CONTEXT-DEPENDENT FLOW-SENSITIVE INTERPROCEDURAL DATAFLOW ANALYSIS AND ITS APPLICATION TO SLICING AND PARALLELIZATION

By

KURT JOHMANN

A DISSERTATION PRESENTED TO THE GRADUATE SCHOOL
OF THE UNIVERSITY OF FLORIDA IN PARTIAL FULFILLMENT
OF THE REQUIREMENTS FOR THE DEGREE OF
DOCTOR OF PHILOSOPHY

UNIVERSITY OF FLORIDA

1992

ACKNOWLEDGEMENTS

I would like to express my appreciation and gratitude to my chairman and advisor, Dr. Stephen S. Yau, for his careful guidance and generous support during this study. I also would like to express my appreciation and gratitude to my previous advisor, Dr. Sying-Syang Liu. Without their supervision and counsel, this work would not have been possible. To Dr. Paul Fishwick, Dr. Richard Newman-Wolfe, and Dr. Mark Yang, members of the supervisory committee, go my thankfulness for their service.

Finally, I want to thank the Software Engineering Research Center (SERC) for providing financial support during this study.

TABLE OF CONTENTS

Abstract of Dissertation Presented to the Graduate School
of the University of Florida in Partial Fulfillment of the
Requirements for the Degree of Doctor of Philosophy

CONTEXT-DEPENDENT FLOW-SENSITIVE INTERPROCEDURAL DATAFLOW ANALYSIS AND ITS APPLICATION TO SLICING AND PARALLELIZATION

By

Kurt Johmann

May 1992

Chairman: Dr. Stephen S. Yau
Major Department: Computer and Information Sciences

Interprocedural dataflow analysis is important in compiler optimization, automatic vectorization and parallelization, program revalidation, dataflow anomaly detection, and software tools that make a program more understandable by showing data dependencies. These applications require the solution of dataflow problems such as reaching definitions, live variables, available expressions, and definition-use chains. When solving these problems interprocedurally, the context of each call must be taken into account.

In this dissertation we present a method to solve this kind of dataflow problem precisely. The method consists of special dataflow equations that are solved for a program flowgraph. Regarding calling context, separate sets, called entry and body sets, are maintained at each node in the flowgraph. The entry set contains calling-context effects that enter a procedure. The body set contains effects that result from statements in the procedure. By isolating calling-context effects in the entry set, a call's nonkilled calling context is preserved by means of a simple intersection operation done at the return node for the call.

Slicing determines program pieces that can affect a value. Logical ripple effect determines program pieces that can be affected by a value. Both slicing and logical ripple effect are useful for software maintenance. The problems of slicing and logical ripple effect are inverses of each other, and a solution of either problem can be inverted to solve the other. Precise interprocedural logical ripple effect analysis is complicated by the fact that an element may be in the ripple effect by virtue of one or more specific execution paths. In this dissertation we present an algorithm that builds a precise logical ripple effect or slice piece by piece, taking into account the possible execution paths. The algorithm makes use of our interprocedural dataflow analysis method, and this method is also used in an algorithm given in this dissertation for identifying loops that can be parallelized.

CHAPTER 1
INTRODUCTION

1.1 Interprocedural Dataflow Analysis

Dataflow analysis refers to a class of problems that ask about the relationships that exist along a program's possible execution paths, between such program elements as variables, constants, and expressions [2, 10]. When dataflow analysis is done for a program by treating its individual procedures as being independent of each other, regardless of the calls made, this is known as intraprocedural analysis. For intraprocedural analysis, assumptions must be made about the effects of calls. By contrast, interprocedural analysis replaces assumptions with specific information about the effects of each call. This information can be gathered by either flow-sensitive [3, 6, 9, 17, 19, 21] or flow-insensitive [4, 7, 18] analysis. When answering a dataflow question, a flow-sensitive analysis will take into account the flow paths within procedures, whereas a flow-insensitive analysis ignores these flow paths. The flow paths are the possible execution paths. Flow-sensitive analysis typically provides more precise information, but at greater cost.

Flow-sensitive interprocedural dataflow analysis has two major problems that make it significantly harder than intraprocedural analysis. First, in intraprocedural analysis, it is assumed that any path in the flowgraph is a possible execution path. By contrast, for interprocedural analysis, it is useful to assume that the possible execution paths conform to the rule that once a procedure is entered by a call, the flow returns to that call upon return. Thus, the set of possible execution paths will typically be a proper subset of the paths in the program flowgraph. This problem will be referred to as the calling-context problem. Second, call-by-reference formal parameters typically cause alias relationships between actual and formal parameters that are valid only for certain calls and apply only to those passes through the called procedure that originate from those calls that establish the specific alias relationship.

There are many applications for a flow-sensitive interprocedural dataflow analysis method that solves the two major problems, assuming that the costs of the method are not too high. Some of the well-known dataflow problems that can be precisely solved by such a method are reaching definitions, live variables, the related problems of definition-use and use-definition chains, and available expressions. Applications that require the solution of one or more of these dataflow problems include compiler optimization, automatic vectorization and parallelization of program code, program revalidation, dataflow anomaly detection, and software tools that show data dependencies.

In this dissertation we present a new method for flow-sensitive interprocedural dataflow analysis that solves the two major problems, and does so at a comparatively low cost [13]. The method consists of special dataflow equations that are solved for a program flowgraph. In deference to calling context, separate sets, called entry and body sets, are maintained at each node in the flowgraph. The entry set contains calling-context effects that enter a procedure. The body set contains effects that result from statements in the procedure. By isolating calling-context effects in the entry set, a call's nonkilled calling context is preserved by means of a simple intersection operation done at the return node for the call. The main advantage of our method is its low complexity, and the fact that the presence of recursion does not affect the preciseness of the result.

The language model assumed for Chapter 2 allows global variables, but the visibility of each formal parameter is limited to the single procedure that declares it. Thus, with the exception of a call and its indirect reference, each formal parameter can only be referenced inside a single procedure. Examples of programming languages that fit this model are C and FORTRAN. This restriction on the visibility of formal parameters is imposed for the sake of the discussions of element recoding in Sections 2.2.2 and 2.3, of implicit definitions in Section 2.2.3, and of worst-case complexity in Section 2.5. Our method can also be used for the alternative language model that allows each formal parameter to have visibility in more than a single procedure, but this is considered only briefly at the end of Section 2.5.

1.2 Slicing and Logical Ripple Effect

Given an actual or hypothetical variable v at program point p, determine all program pieces that can possibly be affected by the value of v at p. This is the logical ripple effect problem. Given v and p, determine all program pieces that can possibly affect the value of v at p. This is the slicing problem. For these two problems, each problem is the inverse of the other, and a solution for one of these problems, once inverted, would be a solution for the other problem.

Logical ripple effect is useful for helping a programmer to understand how a program change, either actual or hypothetical, will impact that program. Making program changes as part of routine maintenance often introduces new errors into the changed program. Such errors typically result because the programmer overlooked some part of the logical ripple effect for that change. By showing a programmer what the logical ripple effect actually is for a program change, mistakes can be avoided.

Slicing is primarily useful for program fault localization [23]. If a variable v at point p is known to have a wrong value, then a slice on v at p will narrow the search for the cause of the error to that part of the program that can truly affect v at p. Thus, the fault is localized. The more precise the slice, the more localized the cause of the error, saving programmer time.

In this dissertation we are concerned only with static logical ripple effect and slicing [11, 12, 16, 24] where the ripple effect or slice is determined from dataflow analysis of the program text. The alternative approach is dynamic logical ripple effect and slicing [1, 14] where the ripple effect or slice is determined by actually executing the program. Whenever we speak of execution paths in Chapter 3, we always mean possible execution paths as determined by dataflow analysis.

Precise interprocedural logical ripple effect analysis is complicated by the fact that a definition may be added to the ripple effect because of one or more specific execution paths. To determine in turn the ripple effect of that added definition, that definition should be constrained to those execution paths that are the possible continuations of the execution paths along which that definition was itself affected and thereby added to the ripple effect. We refer to this as the execution-path problem.

In particular, it is those call instances made in an execution path P that have not been returned to in P that cause the difficulty. This is because of the rule that a called procedure returns to its most recent caller. This means that any continuation of the execution path P must first return to those unreturned calls in P before returns can possibly be made to call instances that precede P. An example will illustrate the problem.

```
procedure main        procedure B          procedure A
begin                 begin                begin
1:  f ← 7             6:  y ← f + 5        7:  call B
2:  call A            end                  8:  x ← y
3:  z ← x                                  end
4:  f ← 1
5:  call B
end
```

For the example, assume that all variables are global, and that the problem is to determine the logical ripple effect for the definition of variable f at line 4. The call to procedure B at line 5 allows the definition of f at line 4 to affect the definition of y at line 6, and the return of procedure B would be to the call at line 5 by which the definition of y at line 6 was affected. The end result is that the ripple effect should include only line 6. However, assume that the execution-path problem is ignored and all returns are possible when the ripple effect is computed. For the same problem, the call at line 5 allows the definition of f at line 4 to affect the definition of y at line 6. Then the definition of y at line 6 affects the definition of x at line 8 by procedure B returning to the call at line 7 in addition to the call at line 5. Then the definition of x at line 8 affects the definition of z at line 3 by procedure A returning to the call at line 2. The end result is a ripple effect that includes lines 3, 6, and 8, but only line 6 should be in the ripple effect.

Although there are a number of papers on logical ripple effect and slicing [11, 12, 16, 24], there appears to be only one [11] that addresses the problems of precise interprocedural logical ripple effect and slicing, and presents a method for it. Weiser [24] was the first to propose an interprocedural slicing method that ignores the execution-path problem and thereby suffers from the resulting loss of precision. Horwitz et al. [11] address the problem of precise interprocedural slicing, and present a method to construct a system dependence graph from which slices can be extracted.

In this dissertation we present an algorithm that builds the logical ripple effect piece by piece, and takes into account the restrictions on execution-path continuation that are imposed by the preceding execution paths up to the point by which the given program piece is affected and thereby included in the ripple effect. In general, the algorithm computes a precise logical ripple effect, but some overestimation is possible, meaning that the computed logical ripple effect may be larger than it actually is. An inverse form of the algorithm is presented for the slicing problem. The languages that our algorithm will work for include many of the common procedural languages such as C, Pascal, Ada, and Fortran.

1.3 Parallelization

Automatic conversion of a sequential program into a parallel program is often referred to as parallelization. Parallelization problems are typically concerned with the conversion of sequential loops into parallel code. In this dissertation, the specific problem considered is the identification of loops in a program that can be parallelized, including those loops that contain calls. A flow-sensitive interprocedural dataflow analysis method has specific applicability to the problem of parallelizing loops that contain calls, because such a method can supply the precise data-dependency information that would be necessary for the parallelization analysis.

The parallelization of a loop would mean that each iteration of the loop can be executed independently of the other iterations of the loop. In theory, this would mean that each single iteration, or each arbitrary block of iterations, can be assigned to a separate processor in a parallel machine. The specific architecture of a particular parallel machine, as well as the programming language to be parallelized, as well as the various loop transformations that are possible to convert sequential loop code into functionally equivalent sequential code that is more parallelizable, will influence the determination in any parallelization tool as to what loops can actually be parallelized, and how they would be parallelized. However, none of the architecture, language, and loop-transformation issues will be considered here. Instead, the problem will be considered solely from the standpoint of data dependence.

After a brief review of the basics regarding data dependence and parallelization, an algorithm is given that identifies loops in a program that can be parallelized, and this algorithm uses our interprocedural dataflow analysis method as an integral part. The potential value of parallelization is clear. On the one hand, parallel machines are becoming more common, and on the other hand, a great number of sequential programs already exist, some of which can benefit from the greater processing power that parallelization would offer.

1.4 Literature Review

Different methods have been offered for solving various flow-sensitive interprocedural dataflow analysis problems. Sharir and Pnueli [21] present a method they name call-strings. The essential idea of their method is to accumulate for each element a history of the calls traversed by that element as it flows through the program flowgraph. The call history associated with an element is used whenever that element is at a return point. The element can only cross back to those calls in its call history. Thus, the call-strings approach provides a solution to the calling-context problem. However, the disadvantage of this approach is the time and space needed to maintain a call history for each element at each flowgraph node.

Let l be the program size. We assume that the number of elements will be a linear function of l. The worst-case number of total set operations required by the call-strings approach would be greater by a factor of l when compared to our method. This is because for each union or intersection of two sets of elements, if the same element is in both sets, then a union operation must also be done for the two associated call histories so as to get the new call history to be associated with that element at the node for which the set operation is being done. A further disadvantage of the call-strings approach is the need to include the associated call histories when set stability is tested to determine termination for the iterative algorithm used to solve the dataflow equations.

Myers [17] offers a solution to the calling-context problem that is essentially the same as call-strings. Allen [3] presents a different method for interprocedural dataflow analysis. The method analyzes each procedure completely, in reverse invocation order. The first procedures to be analyzed would be those that make no calls, then the procedures that only call these procedures would be analyzed, and so on. Once a procedure is analyzed, its effects can be incorporated into those procedures that call it, when they in turn are analyzed. The obvious drawback of this method is that it cannot be used to analyze recursive calls.

Rosen [19] presents a complex method for interprocedural dataflow analysis that is limited to solving the problems of variable modification, preservation, and use. These dataflow problems do not require a solution of the calling-context problem.

Callahan [6] has proposed the program summary graph to solve the interprocedural dataflow problems of kill and use, where kill determines all definite kills that result from a procedure call, and use determines all variables that may be used as a result of a procedure call before being redefined.

As part of the determination of edges in the program summary graph, intraprocedural reaching-definitions analysis must be done for each procedure. Simplifying Callahan's space complexity analysis, we get $O(v_{ga}l)$ as the worst-case size of the program summary graph, where v_{ga} is the number of global variables in the program plus the average number of actual parameters per call, and l is the program size. One limitation of Callahan's method is that it does not correctly handle multiple aliases that result when the same variable is used multiple times as an actual parameter in the same call and the corresponding formal parameters are call-by-reference. By contrast, our method, using element recoding where all the aliases are encoded in a single element, will correctly handle the multiple aliases problem.

Callahan's method offers no solution to the calling-context problem, and could not be used to determine, for example, interprocedural reaching definitions. However, Harrold and Soffa [9] have extended his method so that interprocedural reaching definitions can be determined. They use an interprocedural flowgraph, denoted IFG, that is very similar to the program summary graph. The IFG has inter-reaching edges that are determined by solving Callahan's kill problem. They recommend using his method, so their method inherits Callahan's space and time complexity, as well as its limitation with regard to multiple aliases.

Before the IFG can be used, it must be decorated with the results of intraprocedural analysis done twice for each procedure to determine both reaching definitions and upwardly exposed uses. Then an algorithm is used to propagate the upwardly exposed uses throughout the IFG. This algorithm has worst-case time complexity of $O(n^2)$ where n is the number of nodes in the IFG. Their graph will have the same number of nodes as for Callahan's graph, meaning worst-case graph size will be $O(v_{ga}l)$. Substituting $v_{ga}l$ for n, we get a worst-case time complexity of $O(v_{ga}^2 l^2)$. As the size of our flowgraph is proportional to the size of the program, the worst-case time complexity for solving our equations is only $O(l^2)$.

Weiser [24] was the first to propose an interprocedural slicing method that ignores the execution-path problem and thereby suffers from the resulting loss of precision. Horwitz et al. [11] have presented a method to compute the more precise slice explained in the Introduction. However, they use a more restricted definition of a slice. Their slice is all statements and predicates that may affect a variable v at program point p, such that v is defined or used at point p. Their method consists of constructing a specialized graph called a system dependence graph. Nodes in this graph represent program pieces such as statements, and the edges in the graph represent control or data dependencies. Edges representing transitive data dependencies that are due to procedure calls are computed by first modeling each procedure and its calls with an attribute grammar called a linkage grammar, and then solving the grammar so as to determine the transitive data dependencies represented by it. Once the system dependence graph is complete, any slice based on an actual definition or use occurring at any point p in the program can be extracted from the graph. A major weakness of their method is that it does not allow a hypothetical use to be the starting point of the slice.

The complexity of constructing the system dependence graph is given as $O(G \cdot X^2 \cdot D^2)$ where G is the total number of procedures and calls in the program, X is the total number of global variables in the program plus a term that can be considered a constant, and D is a linear function of X. Once the system dependence graph is complete, any particular slice that is wanted can be extracted from the graph at complexity $O(n)$ where n is the size of the graph. The size of the graph is roughly quadratic with program size, being bounded by $O(P \cdot (V + E) + T \cdot X)$ where P is the number of procedures, V is the largest number of predicates and definitions in a single procedure, E is the largest number of edges in a procedure dependence graph, T is the number of calls in the program, and X is the number of global variables. In their paper, much is made of the fact that once the graph is complete, any slice on an actual definition or use can be extracted from the graph at $O(n)$ cost where n is the size of the graph. However, the number of actual definition and use occurrences in a program is proportional to the program size L. Therefore, any method that can compute a slice at cost $O(Z)$ for some Z, can generate all the slices contained in their graph at cost $O(Z \cdot L)$, spool the slices to disk, and recover them at cost $O(1)$.

Although there are many papers on slicing, it seems that only Horwitz et al. [11] discuss clearly the problem of the more precise interprocedural slice, and present a method to compute it, as well as providing complexity analysis. Our research on slicing is only concerned with computing the more precise slice, so Horwitz et al. is the principal reference.

Zima and Chapman [25] is the principal reference used to study the issues and methods of parallelization. Their book distills the work found in scores of papers and dissertations, and is an excellent survey of parallelization. Interprocedural parallelization is specifically considered by Burke and Cytron [5], and by Triolet et al. [22].

1.5 Outline in Brief

This introductory chapter ends with a brief synopsis of the remaining chapters. Chapter 2 presents in detail our interprocedural dataflow analysis method. The chapter ends with a brief description of the prototypes that were built to demonstrate the method, along with some of the experimental results obtained from these prototypes. Chapter 3 begins with a representation scheme for continuation paths for the interprocedural logical ripple effect problem and then presents our interprocedural logical ripple effect algorithm. A prototype that was built to demonstrate this algorithm is briefly described and experimental results are presented. An inversion of the logical ripple effect algorithm is then presented as a solution to the interprocedural slicing problem. Chapter 4 begins with an explanation of loop-carried data dependence and its relevance to parallelization, and concludes with an algorithm that identifies loops that can be parallelized, including loops that contain calls. Chapter 5 summarizes the major results of the dissertation, and suggests directions for future research.

CHAPTER 2
THE INTERPROCEDURAL DATAFLOW ANALYSIS METHOD

2.1 Constructing the Flowgraph

This section discusses the flowgraph and its relationship to dataflow equations. After the discussion, rules are given for constructing the specific flowgraph required by our interprocedural analysis method. Note that the required flowgraph is conventional and the rules to be given relate only to the representation of calls and procedures in the flowgraph.

A flowgraph is a directed graph that represents the possible flow paths of a program. The nodes of a flowgraph correspond to basic blocks in the program. A basic block is a sequence of program code that is always executed together in the same order. The directed edges of a flowgraph represent possible transfers of control. Figures 2.1 and 2.3 each represent a flowgraph.

Dataflow problems are often formulated as a set of equations that relate the four sets, IN, OUT, GEN, and $KILL$, that are associated with each node in the flowgraph. For any node and its block, the GEN set represents the elements generated by that block. The $KILL$ set represents those elements that cannot flow through the block, because they would be killed by the block. The IN set represents the valid elements at the start of the block, and the OUT set represents the valid elements at the end of the block.

Dataflow problems are typically either forward-flow or backward-flow. For forward-flow, the IN set of a node is computed as the confluence of the OUT sets of the predecessor nodes, and the OUT set is a function of the node's IN, GEN, and $KILL$ sets. For backward-flow, the OUT set of a node is computed as the confluence of the IN sets of the successor nodes, and the IN set is a function of the node's OUT, GEN, and $KILL$ sets. The predecessors of any node n are those nodes that have an out-edge directed to node n. The successors of node n are those nodes that have an in-edge directed from node n. The confluence operator will almost invariably be either set union or set intersection, depending on the problem. Thus, a dataflow problem may be classified as being either forward-flow-or, forward-flow-and, backward-flow-or, or backward-flow-and, where "or" refers to set union and "and" refers to set intersection.

Once the dataflow equations have been defined for a particular problem, and the rules established for creating the GEN and $KILL$ sets, the equations can then be solved for a specific program or procedure and its representative flowgraph. To solve the equations, the iterative algorithm can be used. The iterative algorithm has the advantage that it will work for any flowgraph.

The iterative algorithm repeatedly computes the IN and OUT sets for all nodes until all sets have stabilized and ceased to change. Recomputation of a node is necessary whenever an outside set that it depends on changes. For forward-flow problems, a node must be recomputed if the OUT set of a predecessor node changes. For backward-flow problems, a node must be recomputed if the IN set of a successor node changes. Typically, an evaluation strategy will determine the actual order in which nodes are recomputed.

The flowgraph required by our interprocedural analysis method is conventional, with special nodes and edges as follows. For each procedure in the program, assign an *entry node* and an *exit node*. These nodes have no associated blocks of program code.

The entry node has a single out-edge and as many in-edges as there are calls to that procedure in the program. The exit node has as many in-edges as there are nodes for that procedure whose blocks terminate with a return action. The exit node has as many out-edges as there are calls to that procedure in the program. For every in-edge of the entry node, there is a corresponding out-edge of the exit node.

For the purpose of constructing the flowgraph, calls must be classified as either known or unknown. A *known call* is where the flowgraph for the called procedure will be a part of the total flowgraph being constructed. An *unknown call* is where the flowgraph of the called procedure will not be a part of the total flowgraph being constructed. Unknown calls are common and will occur for two reasons. First, the called procedure may be a compiler-library procedure for which source code is not available. Second, the called procedure may be a separately compiled user procedure for which the source code is not available.

For any unknown call made within the program, if summary information of its interprocedural effects is not available, then conservative assumptions about its effects will have to be made. The actual summary information needed, and the assumptions made in its absence, will depend on the particular dataflow problem. The summary information, if present, would be used when constructing the GEN and $KILL$ sets for any node whose block contains an unknown call.

For any known call made within the program, there will be two nodes in the flowgraph for that call. One node is the *call node*. The call node represents a basic block that ends with the known call. The other node is the *return node*. The return node has an empty associated block.

The call node will have two out-edges. One edge will be directed to the entry node of the called procedure. The other out-edge will be directed to the return node for that call. The return node will have two in-edges. One edge is the directed edge from the call node. The other in-edge is directed from the called procedure's exit node.

In all, each known call results in two nodes and three distinct edges. One edge connects the call node to its return node. A second edge connects the call node to the called procedure's entry node. A third edge connects the called procedure's exit node to the return node.

In constructing the flowgraph, a special problem arises if the programming language allows procedure-valued variables, such as the function pointers of C that when dereferenced result in a call of the function that is pointed at. The problem is to identify what are the possible procedure values when the procedure-valued variable invokes a call. Assuming this information is available from a separate analysis, the flowgraph can be constructed accordingly. For example, if the procedure-valued variable can have three different values when the call in question is invoked and each value is a procedure whose flowgraph will be part of the total flowgraph, then three known calls would be constructed in parallel with a common predecessor node for the three call nodes and a common successor node for the three return nodes.

A procedure-valued variable is in essence a pointer. Note that the problem of determining what a pointer is or may be pointing at when that pointer is dereferenced, can itself be formulated as a dataflow problem, and in particular as a forward-flow-or dataflow problem. If necessary, an initial version of the flowgraph could be constructed that treats all calls invoked by procedure-valued variables as unknown calls, followed by a solving of the dataflow problem for determining possible pointer values whenever a pointer is dereferenced, followed by amendments to the flowgraph using the pointer-value information.

Dataflow analysis makes a simplifying, conservative assumption about the correspondence between paths in the flowgraph and possible execution paths in the program. Let a path be a sequence of flowgraph nodes such that in the sequence node n follows node m only if n is a successor of m in the flowgraph. For intraprocedural analysis, the assumption made is that any path in the flowgraph is a possible execution path. That this assumption may not be true for a particular program should be obvious. However, the problem of determining the possible execution paths for an arbitrary program is known to be undecidable. The simplifying assumption that we use for interprocedural analysis is the same as that used for intraprocedural analysis, but with the added proviso that for any path that is a possible execution path, any subsequence of return nodes must inversely match, if present, the immediately preceding subsequence of call nodes. A return node matches a call node if and only if the return node is the call node's successor in the flowgraph.

2.2 Interprocedural Forward-Flow-Or Analysis

This section begins with our basic approach to solving the calling-context problem. The dataflow equations for forward-flow-or analysis are then given and their correctness is shown. As a part of our interprocedural analysis method, the technique of element recoding is presented as a way to deal with the aliases that result from call-by-reference formal parameters. For some dataflow problems, implicit definitions due to calls require explicit treatment, and this is discussed last.

If certain problems, such as reaching definitions, are to be solved for a program by flow-sensitive interprocedural analysis, then the calling context of each procedure call must be preserved. In general, preserving calling context means that the dataflow effects of an individual call should include those effects that survive the call and were introduced into the called procedure by the call itself, but not those effects introduced into the called procedure by all the other calls to it that may exist elsewhere in the program. We refer to the need to preserve calling context as the calling-context problem.

Our solution to the calling-context problem—and the essential difference between our dataflow equations and conventional dataflow equations—is to divide every IN set and every OUT set into two sets called an *entry set* and a *body set*. The reason for having two sets is that the calling-context effects that enter a procedure from the different calls can be collected and isolated in the separate entry set. This entry set can then have effects in it killed by statements in the body of the procedure, but no additions are made to this entry set by body statements. Instead, any additions of effects due to body statements are made to the separate body set. This body set will also have effects killed in the normal manner, as for the entry set. Because the body set is kept free of calling-context effects, it is empty at the entry node. By contrast, the entry set is at its largest at the entry node and will either stay the same size as it progresses through the procedure's body nodes, or become smaller because of kills. By intersecting the calling context at a call node with the entry set at the exit node of the called procedure, the result is that subset of the calling context that has reached the exit node and therefore will reach the return node for that call. By "reach" we mean that there exists a path in the flowgraph along which the element is not killed or blocked.

2.2.1 The Dataflow Equations

The dataflow equations that define the entry and body sets at every node are now given. The equations are divided into three groups. The first group computes the sets for entry nodes. The second group computes the sets for return nodes. The third group computes the sets for all other nodes. In the equations, B denotes a body set and E denotes an entry set. Two conditions, C_1 and C_2, appear in the equations. C_1 means that x will cross the interprocedural boundary from call node p into the called procedure. C_2 means that x can cross the interprocedural boundary from exit node q into return node n. $\overline{C_i}$ means not C_i. For each node n, $pred(n)$ means the set of predecessors of n. The $RECODE$ set used in Group I is explained in Section 2.2.2. The GEN set used in Group I, and the GEN and $KILL$ sets used in Group II, are explained in Section 2.2.3.

For any node n.

$$IN[n] = E_{in}[n] \cup B_{in}[n]$$

$$OUT[n] = E_{out}[n] \cup B_{out}[n]$$

Group I: n is an entry node.

$$B_{in}[n] = \emptyset$$

$$E_{in}[n] = \bigcup_{p \in pred(n)} \{x \mid x \in OUT[p] \wedge C_1\}$$

$$B_{out}[n] = GEN[n]$$

$$E_{out}[n] = E_{in}[n] \cup RECODE[n]$$

Group II: n is a return node, p is the associated call node and q is the exit node of the called procedure.

$$B_{in}[n] = \{x \mid (x \in B_{out}[p] \wedge (\overline{C_1} \vee (C_1 \wedge C_2 \wedge x \in E_{out}[q]))) \vee (x \in B_{out}[q] \wedge C_2)\}$$

$$E_{in}[n] = \{x \in E_{out}[p] \mid \overline{C_1} \vee (C_1 \wedge C_2 \wedge x \in E_{out}[q])\}$$

$$B_{out}[n] = (B_{in}[n] - KILL[n]) \cup GEN[n]$$

$$E_{out}[n] = E_{in}[n] - KILL[n]$$

Group III: n is not an entry or return node.

$$B_{in}[n] = \bigcup_{p \in pred(n)} B_{out}[p]$$

$$E_{in}[n] = \bigcup_{p \in pred(n)} E_{out}[p]$$

$$B_{out}[n] = (B_{in}[n] - KILL[n]) \cup GEN[n]$$

$$E_{out}[n] = E_{in}[n] - KILL[n]$$

The equations assume that the *GEN* and *KILL* sets for each call node will include only those effects for that call that occur prior to the entry of the called procedure. This requirement is necessary because the *OUT* set of the call node is used by the entry-node equation that constructs the entry set of the called procedure.

Referring to conditions C_1 and C_2, the rules for deciding whether an effect crosses a particular interprocedural boundary will depend on two primary factors, namely the dataflow problem and the programming language. For example, for the reaching-definitions problem and a language such as FORTRAN, any definition of a global variable, and any definition of a variable that is used as an actual parameter whose corresponding formal parameter is call-by-reference, will cross. As a rule, an effect that crosses into a procedure because it might be killed, will also cross back to the return node if it reaches the exit node of the called procedure.

Table 2.1 shows the result of solving the equations for the flowgraph of Figure 2.1. By "solving" we mean that, in effect, the iterative algorithm has been used and all the sets are stable. The dataflow problem is reaching definitions, and variable w is local while variables x, y, and z are global. Reaching definitions is the problem of finding all definitions of a variable that reach a particular use of that variable, for all variables and uses in the program. In Figure 2.1, nodes 1 and 8 are entry nodes, nodes 7 and 10 are exit nodes, nodes 3 and 5 are call nodes, and nodes 4 and 6 are return nodes. Alongside each node is its basic block. Each defined variable is superscripted with an identifier that is the set element used in Table 2.1 to represent that definition.

The correctness of the equations can be seen from the following observations. For a procedure, the entry-node entry set is constructed as the union of all calling-context effects that can enter the procedure from its calls. Within the procedure body, effects in the entry set can be killed, but not added to. For effects in the entry

```
procedure main
begin
   w = 5
   x = 10
   if(w > x)
         z = 10
         call f()
   else
         y = 5
         call f()
 end
```

```
procedure f()
begin
   x = 10
end
```

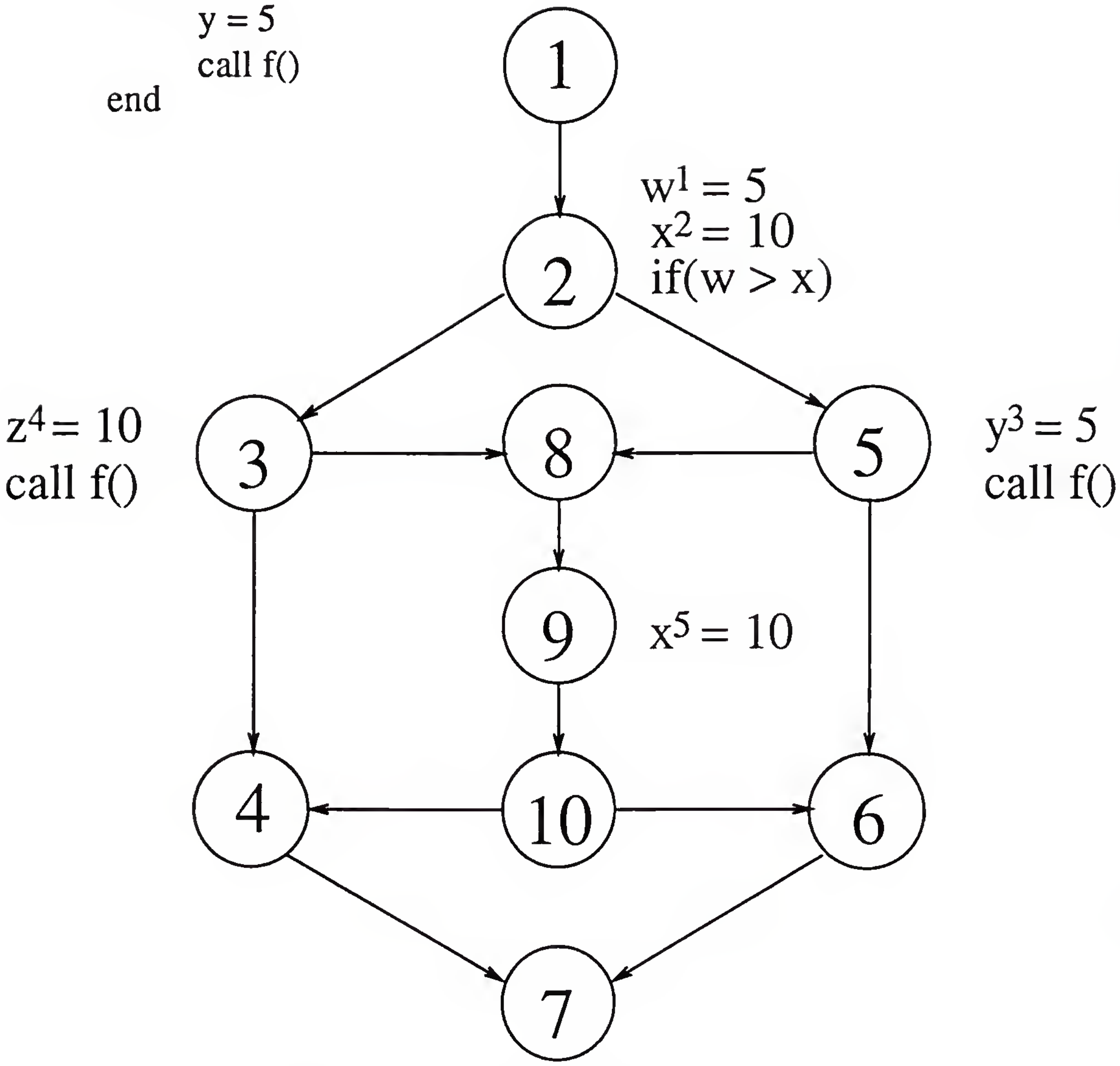

Figure 2.1. A reaching-definitions example.

Table 2.1. Solution of forward-flow-or equations for Figure 2.1.

Node	E_{in}	E_{out}	B_{in}	B_{out}
1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\emptyset$	$\emptyset$	$\emptyset$	{1, 2}
3	$\emptyset$	$\emptyset$	{1, 2}	{1, 2, 4}
4	$\emptyset$	$\emptyset$	{1, 4, 5}	{1, 4, 5}
5	$\emptyset$	$\emptyset$	{1, 2}	{1, 2, 3}
6	$\emptyset$	$\emptyset$	{1, 3, 5}	{1, 3, 5}
7	$\emptyset$	$\emptyset$	{1, 3, 4, 5}	{1, 3, 4, 5}
8	{2, 3, 4}	{2, 3, 4}	$\emptyset$	$\emptyset$
9	{2, 3, 4}	{3, 4}	$\emptyset$	{5}
10	{3, 4}	{3, 4}	{5}	{5}

set that reach a call at a call node, those effects that survive the call are recovered in the entry set constructed by the $E_{in}[n]$ equation for the successor return node n. To see that this is true, observe the following. If an entry-set effect that reaches the call cannot enter the called procedure, then it cannot be killed within the called procedure, so the effect should be added to the return-node entry set without further conditions, and this is done by the selection criterion $(x \in E_{out}[p] \wedge \overline{C_1})$ in the $E_{in}[n]$ equation for the return node. If, on the other hand, an entry-set effect reaches the call and does enter the called procedure, and therefore may be killed by it, then this effect should be added to the return-node entry set only if it reached the entry set of the called procedure's exit node and the effect can cross back into the caller. This is done by the selection criterion $(x \in E_{out}[p] \wedge C_1 \wedge C_2 \wedge x \in E_{out}[q])$ in the $E_{in}[n]$ equation for the return node.

From the equations for the entry set, we see that for any procedure z, the entry set at z's exit node will, as the equations are solved, eventually contain all calling-context effects that entered z and reached its exit node. This characteristic of the exit-node entry set is the requirement placed upon it when it is used in the

$E_{in}[n]$ equation for the return node, so this requirement is satisfied and the entry-set equations are correct.

For any procedure, the B_{in} set is always empty at the entry node, so the B set is free of calling-context effects. Within the procedure body, GEN and $KILL$ sets are used to update the body set as it propagates along the various nodes. For effects in the body set that reach a call at a call node, those effects that survive the call are recovered in the body set constructed by the $B_{in}[n]$ equation for the successor return node n. If a body-set effect that reaches the call cannot enter the called procedure, then it cannot be killed within the called procedure, so it should be added to the return-node body set without further conditions, and this is done by the selection criterion $(x \in B_{out}[p] \wedge \overline{C_1})$ in the $B_{in}[n]$ equation for the return node. If, on the other hand, a body-set effect reaches the call and will enter the called procedure, and therefore may be killed by it, then this effect should be added to the return-node body set only if it reached the entry set of the called procedure's exit node and the effect can cross back into the caller. This is done by the selection criterion $(x \in B_{out}[p] \wedge C_1 \wedge C_2 \wedge x \in E_{out}[q])$ in the $B_{in}[n]$ equation for the return node. In addition, all crossable effects that result from the call, and that are independent of calling context, should also be added to the return-node body set, and this is done by the selection criterion $(x \in B_{out}[q] \wedge C_2)$ in the $B_{in}[n]$ equation for the return node.

From the equations for the body set, we see that for any procedure z, the body set at z's exit node is free of calling-context effects and will, as the equations are solved, eventually contain all body effects that reached the exit node, including those body effects resulting from calls made within z. This characteristic of the exit-node body set is the requirement placed upon it when it is used in the $B_{in}[n]$ equation for the return node, so this requirement is satisfied. The other requirement of this return-node equation is that the exit-node entry set contains all calling-context effects for the procedure that reach the exit node. This requirement has already been shown to be satisfied, so we conclude that the body-set equations are correct.

2.2.2 Element Recoding for Aliases

The *RECODE* set for the entry node has its elements added to the E_{in} set for that node. The idea of the *RECODE* set is that certain elements in the *OUT* set of a predecessor call node, irrespective of their ability to cross the interprocedural boundary when parameters are ignored, should nevertheless be carried over into the entry set of the called procedure as calling-context effects because of an alias relationship established by the call, between an actual parameter and a formal call-by-reference parameter. Any element that enters a procedure because of such an alias relationship between parameters should be recoded to reflect this alias relationship.

A recoded element represents both the base element, which is the element as it would be if there were no alias relationship, and the non-empty alias relationship. Element recoding has two purposes. First, it allows the recoded element within the called procedure to be killed correctly through its alias relationship. Second, it allows the recoded element within the called procedure to be correctly associated with specific references to those aliases that are in the alias relationship.

Element recoding never involves a change of the base element, but only a change of the associated alias relationship, which would be the set of formal parameters to which the base element is, in effect, aliased. Because of element recoding, in effect a new element is generated, hence the separate *RECODE* set.

Figure 2.2 presents an algorithm for generating the entry-node input sets E_{in} and *RECODE*, for a forward-flow-or dataflow problem, for the assumed language model in which the visibility of each formal parameter is limited to the single procedure that declares it. For each element in the $OUT[c]$ set, the algorithm generates at most one element for inclusion in the entry-node input sets. The algorithm is unambiguous, except for line 10. The "can be affected by" test at line 10 is a generalization. The details of this test will depend on the specific dataflow problem being solved. For example, if the dataflow problem is reaching definitions, then each base element w represents a specific definition of some variable z. If the actual parameter p being tested by the algorithm is the variable z, and the corresponding formal parameter is call-by-reference, then the definition that w represents can be used or killed through that formal parameter, so w can be affected by that actual parameter z, and the "affected by" test is therefore satisfied. The $p \in OA$ test at line 10 covers the situation where an actual parameter p that is aliased to the formal f is itself a formal parameter that is effectively aliased to w. In this case f is established as a new effective alias for w, by transitivity of the alias relationship.

Referring to the algorithm, there is no carry over of the old alias relationship into the new alias relationship. The old alias relationship is represented by the OA set, and the new alias relationship is represented by the NA set. That this no-carry-over of the old alias relationship is correct, follows from the assumed language model. The aliases of element recoding are formal parameters, and the model states that each formal parameter is visible in only one procedure. This means there is no need to carry the old alias relationship into a different procedure, because the aliases cannot be referenced outside the single procedure in which the old alias relationship is active. Note that recursive calls are no exception to this no-carry-over rule, because a recursive call will cancel any alias relationship established for a base element by any prior call of the procedure.

In general, the fact that crossing elements are recoded when $NA \neq \emptyset$, and unrecoded when $NA = \emptyset$ and $OA \neq \emptyset$, places an added burden on the return-node equations to recognize an element that should be recovered from the exit-node entry set, necessitating, in effect, additional rules to cover this possibility. After an element is recovered, it would also be necessary to restore the alias relationship, if any, that

```
     — $e$ is an entry node.
     — This algorithm constructs the $E_{in}[e]$ and $RECODE[e]$ sets.
     begin
1       $E_{in}[e] \leftarrow \emptyset$
2       $RECODE[e] \leftarrow \emptyset$
3       for each predecessor call node $c$ of entry node $e$
4          for each element $x \in OUT[c]$
5             let $w$ be the base element of $x$
6             let $OA$ be the set of aliases, if any, associated with $w$, forming $x$
7             let $NA$ be the set of new aliases
8             $NA \leftarrow \emptyset$
9             for each actual parameter $p$ at call node $c$ that is aliased
              to a call-by-reference formal parameter $f$
10               if $(w$ can be affected by $p) \vee (p \in OA)$
11                  $NA \leftarrow NA \cup \{f\}$
                 fi
              end for
12            if $NA \neq \emptyset$
13               $RECODE[e] \leftarrow RECODE[e] \cup \{(w, NA)\}$
14            else if $w$ can cross the interprocedural boundary
15               $E_{in}[e] \leftarrow E_{in}[e] \cup \{w\}$
              fi
           end for
        end for
     end
```

Figure 2.2. Element-recoding algorithm for forward-flow-or dataflow problems.

it had prior to the call. This recognition and restoration problem is perhaps most easily solved by associating with each call node two additional sets, one for body-set elements and another for entry-set elements, where each set consists of ordered pairs. These sets would be determined whenever the entry-node entry set of the called procedure is computed.

The first element of each ordered pair is a crossing element x as it exists in the B_{out} or E_{out} set at the call node, and the second element is element y which is that element effectively generated from element x by the element-recoding algorithm of Figure 2.2 at either line 13 or line 15. If all crossing elements for the call are included in these additional sets, then the return-node equations can use these sets instead of the $B_{out}[p]$ and $E_{out}[p]$ sets to recognize elements to be recovered from the exit-node entry set. Recognition and restoration would be done by trying to match the exit-node entry-set element against the second element of an ordered pair from the appropriate additional set at the call node, and then, if there is a match, restoring the original element by using the first element of the matched pair. For example, if x is a crossing element in the B_{out} set of a call node, and y is the generated element, then (x, y) would be an ordered pair in the additional set for body-set elements. When the B_{in} set for the return node is computed, if y is in the exit-node entry set then it will match the ordered pair (x, y), and element x will be added to the B_{in} set.

As an example of why element recoding is necessary, consider the following. Suppose there are two different calls to the same procedure, and different definitions of global variable g reach each call. At one of the calls, g is also used as an actual parameter and the corresponding formal parameter is call-by-reference. The problem now is what to kill from the entry set whenever that formal parameter is defined in the called procedure. If the individual elements representing the different definitions of g do not somehow identify how they are related to this formal parameter, then the only choice is to kill all of them or none of them, and neither of these choices is correct in this case, as the only definitions of g that should be killed are those that entered the procedure from the call where g is aliased to the call-by-reference formal parameter.

2.2.3 Implicit Definitions Due to Calls

A call with parameters typically has implicit definitions associated with it. For example, if a formal parameter is call-by-reference, then each actual parameter aliased to that formal parameter is implicitly defined at each definition of the formal parameter. If a formal parameter is call-by-value-result, then that formal parameter is implicitly defined each time the called procedure is entered, and the actual parameter at the call is implicitly defined upon return from the call. From the standpoint of solving a dataflow problem such as reaching definitions, all implicit definitions due to calls should be determined, and elements generated at the appropriate nodes to represent these implicit definitions. The remainder of this section discusses the generation of implicit definitions and the determination of what reaches them for the specific problem of reaching definitions.

We assume that a formal parameter may be either call-by-reference, call-by-value, call-by-value-result, or call-by-result. For the reaching-definitions problem, before the iterative algorithm can be used to solve the dataflow equations, all GEN sets must be prepared.

For each point p in the program where a call-by-reference formal parameter is defined, add to the GEN set of the node for point p an implicit definition of each actual-parameter variable that is aliased to that formal parameter in a call. Each added implicit-definition element must be a recoded element that includes the alias relationship for that actual parameter. For example, suppose a procedure named A has two call-by-reference formal parameters, x and y, and inside A at point p there is a definition of x, and there are three calls of procedure A in the program. The first call aliases variable v to x. The second call aliases variable v to both x and y. The third call aliases variable w to x. Thus, at point p there would be three implicit-definition elements generated, namely $(v, \{x\})$, $(v, \{x, y\})$, and $(w, \{x\})$. As an example of what this element notation means, for the $(v, \{x\})$ element the v represents the implicit definition of variable v that occurs at point p, and the x represents the formal parameter that variable v is aliased to. As a special requirement for these implicit-definition elements, for the B_{out} set at the exit node of procedure A, the $(v, \{x\})$ element, if it reaches this set, can only cross from this set to the return node of the first call. Similarly, the $(v, \{x, y\})$ element can only cross to the return node of the second call, and the $(w, \{x\})$ element can only cross to the return node of the third call.

The crossing restrictions in the preceding example are due to a rule, now given. Let A denote a procedure containing a definition at point p of a call-by-reference formal parameter x, $(t, \{x\})$ is the implicit-definition element generated at point p for some specific call c of A that aliases actual-parameter variable t to x, and m is the exit node of A. If $(t, \{x\}) \in B_{out}[m]$, then $(t, \{x\})$ can only cross from $B_{out}[m]$ to the return node of call c, and as $(t, \{x\})$ crosses, it must be recoded as t by having its alias relationship nullified. This crossing-restriction rule is necessary because element $(t, \{x\})$ is both a body effect, because it is generated inside the called procedure, and a calling-context effect, because it is the result of a specific call of that procedure. This dual quality requires the special treatment that the rule provides. Nullifying the alias relationship as the element crosses to the return node is both good practice in general for this element, and a necessity if call c is a recursive call of A. As an example, assume that call c is a recursive call of A, and that variable t is a global variable. If $(t, \{x\})$ reaches the $B_{out}[m]$ set, the rule states that this element can only cross to the return node of call c, and that it be recoded as t. Assuming that this t element then reaches from this return node to the $B_{out}[m]$ set, t can then cross to any return node that has an in-edge from m. Although both the $(t, \{x\})$ and t elements refer to the same implicit definition of variable t occurring at point p, the two elements are not the same, and the crossing-restriction rule applies only to an element that is identical to the element generated at point p, which is $(t, \{x\})$.

The implicit definitions of actual-parameter variables is the most important category of implicit definitions that are due to call-by-reference formal parameters. However, there is also a second, less-important category. At each explicit definition of a variable t at point p inside A, such that variable t is also used in a call of A as an actual parameter aliased to a call-by-reference formal parameter x, then there is an implicit definition of formal parameter x at point p. The implicit-definition element generated at point p would be $(x, \{t\})$, meaning a definition of variable x at point p, aliased to variable t. However, assuming a formal parameter cannot be defined or used outside the procedure for which it is declared, it follows that there is no need for a crossing-restriction rule for these elements, because they cannot cross to any return node.

Normally, a definition of a variable kills all other definitions of that variable. However, the implicit definitions due to call-by-reference formal parameters have no associated kills. Instead, the following rule suffices. For each call-by-reference formal parameter x declared for procedure A, if all calls of A alias the same actual-parameter variable t to x, then each explicit definition inside A of either variable t or x, will kill all definitions of variable t and all definitions of variable x. Otherwise, if all calls of A do not alias the same actual-parameter variable t to x, then each explicit definition inside A of either variable t or x will kill only the definitions of that variable and those recoded elements that are aliased to that variable.

The entry-node GEN set will be used to hold all implicit definitions of formal parameters that occur upon procedure entry. Thus, for each entry node, for each formal parameter of the represented procedure that is call-by-value or call-by-value-result, add to the GEN set of that entry node an element that represents an implicit definition of that formal parameter occurring at that entry node.

The return-node GEN set will be used to hold all implicit definitions of actual parameters that may occur upon return from the called procedure. Thus, for each return node, for each actual parameter of the associated call whose corresponding formal parameter is call-by-result or call-by-value-result, add to the GEN set of that return node an element that represents an implicit definition of that actual parameter occurring at that return node. The return-node $KILL$ set should represent all elements that will be killed by these implicit definitions of actual parameters.

With the GEN sets ready, the iterative algorithm can proceed. Once the iterative algorithm is ended, a follow-on step is done: a) Examine the B_{out} set for each exit node. For each definition d in this set of a formal parameter p, and p is call-by-result or call-by-value-result, then d reaches the implicit use of this formal parameter by those implicit definitions of actual parameters found at the various return nodes whose corresponding formal parameter is p. The element representing d can be added to the B_{in} sets of those return nodes in a way that reflects the reach. b) Examine the OUT set of each call node. For each definition d in this set of a variable that is used as an actual parameter in the call, and the corresponding formal parameter is call-by-value or call-by-value-result, then d reaches the implicit use of the defined variable by the implicit definition of the corresponding formal parameter found at the entry node of the called procedure. The element representing d can be added to the E_{in} set of that entry node in a way that reflects the reach.

2.3 Interprocedural Forward-Flow-And Analysis

This section gives the dataflow equations used by our interprocedural analysis method for forward-flow-and problems. The difference between these equations and the equations for forward-flow-or is explained.

For forward-flow-and problems, some changes are needed to the dataflow equations given in Section 2.2.1. Of course, the confluence operator must be changed from union to intersection. However, it is still necessary to construct the entry-node entry set as the union of all crossing effects from the predecessor-node sets, so that calling context can be properly recovered at the return nodes. At the same time, the entry set must always be constructed as the intersection of predecessor-node sets, if the entry set is to be a part of the IN and OUT sets. These conflicting requirements for the entry-node entry set can be resolved by maintaining two separate entry sets at each node. The revised dataflow equations follow. The two conditions, C_1 and C_2, are explained in Section 2.2.1.

For any node n.

$$IN[n] = E_{in}^{(2)}[n] \cup B_{in}[n]$$

$$OUT[n] = E_{out}^{(2)}[n] \cup B_{out}[n]$$

Group I: n is an entry node.

$$B_{in}[n] = \emptyset$$

$$E_{in}^{(1)}[n] = \bigcup_{p \in pred(n)} \{x \mid x \in (E_{out}^{(1)}[p] \cup B_{out}[p]) \wedge C_1\}$$

$$E_{in}^{(2)}[n] = \bigcap_{p \in pred(n)} \{x \mid x \in OUT[p] \wedge C_1\}$$

$$B_{out}[n] = GEN[n]$$

$$E_{out}^{(1)}[n] = E_{in}^{(1)}[n] \cup RECODE^{(1)}[n] \cup RECODE^{(2)}[n]$$

$$E_{out}^{(2)}[n] = E_{in}^{(2)}[n] \cup RECODE^{(2)}[n]$$

Group II: n is a return node, p is the associated call node and q is the exit node of the called procedure.

$$B_{in}[n] = \{x \mid (x \in B_{out}[p] \wedge (\overline{C_1} \vee (C_1 \wedge C_2 \wedge x \in E_{out}^{(1)}[q]))) \vee (x \in B_{out}[q] \wedge C_2)\}$$

$$E_{in}^{(i)}[n] = \{x \in E_{out}^{(i)}[p] \mid \overline{C_1} \vee (C_1 \wedge C_2 \wedge x \in E_{out}^{(1)}[q])\}; i = 1, 2.$$

$$B_{out}[n] = (B_{in}[n] - KILL[n]) \cup GEN[n]$$

$$E_{out}^{(i)}[n] = E_{in}^{(i)}[n] - KILL[n]; i = 1, 2.$$

Group III: n is not an entry or return node.

$$B_{in}[n] = \bigcap_{p \in pred(n)} B_{out}[p]$$

$$E_{in}^{(i)}[n] = \bigcap_{p \in pred(n)} E_{out}^{(i)}[p]; i = 1, 2.$$

$$B_{out}[n] = (B_{in}[n] - KILL[n]) \cup GEN[n]$$

$$E_{out}^{(i)}[n] = E_{in}^{(i)}[n] - KILL[n]; i = 1, 2.$$

The entry set $E^{(1)}$ is the set used to recover calling context, and the entry set $E^{(2)}$ is the set that is a component of the IN and OUT sets. The $RECODE$ sets appearing in the entry-node equations represent recoded elements as explained in Section 2.2.2. The $RECODE^{(1)}$ set will just be the union of the recoded elements generated from each predecessor call node c, using the algorithm of Figure 2.2 and drawing from the $E_{out}^{(1)}[c]$ and $B_{out}[c]$ sets at line 4 instead of the $OUT[c]$ set.

Similarly, the $RECODE^{(2)}$ set could just be the intersection of the recoded elements from each predecessor call node c, drawing from the $OUT[c]$ set at line 4. However, doing this may cause the unnecessary loss of recoded elements when the same underlying base element w is found in each $OUT[c]$ set. To avoid such loss, an improved rule states that if the same base element w is found in each $OUT[c]$ set, and there is one or more non-empty alias relationships for that w occurring at one or more predecessor nodes c, then a single recoded element for that w that encodes all of these alias relationships would be generated into the $RECODE^{(2)}$ set, otherwise no recoded element for that w would be generated into the $RECODE^{(2)}$ set. For example, suppose c has three different values for a given entry node, and the same base element w is found in each $OUT[c]$ set, and at one c there is an empty alias relationship, at the second c there is an alias relationship to formal parameter x, and at the third c there is an alias relationship to formal parameter y. For this example, the single recoded element would be $(w, \{x, y\})$, and this recoded element can either be killed directly through w, or indirectly through x, or through y. Note that the complete kill of this recoded element at any kill point, even though the kill may have been made through an alias that was not established at each c, is nevertheless correct. The intersection confluence operator associated with $RECODE^{(2)}$ implicitly requires that for base element w to pass a kill point, it must be on every call path past that kill point, which is not the case when w is killed from at least one call path, which happens when that w is killed through an alias that was established by at least one of the c. If the specific dataflow problem being solved allows the base element to be used through one of its effective aliases, then a flag could be associated with each alias in the recoded elements of $RECODE^{(2)}$, and this flag could indicate whether or not the alias was established at each c. In the case of the example, the recoded element with flags would be $(w, \{x_{\mathrm{not}}, y_{\mathrm{not}}\})$. Only a use of the base element through an alias established at each c would be a use through an alias that occurs on every call path, and this kind of use would be the all-paths use that is implicitly required by the specific dataflow problem by virtue of it being forward-flow-and.

With the exception of the confluence operator and the two different entry sets, the equations for forward-flow-and are the same as for forward-flow-or, and are likewise correct. Set $E^{(2)}$ fulfills the requirement for the IN and OUT sets by consistently using the intersection confluence operator for its construction, just as B does. The equations for the $E^{(1)}$ and $E^{(2)}$ sets only differ at the entry node, and there the only difference is the confluence operator, and the way the $RECODE$ sets are built. As set intersection is the confluence operator for $E^{(2)}$, and set union for $E^{(1)}$, and the

Table 2.2. Solution of forward-flow-and equations for Figure 2.3.

Node	$E^{(1)}_{in}$	$E^{(1)}_{out}$	$E^{(2)}_{in}$	$E^{(2)}_{out}$	B_{in}	B_{out}
1	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
2	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	{1, 2}
3	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	{1, 2}	{1, 2, 4}
4	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	{1, 4, 5}	{1, 4, 5}
5	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	{1, 2}	{1, 2, 3}
6	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	{1, 3, 5}	{1, 3, 5}
7	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	{1, 5}	{1, 5}
8	{2, 3, 4}	{2, 3, 4}	{2}	{2}	$\emptyset$	$\emptyset$
9	{2, 3, 4}	{3, 4}	{2}	$\emptyset$	$\emptyset$	{5}
10	{3, 4}	{3, 4}	$\emptyset$	$\emptyset$	{5}	{5}

$RECODE^{(2)}$ set is added to both $E^{(1)}$ and $E^{(2)}$, it follows that $E^{(2)}$ will be a subset of $E^{(1)}$ at every node. Thus, $E^{(1)}$ can be used to recover calling context for $E^{(2)}$. Set $E^{(1)}$ also serves to recover calling context for both $E^{(1)}$ and B, because $E^{(1)}$ is built at the entry node from these two sets, and the use of union as the confluence operator guarantees that all calling-context effects will be collected.

Table 2.2 shows the result of solving the equations for the flowgraph of Figure 2.3. By "solving" we mean that, in effect, the iterative algorithm has been used and all the sets are stable. The dataflow problem is available expressions, and variable w is local while variables x, y, and z are global. Available expressions is the problem of determining whether the use of an expression is always reached by some prior use of that expression, for certain expressions in the program. In Figure 2.3, nodes 1 and 8 are entry nodes, nodes 7 and 10 are exit nodes, nodes 3 and 5 are call nodes, and nodes 4 and 6 are return nodes. Alongside each node is its basic block. Each expression is superscripted with an identifier that is the set element used in Table 2.2 to represent that expression.

```
procedure main
begin
   y = w + 1
   z = x + 1
   if(e)
        a = z + 1
        call f()
   else
        a = y + 1
        call f()
end
```

```
procedure f()
begin
   x = z + 2
end
```

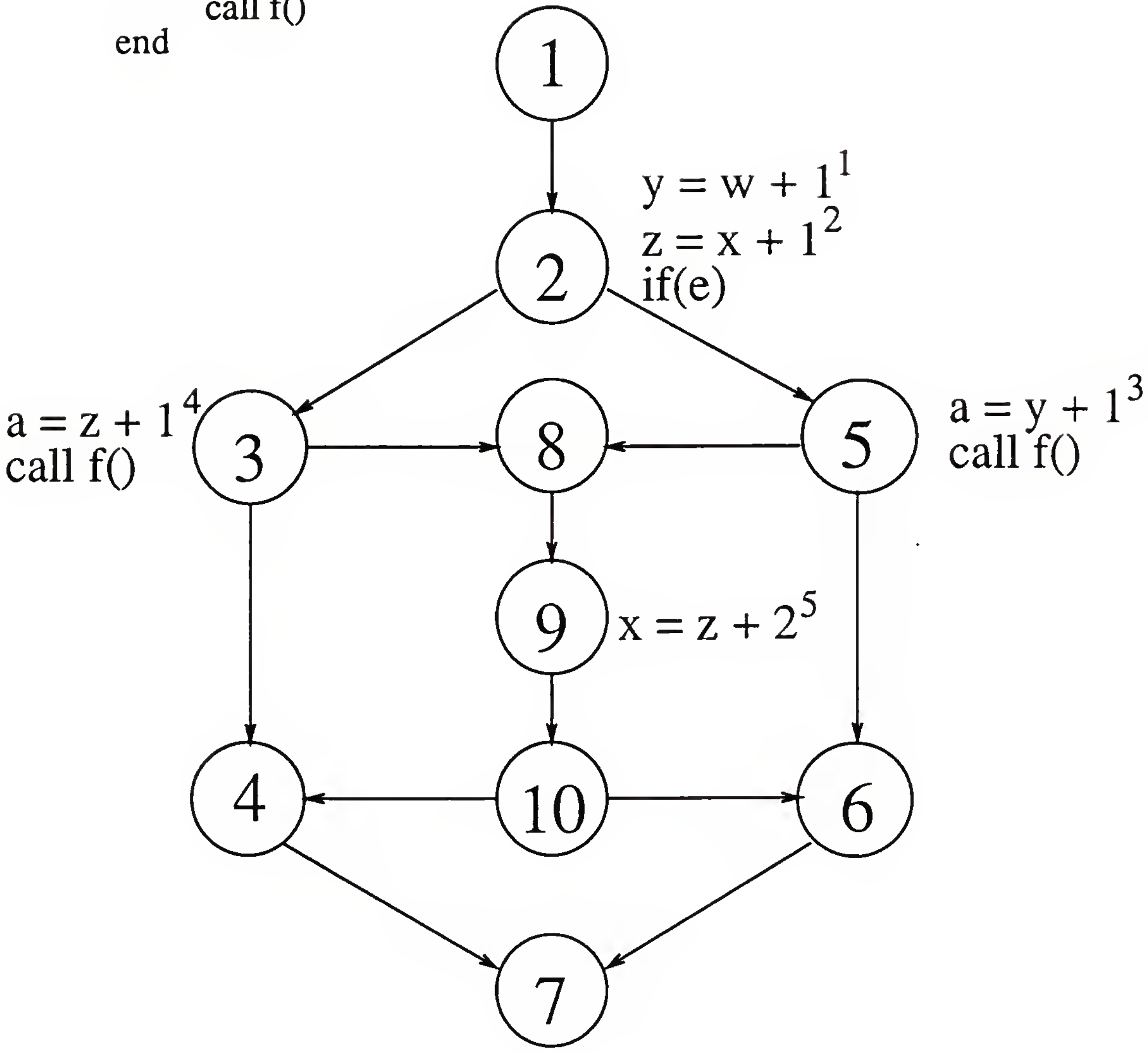

Figure 2.3. An available-expressions example.

2.4 Interprocedural Backward-Flow Analysis

Backward-flow problems are basically forward-flow problems in reverse. However, the same flowgraph is used for both forward-flow and backward-flow problems. To convert the equations for forward-flow-or to backward-flow-or, or for forward-flow-and to backward-flow-and, the transformation is mechanical and straightforward. The same equations are used, but various words and phrases are everywhere changed to reflect the reverse flow. For example, "*pred*(n)" for predecessors becomes "*succ*(n)" for successors, "out" subscripts become "in" subscripts and "in" subscripts become "out" subscripts, IN becomes OUT and OUT becomes IN, "call node" becomes "return node" and "return node" becomes "call node", "entry node" becomes "exit node" and "exit node" becomes "entry node". For backward flow, the nodes requiring special equations are the exit node and call node, and not the entry node and return node as for the forward-flow problems.

2.5 Complexity of Our Interprocedural Analysis Method

To determine the worst-case complexity of our method for the assumed language model in which the visibility of each formal parameter is limited to the single procedure that declares it, we consider the solution of the dataflow equations for only one element at a time. Let n be the number of flowgraph nodes. Let the elementary operation measured by the complexity be the computation of the dataflow equations once at a single, average flowgraph node, for a single element. Only the presence or absence of the single element within a particular body or entry set need be represented, and this requires no more than a single bit of storage for each set referenced by the equations. Thus, computing the dataflow equations once at an average node, for a single element, will consist of a small number of integer operations, assuming that the average in and out-degree of the flowgraph nodes is bounded by a small constant, which will always be the case for flowgraphs generated from real programs, and also assuming that the length of recoded elements will be small. Referring to the algorithm of Figure 2.2, the length of a recoded element is $1 + |NA|$, and $|NA|$ is bounded from above by the number of call-by-reference formal parameters of the given procedure. As a rule, this upper bound will be small.

We next consider the total number of node visits required to solve the dataflow equations for a single element. Prior to solving the equations, all body and entry sets are initialized to empty, at complexity $O(n)$. The empty sets represent the absence of the element. Note that each set has only two states: either the element is present, or it is absent. Assuming a forward-flow problem, each time the equations are computed for a node, if any of the *out* sets have changed from their previous state, then the equations will be computed for all successor nodes. The forward-flow-or equations have only two *out* sets per node, and the forward-flow-and equations have three. It follows that repeated computation of the equations for a single node will cause the successor nodes to be marked for computation at most two or three times, depending on the equations being used. Given that the average number of successor nodes is bounded by a small constant, it follows that the total number of node visits required to solve the dataflow equations for a single element will be bounded from above by $k_1 n$ where k_1 is a constant, giving a worst-case complexity of $O(n)$ for solving the dataflow equations for a single element.

The worst-case complexity of solving the dataflow equations for m total elements will therefore be $O(mn)$. Let b be the number of base elements for the program being analyzed, and let r be the number of recoded elements, giving $m = b + r$. As an example, for the reaching-definitions dataflow problem the base elements will be all the definitions in the program. We assume that for the kind of dataflow problems our method is meant to solve, the number of base elements will be a linear function of the program size, and therefore proportional to n. Let constant k_2 be an upper bound of b/n. We also assume the universe of real, useful programs, written by programmers to solve practical problems. To determine an upper bound for r, let k be the maximum number of formal parameters for a single procedure. That k is a constant independent of program size should be obvious.

Given k and the algorithm of Figure 2.2, and allowing all possible combinations of the formal parameters of any single procedure, the maximum number of recoded elements for any single procedure and base element is $k_3 = \sum_{i=1}^{k} \left(\begin{array}{c} k \\ i \end{array} \right) = 2^k - 1$. Note that k_3 is a constant, albeit an enormous constant. The maximum number of recoded elements for any single procedure will therefore be $k_3 b$. In the assumed language model, each formal parameter is visible in only one procedure, and this means each recoded element is confined to a single procedure when the dataflow equations are solved. Therefore, the total number of node visits required to solve the dataflow equations for all the recoded elements will be bounded from above by $\sum_{i=1}^{j} k_1 s_i k_3 b$ where j is the number of procedures in the flowgraph, and s_i is the number of flowgraph nodes in the ith procedure. This upper bound can be rewritten as $\sum_{i=1}^{j} k_1 k_2 k_3 n s_i$. Ignoring constants and given that $\sum_{i=1}^{j} s_i = n$ and $\sum_{i=1}^{j} n s_i = n^2$, the worst-case complexity of our method for the assumed language model is $O(n^2)$, and the elementary operation measured by the complexity is a small number of integer operations assuming that the average recoded-element length is small.

For a program from the assumed universe of programs, the likelihood of a large complexity constant due to element recoding is very low, for the following reason. In order to increase the number of recoded elements for a given base element and procedure, the given base element must, in effect, be repeatedly aliased to different combinations of formal parameters in the given procedure. The algorithm of Figure 2.2 generates at most a single recoded element for each element in the *OUT* set, so to increase the number of recoded elements as stated, there must be multiple calls to the same procedure, and in these different calls the same base element must be aliased to different formal-parameter combinations. To assess the likelihood of this requirement being met, consider that for any given program from the assumed universe, the type and purpose of a variable determines how that variable is used in that program, and each variable used in a program by necessity has a purpose. Given a number of different calls to the same procedure, and given that a variable appears as one or more of the actual parameters in each of the calls, then as a rule we expect that variable to always occupy the same parameter positions in those calls because there is always a close correspondence between parameter position and the purpose of the variable that occupies that position. Note that by "variable" we mean a variable and any aliases it may have, including formal-parameter aliases. A variable and its aliases are interchangeable and share the same purpose because by definition they reference the same data.

It might be argued that a language such as C has procedures that have a variable number of arguments, such as *printf* and *scanf*, for which the same variable could easily occupy different actual-parameter positions in different calls. This is true, but such library procedures are best treated as unknown calls, and there is no element recoding for unknown calls. For the needs of element recoding in the rare case of a user-written procedure with a variable number of arguments, a single formal parameter could stand for the variable portion of the formal parameters, and conservative assumptions could be made whenever that single formal is, in effect, referenced. Aside from mentioning this, we do not consider such user-written variable-argument procedures further.

For a dataflow problem such as reaching definitions, the base element can only be affected by a single variable. For such a dataflow problem, the purposefulness of variables makes it very unlikely that an increase in the number of recoded elements for a given procedure and base element can even begin, let alone be sustained. However, such an increase would be more likely for a dataflow problem where the base element can be affected by several different variables. An example would be available expressions, because each base element could be affected by as many different variables as compose the expression represented by that base element.

In light of the preceding argument regarding the purposefulness of variables, for the reaching-definitions and similar dataflow problems, we expect the maximum number of recoded elements for any given procedure and base element in the majority of the programs in the assumed universe, to be one, and a little higher than one for the remaining programs in that universe. Given the algorithm of Figure 2.2, we also expect the average length of each recoded element to be slightly more than two, given the preceding expectation that there will be a very small maximum number of recoded elements for any given procedure and base element, and assuming that most base elements when aliased by a call will be aliased to only a single formal parameter, and only occasionally aliased to more than one. Note that this expected average length of the recoded elements is consistent with the claim that the elementary operation measured by the worst-case complexity of our method is a small number of integer operations.

It may be noticed that the complexity of $O(n^2)$ for our interprocedural analysis method is the same as the known worst-case complexity for intraprocedural dataflow analysis, assuming there are no restrictions on the flowgraph. This fact makes it unlikely that it would be possible to improve on our method in terms of complexity, without resorting to flowgraph restrictions. However, although the complexities are the same, this does not mean interprocedural dataflow analysis will now take roughly the same time as intraprocedural dataflow analysis. The following inequality should make this clear. $\sum_{i=1}^{j} s_i^2 \leq n^2$, given that j is the number of procedures in the flowgraph, s_i is the number of flowgraph nodes in the ith procedure, and $\sum_{i=1}^{j} s_i = n$.

Besides the language model that is assumed for this chapter, an alternative model allows each formal parameter to have visibility in more than a single procedure.

Examples of programming languages that fit this alternative model are Pascal and Ada, which allow nested procedures. Element recoding can be used for this alternative model, but unless precision is compromised, the worst-case complexity for solving the equations will be exponential, because the number of recoded elements could grow exponentially assuming that alias information is compounded when a recoded element is recoded. The exponential complexity of tracking aliases due to calls was first considered by Myers [17], and more recently by Landi and Ryder [15]. In practice, the cost of precise element recoding for the alternative language model may be acceptable for the assumed universe of programs, and for the same reason given previously regarding the purposefulness of variables. However, we do not consider the alternative model further.

2.6 Experimental Results

There are experimental data for our interprocedural analysis method. Specifically, two different prototypes have been constructed, and they both solve the reaching-definitions dataflow problem using our method. Both prototypes accept C-language programs as the input to be dataflow analyzed. For simplicity, these prototypes impose some restrictions on the input, such as requiring that all variables be represented by single identifiers, thereby excluding variables that have more than one component, such as structure and union variables. In addition, there is no logic in the prototypes to determine what pointers are pointing at, so pointer dereferencing is essentially ignored. The prototypes do not accept pre-processor commands, so the input programs must be post-preprocessor.

Both prototypes, named prototype 1 and prototype 2, use the same code to parse the input program and construct the flowgraph. However, they differ in how they implement our analysis method. Prototype 1 prepares a single bit-vector format containing all the definitions in the input program, and then solves the dataflow equations once for the program flowgraph. Prototype 2 uses a single integer as the bit vector and solves the dataflow equations for the program flowgraph as many times as there are base elements. For the reaching-definitions dataflow problem, the definitions in the program are the base elements. We call the approach used by prototype 2 one-base-element-at-a-time, and the approach used by prototype 1 is all-at-once.

It might be expected that prototype 2 would be many times slower than prototype 1, because of the big difference in bit-vector sizes, but this is not the case. For prototype 1, calculations using varied test results show that $V \times S_1 \approx D$, where V is the average number of visits per flowgraph node made to solve the dataflow equations, S_1 is the integer size of the bit vector for prototype 1, and D is the number of definitions in the input program. This relationship for prototype 1 means that prototype 2 should run at roughly the same speed as prototype 1, because solving the dataflow equations for a single element will require an average of roughly one visit per flowgraph node and the application of the dataflow equations to a vector of size one. Note that the total amount of work prototype 1 must do per flowgraph node to solve the equations is proportional to the product $V \times S_1 \approx D$, and the total amount of work prototype 2 must do per flowgraph node to solve the equations for the D base elements is proportional to the product $V \times S_2 \times D \approx 1 \times 1 \times D \approx D$, where S_2 is the integer size of the bit vector for prototype 2.

Experimental results have supported the expectation of similar speeds for the two prototypes. When deciding on the design of a practical tool, this finding is important and decisively tips the scales in favor of the one-base-element-at-a-time approach used by prototype 2. For both prototypes, the bit space needed for set storage is nks, where n is the number of flowgraph nodes, k is the average number of sets per node, and $s = \max(\text{average set bit-size for any solving of the equations})$. Note that for prototype 1 there is only one solving of the equations, and for prototype 2 there are as many solving of the equations as base elements. The primary reason

Table 2.3. Typical experimental results for the two prototypes.

defs	defs global	calls	nodes	prototype 1	prototype 2
2126	30%	521	4191	49s	1m21s
2026	60%	472	3948	55s	2m22s
4109	30%	924	7537	4m18s	4m38s
4223	60%	916	7723	4m57s	8m19s
6115	30%	1325	11185	N/A	10m0s
6091	60%	1411	11288	N/A	18m18s
8200	30%	1832	14799	N/A	17m44s
8054	60%	1726	14641	N/A	30m2s
10299	30%	2164	18434	N/A	23m55s
10016	60%	2356	18587	N/A	45m8s

the approach used by prototype 2 is preferable when compared with the all-at-once approach used by prototype 1, is the likelihood of a greatly reduced s value. For example, without element recoding, the s value is 1 for prototype 2, and D for prototype 1. Allowing element recoding, the s value for the prototype-2 approach will be 1 + max(average number of recoded elements per procedure for any solving of the equations). Here we assume that the best way to add element recoding to prototype 2 would be, for each solving of the equations, to solve the equations for both a single base element and all recoded elements generated from that base element.

Table 2.3 presents typical experimental results for the two prototypes. Each table row represents a different input program. The input programs were randomly generated by a separate program generator. The generated input programs are syntactically correct and compile without error, but have meaningless executions. Each input program in Table 2.3 has 100 procedures. Only prototype 1 currently has element-recoding logic, so the input programs do not have call parameters and the table data do not reflect element-recoding costs. Measuring element-recoding costs for randomly generated programs would be somewhat meaningless anyway, since the purposefulness-of-variables principle would be violated.

Referring to the columns of Table 2.3, "defs" is the total number of definitions in the input program, "defs global" is the percentage that define global variables, "calls" is the number of known calls, "nodes" is the number of flowgraph nodes, "prototype 1" is the total CPU usage time in minutes and seconds required by prototype 1 to completely solve the reaching-definitions dataflow problem for the input program and generate a report of all the reaches, and "prototype 2" is the same thing for prototype 2. The hardware used was rated at roughly 23 MIPS. The large space requirements of prototype 1 prevented running it for the larger input programs in the table.

CHAPTER 3
INTERPROCEDURAL SLICING AND LOGICAL RIPPLE EFFECT

3.1 Representing Continuation Paths for Interprocedural Logical Ripple Effect

This section lays the theoretical basis for our algorithm. The problem of interprocedural logical ripple effect is examined from the perspective of execution paths and their possible continuations. First, general definitions are given, followed by three assumptions and a definition of the Allow and Transform sets, followed by Lemma 1, Theorems 1 through 4, and a discussion of the potential for overestimation inherent in the Allow set.

A variable is defined at each point in a program where it is assigned a value. A *definition* is assumed to have the general form of "$v \leftarrow$ expression", where v is the variable being defined and "$\leftarrow$" is an assignment operator that assigns the value of expression to v. If the expression includes variables, then these variables are termed the use variables of the definition. In general, a *use* is any instance of a variable that is having its value used at the point where the variable occurs.

A procedure contains a definition if the statement that makes the definition is in the body of the procedure. Similarly, a procedure contains a call if the statement that makes the call is in the body of the procedure. The body of a procedure is those statements that are defined as belonging to the procedure.

Frequent reference is made in this chapter to a procedure containing a statement, or containing a call, or containing a flowgraph node. For languages that allow nested procedures, such as Pascal and Ada, note that procedure nesting in these languages is a mechanism for controlling variable scope, and not a mechanism for sharing statements, calls, or flowgraph nodes. Throughout this chapter we assume that at most only a single procedure contains any given statement, call, or flowgraph node.

Let d and dd be two definitions, possibly the same, in the same program. Let dd have a use-variable v, let v_{dd} be that use-variable instance, and let d define v. Given a possible execution path between definition d and v_{dd}, along which the definition of v that d represents would be propagated, such a path is referred to as a *definition-clear path* between d and v_{dd} with respect to v. Definition d can only be *propagated* along an execution path to the end of that path if either definition d itself or an element that represents definition d exists at the beginning of that path, and there is no redefinition of v along that path. Definition d is said to *affect* definition dd if there is a definition-clear path between d and v_{dd} with respect to v. Similarly, definition d affects use u if u is an instance of v, and there is a definition-clear path between d and u with respect to v. For convenience, v will not be explicitly mentioned when it is understood. Note that whenever we speak of an execution path between two points, we always mean that the execution path begins at the first point and ends at the second point. For example, an execution path between d and dd begins at the program point where d occurs and ends at the program point where v_{dd} occurs. For convenience, we assume that dd and v_{dd} occupy the same program point.

Assumption 1. A called procedure, if it returns, always returns to its most recent caller. A procedure that returns, always returns to the most recent unreturned call.

Assumption 2. A call has no influence on the execution paths taken inside the called procedure.

Assumption 3. There are no recursive calls.

Assumption 1 reflects the behavior of all the procedural languages that we know of. Regarding Assumption 2, our algorithm may in fact overestimate the logical ripple effect because of both Assumption 2 and the unstated but standard assumption of intraprocedural dataflow analysis that all paths in a procedure flowgraph are possible execution paths. However, these two assumptions are unavoidable because determining all the truly possible execution paths in an arbitrary program is known to be an undecidable problem. Regarding Assumption 3, making this assumption improves the precision of our algorithm because this assumption removes a potential cause of overestimation. The consequence of using our algorithm for a program with recursive calls is discussed at the end of Section 3.2.

To determine what a definition affects when it is constrained by ripple effect, it is useful to introduce two concepts: backward flow and forward flow. Given an execution path, whenever the execution path returns from a procedure to a call, this is termed *backward flow.* All other parts of the execution path may be termed *forward flow.* Note that the possibilities for backward flow are constrained by Assumption 1, and therefore constrained by the relevant execution paths that lead up to the point of the return in question.

Regarding a given execution path, those call instances within that execution path that have yet to be returned to within that path, called *unreturned calls*, are the parts of the path that constrain backward flow. Note that this constraint is a positive constraint, since a call cannot be returned to unless that call exists as an unreturned call in at least one relevant execution path.

Definition 1. Two sets, Allow and Transform, will be used to represent the backward-flow restrictions associated with a particular definition d. Let p be the program point where definition d occurs. The elements in both sets are calls. The Allow set identifies only the calls to which the execution path continuing on from point p may make an unmatched return to—until the backward-flow restrictions represented by this Allow set are effectively cancelled by the interaction between the execution-path continuation and the Transform set, explained shortly. An *unmatched return* is a return made during the execution-path continuation to a call instance that precedes the beginning of that execution-path continuation. The call instance is necessarily an unreturned call, as otherwise it could not be returned to. $|\text{Allow}| \leq$ the total number of different calls represented in the program text. We define Allow $= \emptyset$ to mean there are no backward-flow restrictions for d. The Transform set identifies only the calls to which the execution path continuing on from point p may make an unmatched return to, and upon this unmatched return, the execution-path continuation is no longer constrained by the Allow and Transform sets associated with d. The following relationships hold. Transform $\subseteq$ Allow. If Allow $\neq \emptyset$ then Transform $\neq \emptyset$.

Note that minimizing backward-flow restrictions must be done whenever the possible execution paths allow it, because otherwise the computed logical ripple effect—which is the whole purpose of this formal-analysis section—may be missing pieces that belong in it but were not added to it because backward-flow restrictions were retained that are not valid for all the possible execution paths involved.

Lemma 1. For any execution path P between two program points p and q, if P includes two or more call instances made in P that have not been returned to in P, then for these unreturned calls, c_i calls the procedure containing c_{i+1}, where c_i is the ith unreturned call, in execution order, made in P.

Proof. Assume that the next unreturned call c_{i+1} is not contained in the procedure that was called by c_i. Let X be the procedure called by c_i, and let Y be the procedure that contains c_{i+1}. The execution path in P between making the call c_i and making the call c_{i+1} must include a path out of procedure X and into procedure Y so that the call c_{i+1} can be made. A path out of procedure X can occur in only two ways. Either X returns to a call, or X itself makes a call. If X returns to a call, then by Assumption 1, c_i would be returned to, contradicting the given that c_i has not been returned to. This means X must make a call to get to Y. Let c be the call contained in X that is the last call contained in X on the execution path in P taken from X to Y so as to make the call c_{i+1}. If X makes the call c, and c has not been returned to in P, then c would precede c_{i+1} as an unreturned call following c_i, contradicting the given that c_{i+1} is the next unreturned call in execution order after c_i. If c has been returned to in P, then all calls occurring on the execution path between the call c and the return to c must have been returned to according to Assumption 1. This would mean c_{i+1} has been returned to, contradicting the given that c_{i+1} has not been returned to. Thus, it is true that c_i calls the procedure containing c_{i+1}, as assuming otherwise leads to contradictions. □

Definitions for Theorems 1 through 4. Let d and dd be the two definitions previously defined. Let A and T be the Allow and Transform sets associated with d. Let P be a single execution path between d and dd, and along which d can affect dd, subject to the constraints on P imposed by A and T. P will consist of a sequence of calls and returns, if any, in the order they are made. Any instance of a call made in P that is not returned to in P, is an unreturned call in P.

K is defined for P if and only if P contains an unmatched return—meaning a return to a call instance that precedes the beginning of P—to a call $\in T$. K is that part of P that follows the first unmatched return to a call $\in T$. Thus, K represents the continuation of P after the unmatched return. Any instance of a call made in K that is not returned to in K, is an unreturned call in K.

Referring to each of the four theorems in turn, let AA and TT be the Allow and Transform sets for dd given all the paths P that meet the requirements of P as stated by that theorem. Let AA_P and TT_P be the Allow and Transform sets for dd given a single path P that meets the requirements of P as stated by that theorem. The four theorems that follow each define AA and TT. Note that for any given P, A, and T, one of the four theorems will apply.

Theorem 1. If (1) $A = \emptyset$, and P has no unreturned calls, or (2) $A \neq \emptyset$, K is defined for P, and K has no unreturned calls, then $AA \leftarrow \emptyset$ and $TT \leftarrow \emptyset$.

Proof. For case (1), d is free of backward-flow restrictions and d has affected dd without making an unreturned call, therefore dd will be free of backward-flow restrictions, giving $AA \leftarrow \emptyset$ and $TT \leftarrow \emptyset$. For case (2), as soon as path P makes an unmatched return r to a call $\in T$, then by Definition 1 what d can affect is no longer constrained by A and T, and this freedom from constraint by A and T passes by transitivity to dd because d affects dd.

When K is defined for P, the unmatched return r in P that immediately precedes the beginning of K, means that any unreturned calls in P are also in K. This is because all call instances within P are more recent than the call instance that matches the unmatched return r. Thus, by Assumption 1 all call instances in P preceding the return r must be returned to in P before r can occur. Therefore, P has no unreturned calls because K has no unreturned calls. Thus, dd is free of backward-flow restrictions since A, T, and P contribute nothing in the way of constraint, giving $AA \leftarrow \emptyset$ and $TT \leftarrow \emptyset$. □

Theorem 2. If (1) $A = \emptyset$, and P has at least one unreturned call, or (2) $A \neq \emptyset$, K is defined for P, and K has at least one unreturned call, then $AA \leftarrow \bigcup_{\text{all such } P}$ {the unreturned calls of P}, and $TT \leftarrow \bigcup_{\text{all such } P}$ {the first unreturned call in P}.

Proof. For case (1), A and T contribute nothing in the way of constraint to AA_P and TT_P. Because d affects dd along path P which contains unreturned calls, by Assumption 1 those unreturned calls must be returned to first before any other unreturned calls can be made from the execution-path continuation point of dd onward. Hence, $AA_P \leftarrow$ {the unreturned calls of P}. Because d had no backward-flow restrictions, it follows that once all the unreturned calls of P are returned to by the execution-path continuation, then that continuation would no longer have any backward-flow restrictions. Because of Assumption 3 and Lemma 1, all the unreturned calls of P are returned to when the sequentially first unreturned call in P is returned to. Hence, $TT_P \leftarrow$ {the first unreturned call in P}. For case (2), as shown in the proof of Theorem 1 case (2), A and T contribute nothing to AA_P and TT_P when K is defined for P. Thus, this case (2) is effectively the same as case (1), because the A and T sets contribute nothing and an unreturned call in K is an unreturned call in P. Therefore, $AA_P \leftarrow$ {the unreturned calls of P} and $TT_P \leftarrow$ {the first unreturned call in P}.

From Definition 1 and the general definitions of AA, TT, AA_P, and TT_P, it follows that $AA \leftarrow \bigcup_{\text{all such } P} AA_P$ and $TT \leftarrow \bigcup_{\text{all such } P} TT_P$. Thus, $AA \leftarrow \bigcup_{\text{all such } P}$ {the unreturned calls of P}, and $TT \leftarrow \bigcup_{\text{all such } P}$ {the first unreturned call in P}. $\square$

Theorem 3. If $A \neq \emptyset$, K is not defined for P, and P has no unreturned calls, then $AA \leftarrow \{x \mid x \in A \wedge (x$ is part of a possible execution path that inclusively begins with a call $\in T$ and ends with a call of the procedure containing *dd*, such that each unreturned call in this possible execution path is in $A)\}$, and $TT \leftarrow AA \cap T$.

Proof. Note that only one procedure contains *dd*. Because K is not defined for P, it follows that P was constrained in its entirety by A, never making an unmatched return to a call $\in T$. Because P has no unreturned calls, *d* can only affect *dd* along P by making one or more unmatched returns to calls $\in (A - T)$, unless *d* and *dd* are in the same procedure.

A, in effect, represents possible execution paths with unreturned calls by which *d* was affected. However, once given P, the path P may eliminate some of the paths from A as being possible, and return to some of the unreturned calls in A. Thus, although P contributes nothing directly to AA, it may narrow the unreturned execution-path possibilities that A can contribute to AA. AA as defined for this theorem, captures all execution paths in A that begin with a call $\in T$ and end with a call of the procedure that contains *dd*. Given Assumption 3, it should be obvious that these are all the possible paths in A that are unreturned after P. Note that if *d* and *dd* are in the same procedure, then $AA = A$ and $TT = T$. Assume that

d and dd are in different procedures. Any call $\in A$ that is not part of at least one path in A that makes a call of the procedure containing dd, must be excluded from AA because P requires a path in A that passes through the procedure containing dd, because otherwise P could not make a return to the procedure containing dd. Any call $\in A$ that is on a path in A between the procedure containing dd and the procedure containing d, must be excluded from AA because the procedure containing dd has been returned to by P. The definition of AA for this theorem satisfies these two exclusions.

That $TT \leftarrow AA \cap T$ follows from Definition 1 requiring $TT \subseteq AA$, and from the definition of AA for this theorem. □

Theorem 4. If $A \neq \emptyset$, K is not defined for P, P has at least one unreturned call, and the first unreturned call in P is contained in procedure X, then $S_1 \leftarrow \bigcup_{\text{all such } P \text{ given } X}$ {the unreturned calls of P}, and $S_2 \leftarrow \{x \mid x \in A \wedge (x$ is part of a possible execution path that inclusively begins with a call $\in T$ and ends with a call of the procedure X, such that each unreturned call in this possible execution path is in $A)\}$, $AA \leftarrow S_1 \cup S_2$, and $TT \leftarrow S_2 \cap T$.

Proof. S_1 follows from Definition 1 and the proof of Theorem 2. S_2 follows from Theorem 3, where the specific "procedure containing *dd*" in the expression for AA in Theorem 3 has been replaced by the equally specific "procedure X".

That the union operation of AA, combining S_1 and S_2, does not thereby represent spurious paths in AA, it is only necessary to show that the paths represented in S_1 never cross with the paths represented in S_2. Two paths cross if each path makes an unreturned call to the same procedure. All paths in S_2 end with an unreturned call of procedure X. All paths in S_1 begin with an unreturned call contained in procedure X. Assume that both S_1 and S_2 include an unreturned call to the same procedure. As all paths in S_2 lead to procedure X, this means there exists an execution path that originates in procedure X and eventually calls procedure X. Thus,

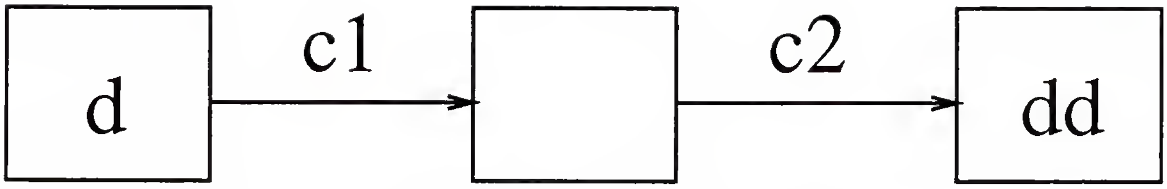

Figure 3.1. An example call structure that does not allow overestimation.

the execution path represents recursion, and this is contradicted by Assumption 3. Therefore, the paths represented in S_1 never cross with the paths represented in S_2.

The first unreturned call in P is not added to TT because the path P is an extension of the unreturned paths represented in S_2. That $TT \leftarrow S_2 \cap T$ follows from Definition 1 requiring $TT \subseteq AA$, and from the definition of AA for this theorem. □

The four theorems given above will be used to build the algorithm given in the next section. In effect, a given Allow set represents possible execution paths with unreturned calls by which the definition associated with that Allow set was affected. Inversely, the Allow set identifies, in effect, those continuation paths that can make unmatched returns. However, missing from the Allow set is the information needed to enforce an ordering of the unmatched returns that the continuation path may make. To a large extent, this missing information is unnecessary because of Lemma 1. Typically, the call structure of the program itself enforces the ordering of the unmatched returns. Figure 3.1 is an example. Assume *d* affects *dd*, giving an Allow set of {c1,c2} for *dd*. Given a continuation path from *dd*, it is not possible for c1 to be returned to before c2, so the correct ordering of unmatched returns is enforced by the program itself. However, there are cases where the missing ordering information can result in a continuation path taking unwanted shortcuts.

Figure 3.2 gives an example of a call structure that allows the continuation path from *dd* to make an unwanted shortcut when given the right circumstances. Assume *d* affects *dd* along the paths c1-c2 and c3-c4, giving an Allow set of {c1,c2,c3,c4} for *dd*. Assume the continuation path is r2-c5-r3, where r2 and r3 are unmatched returns

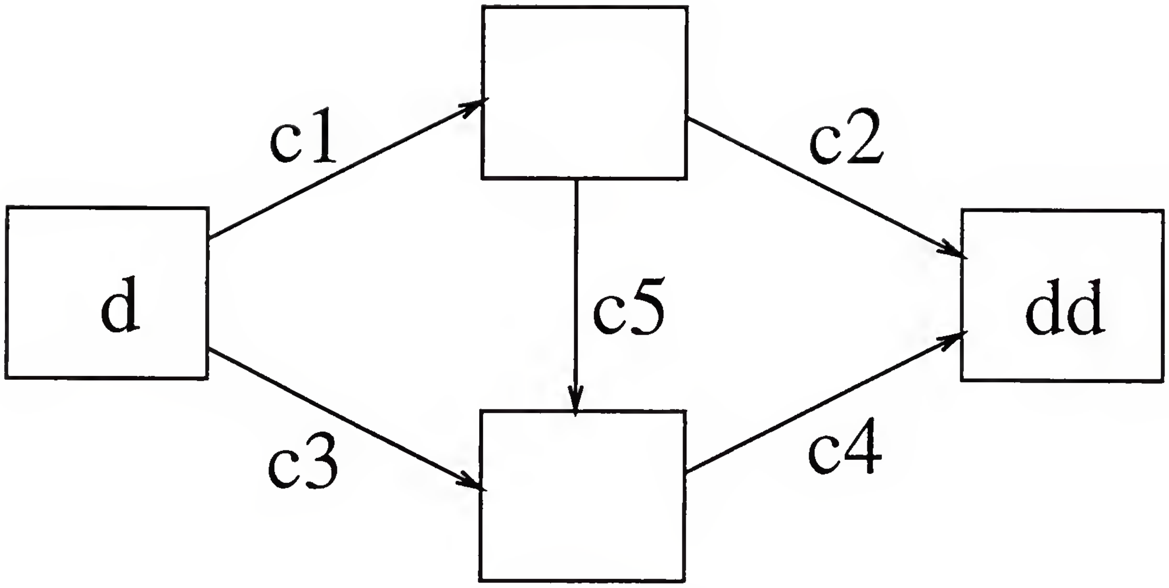

Figure 3.2. An example call structure that allows overestimation.

to calls c2 and c3. The unmatched return r3 should not be allowed to happen before an unmatched return r4, but this unmatched-return ordering will not be enforced by the Allow set defined in this dissertation, so the assumed continuation path is possible. By virtue of such a spurious continuation path, *dd* may be able to affect a definition or use that it would not otherwise be able to affect, assuming *dd* were confined to only legitimate continuation paths. In practical terms, this means that the computed logical ripple effect that consists of affected definitions and uses may in fact be an overestimate because of spurious continuation paths. Although the Allow set does permit spurious continuation paths under the right circumstances, of which Figure 3.2, and the assumed paths by which *d* affected *dd*, are the most simple example, we feel that these circumstances, along with spurious paths that affect what would otherwise be unaffected, will not occur often enough in real programs to undermine the general usefulness of the Allow set in constraining backward flow and permitting computation of a precise or semiprecise logical ripple effect.

3.2 The Logical Ripple Effect Algorithm

This section presents an algorithm for computing a precise interprocedural logical ripple effect. After a brief overview of the algorithm, the dataflow analysis method used by the algorithm is discussed. Then, two important properties of the dataflow sets are detailed, followed by three rules that are used to impose backward-flow restrictions on the dataflow analysis that is done. Last are proofs that the algorithm is correct.

The algorithm to compute logical ripple effect is shown in Figure 3.3. Each statement in the algorithm is numbered on the left. For convenience, algorithm statements will be referred to as lines. For example, a reference to line 28 means the statement at 28 that actually is printed on several lines. Comments in the algorithm begin with —. $\perp$ and $\top$ are just two different, fixed, arbitrary values.

In general, the algorithm works as follows. A definition *d* and its associated Allow and Transform sets are popped from the stack (line 7), and then the reaching-definitions dataflow problem is solved for this definition *d*, imposing any backward-flow restrictions represented by the Allow and Transform sets (line 8). Reaching definitions for a single definition is the problem of finding all uses and definitions affected by the definition. The definition *d* that was dataflow analyzed, and any uses affected by it, are included in the ripple effect (lines 9 to 11). Each affected definition will have its Allow and Transform sets determined in accordance with Theorems 1 through 4 (lines 22 to 46). A check is then made to see if the affected definition and its restriction sets, Allow and Transform, should be added to the stack for dataflow analysis or not (lines 47 to 52). The algorithm ends when the stack is empty. Although the algorithm shows a single definition *b* being added to the stack at line 5, any number of different *b* can actually be added, along with empty restriction sets for each *b*.

```
     — Compute the logical ripple effect for a hypothetical or actual definition $b$
     — Input: a program flowgraph ready for dataflow analysis
     — Output: the logical ripple effect in RIPPLE
     begin
 1     RIPPLE ← $\emptyset$
 2     for each definition $dd$ in the program
 3         $\text{FIN}_{dd}$ ← $\perp$
       end for
 4     $stack$ ← $\emptyset$
 5     push ($b$, $\emptyset$, $\emptyset$) onto $stack$
 6     while $stack \neq \emptyset$ do
 7         pop $stack$ into ($d$, ALLOW, TRANSFORM)
 8         Solve the reaching-definitions dataflow equations for the single definition $d$,
           using Rules 1, 2, and 3.
 9         RIPPLE ← RIPPLE ∪ {$d$}
10         for each use $u$ in the program that is affected by either $d_1$ or $d_2$
11             RIPPLE ← RIPPLE ∪ {$u$}
           end for
12         ROOT1 ← $\emptyset$, LINK1 ← $\emptyset$, ROOT2 ← $\emptyset$, LINK2 ← $\emptyset$
13         for each call node $n$ in the flowgraph
14             if $d_1 \in B_{out}[n]$ and $d_1$ crossed from this call into the called procedure
15                 ROOT1 ← ROOT1 ∪ {the call node $n$}
               fi
16             if $d_1 \in E_{out}[n]$ and $d_1$ crossed from this call into the called procedure
17                 LINK1 ← LINK1 ∪ {the call node $n$}
               fi
18             if $d_2 \in B_{out}[n]$ and $d_2$ crossed from this call into the called procedure
19                 ROOT2 ← ROOT2 ∪ {the call node $n$}
               fi
20             if $d_2 \in E_{out}[n]$ and $d_2$ crossed from this call into the called procedure
21                 LINK2 ← LINK2 ∪ {the call node $n$}
               fi
           end for
```

Figure 3.3. The logical ripple effect algorithm.

```
22      for each definition $dd$ in the program that is affected by either $d_1$ or $d_2$
            — determine Allow and Transform for $dd$ by Theorem 1
23          if $d_2 \in B_{in}$[node where $dd$ occurs]
24              PATHS $\leftarrow \emptyset$, TRANS $\leftarrow \emptyset$
25              call Analyze
            else
                — determine Allow and Transform for $dd$ by Theorem 2
26              if $d_2 \in E_{in}$[node where $dd$ occurs]
27                  PATHS $\leftarrow \emptyset$
28                  PATHS $\leftarrow \{x \mid x \in$ (ROOT2 $\cup$ LINK2) $\wedge$ ($x$ calls the procedure
                            that contains $dd \vee x$ calls a procedure that contains
                            a call c $\in$ (PATHS $\cap$ LINK2))$\}$
29                  TRANS $\leftarrow$ ROOT2 $\cap$ PATHS
30                  call Analyze
                fi
                — determine Allow and Transform for $dd$ by Theorem 3
31              if $d_1 \in B_{in}$[node where $dd$ occurs]
32                  PATHS $\leftarrow \emptyset$
33                  PATHS $\leftarrow \{x \mid x \in$ ALLOW $\wedge$ ($x$ calls the procedure that contains $dd$
                            $\vee\ x$ calls a procedure that contains a call $c \in$ PATHS)$\}$
34                  TRANS $\leftarrow$ TRANSFORM $\cap$ PATHS
35                  call Analyze
                fi
                — determine Allow and Transform for $dd$ by Theorem 4
36              if $d_1 \in E_{in}$[node where $dd$ occurs]
37                  for each procedure $X$ that contains a call $\in$ ROOT1
38                      RT1 $\leftarrow \{x \mid x \in$ ROOT1 $\wedge$ $x$ is contained in procedure $X\}$
39                      PP $\leftarrow \emptyset$
40                      PP $\leftarrow \{x \mid x \in$ (RT1 $\cup$ LINK1) $\wedge$ ($x$ is on a path that inclusively
                                begins with a call $\in$ RT1 and ends with a call of the
                                procedure that contains $dd$, such that each call
                                in this path is in (RT1 $\cup$ LINK1))$\}$
41                      if PP $\neq \emptyset$
42                          PATHS $\leftarrow \emptyset$
43                          PATHS $\leftarrow \{x \mid x \in$ ALLOW $\wedge$ ($x$ calls procedure $X \vee x$ calls
                                    a procedure that contains a call c $\in$ PATHS)$\}$
44                          TRANS $\leftarrow$ TRANSFORM $\cap$ PATHS
45                          PATHS $\leftarrow$ PATHS $\cup$ PP
46                          call Analyze
    end statements: fi, end for, fi, fi, end for, od
end
```

Figure 3.3. – continued.

```
    Procedure Analyze
    begin
      — avoid repetition of dd dataflow analysis if possible
47    if $FIN_{dd} \neq \top \wedge$ (PATHS $= \emptyset$
                          $\vee$ (true for all saved pairs for dd: PATHS $\not\subseteq$ P $\vee$ TRANS $\not\subseteq$ T))
48      if PATHS $= \emptyset$
49        $FIN_{dd} \leftarrow \top$
50        push (dd, $\emptyset$, $\emptyset$) onto stack
        else
51        save PATHS and TRANS as the pair P $\times$ T for dd
52        push (dd, PATHS, TRANS) onto stack
        fi
      fi
    end
```

Figure 3.3. – continued.

The dataflow equations referred to in line 8 are shown in Figure 3.4. These equations are copied from Chapter 2 that presents a method for context-dependent flow-sensitive interprocedural dataflow analysis. The method consists of solving—using the standard iterative algorithm—the dataflow equations shown in Figure 3.4, for the program flowgraph required by the equations. The method in Chapter 2 includes a solution to the problems of parameter aliasing and implicit definitions, that are part of the interprocedural reaching-definition problem. We assume that the full method of Chapter 2 would be used, but we do not discuss these side issues in this chapter as they are not directly relevant to the algorithm. Note that there are other methods for context-dependent flow-sensitive interprocedural dataflow analysis [3, 9, 17, 21], but the method of Chapter 2 has precision and efficiency advantages over the other methods cited.

Referring to the dataflow equations of Figure 3.4, four sets are computed for each flowgraph node: two body sets, B_{in} and B_{out}, and two entry sets, E_{in} and E_{out}. All body and entry sets are initially empty. As the equations will be solved for only a single definition d, the GEN set for the node where d occurs—i.e. the node whose

For any node n.

$$IN[n] = E_{in}[n] \cup B_{in}[n]$$

$$OUT[n] = E_{out}[n] \cup B_{out}[n]$$

Group I: n is an entry node.

$$B_{in}[n] = \emptyset$$

$$E_{in}[n] = \bigcup_{p \in pred(n)} \{x \mid x \in OUT[p] \wedge C_1\}$$

$$B_{out}[n] = GEN[n]$$

$$E_{out}[n] = E_{in}[n] \cup RECODE[n]$$

Group II: n is a return node, p is the associated call node and q is the exit node of the called procedure.

$$B_{in}[n] = \{x \mid (x \in B_{out}[p] \wedge (\overline{C_1} \vee (C_1 \wedge C_2 \wedge x \in E_{out}[q]))) \vee (x \in B_{out}[q] \wedge C_2)\}$$

$$E_{in}[n] = \{x \in E_{out}[p] \mid \overline{C_1} \vee (C_1 \wedge C_2 \wedge x \in E_{out}[q])\}$$

$$B_{out}[n] = (B_{in}[n] - KILL[n]) \cup GEN[n]$$

$$E_{out}[n] = E_{in}[n] - KILL[n]$$

Group III: n is not an entry or return node.

$$B_{in}[n] = \bigcup_{p \in pred(n)} B_{out}[p]$$

$$E_{in}[n] = \bigcup_{p \in pred(n)} E_{out}[p]$$

$$B_{out}[n] = (B_{in}[n] - KILL[n]) \cup GEN[n]$$

$$E_{out}[n] = E_{in}[n] - KILL[n]$$

Figure 3.4. Dataflow equations for the reaching-definitions problem.

associated block of program code contains the definition d—will contain an element representing d, and all the other GEN sets will be empty. The node where d occurs is the natural starting point for the iterative algorithm, that will recompute the body and entry sets for the nodes until stability is attained and the sets cease to change, at which point the equations have been solved. Once solved, an element is in the entry set or body set at a particular node depending on how that element was propagated to that node. The same element may be in both sets at the same node. Properties 1 and 2 listed below, summarize those implications of set membership that are used by the algorithm. The properties follow directly from the dataflow equations.

Property 1. For any node n, an element is in the $E_{in}[n]$ set or $E_{out}[n]$ set if and only if that element entered the procedure that contains node n from a call node, and there is a definition-clear path from that call node to node n. Thus, membership in the entry set of node n implies that the element can propagate to node n by an execution path that makes at least one unreturned call between the point where the element is generated and the point where node n occurs.

Property 2. For any node n, an element is in the $B_{in}[n]$ set or $B_{out}[n]$ set if and only if that element was generated in the same procedure that contains node n, or that element entered the procedure that contains node n from an exit-node B_{out} set. There must also be a definition-clear path to node n from either the element's generation node or from the exit node. If the element entered from an exit-node B_{out} set, then Property 2 applies recursively to the element in that B_{out} set. Thus, membership in the body set of node n implies that the element can propagate to node n by an execution path between the point where the element is generated and the point where node n occurs that does not include any unreturned calls.

The three rules referred to in line 8 are listed below. Rule 1 applies before the dataflow equations are solved. Rules 2 and 3 apply as the equations are being solved. The rules impose the backward-flow restrictions represented by the ALLOW and TRANSFORM sets in line 7.

Rule 1. If ALLOW $= \emptyset$ then element d_2 is generated at the node where definition d occurs, otherwise d_1 is the generated element, meaning the element in the GEN set. Both d_1 and d_2 are base elements that represent the same definition d. Both elements are identical in terms of when they appear in any given $KILL$ set. The only difference between them is that d_1 and d_2 are treated differently by Rules 2 and 3 below.

If the ALLOW set is empty, then by Definition 1 there should be no backward-flow restrictions on d. Rule 1 accomplishes this requirement, as d_2 is immune to backward-flow restrictions which are imposed by Rule 2.

Rule 2. Let n be a return node, p be the associated call node, and q be the exit node of the called procedure. Each time the $B_{in}[n]$ equation is computed, if $d_1 \in B_{out}[q]$, then d_1 cannot cross from $B_{out}[q]$ into the $B_{in}[n]$ set if $p \notin$ ALLOW.

In the dataflow equations, the crossing of an element from an exit-node body set to a return node is the only action in the equations that represents, in effect, an unmatched return to a call instance that was made in an execution path leading up to the program point where definition d occurs, which is the starting point of the reaching-definition analysis done for d. Thus, Rule 2 covers all cases in which an unmatched return occurs. Rule 2 restricts unmatched returns to those call instances that are represented in the ALLOW set, thereby realizing the purpose of the ALLOW set as given by Definition 1.

Rule 3. Let n be a return node, p be the associated call node, and q be the exit node of the called procedure. Each time the $B_{in}[n]$ equation is computed, if $d_1 \in B_{out}[q]$, and, by C_2 and Rule 2, d_1 can cross from $B_{out}[q]$ into the $B_{in}[n]$ set, and $p \in$ TRANSFORM, then as this d_1 element crosses from $B_{out}[q]$ into the $B_{in}[n]$

set, the element is changed to d_2. In effect, d_1 is transformed into d_2, and the return node n becomes a generation node for the d_2 element.

As already mentioned, in the dataflow equations the crossing of an element from an exit-node body set to a return node is the only action in the equations that represents, in effect, an unmatched return to a call instance that was made in an execution path leading up to the program point where definition d occurs, which is the starting point of the reaching-definition analysis done for d. Thus, Rule 3 covers all cases in which an unmatched return occurs. The requirement by Rule 3 that the returned-to call be in the TRANSFORM set satisfies Definition 1 as to when backward-flow restrictions can be ignored. Rule 3 replaces element d_1, which is subject to the backward-flow restrictions, with element d_2, which is free of backward-flow restrictions, at the return point and thereby satisfies Definition 1 regarding removal of backward-flow restrictions on the execution-path continuation, since d_2 now represents the continuation instead of d_1.

Lemma 2. The algorithm computes at lines 23 to 46 the restriction sets for an affected definition in accordance with Theorems 1 through 4.

Proof. We first establish the properties of the LINK and ROOT sets computed at lines 12 to 21. Let p be the node, if any, where d_1 is generated. Let q be any node where d_2 is generated, i.e. those return nodes where d_1 is transformed into d_2, or for ALLOW $= \emptyset$ the node where definition d occurs.

The tests at lines 14 and 18 make use of Property 2: if an element is in the B_{out} set of a call node n, then there exists a definition-clear path between the node where the element is generated and node n, and the path has no unreturned calls. The call at node n would be the first unreturned call on that path by just extending the path to the entry node of the called procedure. Therefore, the ROOT1 set represents all calls that are the first unreturned call on at least one definition-clear path between node p and some other node in the flowgraph. The ROOT2 set represents all calls that are the first unreturned call on at least one definition-clear path between node q and some other node in the flowgraph.

The tests of lines 16 and 20 make use of Property 1: if an element is in the E_{out} set of a call node n, then there exists a definition-clear path between the node where the element is generated and node n, and the path includes the unreturned call that called the procedure containing node n. The call at node n would be at least the second unreturned call on that path by just extending the path to the entry node of the called procedure. Therefore, the LINK1 set represents all calls that are an unreturned call but not the first unreturned call on at least one definition-clear path between node p and some other node in the flowgraph. The LINK2 set represents all calls that are an unreturned call but not the first unreturned call on at least one definition-clear path between node q and some other node in the flowgraph.

The test at line 23 checks for the application of Theorem 1. If $d_2 \in B_{in}$[node where dd occurs], then by Property 2 there exists a definition-clear path P between d and dd that has no unreturned calls, and somewhere along P, d_2 is generated, meaning either ALLOW $= \emptyset$ or K is defined for P. This satisfies the conditions of Theorem 1, and line 24 sets PATHS and TRANS to empty in accordance with the theorem. PATHS and TRANS are the Allow and Transform sets for dd.

The test at line 26 checks for the application of Theorem 2. If $d_2 \in E_{in}$[node where dd occurs], then by Property 1 there exists at least one definition-clear path P between d and dd that has at least one unreturned call, and somewhere along P, d_2 is generated, meaning either ALLOW $= \emptyset$ or K is defined for P. This satisfies the conditions of Theorem 2. Only the d_2 element satisfies the theorem, so it follows that all paths P for the theorem will have to be constructed from the ROOT2 and LINK2 sets exclusively.

Referring to Theorem 2, line 28 computes the AA set, and line 29 computes TT. For line 28, the PATHS set is defined in terms of itself. This recursive reference means that each time a call is added to the PATHS set, the condition containing the recursive reference must be reevaluated, because additional calls may thereby be added to PATHS. Recursive references are similarly used in lines 33 and 43. What line 28 does is extract from all the calls that element d_2 crossed, just those calls that are on a path to *dd*. This is done by building the paths backwards, beginning with those calls that call the procedure containing *dd*. By Lemma 1, any path between d and *dd* consisting of unreturned calls can be found by proceeding in reverse order from *dd* and selecting those calls that call a procedure containing a call already selected. Backward path building and Lemma 1 are similarly used in lines 33 and 43. By the properties of the ROOT and LINK sets, the paths constructed by line 28 will be definition-clear. Notice that a particular call may be in both the ROOT2 and LINK2 sets, but if a call is only in the ROOT2 set, then it cannot be used as the basis for extending further backwards any path, because by Property 1, d_2 does not propagate from the entry node of the procedure that contains that call, to the call node for that call. This is the reason for the (PATHS $\cap$ LINK2) requirement in line 28. Once the PATHS set is computed, line 29 computes TRANS in accordance with the theorem.

The test at line 31 checks for the application of Theorem 3. If $d_1 \in B_{in}$[node where *dd* occurs], then by Property 2 there exists a definition-clear path P between d and *dd* that has no unreturned calls. It also follows that ALLOW $\neq \emptyset$ and P does not make an unmatched return to a call $\in$ TRANSFORM, because d_1 is the element, meaning K is not defined for P. This satisfies the conditions of Theorem 3. Referring to Theorem 3, line 33 computes the AA set, and line 34 computes TT. What line 33 does is extract from ALLOW all paths that end with a call of the procedure containing *dd*. Although Theorem 3 states that the path begin with a call $\in$ TRANSFORM, line 33 does not require a check for this because TRANSFORM is a subset of ALLOW and those first unreturned calls in TRANSFORM that are on a path in ALLOW to *dd*, will unavoidably be picked up as the paths are built backwards from dd. Thus, the PATHS set is computed in accordance with Theorem 3, followed by line 34 that computes the TRANS set in accordance with the theorem.

The test at line 36 checks for the application of Theorem 4. If $d_1 \in E_{in}$[node where dd occurs], then by Property 1 there exists at least one definition-clear path P between d and dd that has at least one unreturned call. It also follows that ALLOW $\neq \emptyset$ and P does not make an unmatched return to a call $\in$ TRANSFORM, because d_1 is the element. This satisfies the conditions of the theorem. Only the d_1 element satisfies the theorem, so it follows that all paths P for the theorem will have to be constructed from the ROOT1 and LINK1 sets exclusively. Referring to Theorem 4, line 40 computes the S_1 set, line 43 computes the S_2 set, line 44 computes the TT set, and line 45 computes the AA set. The reason for the test at line 41 is that although there exists at least one path P satisfying the theorem, there may not be any paths P that begin in the specific procedure X. It can be seen that lines 37 to 46 compute in accordance with the theorem. □

Lemma 3. Let A_i and T_i be one pair of Allow and Transform sets associated with a definition d, and let A_j and T_j be a different pair of Allow and Transform sets associated with the same definition d. Assume $A_i \neq \emptyset$ and $A_j \neq \emptyset$. If $A_j \subseteq A_i$ and $T_j \subseteq T_i$, then dataflow analyzing d with the pair A_j and T_j cannot add anything to the ripple effect that is not added by dataflow analyzing d with the pair A_i and T_i.

Proof. By inspection of Rules 1, 2, and 3, it can be seen that removing some of the calls from A_i or T_i cannot make d affect anything that it does not affect with A_i and T_i as they were. Also, by inspection of lines 23 to 46, the determination of the Allow and Transform sets for any definition dd affected by d, cannot be made to include calls when A_j and T_j are the restriction sets for d, that would not be included when A_i and T_i are the restriction sets for d. □

Lemma 4. Let A and T be Allow and Transform sets associated with a definition d, and let X and Y be a different pair of Allow and Transform sets associated with the same definition d. If $A = \emptyset$, then dataflow analyzing d with X and Y cannot add anything to the ripple effect that is not added by dataflow analyzing d with A and T.

Proof. By Rule 1, d will be represented by d_2 and have no restrictions on its backward flow. Thus, d will affect everything that it is possible for it to affect. If d is dataflow analyzed with X and Y, then any calls found in the ROOT1, ROOT2, LINK1, or LINK2 sets will also be found in the ROOT2 or LINK2 sets when d is dataflow analyzed with A and T. These sets determine the restriction sets associated with a definition dd affected by d. It follows that any dataflow path allowed for a dd affected by d using X and Y, will also be allowed for a dd affected by d using A and T. $\square$

Theorem 5. Given Definition 1 and Theorems 1 through 4, the algorithm will correctly compute the logical ripple effect.

Proof. As shown by Lemma 2, for any affected definition dd, the Allow and Transform sets to be associated with dd are computed in accordance with Theorems 1 to 4. By Lemma 4, if Theorem 1 applies to an affected definition (line 23), then there is no need to check if any other theorem also applies, because additional dataflow analysis resulting from the other theorems cannot contribute to the ripple effect. However, if Theorem 1 does not apply, then the definition must be dataflow analyzed separately in turn for each theorem that does apply. This is done by the sequence of three if statements at lines 26, 31, and 36. Thus, the control logic in lines 23 to 46 is safe.

The Analyze procedure (lines 47 to 52) prepares a definition and its restriction sets for dataflow analysis by adding them to the stack (line 50 and 52). Once a definition will be dataflow analyzed with no restrictions (line 50) it will not be analyzed again (line 47). By Lemma 4, this is safe. Assuming $\text{FIN}_{dd} \neq \top$ and PATHS $\neq \emptyset$, the test at line 47 will not prepare a definition for dataflow analysis if both restriction sets are subsets of any pair of restriction sets used previously to analyze that definition. This follows from Lemma 3. Thus, the Analyze procedure is safe.

The correctness of the dataflow equations (line 8) is established in Chapter 2, and the correctness of the three rules for imposing backward-flow restrictions (line 8) has already been discussed. Regarding the correctness of having no backward-flow restrictions for the initial definition (line 5), let p be the program point where b occurs. For execution to attain point p, any possible execution path between the program's execution starting point and point p can be assumed to have occurred. Thus, there should be no restrictions on the backward-flow possibilities of b, because there were no constraints imposed by the ripple effect on how point p was initially attained. □

Programs with recursive calls can be processed by our algorithm, but there may be some overestimation of the logical ripple effect because of the recursive calls. The dataflow equations (line 8) are not the problem, as they work for recursive programs. Instead, the problem is with the Allow set and its representation of execution paths. If a cyclic execution path is represented in the Allow set, then when the Allow set is used to restrict backward flow by Rule 2, it may be possible for an element moving through the program flowgraph to take a shortcut on its unmatched returns and avoid having to make unmatched returns along the complete cycle before a program point can be attained. This shortcut may permit the element to affect something that it should not be able to affect, possibly adding to the ripple effect beyond what should be there.

3.3 A Prototype Demonstrates the Algorithm

This section first considers the complexity of our interprocedural logical ripple effect algorithm. A prototype that demonstrates the algorithm is then described, and test results presented.

Let n be the number of nodes in the flowgraph of the input program. For a programming language such as C, solving the dataflow equations for a single definition, which is what line 8 does, has worst-case complexity of $O(n)$. Let k be the number of known calls in the input program. Considering line 47, a definition may be dataflow analyzed repeatedly as long as the associated restriction sets are not subsets of any previous pair of restriction sets used to dataflow analyze that definition. The number of different restriction sets possible such that no set is a subset of another set, is clearly a number that will grow exponentially with k. Thus, the worst-case complexity of our logical ripple effect algorithm is exponential, where the exponent is some function of k. However, for the typical input program, the actual number of non-subset restriction sets that can be generated by our algorithm for a given definition, will be severely constrained by a combination of Lemma 1, Theorems 1 through 4, and the typical program call structure that is characterized by shallow call depth.

A prototype that demonstrates our logical ripple effect algorithm has been built. The prototype accepts as input C programs that satisfy certain constraints, such as having only single-identifier variable names. Given an input program, the prototype then requires that one or more definitions be identified as the starting point of the ripple effect. For purposes of comparison, besides using our algorithm to compute a precise logical ripple effect, the prototype also computes an overestimate of the logical ripple effect. The overestimate is computed by simply ignoring the execution-path problem, i.e. there are no backward-flow restrictions when the overestimate is computed. The worst-case complexity of computing the overestimate for C programs is only $O(nd)$ where n is the number of flowgraph nodes and d is the number of definitions in the overestimated ripple effect. This complexity follows from the $O(n)$ complexity of solving the dataflow equations for a single definition, and the fact that the equations will have to be solved d times.

Table 3.1. Experimental results for the prototype.

globals	defs	defs global	depth	nodes	RS_o	RS_p	reduction	$time_o$	$time_p$
50	2420	7%	2/213	3939	2275	936	53.4%	5s	3s
100	2291	15%	2/188	3776	4151	2449	41.0%	17s	13s
200	2294	30%	2/188	3662	5594	3718	33.5%	40s	32s
300	2370	45%	2/231	3962	5897	2607	55.8%	1m5s	27s
50	2225	7%	3/202	3717	1222	633	40.3%	3s	2s
100	2333	15%	3/229	3864	4139	1867	54.9%	17s	7s
200	2211	30%	3/231	3760	4884	2688	45.0%	39s	28s
300	2236	45%	3/205	3737	5308	3505	34.0%	59s	38s
50	2320	7%	4/227	3912	1822	1067	35.1%	5s	3s
100	2211	15%	4/228	3673	4329	1525	64.8%	18s	7s
200	2223	30%	4/227	3705	5019	1918	61.8%	37s	16s
300	2214	45%	4/214	3648	5922	4740	20.0%	1m9s	1m36s
100	4354	7%	2/372	6858	4317	2201	40.0%	19s	10s
200	4467	15%	2/368	7068	8844	6457	27.0%	1m17s	1m12s
400	4261	30%	2/388	6851	9653	2976	69.2%	2m29s	49s
600	4289	45%	2/340	6784	10590	6840	35.4%	4m8s	3m56s
100	4314	7%	3/432	6781	1993	631	52.5%	8s	2s
200	4268	15%	3/395	6876	5795	3236	35.5%	51s	54s
400	4223	30%	3/393	6735	9240	7307	20.9%	2m26s	4m21s
600	4248	45%	3/433	6868	9772	6453	30.6%	3m56s	4m50s
100	4252	7%	4/455	6961	2756	1120	42.6%	14s	5s
200	4276	15%	4/440	6858	7781	5752	26.1%	1m10s	2m35s
400	4228	30%	4/391	6681	9838	8290	15.7%	2m45s	9m20s
600	4112	45%	4/462	6802	10017	9192	8.2%	4m24s	39m55s

Table 3.1 presents test results for the prototype. Each row details relevant characteristics of an input program, and presents the resulting averages of ten different tests of that input program, where each test computed the ripple effect started by a single, randomly chosen definition of a global variable.

The input programs of Table 3.1 were randomly generated by a separate program generator. The generated input programs are syntactically correct and compile without error, but have meaningless executions. Each input program of Table 3.1 has 100 procedures, and exactly the number of global variables listed. Within each input program, each global variable is defined and used at least once. The call structure of each input program was determined randomly by the generator, with the constraint that there be no recursion in the input program, and the given maximum call depth not be exceeded by any call in the input program. All calls in the generated input program are known calls, and approximately $1/(\max + 1)$ of the calls will be at each possible depth from zero to max, where max is the given maximum call depth.

Referring to the columns of Table 3.1, "globals" is the number of global variables in the input program, "defs" is the number of definitions in the input program, "defs global" is the percentage of the definitions that define a global variable, "depth" is the maximum call depth followed by the total number of calls in the input program, "nodes" is the number of nodes in the flowgraph, "RS_o" is the average size of the overestimated ripple effect for the ten test cases where size is the total number of definitions and uses in the ripple effect, "RS_p" is the average size of the precise ripple effect, "reduction" is the average percentage reduction for the ten test cases of the size of the overestimated ripple effect when it is replaced by the precise ripple effect, "time_o" is the average CPU usage time for each test case to compute the overestimated ripple effect, and "time_p" is the average CPU usage time for each test case to compute the precise ripple effect. The hardware used was rated at roughly 24 MIPS. As an example of the time notation used in Table 3.1, time 1m36s would be read as 1 minute, 36 seconds.

Although the worst-case complexity of our algorithm for precise logical ripple effect is exponential, the data of Table 3.1 indicates that the expected complexity for a wide range of input programs, given a programming language such as C, is approximated by $O(nd)$. This follows from the $O(nd)$ worst-case complexity of computing the overestimate, and the typical closeness of time_o and time_p for each row in Table 3.1. However, the last row of Table 3.1 is instructive, because it shows that regardless of what the expected complexity might be, there will always be specific input programs and starting points that require time greatly exceeding the time required to compute the overestimate. In practice, if the computation of the precise logical ripple effect is taking too long, then this computation can be abandoned and the overestimate computed and used in its place. Note that our algorithm can very easily compute the overestimate by simply modifying Rule 1 so that element d_2 is always generated in place of element d_1, thereby avoiding all backward-flow restrictions.

3.4 The Slicing Algorithm

This section presents the inverse form of the precise interprocedural logical ripple effect algorithm, and the inverse form of the associated dataflow equations and backward-flow restriction rules. Our algorithm for precise interprocedural slicing is shown in Figure 3.5. The complexity and expected performance of this algorithm is the same as for the precise interprocedural logical ripple effect algorithm given previously.

For logical ripple effect, the dataflow problem solved at line 8 was reaching definitions for a single definition. For slicing, which is the inverse problem, the dataflow problem solved at line 8 will be reaching uses for a single use. In reaching definitions, the definition flows in the direction of the arcs in the flowgraph, and is killed by definitions of the same variable, and affects uses of the same variable and any definitions directly dependent on an affected use. In reaching uses, the use flows in the reverse direction of the arcs in the flowgraph, and is killed by definitions of the same variable, and affects definitions of the same variable and any uses that directly determine an affected definition. This reverse flow in the flowgraph means that the dataflow equations solved at line 8 for the slicing algorithm must be an inverted form of the dataflow equations that are used for the logical ripple effect algorithm. These inverted dataflow equations are shown in Figure 3.6. The inverted rules that the slicing algorithm uses for backward-flow restriction are given below. Notice that the ALLOW and TRANSFORM sets will contain returns instead of calls.

```
     — Compute the slice for a hypothetical or actual use $b$
     — Input: a program flowgraph ready for dataflow analysis
     — Output: the slice in SLICE
     begin
 1     SLICE $\leftarrow \emptyset$
 2     for each use $uu$ in the program
 3         $\text{FIN}_{uu} \leftarrow \perp$
       end for
 4     $stack \leftarrow \emptyset$
 5     push $(b, \emptyset, \emptyset)$ onto $stack$
 6     while $stack \neq \emptyset$ do
 7         pop $stack$ into ($u$, ALLOW, TRANSFORM)
 8         Solve the reaching-uses dataflow equations for the single use $u$,
           using Rules 1, 2, and 3.
 9         SLICE $\leftarrow$ SLICE $\cup \{u\}$
10         for each definition $d$ in the program that is affected by either $u_1$ or $u_2$
11             SLICE $\leftarrow$ SLICE $\cup \{d\}$
           end for
12         ROOT1 $\leftarrow \emptyset$, LINK1 $\leftarrow \emptyset$, ROOT2 $\leftarrow \emptyset$, LINK2 $\leftarrow \emptyset$
13         for each return node $n$ in the flowgraph
14             if $u_1 \in B_{in}[n] \wedge u_1$ crossed from this return into the returned-from procedure
15                 ROOT1 $\leftarrow$ ROOT1 $\cup$ {the return node $n$}
               fi
16             if $u_1 \in E_{in}[n] \wedge u_1$ crossed from this return into the returned-from procedure
17                 LINK1 $\leftarrow$ LINK1 $\cup$ {the return node $n$}
               fi
18             if $u_2 \in B_{in}[n] \wedge u_2$ crossed from this return into the returned-from procedure
19                 ROOT2 $\leftarrow$ ROOT2 $\cup$ {the return node $n$}
               fi
20             if $u_2 \in E_{in}[n] \wedge u_2$ crossed from this return into the returned-from procedure
21                 LINK2 $\leftarrow$ LINK2 $\cup$ {the return node $n$}
               fi
           end for
```

Figure 3.5. The slicing algorithm.

```
22      for each use $uu$ in the program that is affected by either $u_1$ or $u_2$
           — determine Allow and Transform for $uu$ by Theorem 1
23         if $u_2 \in B_{out}$[node where $uu$ occurs]
24            PATHS $\leftarrow \emptyset$, TRANS $\leftarrow \emptyset$
25            call Analyze
           else
              — determine Allow and Transform for $uu$ by Theorem 2
26            if $u_2 \in E_{out}$[node where $uu$ occurs]
27               PATHS $\leftarrow \emptyset$
28               PATHS $\leftarrow \{x \mid x \in$ (ROOT2 $\cup$ LINK2) $\wedge$ ($x$ returns from the
                          procedure that contains $uu$ $\vee$ $x$ returns from a procedure
                          that contains a return $r \in$ (PATHS $\cap$ LINK2))$\}$
29               TRANS $\leftarrow$ ROOT2 $\cap$ PATHS
30               call Analyze
              fi
              — determine Allow and Transform for $uu$ by Theorem 3
31            if $u_1 \in B_{out}$[node where $uu$ occurs]
32               PATHS $\leftarrow \emptyset$
33               PATHS $\leftarrow \{x \mid x \in$ ALLOW $\wedge$ ($x$ returns from the procedure that
                          contains $uu$ $\vee$ $x$ returns from a procedure that contains
                          a return $r \in$ PATHS)$\}$
34               TRANS $\leftarrow$ TRANSFORM $\cap$ PATHS
35               call Analyze
              fi
              — determine Allow and Transform for $uu$ by Theorem 4
36            if $u_1 \in E_{out}$[node where $uu$ occurs]
37               for each procedure $X$ that contains a return $\in$ ROOT1
38                  RT1 $\leftarrow \{x \mid x \in$ ROOT1 $\wedge$ $x$ is contained in procedure $X\}$
39                  PP $\leftarrow \emptyset$
40                  PP $\leftarrow \{x \mid x \in$ (RT1 $\cup$ LINK1) $\wedge$ ($x$ is on a path that inclusively
                          begins with a return $\in$ RT1 and ends with a return from
                          the procedure that contains $uu$, such that each return
                          in this path is in (RT1 $\cup$ LINK1))$\}$
41                  if PP $\neq \emptyset$
42                     PATHS $\leftarrow \emptyset$
43                     PATHS $\leftarrow \{x \mid x \in$ ALLOW $\wedge$ ($x$ returns from procedure $X$
                                $\vee$ $x$ returns from a procedure that contains
                                a return $r \in$ PATHS)$\}$
44                     TRANS $\leftarrow$ TRANSFORM $\cap$ PATHS
45                     PATHS $\leftarrow$ PATHS $\cup$ PP
46                     call Analyze
   end statements: fi, end for, fi, fi, end for, od
end
```

Figure 3.5. – continued.

```
   Procedure Analyze
   begin
     — avoid repetition of $uu$ dataflow analysis if possible
47   if $\text{FIN}_{uu} \neq \top \wedge (\text{PATHS} = \emptyset$
                     $\vee$ (true for all saved pairs for $uu$: $\text{PATHS} \not\subseteq \text{P} \vee \text{TRANS} \not\subseteq \text{T}$))
48      if $\text{PATHS} = \emptyset$
49         $\text{FIN}_{uu} \leftarrow \top$
50         push $(uu, \emptyset, \emptyset)$ onto *stack*
        else
51         save PATHS and TRANS as the pair $\text{P} \times \text{T}$ for $uu$
52         push ($uu$, PATHS, TRANS) onto *stack*
        fi
     fi
   end
```

Figure 3.5. – continued.

Rule 1. If ALLOW $= \emptyset$ then element u_2 is generated at the node where use u occurs, otherwise u_1 is the generated element.

Rule 2. Let n be a call node, p be the associated return node, and q be the entry node of the returned-from procedure. Each time the $B_{out}[n]$ equation is computed, if $u_1 \in B_{in}[q]$, then u_1 cannot cross from $B_{in}[q]$ into the $B_{out}[n]$ set if $p \notin$ ALLOW.

Rule 3. Let n be a call node, p be the associated return node, and q be the entry node of the returned-from procedure. Each time the $B_{out}[n]$ equation is computed, if $u_1 \in B_{in}[q]$, and, by C_2 and Rule 2, u_1 can cross from $B_{in}[q]$ into the $B_{out}[n]$ set, and $p \in$ TRANSFORM, then as this u_1 element crosses from $B_{in}[q]$ into the $B_{out}[n]$ set, the element is changed to u_2. In effect, u_1 is transformed into u_2, and the call node n becomes a generation node for the u_2 element.

As the usefulness of slicing is primarily for program fault localization, it may be desirable to modify the algorithm so that those uses in control predicates whose subordinate statements have at least one use or definition already in the slice, are themselves added to the slice and propagated in turn. An example of a control predicate is the condition tested by an if statement. By subordinate statements is meant

For any node n.

$$OUT[n] = E_{out}[n] \cup B_{out}[n]$$

$$IN[n] = E_{in}[n] \cup B_{in}[n]$$

Group I: n is an exit node.

$$B_{out}[n] = \emptyset$$

$$E_{out}[n] = \bigcup_{p \in succ(n)} \{x \mid x \in IN[p] \wedge C_1\}$$

$$B_{in}[n] = GEN[n]$$

$$E_{in}[n] = E_{out}[n] \cup RECODE[n]$$

Group II: n is a call node, p is the associated return node and q is the entry node of the returned-from procedure.

$$B_{out}[n] = \{x \mid (x \in B_{in}[p] \wedge (\overline{C_1} \vee (C_1 \wedge C_2 \wedge x \in E_{in}[q]))) \vee (x \in B_{in}[q] \wedge C_2)\}$$

$$E_{out}[n] = \{x \in E_{in}[p] \mid \overline{C_1} \vee (C_1 \wedge C_2 \wedge x \in E_{in}[q])\}$$

$$B_{in}[n] = (B_{out}[n] - KILL[n]) \cup GEN[n]$$

$$E_{in}[n] = E_{out}[n] - KILL[n]$$

Group III: n is not an exit or call node.

$$B_{out}[n] = \bigcup_{p \in succ(n)} B_{in}[p]$$

$$E_{out}[n] = \bigcup_{p \in succ(n)} E_{in}[p]$$

$$B_{in}[n] = (B_{out}[n] - KILL[n]) \cup GEN[n]$$

$$E_{in}[n] = E_{out}[n] - KILL[n]$$

Figure 3.6. Dataflow equations for the reaching-uses problem.

those statements whose execution is decided by the control predicate. Including these control-predicate uses in the slice is advantageous because the cause of a program error may actually be in a control predicate that is not deciding correctly when to execute its subordinate statements. Ferrante et al. [8] present a method to precisely determine the control predicates for each statement.

CHAPTER 4
INTERPROCEDURAL PARALLELIZATION

4.1 Loop-Carried Data Dependence

This section explains loop-carried data dependence and its relevance to parallelization. When a definition of a variable reaches a use of that variable, then a *data dependence* exists such that the use depends on the definition. An example of data dependence can be seen in Figure 4.1. The use of A(I) at line 3, and the use of A(I) at line 4, both depend on the definition of A(I) at line 2. However, when considering whether or not a loop can be parallelized, there is a special kind of data dependence called *loop-carried data dependence* [25]. A data dependence is loop carried if the value set by a definition inside the loop during loop iteration i can be used by a use of that variable inside the loop during loop iteration j, where $i \neq j$. Note that $i \neq j$ is specified instead of the more restrictive and natural seeming $i < j$, because if the loop is parallelized then the ordering of the loop iterations cannot be assumed.

The relationship between loop-carried data dependence and parallelization is straightforward. If there is at least one loop-carried data dependence, then the loop cannot be parallelized, otherwise the loop can be parallelized. Loop parallelization

```
1   DO I = 1,N
2       A(I) = B(I) * C(I) + D
3       B(I) = C(I) / D + A(I)
4       IF C(I) < 0 THEN C(I) = A(I) * B(I) FI
    END DO
```

Figure 4.1. An example loop.

would mean that the ordering of the different iterations of the loop is unimportant, whereas a loop-carried dependence means the opposite. If there are no loop-carried data dependencies then there is no requirement that the iterations be ordered a certain way. However, whenever a loop is parallelized, there should be a following, added, serial step that sets the iteration variables, such as the I in Figure 4.1, to whatever their values would be for the last iteration of the loop, assuming the loop had not been parallelized. This added step would be necessary, assuming the iteration variables of a loop are visible outside the loop and can therefore be referenced after the loop completes. *Iteration variables* are those variables that are incremented or decremented a constant value for each loop iteration. The recognition of iteration variables is language-dependent.

Regarding data dependence and arrays, there are several efficient tests available that determine if a data dependence is possible between a particular definition and use of an array. The tests are the separability test, the gcd test, and the Banerjee test. Details of these three tests can be found in [25]. The number theory behind the tests is linear diophantine equations. A linear diophantine equation can be formed from the array subscripts of the definition and use in question. For example, in Figure 4.2 we want to know if A(3 * I - 5) and A(6 * I) can ever refer to the same array element. The linear diophantine equation that relates these two array references would be $3x - 6y = 5$. The question now becomes does this equation have any integer solutions given the boundary conditions $30 \leq x, y \leq 100$. If there is at least one integer solution, then there would be a data dependence, otherwise there is no data dependence, as is the case with Figure 4.2.

For the discussion that follows, we define the term loop body. The *loop body* of any loop L will be all statements in the program that can possibly be executed during the iterations of loop L. Calls are allowed in a loop, so a single loop body could conceivably include the statements of many different procedures. For example,

```
DO I = 30,100
    A(3 * I - 5) = ...
    ...
    ... = A(6 * I)
END DO
```

Figure 4.2. A loop with array references.

if a loop contains a call of procedure A, and procedure A contains a call of procedure B, then the loop body would include all the statements of procedures A and B. In Figure 4.1, the loop body is the four statements at lines 1 through 4.

With respect to the program flowgraph, the loop body is all flowgraph nodes that may be traversed during the iterations of the loop. Let LB be the set of flowgraph nodes that are in the loop body of loop L. Let n be the first node in the loop body that is traversed during each iteration of the loop. The identification of node n is language-dependent. Within the loop body of L, let definition d be a definition of a non-array variable v, and let use u be a use of the variable v that is reached by definition d. Let d be the node in the loop body where definition d occurs, and let u be the node in the loop body where the use u occurs. To avoid the complications posed by special cases, we assume that d, n, and u are separate and distinct nodes.

Although use u depends on definition d because definition d reaches use u, this data dependence can prevent parallelization of loop L only if the dependence is loop carried. Let P be a sequence of flowgraph nodes drawn from LB, such that P represents a possible execution path along which definition d can reach use u. For definition d to be loop-carried to use u along path P, the three nodes, d, n, and u, must be in P, and in that order, because only the traversal of node n represents the transition to a different iteration of the loop. If v is an array, then we assume that definition d and use u may refer to different array elements during the same iteration. For this reason, a path P that includes the nodes d, u, n, d, u, in that order, must be assumed to show a loop-carried data dependence when v is an array, whereas this path P does not show a loop-carried data dependence if definition d and use u always refer to the same storage location during any iteration, as we assume is the case when v is a non-array, because in any iteration that follows such a path P, the value used at use u is always the value defined at definition d in that same iteration.

4.2 The Parallelization Algorithm

This section presents in Figure 4.3 an algorithm that identifies loops that can be parallelized, including loops that contain calls. The algorithm uses our interprocedural dataflow analysis method as an integral step to determine data dependencies. The loops that can be parallelized are those loops that are not marked by the algorithm as inhibited.

The algorithm has three distinct steps. First, the reaching-definitions dataflow problem is solved for the input program by using our interprocedural dataflow analysis method. Second, the quality of the reaching-definition information computed by the first step is possibly improved in the case of array references by using the separability, gcd, and Banerjee tests. Third, individual d, u pairs that represent data dependence are examined for loop-carried data dependence.

At line 7, the definitions and uses of iteration variables are excluded from testing for loop-carried data dependence, because for any iteration the iteration variables will have constant values that can be precomputed if loop L is parallelized. The test at line 8 is a necessary condition for the P-test procedure to return a T, which is tested for at line 9. The test at line 8 is done as an economy measure to avoid, when possible, the more costly P-test.

Procedure P-test uses a straightforward algorithm that begins with node d and then spreads out examining successors, successors of successors, and so on, until either there are no more acceptable nodes to examine, in which case F is returned, or all the requirements for path P have been met, in which case T is returned. The successors

```
— a $d, u$ pair is a definition $d$ that reaches a use $u$
— $x$ is the dataflow element that represents the definition $d$
— $v$ is the variable referenced by definition $d$ and use $u$

— to avoid complications, $n \neq d \neq u$ is assumed
— $n$ is the first node traversed during each loop L iteration
— $d$ is the node whose basic block contains definition $d$
— $u$ is the node whose basic block contains use $u$

— LB is the set of nodes in the loop body of loop L
— IV is the set of definitions of iteration variables for loop L

    begin
      — step 1, determine reaching definitions for the input program
1     use our method to solve the reaching-definitions dataflow problem

      — step 2, improve the reaching-definition information for array references
2     for all $d, u$ pairs in the program, such that $v$ is an array
3         use the separability, gcd, and Banerjee tests as applicable
4         if definition $d$ and use $u$ can never reference the same element
5             mark the $d, u$ pair as non-reaching
          fi
      end for

      — step 3, identify $d, u$ pairs that inhibit parallelization
6     for each loop L in the program
7         for each reaching $d, u$ pair such that $d, u \in$ LB and definition $d \notin$ IV
8             if $x \in B_{out}[n]$
9                 if P-test($x$, $n$, $d$, $u$, L, LB) = T
10                    mark L parallelization as inhibited by the $d, u$ pair
                  fi
              fi
          end for
      end for
    end
```

Figure 4.3. The parallelization algorithm.

```
    procedure P-test($x$, $n$, $d$, $u$, L, LB)
    — is there a loop-carried data dependence from definition $d$ to use $u$ thru node $n$
    — return T if yes, F if no
    begin
      — part1, is there a path from $d$ to $n$ along which $x$ is found
11    if $v$ is an array
12        DONE $\leftarrow \{d\}$
      else
13        DONE $\leftarrow \{d, u\}$
      fi
14    NEXT $\leftarrow \{d\}$
15    until NEXT $= \emptyset$
16        remove a node from NEXT, denote it $p$
17        for each successor node $s$ of node $p$, such that $s \notin$ DONE
18            DONE $\leftarrow$ DONE $\cup \{s\}$
19            if $s \notin$ LB
              $\vee s$ is an entry node
              $\vee x \notin B_{out}[s]$
20                ignore $s$
21            else if $s = n$
22                goto part2
              else
23                NEXT $\leftarrow$ NEXT $\cup \{s\}$
              fi
          end for
      end until
24    return F
```

Figure 4.3. – continued.

```
      part2:
      — part2, is there a path from $n$ to $u$ along which $x$ is found
25    if $v$ is an array
26       DONE ← $\{n\}$
      else
27       DONE ← $\{n, d\}$
      fi
28    NEXT ← $\{n\}$
29    until NEXT = $\emptyset$
30       remove a node from NEXT, denote it $p$
31       for each successor node $s$ of node $p$, such that $s \notin$ DONE
32          DONE ← DONE ∪ $\{s\}$
33          if $s \notin$ LB
            ∨$s$ is an exit node
            ∨($s$ is contained in the same procedure that contains L ∧ $x \notin B_{out}[s]$)
            ∨($s$ is not contained in the same procedure that contains L ∧ $x \notin E_{out}[s]$)
34             ignore $s$
35          else if $s = u$
36             return T
            else
37             NEXT ← NEXT ∪ $\{s\}$
            fi
         end for
      end until
38    return F
   end
```

Figure 4.3. – continued.

of a node are examined because normally a successor node is assumed to represent a possible continuation of the execution path from the point of the predecessor node. Exceptions in the algorithm involving entry and exit nodes are explained shortly. Note that P-test only determines whether a satisfactory path P exists or not; it does not determine what path P is in terms of an actual node sequence, as there may be many such satisfactory paths P. Lines 13 and 27 are active when v is not an array. In this case, a path P that includes d, u, n, d, u, in that order, is not allowed, and this is prevented by marking the unwanted node u at line 13, and the unwanted node d at line 27.

The test of $x \notin B_{out}[s]$ at line 19 satisfies the requirement that the definition d can reach along the path P. A similar test is made at line 33. At line 19, only the B set is checked because there are no descents into called procedures, as per the rejection of entry nodes at line 19. Entry nodes are rejected at line 19 because any path from d to n will not leave unreturned calls, because n is an outermost node relative to the loop body, and the path is confined to the loop body. As the successors of each call node are an entry node and a return node, it is only necessary to check the *out* set of the return node to know whether the element x survived the call or not, and this is effectively done by the $x \notin B_{out}[s]$ test already mentioned. At line 33, exit nodes are rejected because any path from n to u will not make a return without first making the call. This follows from the fact, already mentioned, that node n is an outermost node relative to the loop body, and the path is confined to the loop body. As the return node can always be added to the path P from the call node, there is no need to add it from the exit node, hence the rejection of the exit node.

For part1 and part2 in procedure P-test, each flowgraph node may appear only once in the NEXT set, hence the complexity of the P-test procedure is $O(n)$ where n is the number of flowgraph nodes. For the entire algorithm, step3 dominates, so the complexity is $O(lpn)$ where l is the number of loops in the program, p is the number of d,u pairs in the program, and n is the number of flowgraph nodes.

CHAPTER 5
CONCLUSIONS AND FUTURE RESEARCH

5.1 Summary of Main Results

The first part of this work presented a new method for context-dependent, flow-sensitive interprocedural dataflow analysis. The method was shown to produce a precise, low-cost solution for such fundamental and important problems as reaching definitions and available expressions, regardless of the actual call structure of the program being analyzed. By using a separate set to isolate calling-context effects, and another set to accumulate body effects, the calling-context problem has been reduced to the problem of solving the dataflow equations that compute the different sets. These equations can be solved by the iterative algorithm. As part of our method, the interprocedural kill effects of call-by-reference formal parameters are correctly handled by the equations-compatible technique of element recoding.

The importance of our interprocedural analysis method lies in the fact that a number of different applications depend on the solution of fundamental dataflow problems such as reaching definitions, live variables, definition-use and use-definition chains, and available expressions. Program revalidation, dataflow anomaly detection, compiler optimization, automatic vectorization and parallelization, and software tools that make a program more understandable by revealing data dependencies, are some of the applications that may benefit by using our method.

The second part of this work presented new algorithms for precise interprocedural logical ripple effect and slicing. The algorithms use our interprocedural dataflow analysis method, and add a control mechanism by which, in effect, execution-path history can affect execution-path continuation as the ripple effect or slice is built piece by piece.

The importance of our algorithms for precise interprocedural logical ripple effect and slicing lies in their applicability to the areas of software maintenance and debugging. A precise interprocedural logical ripple effect can be used to show a programmer the consequences of program changes, thereby reducing errors and maintenance cost. Similarly, a precise interprocedural slice can localize program faults, thereby saving programmer effort and debugging cost.

The third part of this work presented an algorithm that identifies loops that can be parallelized, including loops that contain calls. The algorithm makes use of our interprocedural dataflow analysis method to determine data dependencies, and then the algorithm examines the data dependencies within each loop and determines if any of these data dependencies are loop-carried, in which case parallelization of the loop is inhibited. The algorithm has potential use in parallelization tools.

5.2 Directions for Future Research

There are several topics of possible future research related to our method for interprocedural dataflow analysis. Regarding solving the equations, besides the iterative algorithm there are elimination algorithms [20] that have better complexity. Further studies are needed to determine to what extent these other algorithms can be used to solve the equations. Another topic regards the dataflow problems that can be solved by our method, as the actual universe of solvable problems remains to be determined. We have only mentioned a few of the better known problems. For some dataflow problems, it may be that our method can be used after suitable modification to adapt it to the special needs of the problem.

Regarding possible future research related to our algorithms for precise interprocedural logical ripple effect and slicing, because the algorithms may overestimate when recursive calls are present, or because the Allow set lacks the information needed to enforce the ordering of unmatched returns, one area of future research would be to investigate the possibility of modifying Definition 1, Theorems 1 through 4, and the algorithms, so as to remove the possibility of such overestimation.

REFERENCES

[1] Agrawal, H., and Horgan, J. Dynamic program slicing. *Proceedings of the SIGPLAN 90 Conference on Programming Language Design and Implementation. ACM SIGPLAN Notices*, 25, 6 (June 1990), 246–256.

[2] Aho, A., Sethi, R., and Ullman, J. *Compilers, Principles, Techniques and Tools.* Addison-Wesley, Reading, MA (1986).

[3] Allen, F. Interprocedural data flow analysis. *Proceedings of the IFIP Congress 1974*, North Holland, Amsterdam (1974), 398–402.

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

[5] Burke, M., and Cytron, R. Interprocedural dependence analysis and parallelization. *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, 162–175.

[6] Callahan, D. The program summary graph and flow-sensitive interprocedural data flow analysis. *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation. ACM SIGPLAN Notices*, 23, 7 (July 1988), 47–56.

[7] Cooper, K., and Kennedy, K. Interprocedural side-effect analysis in linear time. *Proceedings of the SIGPLAN 88 Conference on Programming Language Design and Implementation. ACM SIGPLAN Notices*, 23, 7 (July 1988), 57–66.

[8] Ferrante, J., Ottenstein, K., and Warren, J. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9, 2 (1987), 319–349.

[9] Harrold, M., and Soffa, M. Computation of interprocedural definition and use dependencies. *Proceedings of the IEEE Computer Society 1990 Int'l Conference on Computer Languages*, New Orleans, LA (March 1990).

[10] Hecht, M. *Flow Analysis of Computer Programs.* Elsevier North-Holland, New York (1977).

[11] Horwitz, S., Reps, T., and Binkley, D. Interprocedural slicing using dependence graphs. *ACM Transactions on Programming Languages and Systems*, 12, 1 (Jan. 1990), 26–60.

[12] Hwang, J., Du, M., and Chou, C. Finding program slices for recursive procedures. *Proceedings of the IEEE COMPSAC 88* (Oct. 1988), 220–227.

[13] Johmann, K., Liu, S., and Yau, S. *Dataflow Equations for Context-Dependent Flow-Sensitive Interprocedural Analysis.* SERC-TR-45-F, Department of Computer and Information Sciences, University of Florida, Gainesville (Jan. 1991).

[14] Korel, B., and Laski, J. Dynamic program slicing. *Information Processing Letters*, 29, 3 (Oct. 1988), 155–163.

[15] Landi, W., and Ryder, B. Pointer-induced aliasing: a problem classification. *Conference Record of the 18th ACM Symposium on Principles of Programming Languages*, ACM, New York (1991), 93–103.

[16] Leung, H., and Reghbati, H. Comments on program slicing. *IEEE Transactions on Software Engineering*, SE-13, 12 (Dec. 1987), 1370–1371.

[17] Myers, E. A precise interprocedural data flow analysis algorithm. *Conference Record of the 8th ACM Symposium on Principles of Programming Languages*, ACM, New York (1981), 219–230.

[18] Richardson, S., and Ganapathi, M. Interprocedural optimization: experimental results. *Software—Practice and Experience*, 19, 2 (1989), 149–169.

[19] Rosen, B. Data flow analysis for procedural languages. *Journal of the ACM*, 26, 2 (April 1979), 322–344.

[20] Ryder, B., and Paull, M. Elimination algorithms for data flow analysis. *ACM Computing Surveys*, 18, 3 (Sep. 1986), 277–316.

[21] Sharir, M., and Pnueli, A. Two approaches to interprocedural data flow analysis. Muchnik, S., and Jones, N. Eds. *Program Flow Analysis: Theory and Applications*, Prentice-Hall, Englewood Cliffs, NJ (1981), 189–232.

[22] Triolet, R., Irigoin, F., Feautrier, P. Direct parallelization of call statements. *Proceedings of the SIGPLAN 86 Symposium on Compiler Construction*, 176–185.

[23] Weiser, M. Programmers use slices when debugging. *Communications of the ACM*, 25, 7 (July 1982), 446–452.

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

[25] Zima, H., and Chapman, B. *Supercompilers for Parallel and Vector Computers.* Addison-Wesley, Reading, MA (1990).

BIOGRAPHICAL SKETCH

Kurt Johmann was born in Elizabeth, New Jersey, on November 16, 1955. In 1978 he received a B.A. in computer science from Rutgers University in New Jersey. Following graduation, he worked for a shipping company, Sea-Land Service Inc., as a programmer and systems analyst. In 1985 he left Sea-Land and did PC work for three years. Following this, he entered the graduate program of the Computer and Information Sciences Department at the University of Florida in the Fall of 1988. He received an M.S. in computer science, December 1989, and entered the Ph.D. program. Anticipating graduation, he hopes to find a job in academia.

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Stephen S. Yau, Chairman
Professor of Computer and
Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Richard Newman-Wolfe, Cochairman
Assistant Professor of
Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]

Paul Fishwick
Associate Professor of
Computer and Information Sciences

I certify that I have read this study and that in my opinion it conforms to acceptable standards of scholarly presentation and is fully adequate, in scope and quality, as a dissertation for the degree of Doctor of Philosophy.

[illegible]
Mark Yang
Professor of Statistics

This dissertation was submitted to the Graduate Faculty of the College of Engineering and to the Graduate School and was accepted as partial fulfillment of the requirements for the degree of Doctor of Philosophy.

May, 1992

[illegible]
Winfred M. Phillips
Dean, College of Engineering

Madelyn M. Lockhart
Dean, Graduate School