175–204, March 1994.

[JM82a] N. D. Jones and S. Muchnick. Flow analysis and optimization of lisp-like structures. In S. Muchnick and N. Jones, editors, *Program Flow Analysis: Theory and Applications*, pages 102–131. Prentice Hall, 1982.

[JM82b] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Conference Record of the Ninth Annual ACM Symposium on Principles of Programming Languages*, pages 66–74, January 1982.

[KU76] J. B. Kam and J. D. Ullman. Global flow analysis and iterative algorithms. *Journal of the ACM*, 23(1):158–171, 1976.

[KU77] J. B. Kam and J. D. Ullman. Monotone data flow analysis frameworks. *Acta Informatica*, 7:305–317, 1977.

[Lan92] W. Landi. Undecidability of static analysis. *ACM Letters on Programming Languages and Systems*, 1(4):323–337, December 1992.

[LH88] J. R. Larus and P. N. Hilfinger. Detecting conflicts between structure accesses. In *Proceedings of the SIGPLAN '88 Conference on Programming Language Design and Implementation*, pages 21–34, July 1988. SIGPLAN NOTICES, Vol. 23, No. 7.

[LH96] L. Larsen and M. J. Harrold. Slicing object-oriented software. In *Proceedings of the 18th International Conference on Software Engineering*, pages 495–505, March 1996.

[LR91] W. Landi and B. G. Ryder. Pointer-induced aliasing: A problem classification. In *Conference Record of the Eighteenth Annual ACM Symposium on Principles of Programming Languages*, pages 93–103, January 1991.

[LR92] W. Landi and B. G. Ryder. A safe approximation algorithm for interprocedural pointer aliasing. In *Proceedings of the SIGPLAN '92 Conference on Programming Language Design and Implementation*, pages 235–248, June 1992.

[LRS+98] W. Landi, B. G. Ryder, P. Stocks, S. Zhang, and R. Altucher. A schema for interprocedural side effect analysis with pointer aliasing. Technical Report DCS-TR-336, Department of Computer Science, Rutgers University, May 1998. submitted for journal publication.

[LRZ93] W. Landi, B. G. Ryder, and S. 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.

[Mar89] T. J. Marlowe. *Data Flow Analysis and Incremental Iteration*. PhD thesis, Rutgers University, August 1989.

[MLR+93] T. J. Marlowe, W. A. Landi, B. G. Ryder, J. Choi, M. Burke, and P. Carini. Pointer-induced aliasing: A clarification. *ACM SIGPLAN Notices*, 28(9):67–70, September 1993.

[MR90] T. J. Marlowe and B. G. Ryder. An efficient hybrid algorithm for incremental data flow analysis. In *Conference Record of the Seventeenth Annual ACM Symposium on Principles of Programming Languages*, pages 184–196, January 1990.

[MR91] T. J. Marlowe and B. G. Ryder. Hybrid incremental alias algorithms. In *Proceedings of the Twentyfourth Hawaii International Conference on System Sciences, Volume II, Software*, January 1991.

[NN97] F. Nielson and H. R. Nielson. Infinitary control flow analysis: a collecting semantics for closure analysis. In *Conference Record of the Twenty-fourth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 332–345, January 1997.

[NPD87] A. Neirynck, P. Panangaden, and A. Demers. Computation of aliases and support sets. In *Conference Record of the Fourteenth Annual ACM Symposium on Principles of Programming Languages*, pages 274–283, January 1987.

[OO84] K. J. Ottenstein and L. M. Ottenstein. The program dependence graph in a software development environment. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, pages 177–184, May 1984.

[Ost90] Thomas J. Ostrand. Data-flow testing with pointers and function calls. In *Proceedings of the Pacific Northwest Software Quality Conference*, October 1990.

[PLR94] H. D. Pande, W. Landi, and B. G. Ryder. Interprocedural def-use associations for C systems with single level pointers. *IEEE Transactions on Software Engineering*, 20(5):385–403, May 1994.

[PS89] L. Pollock and M. Soffa. An incremental version of iterative data flow analysis. *IEEE Transactions on Software Engineering*, 15(12), December 1989.

[Ram94] G. Ramalingam. The undecidability of aliasing. *ACM Transactions on Programming Languages and Systems*, 16(5):1467–1471, September 1994.

[Rep94] T. Reps. Solving demand versions of interprocedural analysis problems. In *Proceedings of the Fifth International Conference on Compiler Construction*, pages 389–403, April 1994. Appeared as Lecture Notes in Computer Science, Vol 786.

[RHS95] T. Reps, S. Horwitz, and M. Sagiv. Precise interprocedural dataflow analysis via graph reachability. In *Conference Record of the Twenty-second Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 49–61, January 1995.

[RMP88] B. G. Ryder, T. J. Marlowe, and M. C. Paull. Conditions for incremental iteration: Examples and counterexamples. *Science of Computer Programming*, 11:1–15, 1988.

[RP88] B. G. Ryder and M. C. Paull. Incremental data flow analysis algorithms. *ACM Transactions on Programming Languages and Systems*, 10(1):1–50, January 1988.

[RR95] T. Reps and G. Rosay. Precise interprocedural chopping. In *Proceedings of the Third ACM SIGSOFT Symposium on Foundations of Software Engineeering*, pages 41–52, 1995.

[Ruf95] E. Ruf. Context-insensitive alias analysis reconsidered. In *Proceedings of the SIGPLAN '95 Conference on Programming Language Design and Implementation*, pages 13–22, June 1995.

[SFRW90] S. Sagiv, N. Francez, M. Rodeh, and R. Wilhelm. A logic-based approach to data flow analysis. In *Proceedings of the Second International Workshop in Programming Language Implementation and Logic Programming*, pages 277–292, August 1990. Volume 456 of Lecture Notes in Computer Science.

[SH97a] M. Shapiro and S. Horwitz. The effects of the precision of pointer analysis. In *Proceedings of the Fourth International Symposium on Static Analysis (SAS'97)*, pages 16–34, September 1997.

[SH97b] M. Shapiro and S. Horwitz. Fast and accurate flow-insensitive points-to analysis. In *Conference Record of the Twenty-fourth Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 1–14, January 1997.

[Shi91] Olin Shivers. *Control-flow Analysis of Higher-Order Languages or Taming Lambda*. PhD thesis, Carnegie-Mellon University School of Computer Science, 1991.

[SHR99] S. Sinha, M. J. Harrold, and G. Rothermel. System-dependence-graph-based slicing of programs with arbitrary interprocedural control flow. In *Proceedings of the 21st International Conference on Software Engineering*, pages 432–441, May 1999.

[SP81] 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 189–234. Prentice Hall, 1981.

[SRLZ98] P.A. Stocks, B.G. Ryder, W.A. Landi, and S. Zhang. Comparing flow- and context-sensitivity on the modification side-effects problem. In *Proceedings of the International Symposium on Software Testing and Analysis*, pages 21–31, March 1998. Also available as DCS-TR-335.

[SRW98] Mooly Sagiv, Thomas Reps, and Reinhard Wilhelm. Solving shape-analysis problems in languages with destructive updating. *ACM Transactions on Programming Languages and Systems*, 20(1):1–50, January 1998.

[SS78] J. T. Schwartz and M. Sharir. Tarjan's fast interval finding algorithm. SETL Newsletter No. 204, March 3, 1978, 1978.

[Ste96] Bjarne Steensgaard. Points-to analysis in almost linear time. In *Conference Record of the Twenty-third Annual ACM SIGACT/SIGPLAN Symposium on Principles of Programming Languages*, pages 32–41, 1996.

[TAFM97] P. Tonella, G. Antoniol, R. Fiutern, and E. Merlo. Flow-insensitive c^{++} pointers and polymorphism analysis and its application to slicing. In *Proceedings of the 19th International Conference on Software Engineering (ICSE97)*, pages 433–443, 1997.

[TCFR96] F. Tip, J-D Choi, J. Field, and G. Ramalingam. Slicing class hierarchies in c^{++}. In *Proceedings of OOPSLA'96: Conference on Object-oriented Programming Systems, Languages and Applications*, pages 179–197, October 1996.

[Tip96] Frank Tip. A survey of program slicing technques. *Journal of Programming Languages*, 3(3):121–189, 1996.

[Ven91] G. A. Venkatesh. The semantic approach to program slicing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 107–119, June 1991.

[Wei80] W. E. Weihl. *Interprocedural Data Flow Analysis in the Presence of Pointers, Procedure Variables and Label Variables*. Master's thesis, M.I.T., June 1980.

[Wei84] Mark Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[Wey94] E. Weyuker. More experience with data flow testing. *IEEE Transactions on Software Engineering*, 19(9):912–919, September 1994.

[WL95] Robert Wilson and Monica 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. also available as SIGPLAN Notices, 30(6).

[YRL98] Jyh-shiarn Yur, Barbara G. Ryder, and William. A. Landi. Incremental algorithms and empirical comparison for flow- and context-sensitive pointer alias analysis. Department of Computer Science Technical Report DCS-TR-348, Department of Computer Science, Rutgers University, October 1998.

[YRL99] J. Yur, B. G. Ryder, and W. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *Proceedings of the Twenty-First International Conference on Software Engineering*, pages 442–451, May 1999.

[YRLS97] J. Yur, B. G. Ryder, W. Landi, and P. Stocks. Incremental analysis of side effects for C software systems. In *Proceedings of the Nineteenth International Conference on Software Engineering*, pages 422–432, May 1997.

[Zad84] F. K. Zadeck. Incremental data flow analysis in a structured program editor. In *Proceedings of the ACM SIGPLAN Symposium on Compiler Construction*, pages 132–143, June 1984. SIGPLAN Notices, Vol 19, No 6.

[ZRL96] Sean Zhang, Barbara G. Ryder, and William Landi. Program decomposition for pointer aliasing: A step towards practical analyses. In *Proceedings of the 4th Annual ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 81–92, October 1996.

[ZRL98] Sean Zhang, Barbara G. Ryder, and William A. Landi. Experiments with combined analysis for pointer aliasing. In *Proceedings of ACM SIGPLAN Workshop on Program Analysis and Software Tools for Engineering*, pages 11–18, June 1998.

Vita

Jyh-shiarn Yur

1987 B.S. in Information Science, Tunghai University, Taichung, Taiwan.

1987-89 M.S. in Information Engineering, Tatung Institute of Technology, Taipei, Taiwan.

1989-91 Research Engineer, Matsushita Institute of Technology, Taipei (MITT), Taiwan.

1991-93 Teaching Assistant, Department of Computer Science, Rutgers, The State University of New Jersey.

1994 M.S. in Computer Science, Rutgers, The State University of New Jersey.

1993-99 Research Assistant, Department of Computer Science, Rutgers, The State University of New Jersey.

1997 Jyh-shiarn Yur, Barbara G. Ryder, and William. A. Landi, and and Phil Stocks. Incremental analysis of side effects for C software systems. In *Proceedings of the Nineteenth International Conference on Software Engineering*, May 1997.

1999 Jyh-shiarn Yur, Barbara G. Ryder, and William. A. Landi. An incremental flow- and context-sensitive pointer aliasing analysis. In *Proceedings of the Twenty-first International Conference on Software Engineering*, May 1999

1999 Ph.D. in Computer Science, Rutgers, The State University of New Jersey.