

Incremental Dependence Analysis

Carl M. Rosene

CRPC-TR90044
March, 1990

Center for Research on Parallel Computation
Rice University
P.O. Box 1892
Houston, TX 77251-1892

RICE UNIVERSITY

Incremental Dependence Analysis

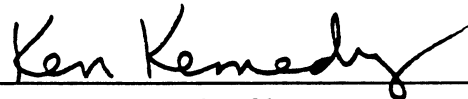
by

Carl M. Rosene

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Doctor of Philosophy

APPROVED, THESIS COMMITTEE:



Ken W. Kennedy, Chairman
Noah Harding Professor of Mathematics



Lori Pollock
Assistant Professor of Computer Science



John E. Dennis
Noah Harding Professor of Mathematical
Sciences

Houston, Texas

March, 1990

Contents

Abstract	iv
Acknowledgments	v
List of Illustrations	vi
List of Tables	vii
List of Algorithms	viii
1 Introduction	1
1.1 Dependence Analysis	1
1.2 PFC-Plus	4
1.3 Parallel Programming Environments	6
1.3.1 PTOOL	6
1.3.2 ParaScope	7
1.4 A Model Environment	8
1.5 Overview of Incremental Dependence Analysis	9
2 Analysis of Control Dependence	12
2.1 The Control Dependence Graph	15
2.2 CFG Edge Addition	19
2.2.1 Control Dependence Addition	23
2.2.2 Control Dependence Deletion	38
2.3 CFG Edge Deletion	41
2.3.1 Control Dependence Addition	42
2.3.2 Control Dependence Deletion	50
2.4 Complexity Analysis	54
3 Data Dependence	56
3.1 Definitions	56
3.1.1 Scalars and Arrays	57
3.1.2 Independence Tests	59

3.1.3	A Stronger Notion of Dependence	62
3.2	Algorithms	66
3.2.1	Batch Algorithm	68
3.2.2	An Incremental Calculation	77
3.3	Generalizing the Algorithms	100
3.3.1	Extended Array Descriptors	102
3.3.2	Covering by a Loop	104
3.3.3	Nested Loops	109
3.4	Calculation of Anti and Output Dependences	114
3.5	Complexity Analysis	114
4	Symbolic Analysis for Subscript Testing	117
4.1	Integer Expression Folding	118
4.2	Loop Invariant Testing	135
4.3	Induction Variable Identification	138
5	Experiments and Results	143
5.1	Measurements	144
5.1.1	$\ \eta_s\ $ and $\ \eta_t\ $	144
5.1.2	$\ \eta\ $	147
5.2	Implications and Further Arguments	149
6	Related Work	152
6.1	Dependence Analysis	152
6.1.1	Parallelism Detection	152
6.1.2	Intermediate Analysis	154
6.2	Programming Environments	154
6.3	Incremental Techniques	155
7	Conclusions and Future Work	157
	Bibliography	159

Illustrations

2.1	Post Dominator Relation	13
2.2	A Control Flow Graph and Its Control Dependence Graph	16
2.3	Edits Producing an If-Then-Else	20
2.4	Splitting a CFG Node	21
2.5	Example for Lemma 2.3	24
2.6	CFG Edge Addition	26
2.7	Example for Theorem 2.2	27
2.8	Example for Lemma 2.4	29
2.9	Example for Lemma 2.5	31
2.10	Example for 2.9	39
2.11	Dependence formed by postdomination of immediate successor	43
2.12	A multi-exit loop	47
2.13	Deletion of a Control Flow Edge	50
3.1	Data Statements in the Control Flow Graph	79
3.2	Set Storage Scheme	80
3.3	Combs	106
4.1	Shadow Expressions and Links	123
5.1	CFG fragment from SIMPLE	148

Tables

5.1	Programs from RiCEPS	144
5.2	Complexities of the Dependence Update Algorithms	145
5.3	Distribution of values for η_s and η_t	147
5.4	Distribution of values for η	150

Incremental Dependence Analysis

Carl M. Rosene

Abstract

New supercomputers depend upon parallel architectures to achieve their high rate of computation. In order to take advantage of the power of such a machine, programs must be executed *in parallel*, that is, parts of the program must execute on different processors. When a program is executed in parallel it is impossible to guarantee the *execution order* of the parts of the program being executed by different processors. *Dependence analysis* identifies the statements whose execution order must be preserved in order for the program to be correct. Making available the results of dependence analysis in an interactive programming environment can aid a programmer in writing programs which will execute more efficiently on a parallel machine. In order to provide the results of dependence analysis to the programmer in as timely a manner as possible, incremental methods of dependence analysis have been developed. The performance of these algorithms has been estimated based on static measurements of FORTRAN programs. This dissertation presents the incremental methods and the results of estimates of their performance.

Acknowledgments

Any attempt to adequately acknowledge all the people who have contributed to this work is hopelessly doomed to failure. Nonetheless, in the tradition of PhD dissertations everywhere, I make the attempt.

First, I would like to acknowledge the efforts of my parents who started all this twenty-eight years ago. My chairman, Dr. Ken Kennedy, has supported me throughout the effort and has been insistent on my presentation of the best work possible. Any failure is wholly mine. Dr. Lori Pollock talked with me at an early stage of the thesis while many of the ideas had yet to solidify. Marina Kalem helped me collect much of the experimental results presented in Chapter 5. Dr. Randy Allen suggested the topic. My fellow graduate students have all been eager to serve as patient listeners and active participants while I have rambled on about the problems investigated here. The members of my committee and many of my fellow graduate students have labored mightily to save me from the tortured syntax and awkward sentence structures that it is all too easy to fall victim to. To all of them, "Thank You."

Besides these academic contributions, my many friends have helped make my graduate school experience what it was. I would especially like to mention the members of World Tour, Sixth Floor, The Rice Sailing Club, the Durcans, Tripp and John, Banks, my fellow graduate students, the residents of Crashwood, and everybody who just hung out there. You have my deep and abiding gratitude for providing me some very good times.

Algorithms

2.1	Adding Control Dependences due to Condition 1 of Lemma 2.2 . . .	32
2.2	Adding control dependences due to second condition of 2.2	34
2.3	CDG Edge Deletion in Response to CFG Edge Addition	40
2.4	Control Dependence Additions in response to CFG Edge Deletion . . .	49
2.5	Control Dependence Deletion in Response to Control Flow Edge Deletion	53
3.1	Main Program AllDeps for Batch Algorithm	73
3.2	Procedure LoopIndependent called from AllDeps	74
3.3	Procedures LoopIndependent and LoopCarried called from AllDeps .	75
3.4	Updating Dependences in Response to Addition of a Use	82
3.5	Updating Dependence in Response to the Deletion of a Use	84
3.6	Update In Response to Addition of a Definition	86
3.7	Routine to Propagate Added Definitions to May Sets	87
3.8	Routine to Propagate Deletion of Definitions from May Sets	88
3.9	Routine to Propagate Additions to Must sets	89
3.10	Must Propagation Routines	90
3.11	Updating Dependences Due to Deletion of a Definition	96
3.12	Updating Data Dependence After Control Flow Edge Addition	98
3.13	Updating Data Dependence After Control Flow Edge Deletion	101
3.14	Finding Array Kills	107
3.15	Incremental Discovery of Array Kills	108
3.16	Updating Dependences in Response to Addition of a Use	112
3.17	Update for Addition of a Definition in Presence of Nested Loops . . .	113
4.1	Fold Update for Added Use	125
4.2	Fold Update for Deleted Use	125
4.3	Fold Update for Added Definition	126
4.4	Fold Update for Deleted Definition	127
4.5	Fold Update for Added Control Flow Edge	128

4.6	Fold Update for Deleted Control Flow Edge	128
4.7	Delete Folds for Deleted Use	129
4.8	Delete Folds for Changed Definition	129
4.9	Delete Folds for Changes in Reaching Sets	130
4.10	Create Folds for Changes in Reaching Sets	130
4.11	Attempt a Particular Fold	131
4.12	Finding IAIVs	141

Chapter 1

Introduction

Efficient scientific programming has always required attention to the low-level details of how a machine operates. As proof, consider the thousands of FORTRAN programmers who know to access two-dimensional arrays in column-major order. Despite the best efforts of language designers and compiler optimization experts, the emergence of parallel and vector processors and the existence of still insoluble problems promise that programmers' preoccupation with using their machines efficiently will increase rather than decrease.

More than any other innovation in computer architecture, the parallel nature of new machines creates the need for sophisticated software tools to help the programmer exploit the power of his machine. These tools will operate within larger integrated programming environments that already provide some aid in the form of language-smart editors and sophisticated debuggers. Time will determine the ultimate form of these new tools, but by looking at experiences with automatic vectorizers and parallelizers we can gain insight concerning the kind of information that the tools must provide to the programmer. This examination reveals that at the center of automatic vectorizers and parallelizers is a form of data flow analysis called *dependence analysis*.

1.1 Dependence Analysis

The central problem in programming parallel machines is guaranteeing correctness in the absence of knowledge on the part of the programmer about the *execution order* of statements run in parallel. The execution order of statements is determined by their location in the control flow of the program.

For instance, in the following code segment,

```

DO I = 1, N
S1:    A(I) = B(I) + J
S2:    B(I) = B(I+1)
ENDDO

```

statement S1 is clearly executed before S2 during any iteration of the loop. But there are N executions of S1 and S2 over the N iterations of the loop. The execution of S2 that occurs during the k^{th} iteration of the loop happens before the execution of S1 during the $(k+1)$ th iteration. If the iterations of the loop execute in parallel, we have no knowledge about the relative execution order of statements in different iterations.

To preserve the semantics of the original sequential program, it is necessary to preserve the execution order of some statements in the program. However, this restriction prevents some parallelization. Maximizing parallelism while maintaining the semantics of the original program requires a knowledge of how the results of the program depend on the execution order of its statements. This requires knowledge about how the results of statements within the program depend on the execution of other statements. When the result of a statement x can change based on the result of the execution of another statement y , then we say that x *depends* on y , or, equivalently, a *dependence* exists from y to x . The dependence in the example above is known as a *data dependence*. It results from the interaction of the statements S1 and S2 via a shared memory location or locations, namely, elements of the array B.

Statements also interact if one statement determines the control flow around the other statement. A statement s is said to be *control dependent* on a control statement (i.e., a statement with more than one immediate successor in the control flow graph) whose result determines whether s is executed.

Automatic parallelizers and vectorizers use *dependence analysis* to discover how statements depend on one another. This information is represented by a directed

or return answers takes so long that the user takes a coffee break, then we have lost the advantage of working in an interactive system, and we can return to batch systems for our programming.

The required response time mandates the use of incremental techniques for the analysis so that whenever the user needs information about his program a minimum of work must be done to obtain the current information. This dissertation describes an incremental method for dependence analysis appropriate for use in an environment for programming multiprocessors. It provides some of the techniques essential for the development of effective parallel programming environments.

We next describe in the next section of this chapter an automatic vectorizer and parallelizer known as PFC-Plus which was developed at Rice University . Knowledge of PFC-Plus will help in the discussion of our own methods for dependence analysis presented in later chapters. To further motivate and justify this work, the following section discusses two tools for parallel programming and the use of dependence information to aid the programmer of a parallel machine. We follow our discussion of these two environments with a description of a simple programming environment which will provide a context in which to consider the new incremental methods of dependence analysis. At the end of this chapter, we describe the structure of the dependence analysis phase for a possible programming environment and present an overview of the dissertation.

1.2 PFC-Plus

PFC-Plus is a source-to-source optimizer that takes serial FORTRAN programs and produces FORTRAN programs with vectorization and parallelism directives. The directives for vectorization are in the form of FORTRAN 8X-like triplet notation. Shared memory multiprocessor parallelism directives take the form of VM/EPEX-style directives. The following describes in a general way the phases of transformation and analysis that occur in PFC-Plus.

graph called a *dependence graph*. Nodes of the dependence graph represent statements, while edges represent the dependences between statements. The dependence graph represents the statement execution orders in the program that must be preserved. Typically, the programmer using an automatic vectorizer or parallelizer does not see the results of the analysis, and indeed he might not even know what parts of his program can be parallelized and what parts cannot be. The programmer is given no aid in writing programs that would parallelize more completely, and the programmer is unable to help the automatic parallelizer transform his program.

So far, aid to programmers has been limited to automatic vectorizers or parallelizers attached to either the front or back end of a compiler for a particular machine. While some parallelism can be exploited in this way, we ask if it is possible to produce a synergy between the programmer and the optimization process. Is there some way to help the programmer to write his program so that it will run on the target machine substantially faster than the product of an unaided programmer and an automatic parallelizer working separately?

We believe that effective aids for parallel programming can be developed based on an understanding of how the execution order of statements in a program must be preserved. With dependence information, a sophisticated compiler embedded in a programming environment could interactively

- suggest ways of changing the program to obtain greater parallelism,
- perform sequential to parallel transformations under user direction,
- query the user to establish the correctness of particular transformations, or
- confirm that the algorithm itself is inherently sequential.

Such interactive features would justify embedding a compiler in an interactive programming environment, but only if the response times remain reasonable. That is, if the analysis required for the environment to make suggestions, present questions,

with the equivalent linear expression involving the loop induction variable. Failure to substitute the actual induction variable for its auxiliary in a subscript expression results in a much less accurate test for independence between the reference involving the subscript expression and other references to the same subscripted variable within the loop. The inaccuracy arises from the necessity of treating the unsubstituted auxiliary induction variable as an unknown symbolic value.

After induction variable substitution, dependence analysis begins. Dependences are calculated for both scalar and array variables during this phase. For scalars, all possible dependence edges are added for all the loops surrounding both references. For subscripted variables, *independence tests* are performed on each pair of references contained within common loops to attempt to prove that, due to the sequence of values of the subscript expressions of the references, the two references cannot access the same memory locations in a particular order during the execution of the loop. Both the textual order of the references and the order of the loop iterations is considered. If independence cannot be proven, then dependence is assumed and the appropriate edge is added to the dependence graph.

Code generation follows dependence analysis. Since this dissertation does not treat code generation we will not discuss it here.

1.3 Parallel Programming Environments

1.3.1 PTOOL

At Rice University, we have implemented and experimented with a parallel programming aid called PTOOL [ABKP86], which has been used to help debug parallel programs at Los Alamos National Laboratory and the Cornell Theory Center. PTOOL identifies variables that must be in shared storage for a loop to be correctly parallelized and, on request, displays dependences that inhibit parallelization. Thus the user is directed to portions of his code that do not allow parallelism or are most

The input program is first scanned and parsed into an abstract syntax tree (AST). All other phases operate on this AST. In the output phase, the transformed AST, with parallelism directives, is converted to the extended FORTRAN output language.

In order to simplify dependence tests on array accesses, do-loops are *normalized*. This involves taking inductive do-loops and converting the upper and lower bounds and the step of the induction variable of the loop so that the lower bound and step are both 1. The necessary upper bound is easily calculated. All uses of the old induction variable in the loop are replaced with equivalent expressions involving the new induction variable.

PFC-Plus then performs *if conversion*. If conversion is performed for two reasons. First, in order to translate a statement that is only conditionally executed in a loop body into a vector statement, a mask must be calculated to control the corresponding vector instruction performing the specified operation so that it affects only the same elements of the vector affected by the serial code. Second, if conversion transforms control dependences into data dependences. In this way, a single phase can be used to calculate both types of dependence in the dependence graph.

Neither normalization nor if conversion have a direct positive effect on the accuracy or precision of the data dependence calculation. Do-loop normalization is performed for convenience in the coding of *independence tests* for array variables, and if conversion is performed as preparation for vector code generation and to avoid a separate control dependence calculation. The next phase, in contrast, has a very direct effect on the precision of the data dependence calculation on statements referencing array variables.

Induction variable substitution identifies *auxiliary induction variables* within a loop body and replaces their occurrences with expressions in terms of the actual induction variable of the loop. An auxiliary induction variable is a variable in a loop whose value at all uses within the loop body is a linear expression in the actual induction variable of the loop. PFC-Plus replaces references to auxiliary induction variables

likely to produce indeterminate results. Bringing the user's attention to these regions provides significant help in parallelizing or debugging code.

Despite its success, PTOOL is severely limited by its nature. It is merely a browser and does not support changes of the program to allow its user to attempt to eliminate a parellism preventing dependence. After the user changes his program using some editor unrelated to PTOOL, he must resubmit the entire program to PTOOL and wait for all the dependences to be recalculated, regardless of the significance of the change. This process can take several minutes even for small programs.

The next tool described, ParaScope, attempts to address this shortcoming of PTOOL.

1.3.2 ParaScope

ParaScope is an interactive parallel programming environment under development at Rice University. Evolved out of both the Rn programming environment project [CCH⁺87] and the PFC project [AK87], it provides an interactive editor along with integrated program composition, execution, and debugging tools. In its ultimate form, it is intended to provide those desired features absent in PTOOL.

ParaScope's parallel programming tools will be embedded in an interactive editor. In response to user queries, parallelization preventing dependences will be displayed, and the user will be able to attempt to eliminate these dependences on his own. Or the use can request that the environment perform some particular parallelizing transformation on the code. Alternatively, the environment could select and perform the transformations automatically. These levels of support are described as manual steering, power steering, and autopilot respectively.

This dissertation addresses the third limitation of PTOOL, the lack of an incremental dependence analysis. The techniques developed here are intended for incorporation in ParaScope.

1.4 A Model Environment

In order to help motivate the discussion of our methods in the rest of this work, we will consider them in the context of a model environment. The model is a dependence browser integrated with some kind of syntax-directed editor. The editor maintains an intermediate form of the program which includes a control flow graph and a symbol table.

We assume that the user is editing a procedure. Occasionally he will stop and browse the dependences that have changed as a result of his activities. All editing stops when the user indicates that he wants to browse the dependences. He may be modifying an already existing procedure or creating one from scratch.

The goal is to provide the user with the dependences as quickly as possible when he asks to browse them. The advantage of this model is that it focuses attention on the building of the dependence graph itself while demanding as much from the calculation of the dependences in terms of precision and response times as any of the other features of an interactive system we described. If we can satisfy the time and precision requirements on the calculation of dependences in this application, we will be able to do so for other interactive applications of dependence analysis that arise in an effective parallel programming environment.

We also assume the existence of a *do-it* button which indicates to the environment when a change is complete. "Change" in this context refers not to the change of a syntax tree node (which is what the editor sees), but rather the user's idea of a change. Thus, a single change might refer to changing all of the subscript expressions in a loop. The do-it button can be implemented in a number of ways. The user could simply press a key when he considered a change complete. Dependence analysis will begin when the button is pressed. This device will avoid *false updates*—changes that are undone by a subsequent change. It will be the editor's job to present a minimal set of changes to the rest of the analysis.

1.5 Overview of Incremental Dependence Analysis

The processes necessary for dependence analysis in response to arbitrary editing changes can be divided into five phases:

1. The Editor
2. Control Dependence Analysis
3. Scalar Dependence Analysis
4. Symbolic Analysis to Support Subscript Testing
5. Array Dependence Analysis

Each phase accepts a different kind of information about the changes to the program to produce the information for which it is responsible. Likewise, each phase produces a different kind of information about the changes to the program.

The editor communicates directly with the user. It is the original source for program changes seen by the later stages of analysis. The form of the editor is outside the scope of the present work. However, we require that the editor

- maintain the program structure as a control flow graph using basic blocks,
- refine the changes ordered by the user into a minimal set of changes, and
- pass these changes to the later analysis phases and invoke the phases as appropriate.

The control dependence phase accepts from the editor changes in control flow edges and basic blocks. This phase performs all of its analysis on the nodes and edges of a control flow graph representation of the program, based on information from the editor concerning the addition and deletion of flow graph nodes and edges.

The algorithms necessary to update the control dependence graph in response to the changes presented by the editor are discussed in chapter 2.

Scalar dependences can be calculated using Zadeck's method[Zad84] for calculating def-use chains. Zadeck's algorithm must be modified slightly to include loop-carried and loop-independent annotations on the dependences. In addition, the technique must be applied to calculation of anti (use-def) and output (def-def) dependences. The scalar dependence phase

- uses information from the editor concerning added and deleted variable definitions and uses, control flow edges, and basic blocks;
- performs its analysis on statements of the program; and
- provides lists of added and deleted dependences on scalar variables.

Since this problem has been addressed elsewhere[Zad84], we will not describe this phase further.

Data dependence on arrays is a harder problem. In order to support the kind of calculation of data dependence required to find all the parallelism in a multiprocessor, tests for dependence must be applied to the subscripts of the array references. A symbolic analysis phase (to be described next) contributes to the information available about the subscripts. Thus, the array dependence analysis

- uses information from the editor and from the symbolic analysis phase to determine which reference pairs require testing of their independence or covering, and
- performs independence and covering tests on the appropriate array references, placing the appropriate annotated edges into the data dependence graph indexed by statement.

Dependence analysis on arrays is presented in chapter 3.

The intermediate symbolic scalar analyses all have as their goal the support of subscript testing for dependence. This phase

- uses the updated scalar dependences to generate new information about symbolic values in subscripts and
- uses the *old* control flow graph and scalar dependences to mark the array dependences that must be recalculated due to changes in the results of the intermediate analyses.

This analysis phase is described in Chapter 4.

Since the kinds of tools that are actually provided in the programming environment will affect the kinds of edits that occur, the actual performance of our techniques can be accurately measured only in an actual implementation under use by real programmers. Nonetheless, chapter 5, considers the expected performance of these techniques, based on measurements of real FORTRAN programs. Chapter 6 describes related work in the area. Finally, Chapter 7 presents the conclusions.

The definition of control dependence follows[FOW87].

Definition 2.2 In a program, a statement v is control dependent on a statement u if and only if

1. a path exists from u to v such that all nodes on the path not including u or v are postdominated by v and
2. either v does not postdominate u or $v = u$.

It is easy to prove from the definition of the postdominator relation that a statement, r , is postdominated by another statement, v , if and only if a path exists from r to v such that all nodes on the path are postdominated by v . This gives rise to the following equivalent definition¹.

Definition 2.3 In a program, a statement v is control dependent on a statement u if and only if

1. there exists r , an immediate successor of u , such that r is post dominated by v and
2. either v does not postdominate u or $v = u$.

Intuitively, a control dependence exists from u to v if and only if there is some outcome of u (i.e. some branch taken from u) that implies the execution of v and some other outcome that does not imply the execution of v (note that it is unnecessary that the other outcome imply that v is *not* executed). The postdominator relation between two nodes tells us whether the execution of one node implies the execution of the other node. By convention, the nodes of a CFG that postdominate the beginning of the program are control dependent on a special node, START.

¹In the original paper by Ferrante et. al., the definition of the postdominator relation does not allow a node to postdominate itself. This results in their original control dependence definition. By extending slightly the definition of the post dominator relation we obtain a simpler and more natural alternative definition of control dependence.

The control dependence relation is represented by the control dependence graph (CDG). The next section describes the CDG. The two sections following describe how to update the CDG in response to the addition or deletion of edges in the program's control flow graph. We close the chapter with an analysis of the time complexity of the update algorithms.

2.1 The Control Dependence Graph

In order to assure the greatest generality of the algorithms, we make few assumptions about the shape of the CFG. In particular, we place no limits on the number of immediate successors or predecessors of a node in the CFG.

We have discussed the control dependence relation as it applies to individual statements. Since the execution of any statement within a basic block implies the execution of all the statements in the basic block [AU77], all the statements within a basic block are control dependent on the same statements. Hence the control dependence relation can be thought of as one between basic blocks, or between control statements (necessarily the last statement in a block) and basic blocks. An implementation of this view of control dependence has a practical advantage in that it reduces the number of nodes and edges in the control dependence graph. Our algorithms apply whether the nodes in the control flow graph represent statements or blocks, even where our discussion refers to one or the other.

Each node in the CDG represents a node in the control flow graph of the program and the edges of the CDG represent the control dependence relation. An edge $\langle u, v \rangle^r$ exists in the CDG if and only if v is control dependent on u and v postdominates r , some immediate successor of u . u is the source of the edge and v is the sink. The edge will be *labelled* by r . In control flow graphs with out degrees greater than two, if v postdominates more than one immediate successor of u but not all of them then an edge with label r_i will exist for every such immediate successor, r_i , of u that v postdominates. For clarity, we introduce the active term and say that u *control*

determines. By the transitivity of the reaches relation, p_n reaches x in the CFG. Suppose that w does not postdominate x . Since p_n reaches x and p_n is postdominated by w and x is not postdominated by w , all paths in the CFG from p_n to x must include w . One of these paths includes the set of nodes $\{p_n, p_{n-1}, \dots, p_1\}$. Some part of this path will include a segment $[p_j, \dots, w, \dots, p_{j-1}]$ such that w postdominates p_j , but not p_{j-1} . Since w postdominates p_j , all of p_j 's immediate successors must reach p_{j-1} by paths involving w . p_{j-1} may or may not postdominate w . If p_{j-1} postdominates w then p_{j-1} will postdominate all of p_j 's immediate successors; thus p_j does not control determine p_{j-1} . Otherwise, p_{j-1} will postdominate none of p_j 's immediate successors; again, p_j does not control determine p_{j-1} . Hence, p_j cannot control determine p_{j-1} , thus contradicting the assumption that p_n was an ancestor of x in the CDG. The if portion of the theorem is proved.

Now we prove the only-if portion. Suppose that w postdominates x . We must show that w is either a right sibling of x or a right sibling of some ancestor of x . Consider a path in the CFG from the beginning of the program to w including x , $P = [\text{ENTRY}, p_1, \dots, x = p_k, \dots, w]$. If w postdominates ENTRY, then w will be a child of the START node in the CDG. Some other child of START will be an ancestor of x or possibly x itself. Since w will postdominate this ancestor of x , it will be a right sibling of the ancestor.

Suppose w does not postdominate ENTRY. The path P will contain a segment $[p_j, p_{j+1}]$, $j < k$, where p_{j+1} is postdominated by w and p_j is not postdominated by w . The edge $\langle p_j, w \rangle^{p_{j+1}}$ will appear in the CDG. First suppose x postdominates p_{j+1} . Since w postdominates x and w does not postdominate p_j , x cannot postdominate p_j . Hence there is a control dependence from p_j to x with label p_{j+1} . From the construction of the CDG w is x 's right sibling, which satisfies the theorem.

Suppose x does not postdominate p_{j+1} . We show that a path exists in the CDG from p_j to x and that the first edge on the path has label p_{j+1} . Since x is an immediate successor of p_{k-1} and it postdominates an immediate successor of p_{k-1} (x itself) it

either postdominates p_{k-1} or is control dependent on p_{k-1} . The same reasoning can be applied to p_{k-1} 's predecessors in the path from ENTRY. By induction, for some p_i , $k - 1 > i > j + 1$, the edge $\langle p_i, x \rangle^{p_{i+1}}$ will exist in the CDG of G . The same argument can be applied for p_i , with p_i replacing x . Thus, by induction, we prove that a path exists in the CDG from p_j to x . The edge from p_j must necessarily have as its sink a node postdominating p_{j+1} . Hence, the label on the edge will be p_{j+1} and the theorem is proved. \square

2.2 CFG Edge Addition

Before we examine the addition of edges between nodes that are already present in the CFG, we will examine the addition of nodes to the CFG and the new edges that arise as a result of the added nodes. Most edits involving the deletion or addition of control flow nodes in the CFG imply the addition or deletion of edges as well. When a node is added to the CFG new edges are necessarily added which connect this node to the rest of the graph. These new edges, though not previously present in the CFG, do not necessarily imply a new alternate path of control. By restricting how new nodes are specified we can avoid updating the CDG in response to those new edges that do not represent new paths of control in the CFG.

When a node is added to the control flow graph it is not initially present in the control dependence graph. A node in the CDG must be created that corresponds to the new node in the CFG. The position of the new node in the CFG can be specified by its immediate successors and predecessors. However, the task of the update algorithms will be easier if, instead, the node is specified as having been split from a previously existing node or added on an existing edge. In Figure 2.3 a series of edits is shown. The first two edits, from a to b to c , show the insertion of an if clause. This requires the splitting of a node and then the addition of an edge from the node containing the new if. The first edit, the split of a node, only requires the update to add the node

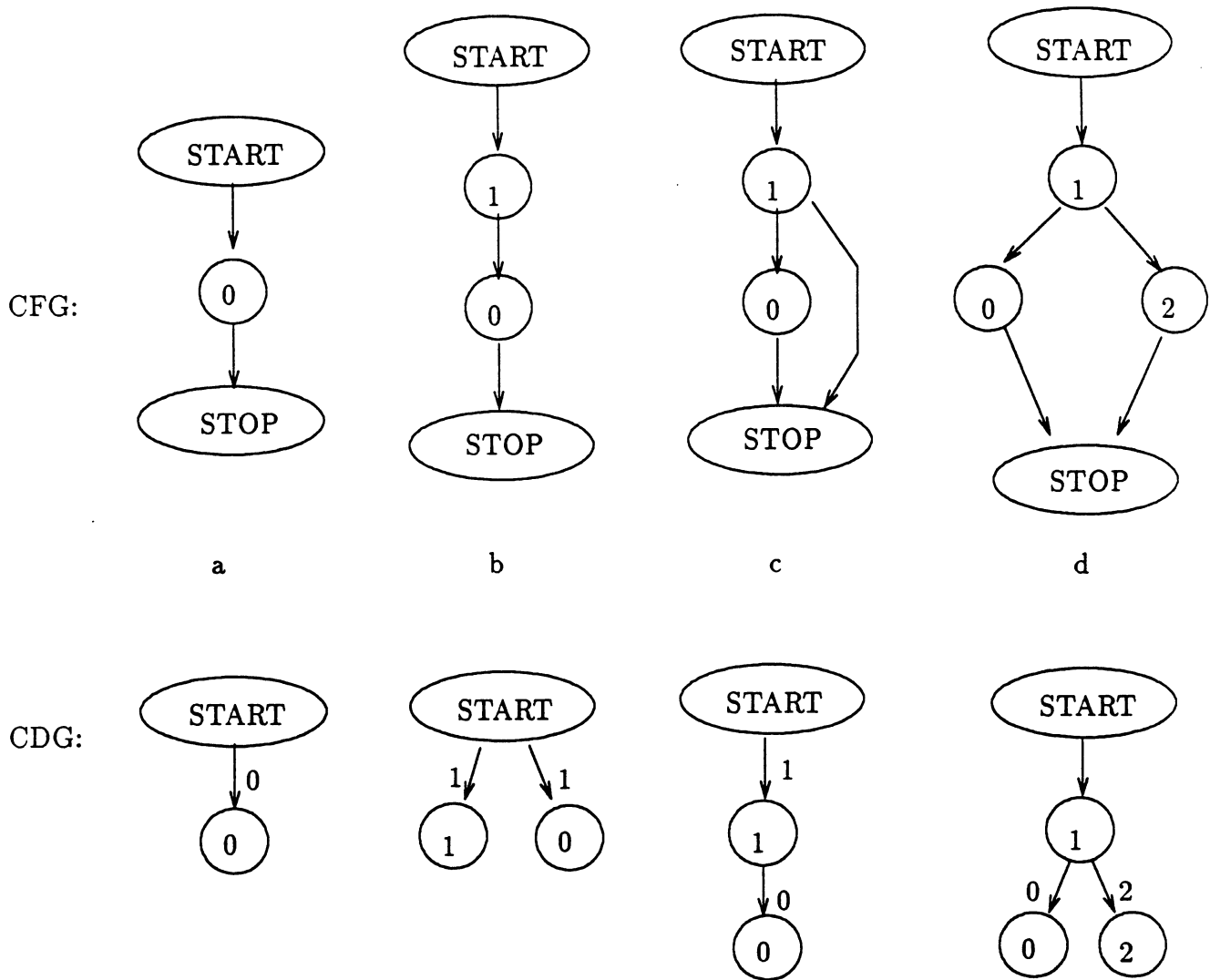


Figure 2.3 Edits Producing an If-Then-Else

to the CDG with the correct edges leading to it. However, the addition of the edge bypassing the second node requires a broader update since the new edge represents the addition of a new path to the CFG. From c to d , the else clause is added. This addition requires that a node be added on the existing edge representing the previously empty else. Since this edit does not add any paths of control that did not already exist, it will only be necessary to add the new node to the CDG.

When a new node is created by splitting a previous one, two nodes are created. One of these nodes will be distinguished as "new" and the other as "old". That is, the "old" node will be given the same identifier as the node that was split; the "new" node must be given a new identifier. Figure 2.4 shows a node split into two. On the right side, either the top or bottom node can have the same name as the old node. If we give the new node name to the bottom node, then we must change all the control dependences on the old node with the name 1 to be on the new node named 2. If instead, we assign the new node identifier to the top node, then the second node can be left as the old node, and the update of those dependences can be avoided. The new node, 2, will be control dependent on the same nodes as the old node 1. Adding the new node to the CDG consists of adding the node to the CDG as a sibling of the old node with identical edge labelling as the old node, but to the left of the old node.

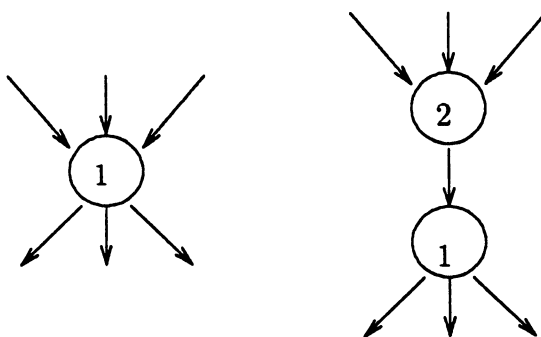


Figure 2.4 Splitting a CFG Node

If a node is added on an existing edge, it has a single immediate successor and a single immediate predecessor. Call the new node v , its immediate predecessor u , and its immediate successor w . Since v has a single immediate successor, w , it is postdominated by whatever nodes postdominate w . Hence, no nodes are control dependent on v . If v 's immediate predecessor, u , has more than one immediate successor (before the node addition), then the new node v is control dependent on u , and the label of the new edge in the CDG from u to v is v . The new node v replaces w in the CFG as the immediate successor of u that corresponds to the particular branch that results in the execution of w and v . Hence, any edges in the CDG labelled with w must be changed to have the label of the new node, v . If u has only a single immediate successor, then v will be control dependent on exactly the same nodes that control determine u . v is added to the CDG by duplicating the current dependences into u with the v as the sink. These added edges be placed to the right of the corresponding edges to u .

We now describe updating the CDG in response to new edges that result in the addition of control paths that do not merely serve to connect new node(s) to the CFG. Our algorithm for updating the CDG in response to the addition of edges to the CFG only needs to deal with the addition of an edge between nodes already present in the CFG. Given an edge (s, t) added to a CFG G , this algorithm will discover all the edges $\langle u, v \rangle^l$ added to, or deleted from, the corresponding CDG.

By considering carefully the definition of the postdominator relation and the possible results of the addition of edges on the availability of paths, we derive the following lemma concerning the possible changes in the postdominator relation caused by the addition of an edge to the CFG.

Lemma 2.1 Given two complete CFGs $G=(V,E)$ and $G'=(V,E')$, where $E'=E+(s,t)$, a node x postdominates a node w in G' only if x postdominates w in G .

Proof Assume that x does not postdominate w in G . Since G is complete, w can reach STOP. Since w is not postdominated by x in G , there exists a path from w to STOP that does not involve x . Since the only difference between G and G' is the addition of (s, t) , this path will exist in G' . Therefore, x cannot postdominate w in G' . \square

This lemma serves to limit how the postdominator relation, and hence the control dependence relation, between a pair of nodes can change in response to the addition of an edge to the CFG.

2.2.1 Control Dependence Addition

The definition of control dependence leads to the following lemma.

Lemma 2.2 Given CFGs $G=(V,E)$ and $G'=(V,E')$, the control dependence edge $\langle u, v \rangle^r$ is present in $CDG(G')$ and not in $CDG(G)$ only if one of the following is true.

1. v postdominates an immediate successor of u , r , in G' but not in G .
2. v postdominates u in G but not in G' .

The proof follows immediately from the definition of control dependence.

Hereafter in this section, unless otherwise stated, $G = (V,E)$ represents a control flow graph and G' represents the control flow graph $(V, E+(s,t))$.

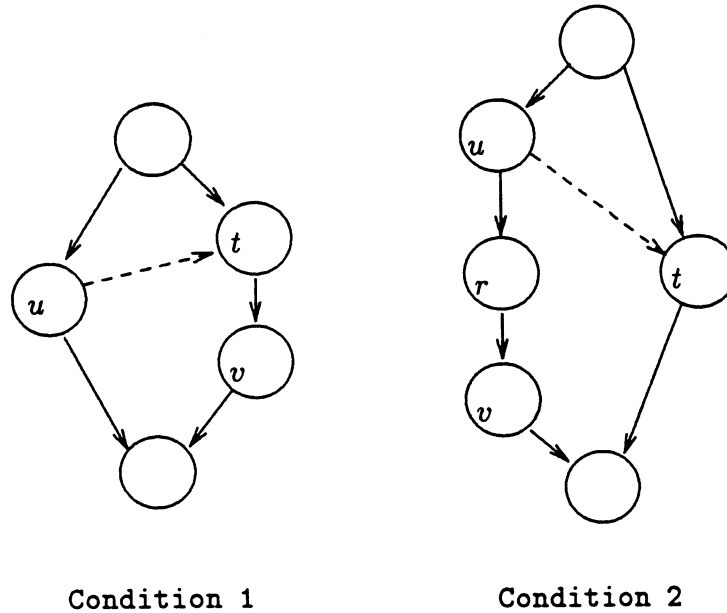


Figure 2.5 Example for Lemma 2.3

From the previous two lemmas we can establish a result that limits the scope of an algorithm for updating control dependence in response to edge addition.

Lemma 2.3 Given CFGs G and G' , the control dependence $\langle u, v \rangle^r$ exists in $CDG(G')$ and not in $CDG(G)$ only if

1. v postdominates t but not s in G and $u = s$ and $r = t$, or
2. v postdominates u and r in G but only r in G' .

See Figure 2.5.

Proof The control dependence $\langle u, v \rangle^r$ implies that r is an immediate successor of u in G' . By Lemma 2.1, if v postdominates r in G' then it does so in G . Hence, the first condition of Lemma 2.2 cannot be satisfied. Therefore a new control dependence cannot be formed by the first condition in Lemma 2.2 if r is an immediate successor of u in G . On the other hand, the addition of (s, t) causes t to become an immediate successor of s in G' when it was not in G . If $u = s$ and $r = t$, r is not an immediate

successor of u in G , thus allowing the first condition of Lemma 2.2 to be satisfied, resulting in a new control dependence. By the definition of control dependence, v postdominates $r = t$ in G' . By Lemma 2.1, v postdominates t in G . By the definition of control dependence, v does not postdominate s in G' . Thus a path exists in G' from s to STOP that does not include v . Since v postdominates t , this path does not include (s, t) , and thus the path exists in G . Hence, v does not postdominate s in G . Thus, the first part of the theorem is proved.

Suppose again that r is an immediate successor of u in G . By the reasoning above, a control dependence can be formed only by the second condition of Lemma 2.2, that is, v postdominates u in G and not in G' . The edge $\langle u, v \rangle^r$ requires that r be postdominated by v in G' and Lemma 2.1 implies that the same holds in G . Thus the theorem is proved. \square

The first condition is easy to understand. If execution follows the new edge (s, t) then of all the nodes postdominating t will be executed. For instance, in figure 2.6, the new edge from 2 to 5 causes 5 to become control dependent on 2.

The second condition is satisfied when the new edge forms a new path from u to STOP that does not involve v . In G , the execution of u implied the execution of v . In G' , the execution of v is dependent on which branch is taken from u . In Figure 2.6, after the addition of the edge from 2 to 5, the execution of 2 no longer implies the execution of 3, hence a new control dependence is created from 2 to 3.

The second condition gives rise to the question, "When does v postdominate u and r in G , but only r in G' ?" This can occur only when the new edge (s, t) creates a new path from some immediate successor of u , $r_2 \neq r$, to STOP, not involving v . Call this new path P . Since we know that P exists only in G' , it must include s and t . The path begins with r_2 , so the portion of P up to s comprises a path from r_2 to s . Thus, the only nodes that can possibly be the source of, or the label on, a new edge

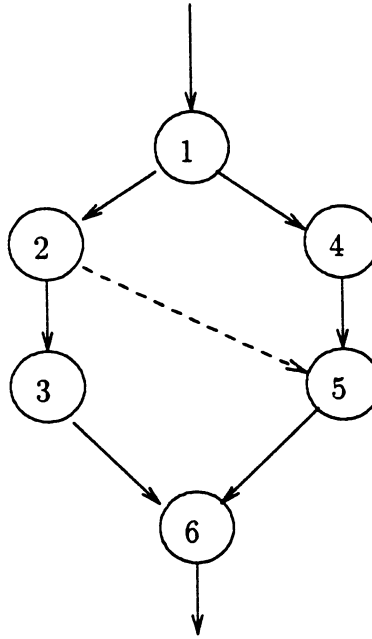


Figure 2.6 CFG Edge Addition

in $CDG(G')$ are those that can reach s in G . This information can be derived from the CDG as the following theorem shows.

Theorem 2.2 In a complete CFG G there exists a path P from x to w such that x is not postdominated by w or any other node in P if and only if there exists a path Q from x to w in the $CDG(G)$, all of whose vertices are contained in P . See Figure 2.7.

Proof We prove the if part first. Consider the path $P=[x, p_1, \dots, p_n, w]$ in G . Since it is known that no element of P postdominates x in G , then, in particular, p_1 does not. Since, in addition, p_1 is an immediate successor of x , p_1 must be control dependent on x . Thus, the edge $\langle x, p_1 \rangle^{p_1}$ must be present in the CDG, thereby establishing a path in the CDG from x to p_1 containing only elements of P . Now consider the adjacent pair p_i and p_{i+1} . Assume a path exists in the CDG from x to p_i such that the path contains only vertices in P . Either p_{i+1} is control dependent on p_i or p_{i+1} postdominates p_i .

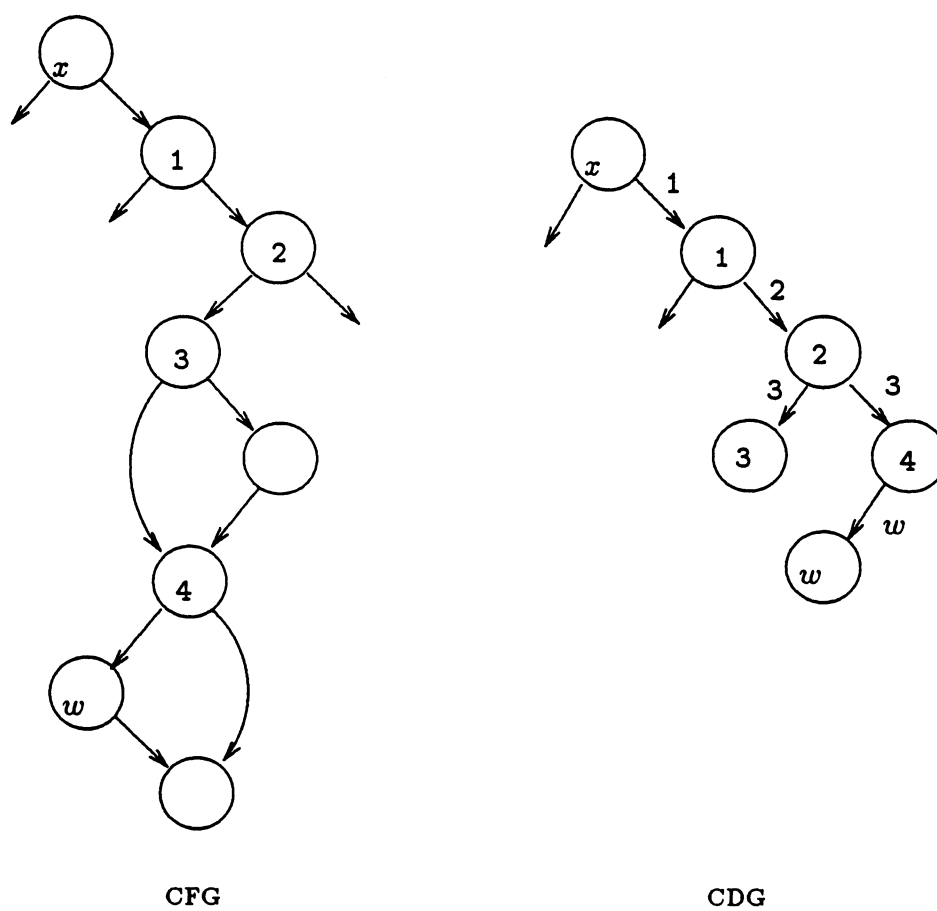


Figure 2.7 Example for Theorem 2.2

If the p_{i+1} postdominates p_i then p_{i+1} is control dependent on whatever nodes p_i is control dependent on. In either case, a path is established from x to p_{i+1} such that the path contains only vertices contained in P . This completes the inductive argument. The same argument applies to p_n and w . By induction the only-if portion of the theorem follows.

Now we prove the only if part of the theorem. Consider the path $Q = \langle x, q_1 \rangle, \langle q_1, q_2 \rangle, \dots, \langle q_n, w \rangle$ in the CDG. Since q_i can control determine q_{i+1} only if a path exists in the CFG from q_i to q_{i+1} , an inductive argument similar to that above proves that the path P must exist in the CFG. To prove that neither w nor any element of P postdominates x , suppose that some node $p_i = q_j$ postdominates x . Then p_i must post dominate every node reached by x in the CFG by a path not including p_i . In particular, p_i postdominates q_{j-1} . However p_i would not then be control dependent on q_{j-1} , causing a contradiction. The theorem follows. \square

To find all of the nodes that reach s in the CFG G , we can trace backwards from s along the edges of the CDG(G). By Theorem 2.2 only a node on this path can postdominate some node in G that it does not postdominate in G' . Hence, only a node on this path can control determine some node in G' that it did not control determine in G via the second condition of lemma 2.2. The next lemma further refines the conditions under which a node can be the source of a new edge in the CDG via the second condition in lemma 2.2.

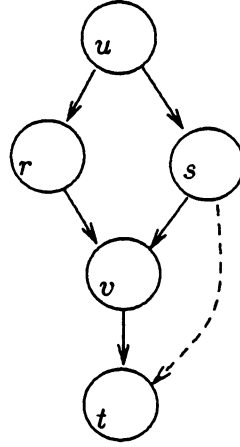


Figure 2.8 Example for Lemma 2.4

Lemma 2.4 Given CFGs G and G' , u can be the source of a new dependence, $\langle u, v \rangle^r \in \text{CDG}(G')$, via the second condition in Lemma 2.2 only if

1. u reaches s in G ,
2. r , an immediate successor of u , does not reach s in G , except through v , and
3. u does not reach t by any path not involving v in G .

See Figure 2.8

Proof By the second condition of Lemma 2.2, v postdominates u in G but not G' . This implies that there exists a path from u to STOP not involving v in G' , but not in G . Call this path P . Since the only difference between G and G' is the addition of (s, t) , P must include (s, t) . Hence, P contains a path from u to s . Thus, condition one is proved.

The existence of P implies that v does not postdominate s in G' . The control dependence $\langle u, v \rangle^r$ implies that v postdominates r in G' . Hence, any path from r to

s in G must include v , else r would not be postdominated by v in G' . Thus condition two is proved.

The second condition of Lemma 2.3 states that v post dominates r in G . Since the path P in G' includes a segment from t to STOP which does not include v and this segment is present in G , t is not postdominated by v in G . If u reaches t in G via a path not involving v then u would not be postdominated by v in G and the control dependence $\langle u, v \rangle^r$ would exist in the CDG of G , contradicting the premise of the lemma. Thus condition three is proved. \square

The set of nodes that can be the sink of a new dependence is limited as shown in the following lemma.

Lemma 2.5 Given CFGs G and G' a node v can be the sink of a new dependence, $\langle u, v \rangle^r \in \text{CDG}(G')$, via the second condition in Lemma 2.2 only if

1. v postdominates r , an immediate successor of u , in G ,
2. v does not postdominate t in G , and
3. v postdominates s in G .

See Figure 2.9.

Proof By the definition of control dependence we know that v postdominates r in G' . Lemma 2.1 implies this is true in G . This proves the first condition.

See Figure 2.9. The second condition of Lemma 2.2 states that u is post dominated by v in G , but not in G' . For the postdominator relation between u and v to change in this way, a new path from u to STOP, which does not involve v , must be created by the edge addition. Hence, the new path includes (s, t) . Since this path from u to STOP in G' includes t and not v , we know that v does not postdominate t in G . This proves the second condition.

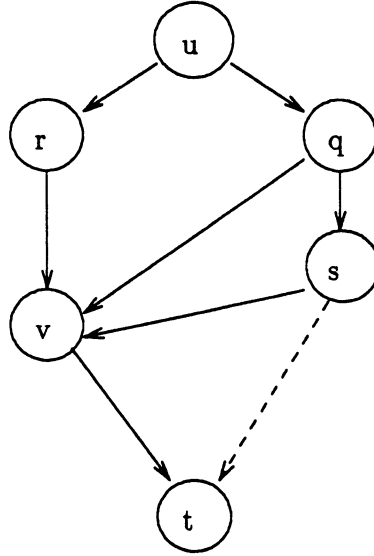


Figure 2.9 Example for Lemma 2.5

u reaches s in G via a path not involving v . Assume that v does not post dominate s in G . Then v can not postdominate u in G . This violates the second condition of Lemma 2.2. Thus the third condition is proved. \square

We are now ready to present the algorithm for adding edges to the CDG in response to the addition of an edge to the CFG. Lemma 2.3 divides all the control dependence edges that must be added in response to the addition of a control flow edge (s, t) into two distinct groups.

The first group of control dependences edges arise from the first condition of Lemma 2.2 and all have as their source s and as their label t . Lemma 2.3 states that the sink of every such new dependence post dominates t in the control flow graph before the update. Theorem 2.1 specifies how these may be found from a walk backward from t on the CDG before the update. During the walk from t , a control dependence with label t is added from s to all the nodes that postdominate t and do not postdominate s . This is done in CFAddCDAddnt shown in Algorithm 2.1. The walk ends when a node is found that reaches s in the CDG. Any node reached by the

```

procedure CFAddCDAddnt(curnode)
  if ReachesS(curnode) then return

  for every edge  $\langle x, \text{curnode} \rangle^l \in \text{CDG}(G)$  (incoming to this node)
    for every edge  $\langle x, v \rangle^l$  to the right of curnode with respect to x
      add  $\langle s, v \rangle^t$  to CDG
      call CFAddCDAddnt(x)
    endfor
  end CFAddCDAddnt

  call CFAddCDAddnt(t)

```

Algorithm 2.1 Adding Control Dependences due
to Condition 1 of Lemma 2.2

reverse walk from a node that reaches s would postdominate both s and t in G or neither. In any case it cannot be the source of a new control dependence.

The second group of control dependence edges arise from the second condition of Lemma 2.2 and are the subject of lemmas 2.4 and 2.5. Lemma 2.4 states that the source, u , of any new edge must reach s in G and that the label, r , of this edge must be a node that does not reach s in G . Hence, by theorem 2.2 we can find all the nodes that are the source of a new dependence by tracing backwards from s in the CDG. The same walk determines that r does not reach s . By Lemma 2.5 the sink, v , of a new dependence must postdominate s and r in G . Hence, Theorem 2.1 shows how to find all the possible sinks of a new dependence during the backward walk from s and prove that v postdominates r . By lemmas 2.4 and 2.5 and Theorem 2.2 the walk terminates at a node that reaches s in the CDG.

This is done in CFAddCDAddns shown in Algorithm 2.2. CFAddCDAddns uses a set U to maintain pairs containing possible sources of dependences and their immediate successors during the walk. The first member, u , of a pair is a possible source of a control dependence in G' . The second member, r , of the pair is an immediate

successor of u that has not been visited by the algorithm. Let *curnode* be the name of the node being visited in the algorithm. A node, v , that postdominates *curnode* in G postdominates the both the first and second member of every pair in U in G . But, in G' , only the second members of the pairs in U are postdominated by v . If r is postdominated by v , then a control dependence is added from u to v and labelled with r . The appearance of r as the second member of a pair in U during the execution of the algorithm does not imply that r does not reach s in the CDG. r might be visited at a later point in the execution. So the dependence $\langle u, v \rangle^r$ is only tentatively added until all the nodes that reach s in the CDG are visited. If r is visited during the walk and if it appears as the second member of a pair in U then the pair is deleted from U . At termination of the algorithm, if r is still present as the second member of the pair (u, r) in U , then it does not reach s in the CDG(G), and the tentatively added dependence edge is added in fact. See Algorithm 2.2.

Both CFAddCDAddnt and CFAddCDAddns make use of the functions ReachesS and ReachesT. ReachesS and ReachesT take a node in the CFG and return true if the node reaches s or t , respectively, in the CDG.

Theorem 2.3 CFAddCDAddnt in Algorithm 2.1 adds all control dependences that result from the first condition of Lemma 2.2.

Proof From Lemma 2.3 we know that any dependence added via the first condition of Lemma 2.2 is of the form $\langle s, v \rangle^t$ such that v postdominates t by not s in G . From Theorem 2.1 we know that if v postdominates t in G , then v is either a right sibling of t or the right sibling of some ancestor of t .

CFAddCDAddnt visits ancestors of t and searches for right siblings of the ancestor. As they are found a dependence to the right siblings are added. CFAddCDAddnt terminates when an ancestor would also post dominate s . This occurs when the node being walked is an ancestor of s . By Lemma 2.3 we know that no dependence is added to any such node. \square

```

procedure CFAddCDAddns(curnode,label,U,edge)
  Delete any edges with curnode as label from set NewEdges
  Delete any pair (x,curnode) from U
  for every edge  $\langle \text{curnode}, x \rangle^l$  to right of edge with respect to curnode with label label
  but left of t (if it exists as child)
    for every element of U, (u,y)
      add  $\langle u, x \rangle^y$  to NewEdges
    endfor
  endfor

  if ReachesT(curnode) then return

  for every immediate successor, l, of curnode which is not marked visited
    U = U + (curnode,l)
  endfor

  Mark curnode visited; Mark edge walked

  for every CDG edge incoming to curnode,  $\text{edge} = \langle u, \text{curnode} \rangle^l$ , which is not marked as walked
    call CFAddCDAddns(u,l,U,edge)
  endfor
end CFAddCDAddns

NewEdges =  $\emptyset$ 
U =  $\emptyset$ 
for every immediate successor of s, r
  U = U + (s,r)
endfor
for every CD edge incoming to s
   $\text{edge} = \langle u, s \rangle^l$ 
  call CFAddCDAddns(u,l,U,edge)
endfor

```

Algorithm 2.2 Adding control dependences due
to second condition of 2.2

Theorem 2.4 Algorithm 2.2 correctly adds to the CDG all the control dependences which result from the addition of a CFG edge (s, t) and have a label not equal to t .

Proof In order to prove this theorem we need to prove that every control dependence that should be added is found by the algorithm and that every control dependence the algorithm identifies truly exists in the CDG of G' .

We use lemmas 2.4 and 2.5 to show that every control dependence is added. Lemma 2.4 states that for any added dependence $\langle u, v \rangle^r$, u reaches s in the CFG and does not reach t except through v . By theorem 2.2, we know that this implies that u reaches s in the CDG of G . Hence a path, $Q=[q_1, q_2 \dots q_n]$, exists from u to s in the CDG. By induction on this path, we know that eventually CFAddCDAddns will be called with *curnode* equal to u . Thus we know that all possible sources of new CDG edges are examined. Each of these nodes and any immediate successor not yet visited are added together as a pair to the set U .

By Lemma 2.5, we know that v postdominates s , but not t in G . We also know that v post dominates u in G . The algorithm identifies all of these postdominator relationships during the same walk. Theorem 2.1 states which these are. The search ends when all the nodes yet to be found would also postdominate t . The set U contains all the nodes that v postdominates and are possible sources of control dependences in G' . When a possible sink is discovered, a dependence is added to it from all the first elements of pairs in U ; for each dependence, the label is set to be the second element of the pair. Therefore we know that all possible control dependences are found.

To prove that only correct control dependences are found, we prove the following two lemmas.

Lemma 2.6 Upon entry to CFAddCDAddns in Algorithm 2.2, *curnode* is not postdominated in G' by any node which is not equal to s and which postdominates s but not t in G .

Proof CFAddCDAdd is first called with *curnode* equal to s . Inside CFAddCDAddns, CFAddCDAddns is called with *curnode* equal to some node with a control dependence leading to *curnode*, (i.e., some parent of *curnode* in the CDG). Thus, a path exists in the CDG of G from *curnode* to s . By Theorem 2.2, this implies that, in the CFG G , *curnode* reaches s . Suppose that the lemma is false and *curnode* is postdominated in G' by some node v which postdominates s , but not t in G . Since v does not postdominate t , every in G' path from *curnode* to s must involve v . Since the only change is the addition of (s, t) , this must also be true in G . By Theorem 2.2, *curnode* cannot reach s in the CDG of G unless *curnode* reaches s by a path in G which does not include a node, v , that postdominates *curnode*. Thus, a contradiction is shown, and the lemma is proved. \square

Lemma 2.7 If CFAddCDAddns is never called with *curnode* equal to some node r , or with edge labelled by r , and an immediate predecessor of r is visited and does not reach t in the CDG of G , then r is postdominated by the same nodes in G' as in G .

Proof Since CFAddCDAddns is never called with *curnode* equal to r , one of the following must be true in G .

1. r reaches t
2. r does not reach s
3. r reaches s only by paths that include nodes that postdominate r .
4. r is postdominated by s .

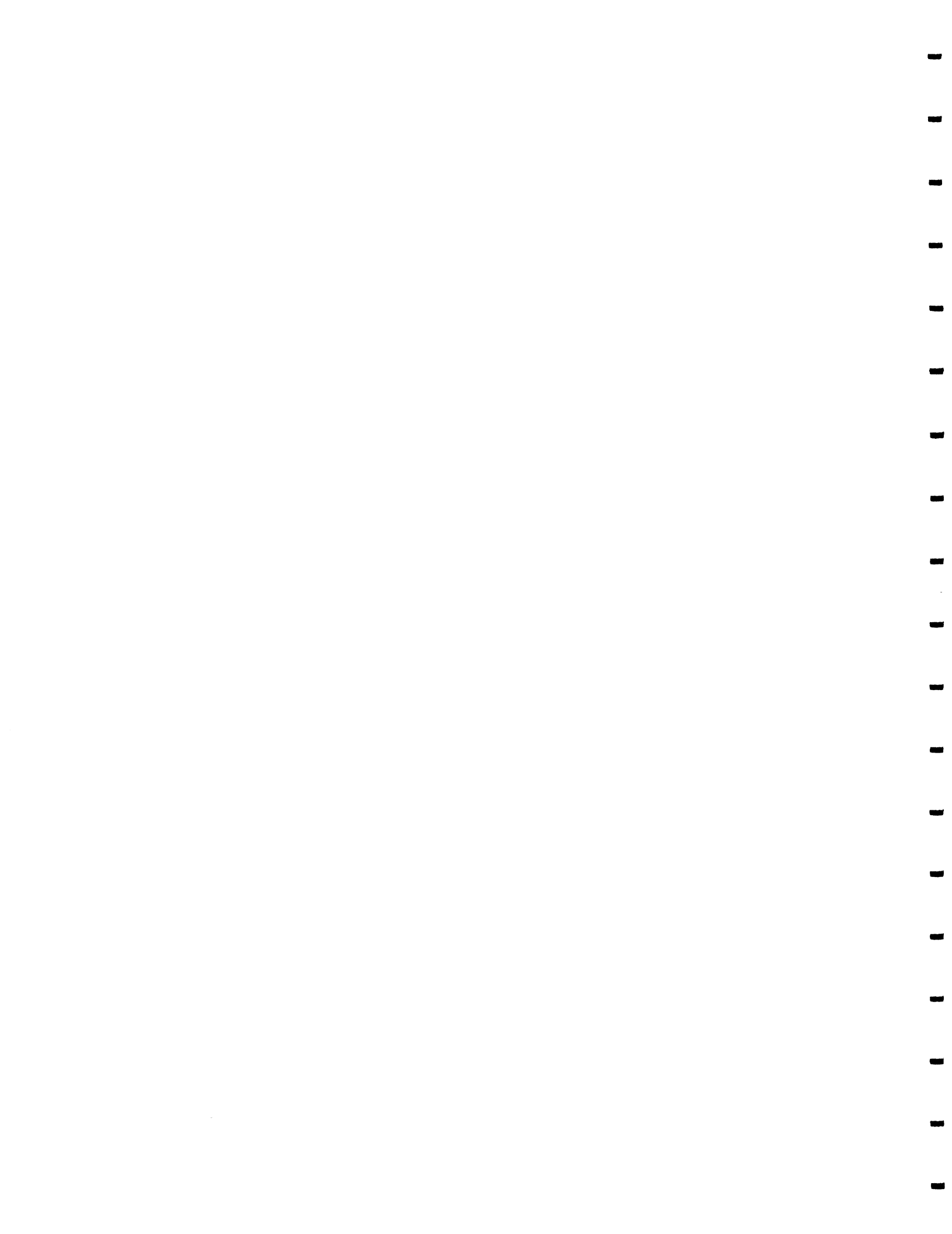
r cannot reach t in the CDG of G because then its immediate predecessor would reach t in the CDG of G , contradicting the premise of the lemma. If r does not reach s in G then the addition of (s, t) cannot add a path to STOP such that r is not

postdominated by some node in G' that postdominated r in G . Hence, we need only consider cases 3 and 4 above where r reaches s but is postdominated by s or by nodes on every path to s .

Let u be r 's visited immediate predecessor in G . Since u is visited, it is not postdominated by s in G . Thus, if r is postdominated by s in G , then the control dependence $\langle u, s \rangle^r$ would exist in the CDG of G . The edge $\langle u, s \rangle^r$ would be walked and the premise of the lemma is contradicted. Thus, case 4 cannot occur.

Finally, suppose that there is some node q that postdominates r and appears on every path to s from r . Let q be the last such node on the path. Then we can consider two cases. First, if u is not postdominated by q , then q is control dependent on u with label r . A path in the CDG of G exists from q to s , so the premise of the lemma is contradicted when the edge from u to q with label r is walked. Second, suppose that u is postdominated by q . We must consider two cases. First, suppose that s is postdominated by q . This implies that an infinite loop exists in the CFG G , but we have assumed there are no infinite loops in G . Thus a contradiction occurs. Second, if s is not postdominated by q but u is, then all paths from u to s must involve q . Since q postdominates u Theorem 2.2 implies that there is no path in the CDG of G from s to u and u could never be visited. This proves the last contradiction and the lemma follows. \square

These two lemmas directly prove the necessary result that if a node n is visited by CFAddCDAddns, and one of n 's immediate successors is not visited, then a control dependence is added between n and the nodes postdominating n 's immediate successor that do not also postdominate t . Since this is just what the algorithm does, the correctness theorem follows. \square



2.2.2 Control Dependence Deletion

By switching the roles of G and G' in lemma 2.2, a corresponding lemma can be derived to give the conditions under which a control dependence is deleted upon the addition of an edge.

Lemma 2.8 Given the CFGs G and G' , the control dependence edge $\langle u, v \rangle^r$ is present in $CDG(G)$ and not in $CDG(G')$ only if one of the following is true

1. v postdominates r , an immediate successor of u , in G but not in G' .
2. v postdominates u in G' but not in G .

From Lemma 2.1 we know that the second condition cannot occur in a complete CFG. The first condition results in the following lemma.

Lemma 2.9 Given the CFGs G and G' , the control dependence $\langle u, v \rangle^r$ exists in $CDG(G)$ and not in $CDG(G')$ only if

1. r reaches s in G ,
2. v postdominates s in G , and
3. v does not postdominate t in G .

See Figure 2.10.

Proof Condition 1 of Lemma 2.8 requires r to be postdominated by v in G but not in G' . This implies that a path not involving v exists from r to STOP in G' . This path must involve (s, t) otherwise r would not be postdominated by v in G . Thus, r must reach s in G .

Since r reaches s in G' via a path not involving v and v postdominates r , v must postdominate s in G .

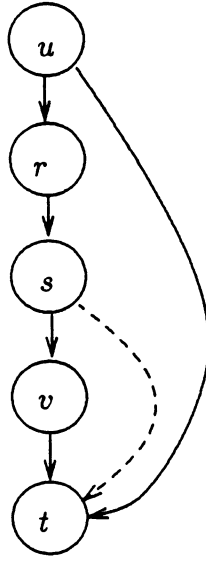


Figure 2.10 Example for 2.9

If v postdominates t in G , then r is postdominated by v in G' . This contradicts condition 1 in Lemma 2.8 Hence, t is not postdominated by v in G . \square

We observe that the nodes v which are the sinks of the deleted control dependences are exactly those nodes that were sinks of added control dependences. Also, the nodes r labelling the deleted edges and whose immediate predecessors are the source of the deleted edges reach s are the same nodes that CFAddCDAddns considers as sources of new control dependence edges. Hence, a walk similar to that which occurs during CDG edge addition will identify these nodes.

The algorithm for CDG edge deletion in response to a CFG edge addition consists of walking the CDG of G backward and if

1. the dependence $\langle u, v \rangle^r$ exists where u is the node being visited,
2. r is the label of the edge traveled backward to u , and
3. v postdominates s but not t ,

then delete the edge $\langle u, v \rangle^r$ from the CDG.

program CFAddCDDel

procedure Visit(*curnode*,*label*)
 if ReachesT(*curnode*) **then return**
 for every edge $\langle u, \text{curnode} \rangle^l \in \text{CDG}(G)$
 for every right sibling with label *l*
 $(\langle u, v \rangle^l \in \text{CDG to right of } \langle u, \text{curnode} \rangle^l)$
 delete $\langle u, v \rangle^l$ from CDG
 endfor
 call Visit(*u*)
 endfor
end Visit

for every CD edge $\langle u, s \rangle^l$
 call Visit(*s*, $\langle u, s \rangle^l$)
endfor

end CFAddCDDel

Algorithm 2.3 CDG Edge Deletion in Response to CFG Edge Addition

Theorem 2.5 Algorithm 2.3 correctly identifies the control dependences deleted from the CDG of G in response to the addition of a control flow edge.

Proof By Lemma 2.6, the nodes visited by the walk of the algorithm are not postdominated in G' by any node v that postdominates s in G . The algorithm deletes all dependences to v with labels equal to a node visited by the walk. By the definition of control dependence, all these dependences must be deleted. By lemmas 2.8 and 2.9, these edges are all of the edges that must be deleted. \square

The walks of the algorithms for CDG edge addition and deletion in response to CFG edge addition are identical. In fact, the deletion of a control dependence edge implies the addition of another control dependence edge.

2.3 CFG Edge Deletion

The update of the CDG in response to the deletion of CFG nodes during editing is a straightforward inverse of the method for addition of CFG nodes. If a node has a single immediate successor and a single immediate predecessor, then it can be deleted from the CFG and CDG and any edge leading to deleted node is deleted from the CDG as well. If the name of the deleted node exists as a label on any CDG edge which is not deleted when the node is deleted, then that label must be replaced with the name of the immediate successor of the deleted node in the CFG.

If two nodes x and w have a single edge between them, that is, if x is the sole immediate predecessor of w and w is the sole immediate successor of x , then they can be joined. The node w will retain its identifier in order to avoid updating edges for which it is the source. Node x must be deleted from the CDG and any edges for which it is the sink must be redirected to w . If x appears as a label on any edge then x must be replaced by w in the label.

We now turn to the update of the CDG in response to the deletion of a control flow edge. For the rest of this section, $G=(V,E)$ is a complete CFG and $G'=(V,E-(s,t))$ is the corresponding complete CFG with a single edge deleted.

2.3.1 Control Dependence Addition

Lemma 2.10 corresponds to Lemma 2.1 for CFG edge deletion.

Lemma 2.10 Let $G=(V,E)$ and $G'=(V,E-(s,t))$. If a node x is postdominated by a node w in G then x is postdominated by w in G' .

Lemma 2.10 and Lemma 2.2 imply the next lemma which corresponds to Lemma 2.3.

Lemma 2.11 Given complete graphs G and G' , $\langle u, v \rangle^r \in \text{CDG}(G')$ and $\langle u, v \rangle^r \notin \text{CDG}(G)$ only if r is postdominated by v in G' and not in G .

Lemma 2.4 serves to restrict the nodes that can be the source of a dependence formed as a result of a control flow edge addition. For updating in response to the deletion of a control flow edge, we find it more natural to state a restriction on the nodes that can appear as labels on a new control dependence edge. The number of potential nodes for r in Lemma 2.11 can be restricted as follows.

Lemma 2.12 Given complete graphs G and G' , a node r is postdominated by v in G' and not in G only if a path not involving v exists in G from r to s .

Proof By Lemma 2.11, r is not postdominated by v in G . Hence, there must exist some path in G not involving v from r to STOP. Since this path is not present in G' , it must include the edge (s,t) . Thus, a path from r to s , not involving v , is shown to exist in G . \square

The possible sinks of a new dependence are described by the next lemma.

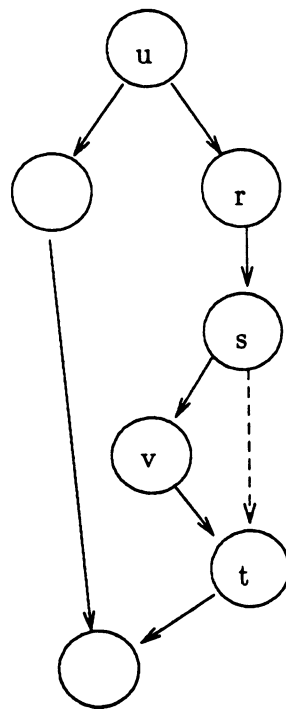


Figure 2.11 Dependence formed by postdomination of immediate successor

Lemma 2.13 Given complete graphs G and G' , $\langle u, v \rangle^r \in \text{CDG}(G')$ and $\langle u, v \rangle^r \notin \text{CDG}(G)$ only if

1. s is postdominated by v in G' , and
2. v is control dependent on s in G .

Proof By Lemma 2.12 a path not involving v from r to s exists in G . Since only (s, t) is deleted, the portion of the path up to s also exists in G' . For r to be postdominated by v in G' , all the nodes that r can reach via a path not involving v must also be postdominated by v . Hence, s is postdominated by v in G' and the first condition is proved.

All paths in G from r to STOP that do not include (s, t) are also present in G' . If any of these paths do not include v in G , then r is not postdominated by v in G' . Therefore, the only paths in G from r to STOP that do not include v involve s . If s were postdominated by v in G , r would also be postdominated by v in G . Thus, s is not postdominated by v in G . The first condition implies that any immediate successor of s , $w \neq t$, is postdominated by v in G . This and the fact that s is not postdominated by v in G imply that $\langle s, v \rangle^w \in \text{CDG}$ of G for any immediate successor of s , w . The nodes v that postdominate s in G' , but not G , are all the possible sinks of new control dependences in G' . \square

The next lemma restricts the nodes that can be a source of a new control dependence so that they may be found via a simple walk from s following the dependence edges backwards.

Lemma 2.14 Given a control dependence $\langle u, v \rangle^r$, such that $\langle u, v \rangle^r \in G'$ and $\langle u, v \rangle^r \notin \text{CDG}(G)$, a path exists from u to s in $\text{CDG}(G)$.

Proof From Lemmas 2.11 and 2.12 we know that, given a control dependence $\langle u, v \rangle^r$ deleted from $\text{CDG}(G)$ to form $\text{CDG}(G')$, a path, P , exists from r to s in G . From Theorem 2.2 we know that either

1. r is an ancestor of s in the $CDG(G)$,
2. r is postdominated by s in G , or
3. r is postdominated by some node, w , on the path from r to s in G .

If 1, then, since the edge $\langle u, r \rangle^r$ is present in $CDG(G)$, u is an ancestor of s in $CDG(G)$.

By Lemma 2.13 we know that v postdominates s in G' . u is not postdominated by s in G because if it were then it be postdominated by s in G' and hence u would be postdominated by v in G' and $\langle u, v \rangle^r$ would not exist in $CDG(G')$. Thus 2 implies that the control dependence $\langle u, s \rangle^r$ is present in $CDG(G)$. Hence, a path exists from u to s in $CDG(G')$.

If 3, then assume, without loss of generality, that w is the last node on P that postdominates r in G . Since the segment of P beginning at w is a path from w to s , and there are no elements of this segment of P that postdominate w in G , either condition 1 or 2 above apply with w taking the place of r . Thus, a path will exist from w 's immediate predecessor in P to s in $CDG(G)$. By induction, a path from w to s is shown to exist in $CDG(G)$. \square

We now present the algorithm. Let V be an ordered list of nodes such that V contains the name of every node v for which a control dependence edge exists in $CDG(G)$ with label equal to $w \neq t$ for every immediate successor w of s . That is, v is contained in V if and only if for every immediate successor $w \neq t$ of s the control dependence edge $\langle s, v \rangle^w$ is present in $CDG(G)$. The members of the list are ordered so that every node postdominates the nodes that come after it in the list. If r is not postdominated by some node v on the list then it is not postdominated by any node that follows v in the list. Similarly, if u is postdominated by a node v on the list then it is also postdominated by every node preceding v on the list.

Consider a node, u , that control determines s . Let r_1 be the label of the control dependence from u to s . r_1 is an immediate successor of u that is postdominated by

s. For every node $v \in G$, if the control dependence $\langle u, v \rangle^{r_1}$ does not already exist in the CDG of G and if some immediate successor of u , $r_2 \neq r_1$, is not postdominated by v in G' , then the control dependence $\langle u, v \rangle^{r_1}$ must be added to the CDG. No node that reaches u in $\text{CDG}(G)$ will be postdominated by v in G' since u is not postdominated by v in G' . Hence, none of the nodes reaching u can be the label of a new control dependence with the sink at v . Thus the addition of a dependence to v is the termination condition for the search for dependences to v . In our algorithm we will delete v from V upon adding a dependence $\langle u, v \rangle^r$. The algorithm terminates when V is empty.

On the other hand, it is possible that all of u 's immediate successors not equal to r_1 are postdominated by v . In this case, u is postdominated by v in G' . We can then consider all the nodes on which u is control dependent as potential sources of a new control dependence on v . In this case, v is not deleted from V and the algorithm continues.

A naive way to see if all of u 's immediate successors not equal to r_1 are postdominated by v is to search for the control dependence $\langle u, v \rangle^{r_i}$ for all of u 's immediate successors $r_i \neq r_1$. The problem with this scheme is illustrated by Figure 2.12. In the figure we see that node 1 becomes postdominated by v upon the deletion of (s, t) . However, in the $\text{CDG}(G)$, there is no control dependence from 1 to v with the label 2, because 2 is not post dominated by v in G . In fact, there is no way to prove that either 1 or 2 becomes postdominated by v by examining 1 and 2 and the nodes that they control determine separately. Instead we must consider 1 and 2 as a unit. This is true of multi-exit loops in general; because any loop exit is postdominated only by those nodes that postdominate all of the loop exits, all of the loop exits must be considered together. In our example, looking at 1 and 2 together, we can conclude at 1 that both 1 and 2 are postdominated by v in G' since 2 control determines v . We thus avoid placing an incorrect control dependence into the CDG and can move

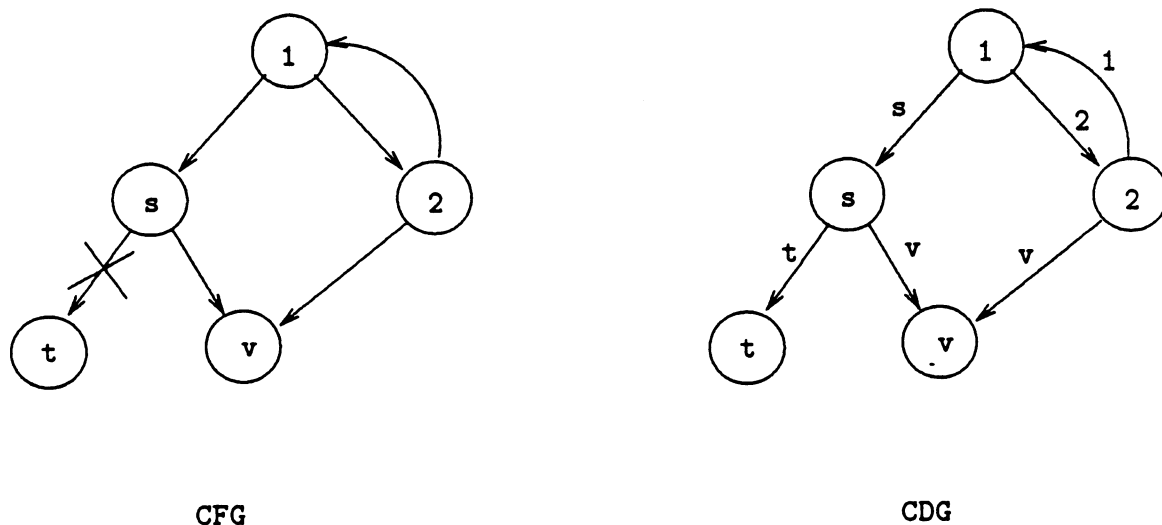


Figure 2.12 A multi-exit loop

on to examine the node control determining 1 as a possible source of a new control dependence.

Some nodes or loops will still have immediate successors that will not be postdominated by v . These nodes can be discovered during the update process as follows. For each of these immediate successors, a new control dependence is added when the walk from s traverses the CDG edge to the immediate successor. When the last immediate successor is reached, control dependences will be already present in the CDG for all the other immediate successors. At this point, it can be discovered that u is post dominated by v . The control dependences previously added to the other immediate successors can be deleted and the process can continue onto the nodes which control determine u .

The algorithm shown in Algorithm 2.4 follows the preceding discussion. The CDG is walked backward from s examining each node n to see whether n newly control determines any node in V or whether n is newly postdominated by the nodes in V . The walk continues along all possible paths backwards from s . Each time a control dependence is added, the sink of the dependence is deleted from V . The walk ends

when V is empty. When multi-exit loops are encountered, in order to determine post-domination, the exits of the loop and their dependent nodes are considered together. The loops are recognized as they are created during the CDG update in response to CFG edge addition. See Figure 2.13.

Theorem 2.6 A control dependence $\langle u, v \rangle^r \in \text{CDG}(G')$ and $\langle u, v \rangle^r \notin \text{CDG}(G)$ if and only if CFDeICDAdd in Algorithm 2.4 will add $\langle u, v \rangle^r$ to the CDG.

Proof From Lemma 2.14 we know that a path exists from u to s in $\text{CDG}(G)$. This path specifies a call chain from s to u . This chain will be followed and visit called with *curnode* equal to u and V containing v unless a dependence $\langle q, v \rangle^l$ is added when visit is called with *curnode* equal to q where q is some node in the call chain. Suppose $\langle q, v \rangle^l$ is added to the CDG. This implies that q is not postdominated by v in G' . Thus, since r reaches q in the CFG G , r is not postdominated by v in G' and $\langle u, v \rangle^r \notin \text{CDG}(G')$ and a contradiction proved. Hence visit will be called with *curnode* equal to u , *label* equal to r , and v contained in V . Since $\langle u, v \rangle^r \in \text{CDG}(G')$, there is at least one immediate successor, l , of u that is not postdominated by v , and hence the $\langle u, v \rangle^l$ will not be present in $\text{CDG}(G)$. Thus, the test for edges to v from u for all of u 's immediate successors will fail and $\langle u, v \rangle^r$ will be added to the CDG.

Upon every call to visit, *label* is postdominated by every element of V . Hence, the existence of another immediate successor of *curnode* l which is not postdominated by v in G' is sufficient to prove that $\langle \text{curnode}, v \rangle^{\text{label}} \in \text{CDG}(G')$ where $v \in V$. visit determines this by testing if the control dependence $\langle \text{curnode}, v \rangle^l$ is absent from the CDG for some immediate successor, l , of *curnode*. If later, visit is called with *label* equal to l , then visit will determine that *curnode* is postdominated by v and the control dependence $\langle \text{curnode}, v \rangle^{\text{label}}$ will be deleted from the CDG.

Thus, by the time of termination, a new control dependence edge is added to the CDG if and only if it is present in $\text{CDG}(G')$.

program CFDeICDAdd:

Create the ordered list V containing the nodes control dependent on s in G with labels for every immediate successor of s not equal to t in G .

Thus V contains those nodes that postdominate s in G' but not G .

procedure visit($currnode, label, V$)

$upostdominated = false$

do until $upostdominated$ or V is empty

$v =$ last element of V

if $currnode$ is an exit from a multi-exit loop **then do**

if $\langle enode, v \rangle^l \in CDG$ for all exit tests, $enode$ and exit branches, l

then $looppostdom = true$

endif

if $looppostdom$ or $\langle currnode, v \rangle^l \in CDG$ for all successors, $l \neq label$, of $currnode$

then do

for all u, r where $\langle u, currnode \rangle^r \in CDG$

call visit(u, r, V)

endif

$upostdominated = true$

 /* Deletion of dependences will be shown in Algorithm 2.5 */

endif

else

 add $\langle currnode, v \rangle^{label}$ to CDG

$V = V - v$

endelse

end visit

for all u, r where $\langle u, s \rangle^r \in CDG$

call visit(u, r, V)

endfor

end CFDeICDAdd

Algorithm 2.4 Control Dependence Additions in
response to CFG Edge Deletion

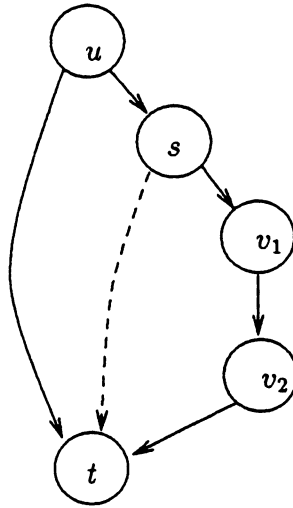


Figure 2.13 Deletion of a Control Flow Edge

□

2.3.2 Control Dependence Deletion

Lemma 2.8 specifies the conditions for the elimination of a control dependence due to the addition of control flow edge. The lemma applies without change to the case of a deletion of a control flow edge. Lemma 2.8 plus Lemma 2.10, which specifies the possible change to the postdominator relation as a result of the deletion of a control flow edge, lead to the following lemma.

Lemma 2.15 Given G and G' , $\langle u, v \rangle^r \in \text{CDG}(G)$ but not G' only if

1. r is no longer an immediate successor of u , or
2. u is postdominated by v in G' .

Assuming G and G' are complete, the first condition implies that $u = s$, $r = t$, and v is a node postdominating t . The deletion of the CFG edge (s, t) is sufficient

for the removal of any dependence edge from s with label t in the CDG. Thus, any dependence $\langle s, v \rangle^t$ must be deleted.

The second condition implies that all of u 's immediate successors are postdominated by v . The existence of the control dependence $\langle u, v \rangle^r$ in the CDG of G implies that r is postdominated by v in G , but some other immediate successor of u was not postdominated by v in G . Any such immediate successor becomes post dominated by v in G' . Lemma 2.12 specifies how a node can become post dominated by another as a result of a control flow edge deletion. A lemma similar to Lemma 2.13 describes the sink of any deleted dependence.

Lemma 2.16 Given complete graphs G and G' , $\langle u, v \rangle^r \in \text{CDG}(G)$ and $\langle u, v \rangle^r \notin \text{CDG}(G')$ only if

1. s is postdominated by v in G' ,
2. s is not postdominated by v in G , and
3. v is control dependent on s in G .

Proof The edge $\langle u, v \rangle^r$ is deleted from the CDG only if v postdominates u in G' and not in G . This implies that some immediate successor, r , of u is postdominated by v in G' and not G .

The rest of the proof is identical to the proof of Lemma 2.13. □

The same walk as for control dependence addition contained in Algorithm 2.4 can be used to find when all the immediate successors of some node u are postdominated by v as a result of the edit. The difference is that we now delete dependences from u when all its immediate successors are postdominated by v , instead of continuing the search for new control dependences.

Lemma 2.17 Given a control dependence $\langle u, v \rangle^r$, such that $\langle u, v \rangle^r \in G$ and $\langle u, v \rangle^r \notin \text{CDG}(G')$ as a result of condition 2 of Lemma 2.15, a path exists from u to s in $\text{CDG}(G)$.

Proof If u is equal to s then the lemma is true. If $\langle u, v \rangle^r$ is deleted as a result of condition 2 of Lemma 2.15 and u is not equal to s , then u is postdominated by v in G' but not G . By Lemma 2.12, a path exists from u to s in G .

The rest of the proof is similar to that for Lemma 2.14. \square

Algorithm 2.5 shows the deletion of control dependences in response to the deletion of a control flow edge.

Theorem 2.7 A control dependence $\langle u, v \rangle^r \in \text{CDG}(G)$ and $\langle u, v \rangle^r \notin \text{CDG}(G')$ if and only if CFDeICDDel in Algorithm 2.5 will delete $\langle u, v \rangle^r$ from the CDG.

Proof Dependences deleted as a result of condition 1 in Lemma 2.15 are deleted explicitly by CFDeICDDel when control dependences of form $\langle s, v \rangle^t$ are deleted from the CDG. Since t is no longer an immediate successor of s , any control dependence of form $\langle s, v \rangle^t$ is deleted from the control dependence relation upon the removal of the control flow edge (s, t) .

CFDeICDDel forms V from all the nodes that are control dependent on s in G and which postdominate s in G' . Thus, by Lemma 2.16, if $\langle u, v \rangle^r$ is deleted as a result of condition 2 in Lemma 2.15, then v is contained in V when V is formed.

By Lemma 2.17 we know that a path exists from u to s in $\text{CDG}(G)$. Since u is postdominated by v in G' , any node on the path in the CDG from u to s is also postdominated by v in G' . Hence at every node, q , on the reverse path from s to u dependences will be found to v for all of the node's immediate successors not equal to the label of the edge traveled to q and v will not be deleted from V . Thus, a call chain is shown such that visit will be called with *currnode* equal to u , *label* equal to r , and v contained in V . Thus, at u a test is made for the existence of control dependences $\langle u, v \rangle^w$ for all of u 's immediate successors $w \neq l$ where l is the label of the edge traveled to u . If these dependences are found, since we know that l is

```

program CFDelCDDel
delete all edges  $\langle s, v \rangle^t$  from CDG
Create the ordered list  $V$  containing the nodes control dependent on  $s$  in  $G$ 
with labels for every immediate successor of  $s$  not equal to  $t$  in  $G$ .
Thus  $V$  contains those nodes that postdominate  $s$  in  $G'$  but not  $G$ .

Delete all the dependences from  $s$  to a member of  $V$ 

for all  $u, r$  where  $\langle u, s \rangle^r \in \text{CDG}$ 
    call visit( $u, r$ )
endfor

procedure visit( $curnode, label$ )

     $unotpostdominated = \text{false}$ 
    for every element,  $v$ , of  $V$ , in order do until  $unotpostdominated$ 
        if  $\langle curnode, v \rangle^l \in \text{CDG}$  for all immediate successors,  $l$ , of  $curnode \neq \text{label}$ 
            then delete all  $\langle curnode, v \rangle^l$  from CDG
        else do
             $unotpostdominated = \text{true}$ 
            delete from  $V$ ,  $v$  and all the elements following  $v$  in  $V$ 
        endifor-until

    for all  $u, r$  where  $\langle u, curnode \rangle^r \in \text{CDG}$ 
        call visit( $u, r$ )
    endfor
end visit

end CFDelCDDel

```

Algorithm 2.5 Control Dependence Deletion in
Response to Control Flow Edge Deletion

postdominated by v , all dependences from u to v with any label, and in particular label r are deleted.

□

2.4 Complexity Analysis

Algorithms 2.1 and 2.2 involve walks on two distinct sets of nodes in the CDG. The first set, η_t , contains the nodes in the CDG which can reach t but not s in the CDG. CFAddCDAddnt is called for each member of this set.

For each node, *curnode*, on which CFAddCDAddnt is called, it adds control dependences to the nodes which postdominate *curnode* in G . This involves finding all of *curnode*'s right siblings. To do this, it examines all the control dependence edges for which *curnode* is a sink. For each of these edges the algorithm determines whether any edges out of the source are to the right of the incoming edge to *curnode*. The sinks of all the edges to the right postdominate *curnode* in G . A control dependence edge must be added for each of these nodes. The amount of work required for this determination is bounded by the sum of the out degrees of all the nodes which control determine *curnode*. This, in turn, is loosely bounded by the product of the in-degree of *curnodes* parents in the CDG. Over the whole graph G , we can bound the response time to an edit to be proportional to the product of the maximum in-degree and out-degree of nodes in the CDG times the size of η_t , $\|\eta_t\|$.

The second set, η_s , walked by CFAddCDAddns, contains the set of nodes that reach s , but not t , in the CDG of G . CFAddCDAddns is called on all of the nodes in η_s . The sinks of the new control dependence edges are the children of *curnode* to the right of the edge walked to *curnode*. Dependences are added to each of these nodes. The sources of the dependences are the members of U , U contains node pairs consisting of nodes reaching s and their immediate successors that do not reach s .

The time for the addition of these dependences is bound by the number of members of U , $\|U\|$ times the number of children of *curnode*.

CFAddCDDel in Algorithm 2.3 visits the same set of nodes as CFAddCDAAdd and thus their time complexities are equal.

Algorithms 2.4 and 2.5 update the CDG in response to the deletion of a control flow edge. It walks the set of nodes η_s . During each visit, all of *curnode*'s immediate successors are examined to see if a control dependence edge exists from *curnode* to v with the label on the edge equal to *curnode*'s immediate successor. If an immediate successor is found for which this the edge does not exist then a dependence edge with labelled by the immediate successor is added from *curnode* to all the nodes control dependent on s , in G . Otherwise dependences for *curnode* to the nodes control dependent on s in G are deleted. Thus, during a single visit, the work is bounded by the out-degree of s in the CDG of G . Thus, in response to the deletion of a CFG edge (s, t) , the worst case complexity for an update is $O(\|\eta_s\| \cdot \text{out-degree}(s))$.

Chapter 3

Data Dependence

3.1 Definitions

As defined by Allen [All83], a data dependence arises when one statement can affect the result of another statement through a change to a memory location accessed by both statements. A data dependence between two statements can be characterized by the types of the two memory accesses involved.

Definition 3.1 Given two statements $S1$ and $S2$ such that the execution of $S1$ precedes $S2$,

- a *true dependence* exists from $S1$ to $S2$ if $S1$ assigns to a memory location that $S2$ uses;
- an *anti dependence* exists from $S1$ to $S2$ if $S1$ uses a memory location that $S2$ assigns;
- an *output dependence* exists from $S1$ to $S2$ if $S1$ and $S2$ both assign to the same memory location; and
- an *input dependence* exists from $S1$ to $S2$ if $S1$ and $S2$ both use the same memory location. ¹

Note that “a dependence exists from $S1$ to $S2$ ” is equivalent to “ $S2$ depends on $S1$ ”. This can be written $S1\sigma_t S2$ with the subscript, t , denoting the type of dependence.

¹Under most models of parallel computation the order of two reads from a single memory location cannot affect the result of any computation and thus would not truly represent a dependence or execution order constraint. Nonetheless, these so-called dependences have become useful in other optimizations and we include them for completeness' sake.

The rest of our discussion will concentrate on the calculation of true dependences. Generalization to other types of dependences is discussed in section 3.4.

3.1.1 Scalars and Arrays

True dependences correspond to def-use chains defined for scalars. But array variables are much more difficult to analyze. Consider the following example.

```

DO I = 1, N
    DO J = 1, N
S1      A(I,J) = ...
S2      ... = A(I,J+1)
    ENDDO
ENDDO

```

If we treat array variables like scalar variables, then we find a true dependence from S1 to S2. In fact this true dependence does not exist, since no location that is written by S1 is afterward used by S2.

The dependence relation for subscripted variables is fundamentally different from that for scalar variables because the name of the variable alone does not determine the memory location involved in an array reference; the memory location involved in an array reference is also determined by the subscript expression in the reference. For two references to access the same member of an array, their subscript expressions must be equal at the times of the respective references.

Array references may appear within loops and their subscripts may contain variables whose values change during execution of the loop. Consider two subscripted references inside a single loop body with induction variable I , a definition, $A(f(I))$, and a later use, $A(g(I))$.

```

DO I = L, U
    A(f(I)) = ...
    ...= A(g(I))
ENDDO

```

If f equals g , then the two references access the same memory location during every iteration. This results in a *loop-independent* true dependence, so named because the dependence exists regardless of any action of the loop. In contrast, if f is not equal to g , but $f(j)$ equals $g(k)$ for some j less than k within the bounds of the loop induction variable I , then the two references access the same memory location in different iterations of the loop and a *loop-carried* true dependence exists from the first to the second reference. We say that the loop “carries” the dependence. We define the *threshold* or *distance* of the dependence to be the smallest positive $t = k - j$, where j and k are contained within the bounds for I , and $f(j) = g(k)$ ². The threshold represents the minimum number of iterations between which the two references access the same memory location. If $f(j + c) = g(k)$ for a constant c and all values of j and k , then we call c a *constant threshold*. Constant thresholds will be important in the algorithms presented later in this chapter.

In the above example, the two references are contained in a single loop and hence there is no ambiguity concerning which loop is carrying the dependence. If the two references of a loop-carried dependence are contained in more than one loop, we define the *level* of the loop carrying the dependence to be the level of the lowest numbered loop for which the references forming the source and sink of the dependence occur on different iterations and where levels are numbered from outermost to innermost.

Subscript expressions can be quite complex. But even for the relatively simple case of affine expressions involving induction variables of the surrounding loops, determining exactly when a dependence exists between two statements with references to a

²More formal definitions are given by Allen and Kennedy [AK87].

common subscripted variable requires integer programming techniques [All83, Wol82]. As a consequence, a conservative approximation is used. All possible dependences between subscripted variables are assumed to exist unless a simpler analysis of their subscripts proves independence. Typically, attention is restricted to pairs of array references with subscript expressions which are affine functions of the loop induction variables. All other pairs of array references are assumed to be dependent. One or more *independence tests* are applied. These tests cast the problem as proving that there are no integer solutions for the loop induction variables in the equation obtained by setting the difference of the subscript expressions equal to zero[Ban79].

For instance, in the previous example, where the induction variable of the loop ranges from L to U, we determine whether there exist any integral solutions for x and y in the following equation.

$$\begin{aligned} f(x) - g(y) &= 0; \\ L \leq x \leq y \leq U \end{aligned}$$

If we can prove that no solution exists, then no true dependence can exist between the references.

3.1.2 Independence Tests

Independence tests may be divided broadly into three types;

1. range tests,
2. integral tests, and
3. exact tests.

Range tests prove that the maximum and minimum values of two subscript expressions do not overlap in the range of the possible values for the induction variables contained in the subscript expressions. This is done by finding the maximum and

minimum values for the two expressions over the range of the induction variables and comparing them. Banerjee's test is an example of this type of test[Ban79]. In the following,

```

DO I = 1, N
S1      A(I) = ...
S2      ...= A(I + N)
ENDDO

```

the maximum value of the subscript in the definition in S1 is less than the minimum value of the subscript in the use in S2. Thus, Banerjee's test (as implemented at Rice) would conclude independence. Banerjee's test is formulated to work correctly in nested loops with subscript expressions involving more than one induction variable.

Integral tests prove that the only possible values of induction variables which could result in equality of the two subscript expressions are nonintegral. Since the induction variables are necessarily integer quantities, this proves independence. The GCD test described by Cohagen[Coh73] is an example of this type. In

```

DO I = 1, N
S1      A(2I) = ...
S2      ...= A(2I+1)
ENDDO

```

the definition in S1 only writes to the even numbered array elements and the use in S2 only reads the odd numbered elements. Therefore, they are independent. The general case is proved by comparing the greatest common divisor (GCD) of the coefficients of the induction variables in the subscript expressions with the difference of the constant terms of the expressions. If the difference is not evenly divided by the GCD then independence is proved.

Finally, there are a number of *exact tests*. Although exacts tests are too expensive for the general case, a number of special cases of subscript expressions are susceptible

to proving inexpensively that there exist values of the induction variables which result in equality of the subscript expressions. Since the existence of such values of the induction variables proves a dependence, these tests are called "exact" to distinguish them from the approximate range and integral tests.

Exact tests normally apply only to arrays with a single subscript and subscript expressions involving only a single induction variable. These tests are called Single Induction Variable (SIV) tests. An even more restricted test is the Simple SIV test, which only applies to subscript expression pairs where the coefficient of the induction variable in each expression is the same. Since these subscript pairs also have the property that they result in constant thresholds, we shall consider this test in detail.

Consider a pair of subscript expressions.

$$aI+b$$

$$aI+c.$$

These expressions result in the dependence equation,

$$ax+b=ay+c,$$

which, when solved for $(y - x)$ yields

$$y - x = (b - c) / a = t.$$

The threshold of the dependence, if one exists, is given by t . Since the threshold of any dependence must be an integer, we can conclude independence if $(b - c) / a$ is not an integer. If the bounds of the loop can be determined, and the possible threshold is greater than the maximum number of iterations between loop iterations, then there can be no dependence. If this test fails to prove independence and $t = (b - c) / a$ is an integer, then t is said to be a *constant threshold*, because the number of iterations between the definition and use that cause the dependence is the same over all iterations of the loop. For a loop with inductive step 1, upper bound U , and lower bound L , if this constant threshold is less than the number of iterations of the loop, then there exist at least $U-L-t$ iterations of the loop during which a statement on

one iteration shares a memory location with a statement on another iteration. Hence, the dependence is proved.

3.1.3 A Stronger Notion of Dependence

The data dependence calculation that we have described so far ignores the effect of a definition to an array on the propagation of information from an earlier definition. Consider the following example.

```

DO I = 1, N
  S1      A(I) = 0.0
  S2      A(I) = F(I)
  S3      B(I) = G(A(I))
ENDDO

```

The method that we have described for discovering data dependence we have described will detect a true data dependence from S1 to S3 and from S2 to S3. But, there is no memory location that is written by S1 and is read by S3 that has not in the between time been written to by S2. In other words, there is no way for the *value* in the definition at S1 to reach the use in S3. The definition of A in S2 *kills* the previous definition in S1. Kuhn called this effect *covering* [Kuh80]. The data dependence calculated from S1 to S3 is the result of ignoring the covering effect of a definition to an array variable or one of its elements.

A more precise or minimal data dependence graph would not include the edge from S1 to S3. Instead the execution order constraint between S1 and S3 would be represented as one data dependence between S1 and S2 and another dependence from S2 to S3.

The transitive closure of this minimal dependence graph includes dependence edges $\langle u, v \rangle$ for each pair of statements where the minimal dependence graph has edges $e_1 e_2 \dots e_n$ such that $e_1 = \langle u, s_1 \rangle$, $e_n = \langle s_{n-1}, v \rangle$, and all other $e_i = \langle s_{i-1}, s_i \rangle$.

We call edges $\langle u, v \rangle$ *transitive edges* whenever the same edge does not exist in the minimal dependence graph.

In automatic vectorizers and parallelizers, the transitive closure of the dependence graph is used to test for the correctness of a parallelizing transformation. Since this is the only form of the dependence graph that is required for the generation of parallel code, no attempt is made to distinguish between transitive and nontransitive edges as data dependences are calculated.

For our purpose of providing a dependence graph to be inspected by the user, this strategy is not acceptable. The transitive dependence graph contains dependences that are confusing to a user and typically are so numerous that they obscure the information that may be helpful to the user. Consequently, we desire a graph containing as few transitive edges as possible.

Kuck [Kuc78] defines data dependence with respect to the flow of values through a variable rather than with respect to the assignments and uses of a memory location. A kill represents the end of the presence of a previous value contained in a variable. Hence, Kuck's idea of defining data dependence in terms of the flow of value corresponds to the minimal dependence graph we have just described.

In order to draw a firm distinction between these two approaches to dependence, we offer the following definition.

Definition 3.2 A *strong dependence* exists on the variable A between two statements, S_1 and S_2 , if and only if

1. S_1 and S_2 access the same memory location in their references to identified by A and
2. there exists some execution path such that the memory location within A accessed by S_1 and S_2 is not written between the executions of S_1 and S_2 .

A strong dependence is characterized as true, anti, output, or input under the same conditions as in definition 3.1. Only definitions can act as covers, regardless of the type of dependence being calculated. In the following we discuss the calculation of strong true dependences. Section 3.4 will show how to extend the algorithms to calculate strong anti and output dependences.

If only scalars are considered, a dependence graph containing only strong true dependences corresponds exactly to def-use chains. When we extend the notion of strong dependence to arrays, we gain the same kind of information about subscripted variables as def-use chains give for scalars.

Arrays are not atomic data structures. It is insufficient to know that a statement assigns to or reads from a particular array. Instead, we need to know what *part* of the array is defined or used by the statement. For this purpose, we will associate with the statement s an *access descriptor*, denoted \mathcal{A}_s^+ , for the set of elements of the array defined by s and \mathcal{A}_s^- for the set of elements of the array used by s ³. These access descriptors are functions of the context or state of the program at the time of a particular execution of s . For an assignment statement contained in a loop and with left hand side containing the array A , with the subscript expression being a function of the induction variable of the loop, \mathcal{A}_s^+ is equal to the subscript expression on the left hand side and the context of the execution is the value of the induction variable at the time of execution of s . For example, the access descriptor of an assignment statement with left hand side equal to $A(I+1)$ is $I+1$. If a call is made to a subroutine that defines the I th column of a two dimensional array A , then the access descriptor is a set containing the first column of the array. This set can be denoted by $(*,I)$. Since these access descriptors are sets of array elements, set arithmetic may be performed on them. Whenever we show arithmetic operations on access descriptors, the operations will denote set arithmetic.

³In our discussion we will concentrate on the calculation of dependences on the ubiquitous array A . In practice, access descriptors for s can be calculated for all the arrays of the program. If the array does not appear in s , then its access descriptor will be equal to the empty set.

According to Definition 3.2, a definition at a statement d can be the source of a strong dependence at a statement s only if the definition reaches s ; that is, some memory location x is written by d and there exists a control path in the program from d to s where no statement on the path writes to x . We collect the definitions that can be the source of a strong dependence at s in the set $may(s)$. The set $may(s)$ consists of pairs containing the name of a statement, d , and the part of A that d defines and which has not been defined by some other statement between the execution of d and s . We will denote a pair by enclosing a statement identifier and the access descriptor in parentheses separated by a colon. For instance, the pair $(11:2I+1)$ indicates that at statement 11, a single array element is defined and, during a single execution of the statement, it is the element given by $(2I + 1)$ where I is the induction variable of the surrounding loop.

The *propagation function* $f(s)$ specifies how the value of the set $may(s)$ changes as the value of s changes from s to one of its immediate successors. The value of $may(s)$ is calculated by performing a *meet* operation on the values of $f(p_i)$ for all of s 's immediate predecessors, p_i . If s does not define A then $f(s)$ is equal to $may(s)$. Otherwise $f(s)$ is calculated as follows. All the memory locations defined by s must be subtracted from the access descriptors contained in the pairs which are elements of $may(s)$. So, for each pair $(u : \mathcal{X})$ in $may(s)$, if $\mathcal{X} - \mathcal{A}_s^+$ is not equal to the empty set then the pair $(u : \mathcal{X} - \mathcal{A}_s^+)$ is placed in $f(s)$. Next, the pair $(s : \mathcal{A}_s^+)$ is added to $f(s)$ to represent the definition at s . The set $may(s)$ is formed from the union of the propagation functions of all of s 's immediate predecessors.

A strong true dependence exists from a statement $S1$ to a statement $S2$ if $(S1:\mathcal{X})$ is an element of $may(S2)$, there exist iterations i and j of the loops surrounding $S1$ and $S2$ such that \mathcal{X} evaluated on i is equal to \mathcal{A}_{S2}^- evaluated on j . If this occurs we say there is a non-empty intersection between \mathcal{X} and \mathcal{A}_{S2}^+ .

Unfortunately, the test for a non-empty intersection between two access descriptors is equivalent to an exact test for weak dependence between general subscripts

and hence is NP-hard for affine expressions. We are forced, therefore, to seek an approximation to the notion of strong dependence. The approximation should be conservative in the sense that all strong dependences *must* be represented in the approximation.

Our approximation involves an attempt to prove covering between a restricted form of array references within and between iterations of *inductive loops*. Inductive loops have an associated *induction variable* whose value is an affine function of the number of the iteration where the use of the variable occurs. If analysis of the control flow shows a path between a definition and a use of an array variable, and the independence tests fail, and we are not able to show that a cover exists between the two references, we assume a dependence.

3.2 Algorithms

In the following discussion, unless otherwise stated, “dependence” denotes strong dependence.

In order to motivate our algorithm we make the following observations:

1. *A use cannot be the sink of a loop-carried dependence if the memory location that it uses is always defined previously in the loop body.*

If a use is covered from above by a definition d in the same loop iteration, then d prevents any possible loop-carried dependence with its sink at the use. For instance, in the following example, the use of $A(I)$ cannot be the sink of any loop-carried dependence because it is defined within the same iteration that it is used and textually before the use in the loop body.

```

DO I = 1, N-1
    A(I) = ...
    ... = A(I)
    A(I+1) =
ENDDO

```

2. *If a memory location used on iteration i is always defined on iteration $(i - c)$ where c is a known constant, then no definition from an iteration earlier than $(i - c)$ can reach the use on iteration i .*

A definition, d , that results in a loop-carried dependence can cover the sink of the dependence from any definitions that occurred before d . A definition only covers in this way if it is one that occurs on every loop iteration. In the following example, the use of $A(I)$ would be the sink of a dependence from $S3$ except that definition in $S2$ occurs on an iteration that is closer in time to the use. Since the definition to $A(I+1)$ occurs on every iteration, no dependence with its sink at the use of $A(I)$ could possibly exist at this level with a threshold greater than one.

```

DO I = 1, N-2
S1      ... = A(I)
S2      A(I+1)=
S3      A(I+2)=
ENDDO

```

3. *Information about the pattern of definitions that can occur between loop iterations can be derived from a set representing all the definitions that may occur during a single iteration and a set representing all the definitions that must occur during any iteration.*

Without simulating the execution of conditionals, all iterations must appear alike to the dependence analysis. The set of all definitions that can occur during

the execution of the loop body provides all the possible sources of dependences.

The set of all the dependences that must occur during any iteration provides all the definitions that can prevent some loop-carried dependence from occurring.

We first present a batch algorithm to calculate our approximation of strong dependence. Using the batch algorithm as a guide, we will develop an algorithm for updating the dependence graph after arbitrary editing changes. The algorithms first presented are correct for innermost loops without call sites. That is, they are designed to work in the absence of call sites or loops nested within the loop carrying the loop-carried dependences that we are testing. In a later section we will show how to generalize these algorithms to handle call sites and multiple nested loops.

3.2.1 Batch Algorithm

We assume that the references in which we are interested reside in a single loop with a single entry point called the *loop header*; all iterations of the loop begin at the loop header. Given a particular use of an array A , we calculate all the dependences with their sink at this use. Reflecting the observation above, we need the following information.

- The definitions of A that can reach this use from within the same iteration of the loop.

The set $may(s)$ consists of the same pairs as described in subsection 3.1.3. The propagation function associated with $may(s)$ is $f(s)$.

- Definitions of A that can reach the beginning of an iteration of the loop from an earlier iteration.

The set $loopmay(s)$ contains the same kind of pairs as $may(s)$, but contains a pair for each definition that can reach an iteration from an earlier iteration. $loopmay(s)$ is calculated by combining the may sets for all the statements that reach the loop header from inside the loop.

- The part of A that must be defined previous to the use in the same iteration.

Information about what elements of A must be defined during a single iteration from the beginning of the loop body up to s is contained in the set $must(s)$. The elements of $must(s)$ are access descriptors describing the part of A defined by some statement in the loop. The propagation function associated with $must(s)$ is $g(s)$.

- The part of A that must be defined during a previous iteration and the number of iterations that must have occurred between the iteration containing the definition and the current iteration .

The elements of A that must be defined during any iteration is contained in $loopmust$.

The set equations defining this information are presented below.

Definition 3.3

$$g(s) = must(s) + \mathcal{A}_s^+$$

$$f(s) = may(s) \text{ if } s \text{ does not assign to } A$$

$$f(s) = \{(\psi : \mathcal{X}) \text{ where } (\psi : \mathcal{Y}) \in may(s) \text{ and } \mathcal{X} = \mathcal{Y} - \mathcal{A}_s^+ \neq \emptyset\} + (s : \mathcal{A}_s^+)$$

$$may(s) = \bigcup_{p \in pred(s)} f(p)$$

$$must(s) = \bigcap_{p \in pred(s)} g(p)$$

$$loopmay = \bigcup_{i \in \{\text{back edge to loop header}\}} may(i)$$

$$loopmust = \bigcap_{i \in \{\text{back edge to loop header}\}} must(i)$$

Theorem 3.1 Given a singly-nested inductive loop with lower bound L and upper bound U , array accesses restricted to affine functions of the induction variable, and the solution to the set of equations above, a loop-independent strong dependence exists from a definition contained in a statement u to a use contained in a statement v if and only if there exists $(u : \mathcal{X}) \in \text{may}(v)$, and integer k , $L \leq k \leq U$, such that $\mathcal{X}(k) \cap \mathcal{A}_v^- \neq \emptyset$.

Proof We prove the if part of the theorem first. The use at v intersects with some element $(u, \mathcal{Y}) \in \text{may}(v)$. Hence, we know u writes a location which v reads. Now, we show that this intersection is not written in the interim between their executions.

Suppose that the intersection of the use in v and \mathcal{Y} in the element of $\text{may}(v)$ is \mathcal{X} . Then there exists a path in the program, u, s_1, \dots, s_n, v , such that

$$f(s_n) \circ f(s_{n-1}) \dots f(s_1) \circ f(u) \supseteq (u : \mathcal{Y}) : \mathcal{Y} \supseteq \mathcal{X}.$$

From the definition of f , it is clear that this implies that no statement s_i writes to \mathcal{X} . Hence a strong true dependence exists from u to v . This completes the proof of the if portion of the theorem.

A loop-independent strong dependence implies that u writes to some memory location \mathcal{Y} that is read by v . Also, it implies that there exists a path, $[s_1, s_2, \dots, s_n]$ from u to v , such that no s_i on the path writes to \mathcal{Y} . From the definition of f , we know that $(u : \mathcal{A}_u^+) \in f(u)$. Also from the definition of the propagation function f , since s_i does not write to \mathcal{Y} , if $(r, \mathcal{W}) \in \text{may}(s_i)$ and $\mathcal{W} \supseteq \mathcal{Y}$ then

$$(u : \mathcal{Z}) \in f(s_i) \text{ where } \mathcal{W} \supseteq \mathcal{Z} \supseteq \mathcal{Y}.$$

This along with the definition of the meet operation proves that there exists \mathcal{X} such that $\mathcal{X} \supseteq \mathcal{Y}$ and $(u : \mathcal{X}) \in \text{may}(v)$. Hence, from the definition of strong dependence, since $\mathcal{Y} \subseteq \mathcal{X}$, there exists k , $L \leq k \leq U$, such that $\mathcal{X} \cap \mathcal{A}_v^- \neq \emptyset$. Thus, the theorem is proved. \square

Theorem 3.2 Given a singly nested inductive loop with lower bound L and upper bound U , array accesses restricted to affine functions of the induction variable, and the solution to the set of equations above, there is a loop-carried strong dependence between the statements u and v if and only if

1. $(u : \mathcal{X})$ is an element of *loopmay*,
2. there exist i and j , $L \leq i \leq j \leq U$, such that $\mathcal{Z} = \mathcal{A}_v^- \cap \mathcal{X} \neq \emptyset$,
3. \mathcal{Z} is not contained in any element within $\text{must}(v)$ ($\mathcal{Z} \not\subseteq \text{must}(v)$),
and
4. the threshold between u and v is less than or equal to any threshold calculated between an element of *loopmust* and v .

Proof We prove the if portion first. Since u appears in *loopmay*, there exists a path from u to the end (or the beginning) of the loop body such that the array element that u defines during some iteration is not defined along the path. This particular element is a function of the particular iteration of the loop. Call this element Λ_u^i . Condition 2 above implies that there exist iterations i and j such that Λ_u^i is used by statement v during iteration j . Condition 4 tells us that there is no iteration between i and j such that a definition must occur to the memory location(s) that u and v share. Hence, there is a path from the end of iteration i to the beginning of iteration j such that no statement on the path defines Λ_u^i . Condition 3 implies that there is a path from the beginning of an iteration to v such that no statement on the path defines the element that v uses. During iteration j this element is Λ_u^i . Thus we have proven the existence of a path from u to v such that no statement defines Λ_u^i . Since u and v share Λ_u^i , the if part of the theorem is proved.

Given a loop-carried dependence from u to v , we know that there must exist a path, P , from u to the beginning of the next iteration, through some number of iterations of the loop and then from the beginning of the loop body to v such that

no statement on this path defines the array element that u defines and v uses. The existence of the portion of P from u to the beginning of the next iteration proves that u will appear in *loopmay*, thus proving item 1 above. By the definition of strong dependence, there must be an intersection between the memory location written by u and that read by v , thus proving item 2. The existence of the portion of P from the beginning of the loop body to v proves that no member of $\text{must}(v)$ can write to the element that v uses. Thus, item 3 is proved. Finally, the number of iterations between the definition at u and the use at v establishes that no element of *loopmust* defines the array during an iteration closer to the use in v than u , and hence no element of *loopmust* can possibly have a lower threshold with respect to v than u . Thus item 4 is proved and the theorem follows. \square

Given precise tests on array subscripts and precise calculation of the sets, this theorem describes a precise calculation of strong dependence within a single inductive loop. Within a singly nested loop and with subscript expressions restricted to those on which the SSIV test can be applied, these tests are precise. For subscript expressions outside outside the restricted set, we approximate our sets and our tests.

In the algorithm we present we assume the existence of three routines:

1. **TestForLoopIndependent(def,use)** uses information about the surrounding loop to perform an independence test on **def** and **use**, checks for a loop-independent dependence, and places the edge in the dependence graph if independence is not proved.
2. **TestForContainment(AD1, AD2)** takes two access descriptors, **AD1** and **AD2**, and information about the surrounding loop structure, and tests if **AD2** covers **AD1** in a loop-independent way. If the access descriptors represent subscript expressions of the restricted form, the obvious test is placing the two access descriptors into a normal form and testing for equality. If **AD1** or **AD2**

```

program AllDeps(l):
/* l is the header of an innermost loop */
may(*) =  $\emptyset$ 
must(*) =  $\emptyset$ 
loopmay =  $\emptyset$ 
loopmust =  $\Omega$  /* Universal Set */

for all successors, succ, of l
    if succ  $\neq$  l then place succ on worklist
    else loopmust =  $\emptyset$ 
    endfor
do until worklist is empty
    take s from worklist
    call LoopIndependent(s)
call LoopCarried
end AllDeps

```

Algorithm 3.1 Main Program AllDeps for Batch Algorithm

represent ranges, then the test will be more complex. We will discuss this in more detail later.

3. **FindThreshold**(AD1, AD2) takes two access descriptors, AD1 and AD2, and performs a test similar to the SIV test to discover, if possible, a constant minimum threshold. If a constant threshold is found, it is returned.
4. **TestForCarriedIndependence**(defpair, *u*, *M*) takes the access descriptor of an element of a *mayset*, *defpair*, and a use, *u*, and information about the surrounding loop and performs an independence test searching for a loop-carried dependence. If independence is shown, or a minimum threshold is found that is greater than *M*, then the routine returns without adding the dependence. Otherwise a dependence from the definition in *defpair* to *u* is added to the dependence graph.

```

procedure LoopIndependent( $s$ ):
 $may(s) = \emptyset$ ;  $must(s) = \emptyset$ 
 $inmay = \emptyset$ ;  $inmust = \text{emptyset}$ 

/* Calculate  $may(s)$  and  $must(s)$  */
for all predecessors of  $s$ ,  $p$ 
     $inmay = may(p) \cup inmay$ 
     $inmust = must(p) \cap inmust$ 
endfor
for all  $(u: \mathcal{A}_i^+) \in inmay$ 
    call TestForLoopIndependent( $(u: \mathcal{A}_i^+), s$ )
    if  $s$  assigns to  $A$  then do
         $diff = \mathcal{A}_i^+ - \mathcal{A}_s^+$ 
        if  $diff \neq \emptyset$  then  $may(s) = may(s) + (u : diff)$ 
        endif
    endfor

/* Store  $f(s)$  in  $may(s)$  */
 $may(s) = may(s) + (s: \mathcal{A}_s^+)$ 

for all  $w \in inmust$ 
    if TestForContainment( $(\mathcal{A}_s^-, w)$ ) succeeds then mark  $s$  as "covered"
    endif

/* Store  $g(s)$  in  $must(s)$  */

 $must(s) = inmust + \mathcal{A}_s^+$ 
for all successors,  $succ$ , of  $s$ 
    if  $succ \neq l$ 
        then place  $succ$  on worklist
        else do
             $loopmay = loopmay \cup may$ 
             $loopmust = loopmust \cap must$ 
        endelse
    endfor
until worklist is empty
    take  $n$  from worklist
    if LoopIndependent has been called on all of  $n$ 's predecessors
        then call LoopIndependent( $may, must, succ$ )
    end until
end LoopIndependent

```

Algorithm 3.2 Procedure LoopIndependent called from AllDeps

```

procedure LoopCarried:
for all statements in loop  $l$  not marked as “covered”,  $s$ 
  for all  $w \in loopmust$ 
     $M = \min(M, \text{FindThreshold}(w, s))$ 
  endfor
  for all  $(u:A_s^+) \in loopmay$ 
     $\text{TestForCarriedDependence}((u:A^+), s, M)$ 
  endfor
endfor
end LoopCarried

```

Algorithm 3.3 Procedures LoopIndependent and
LoopCarried called from AllDeps

All these routines are based on the independence tests described in 3.1.2. The algorithm consists of two passes over the loop. During the first pass, the algorithm calculates the sets $may(s)$ and $must(s)$ sets for each statement s contained in the program. At the same time loop-independent dependences with their sink at s are calculated. If a use of a variable x is covered by the $must$ set for x at the statement containing the use of x , then it is marked as one that cannot be the sink of a loop-carried dependence. However, in the set variables $may(s)$ and $must(s)$ is stored the values of the propagation functions $f(s)$ and $g(s)$, respectively. These are stored in the batch algorithm in order to ease later comparison of the batch and incremental algorithms.

After $may(s)$ and $must(s)$ have been calculated for all the statements of the loop body, we construct the two sets $loopmay$ and $loopmust$. $loopmay$ is formed from the union of the may set of all those statements with a back edge to the loop header. Similarly, $loopmust$ is formed from the intersections of the $must$ sets of these statements.

With these sets in hand, the second stage of the algorithm calculates loop-carried dependences. For each use contained in a statement v in the loop body not marked as

covered from carried dependences, the use is compared to each element of *loopmust*. Each comparison yields a constant threshold if one exists. Call the minimum of these thresholds M . Since for every iteration, the part of A which v uses has been defined only M iterations previously, M represents the maximum possible threshold of any loop-carried dependence with v as a sink.

We then compare the use in v with each element of *loopmay*. For each element, if the independence test succeeds, then no dependence exists. If the test fails, but a constant or minimum threshold can be calculated, and this threshold is greater than M , then no dependence is added. Otherwise a loop-carried dependence is added. See Algorithm 3.1.

Before proving AllDeps correct we prove the following lemma.

Lemma 3.1 AllDeps calculates $may(s)$ for each statement s in program P .

Proof By Definition 3.3, $may(s)$ is equal to the union of $f(p)$ for all of s 's immediate predecessors in P . AllDeps performs this calculation in the first loop in procedure LoopIndependent which is called for every statement in P . The proper initial conditions are set at the beginning of AllDeps. Thus the lemma is proved. \square

Theorem 3.3 AllDeps terminates after calculating dependences according to theorems 3.1 and 3.2.

Proof AllDeps calls TestForLoopIndependent to test whether the set of elements described by some access descriptor contained in an element of $may(s)$ intersects with the set of elements used by the statement. Thus AllDeps calculates the loop-independent dependence according to Theorem 3.1.

loopmay is equal to the union of the sets $f(i)$ where i is a statement with an edge to the loopheader. In the third loop of LoopIndependent, if a successor of a statement s is the loop header, then the set $may(s)$ which contains $f(s)$ is unioned with *loopmay*.

Since union is commutative and associative this is equivalent to unioning $f(s)$ for all the statements with edges to the loop header. Thus *loopmay* is calculated correctly. The proof that *loopmust* is calculated correctly is similar.

$must(s)$ is equal to the intersection of $g(p_i)$ for all of s 's immediate predecessors, p_i . The third loop of LoopIndependent calculates the propagation function $g(p)$ and performs the intersection to create $must(s)$.

LoopCarried determines whether a use in a statement s is covered by an element in $must(s)$. If not, it forms the maximum threshold M from the elements in *loopmust* and the use. Then this threshold is compared to the threshold between each member of *loopmay* and the use. A dependence is added from each element of *loopmay*, d , to s for which independence is not shown between d and s and the threshold between d and s is less than M . Thus, by theorem 3.2. The theorem follows. \square

3.2.2 An Incremental Calculation

Before presenting the incremental algorithms we describe the notion of a *correct update* to be used as the criterion for the correctness of our incremental algorithms.

Definition 3.4 An algorithm performs a *correct data dependence update* if and only if given a program P , the sets *loopmust* and *loopmay* for each loop in P , and $may(s)$ and $must(s)$ for all statements in P as calculated by AllDeps, the algorithm, given an edit to produce P' , produces the new data dependence graph and the sets *loopmust*, *loopmay*, $must(s)$, and $may(s)$ as produced by algorithm AllDeps on P .

All edits can be classified as one of three categories[Zad84]. These are

1. the addition or deletion of a node from the control flow graph,
2. the addition or deletion of a variable definition or use, and
3. the addition or deletion of a control flow edge

We will treat each category of edit separately, describing how to update the data dependence information and proving that the resulting dependence graph is the same as obtained when using the batch version of the algorithm. First, we discuss some details of managing and organizing the information that is maintained by the algorithm between updates.

Set Information Management and Organization of Statements

Assignment statements are contained in control flow graph nodes which represent basic blocks. Figure 3.1 shows part of a control flow graph and the data statements contained in the nodes of the graph. A control flow graph node can have more than one predecessor. To simplify the algorithms, we assume that the first statement in each node is a pseudo statement called the *node header*, which contains no definitions or uses. The node header serves as target for all the node's predecessors. The *may* and *must* sets for this pseudo statement are equal to the union or intersection of the predecessors of the node, respectively. Thus, all statements containing a use or definition have only a single predecessor. This allows us to calculate $may(s)$ and $must(s)$, for a statement s that contains a use or definition, by examining just $may(p)$, $must(p)$ and s , where p is s 's predecessor.

For a particular array variable, the *may* and *must* information does not necessarily change from one statement to the next. We would like to maintain distinct sets only for each point in the control flow graph where the sets actually change. However, given any statement, we must be able to quickly find the correct sets for the statement. This is done by keeping, for each variable, a table of pointers to distinct values of *may* and *must*. For each statement, s , we keep a vector of indices into these tables. See figure 3.2. For each variable there is a vector element which indexes into that variable's table. The entry indexed by this vector element holds the pointer to the appropriate set for s . Thus, two indirections are required to access a set associated with a particular statement and variable.

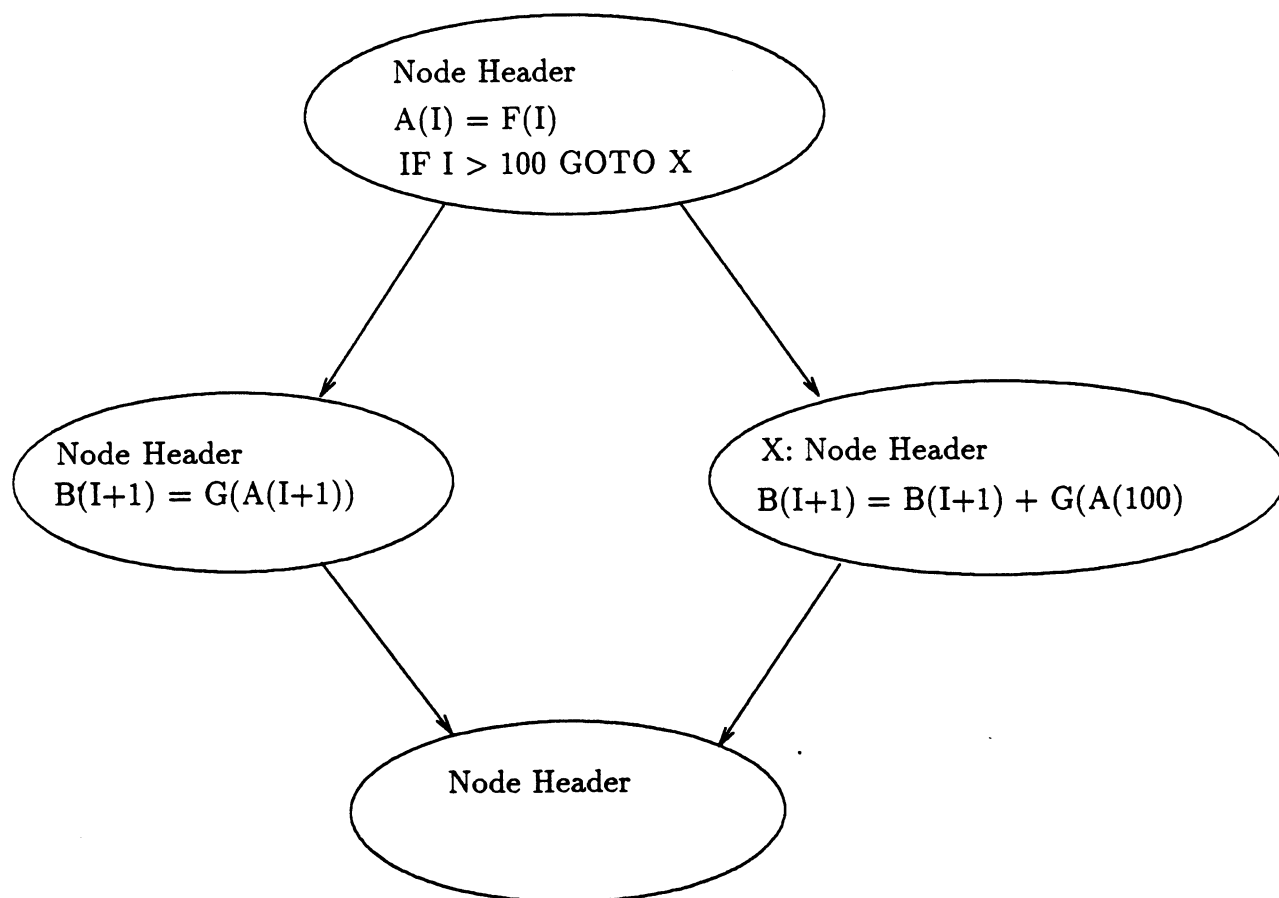


Figure 3.1 Data Statements in the Control Flow Graph

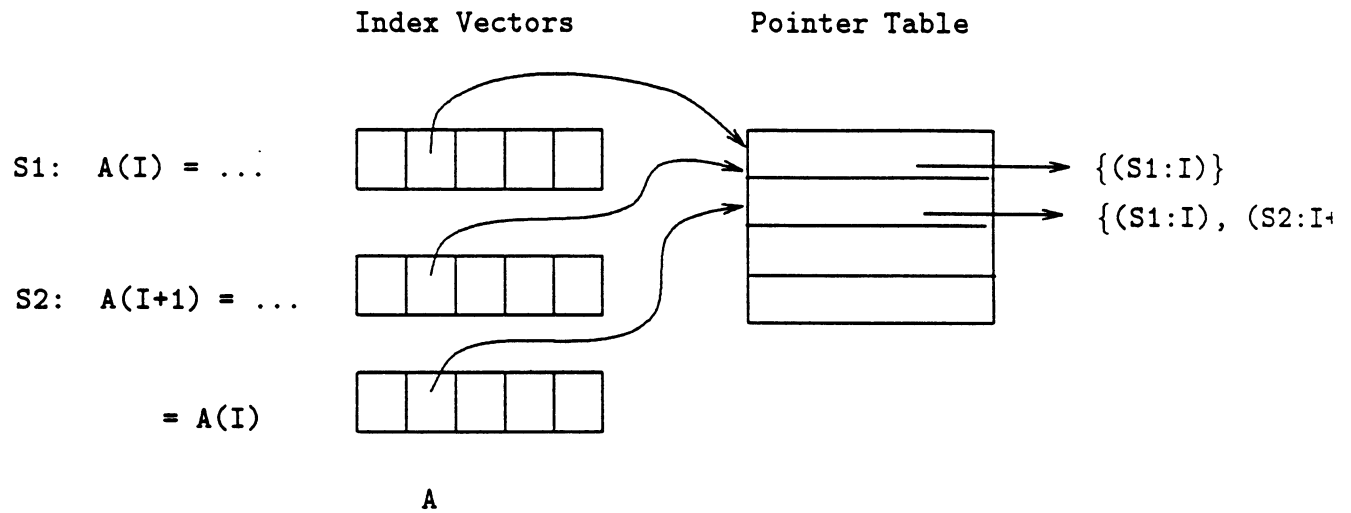


Figure 3.2 Set Storage Scheme

Addition and Deletion of a Node

In chapter 2, we have discussed how a node might be added or deleted from the control flow graph. A node can be created by splitting an existing node or placing a node on an existing edge. To be consistent with the convention developed for control dependence, we call the second node of a split pair the old node and the first node of the pair the new node. The second (old) node will require a new pseudo statement as its head. Since our sets and data dependences are kept on the statements contained within a node, no update is required in response to a name change of the node. If a node is placed on an existing edge, then it is initially empty; the new node will initially contain only the pseudo statement which acts as the node's single entry. Again, no update of the data dependence information is required. In both cases, a new variable vector must be created for the pseudo statement contained in the new node, and its elements copied from the last statement of the node preceding the new node.

Flow graph nodes that are deleted by joining two nodes require no special update. If a node is deleted from an edge we assume that the deleted node is empty, and again no update of dependences or sets is required.

Addition of a Use

A use can be added to an existing statement or as part of a new statement. In either case the new use will be part of an existing node in the CFG. If the use is part of a new statement, then new *may* and *must* sets must be created for the new statement. Since only the first statement in a node can have more than one predecessor and the first statement in a node will always be an empty pseudo statement, we know that any new statement containing a new use will have only a single immediate predecessor. Hence, the *may* and *must* sets of the new statement are calculated by computing the appropriate propagation function on the new statement's predecessor.

After the *may* and *must* sets are updated the possible dependences resulting from the added use can be determined. A use blocks no dependences and is not the source of any new dependences. But an added use can be the sink of new dependences.

A loop-independent dependence is added if and only if the access descriptor of one of the elements of $may(p)$ has a non-empty intersection with the new use where p is the predecessor of the statement containing the new use.

In order to find loop-carried dependences, we first check the new use against all the elements of $must(p)$. If the use is covered by a preceding definition in the loop body, then, as in the batch version, no loop-carried dependences can have this use as their sink. This is tested by determining whether the use is contained within any of the elements of $must(p)$ where p is the predecessor of the statement containing the new use. If the use is not covered by any of the elements in $must(p)$, then we calculate the constant thresholds between the new use and each element of *loopmust*. The minimum of the thresholds above zero, M , is the maximum threshold of any possible loop-carried dependence to this use. Independence tests are performed between all

```

program AddUse(s,var,use)
/* s is the statement location containing the new use          */
/* var is the variable involved in the new use                */
/* use holds the expression containing the new use. e.g. A(I) */

p = predecessor of s
for all elements, (u: $\mathcal{A}^+$ ), of may(p)
    call TestForLoopIndependent((u: $\mathcal{A}^+$ ),use)
endfor

for all elements,  $\mathcal{W}$ , of must(p) do until covered
    covered = TestForContainment( $\mathcal{W}$ ,use)
endfor

if not covered then do
    for all elements,  $\mathcal{W}$ , of loopmust
        M = min(M, FindThreshold( $\mathcal{W}$ ,s))
    endfor

    for all elements, (u,  $\mathcal{A}^+$ ), of loopmay
        call TestForCarriedIndependence((u,  $\mathcal{A}^+$ ),use,M)
    endfor
endif
end AddUse

```

Algorithm 3.4 Updating Dependences in Response to Addition of a Use

the elements of the set *loopmay* and the added use. If the independence test indicates possible dependence, and with a threshold less than or equal to M then a loop-carried dependence is added. See Algorithm 3.4

Theorem 3.4 AddUse performs a correct data dependence update according to definition 3.4.

Proof Since a use cannot be the source of any true dependences and cannot block any definitions from reaching any other uses, the only possible change to the dependence graph is the addition of some number of edges with v as their sink. Thus, in order to ensure that the dependence graph is correctly updated by AddUse, we only have to consider the addition of dependences with their sink at the new use. $may(p)$ in AddUse, where p is s 's predecessor, is identical to the set $may(s)$ in algorithm 3.1. Inspection of the first loop in AddUse shows that the same tests are performed for members of $may(p)$ against v as are done for members of $may(s)$ in the first loop of LoopIndependent in Alldeps. This proves that loop-independent dependences are calculated identically to the batch version.

In LoopIndependent, there is a test for covering of v by members of $must(s)$. This same test is performed by the second loop in AddUse. In both, when a cover is not found, loop-carried dependences are tested. In AllDeps the testing for loop-carried dependences occur in LoopCarried. In AddUse the tests are performed in the third loop. Both algorithms search for the minimum threshold, M , between the use and elements in *loopmust*, and then add loop-carried dependences from definitions in *loopmay* with thresholds with respect to v that are less than M . Thus, the data dependence graph updated by AddUse will be identical to the graph calculated by AllDeps on the updated program.

AddUse makes no changes to any sets $may()$, $must()$, *loopmust*, or *loopmay*. In AllDeps the calculation of these sets is strictly dependent on the definitions in the loop, and since these definitions are identical in P and P' , the sets *may*, *must*,

loopmust, and *loopmay* will be identical for the two programs. Hence AddUse, making no changes to the sets, will produce the same sets during its update as AllDefs when called on P' . This concludes the proof. \square

Deletion of a Use

Updating data dependences in response to the deletion of a variable use is very simple. Since none of the *may* or *must* sets of any statements change as the result of the deletion of a use, only dependences to the deleted use must be deleted. Assuming a mechanism to translate from a statement in the editor's intermediate form of the program to the corresponding node in the dependence graph, it is easy to identify to the appropriate dependences in the dependence graph and delete them. Algorithm 3.5 shows the update of the dependence graph in response to the deletion of a variable use.

Theorem 3.5 DelUse performs a correct data dependence update according to definition 3.4.

Proof The same reasoning of the proof for theorem 3.4 applies here to show that the sets *may*(), *must*(), *loopmust*, and *loopmay* will be correctly updated (i.e. the sets

```

program DelUse(s, u)
/* u is a use contained in s. */
for all the dependences  $\langle d, u \rangle$  with sink u in s
    delete the dependence  $\langle d, u \rangle$ 
endfor

end DelUse

```

Algorithm 3.5 Updating Dependence in
Response to the Deletion of a Use

remain unchanged). Also, by the same reasoning, we can restrict our attention to dependences having their sink at the deleted use v .

Since the use v does not appear in P' , AllDeps cannot add any dependences with v as their sink. Since the deletion of these dependence edges is the change made to the dependence graph by DelUse, the theorem follows. \square

Addition of a Definition

Updating the dependence graph in response to the addition of a variable definition is more difficult than updating the dependence graph in response to the addition of a use, because the change in the *may* and *must* information is not limited to the statement containing the new definition. Changes to these sets must be propagated to all the statements that the new definition reaches during a single loop iteration and, if necessary, to the *loopmust* and *loopmay* sets. This propagation is accomplished by following the possible flow of control, updating the sets for each statement as necessary. At the same time the algorithm accounts for the covering action of the statements through which the changes are being propagated. If after updating the *may* and *must* sets for some statement s , the value of one of the updated sets is identical to its value before the update, then we need not propagate the change to that set any further from that statement. At some point the sets will either cease changing, or all the sets for the loop body reachable during a single iteration from the inserted statement will have been updated. During the propagation of the changes to *may* and *must* sets, every time the *may* set of a variable used in a statement changes, the existence of a new loop independent dependence from the added definition to the use is tested.

For loop-carried dependences there are three cases to consider. First, neither the *loopmay* nor *loopmust* sets change. In this case, there is no change in the loop-carried information. Second, only the *loopmay* information changes. In this case, all the uses

```

program AddDef(s, var, def)
/* s is the location of the new definition                                */
/* var is the variable involved in the new definition                    */
/* def holds the added definition                                        */
/*                                                                    */

p = predecessor(s)

oldmay = may(s)
oldmust = must(s)
may(s) = may(p) - { definitions covered by def } + (s:def)
must(s) = must(p) ∪ def

/* Propagate deletion of the covered definition                        */
/* and the addition of the new new definition                          */
DeltaMayMinus = oldmay - may(s)
DeltaMayPlus = may(s) - oldmay
DeltaMust = oldmust - must(s)

for all successors, r, of s
    if DeltaMayMinus is not empty then call PMCMinus(s, r, DeltaMayMinus)
    if DeltaMayPlus is not empty then call PMCPlus(s, r, DeltaMayPlus)
    endfor

if DeltaMustPlus is not empty then do
    for all successors, r, of s
        call PMustCPlus(s, r, DeltaMust)
    endfor
endif
if MustXIteration
    then calculate all carried dependences in loop body
else if MayXIteration
    then calculate carried dependences on the added definition
end AddDef

```

Algorithm 3.6 Update In Response to Addition of a Definition

```

procedure PMCPlus(src, sink, DeltaMay)

if DeltaMay − may(sink) = ∅ then return

if sink is the loop header then do
    loopmay = loopmay ∪ DeltaMay
    MayXIteration = true
    return
endif

for all elements, (d :  $\mathcal{A}^+$ ), of DeltaMay
    for all uses, u, at sink
        call TestForLoopIndependent((d :  $\mathcal{A}^+$ ), u)
    endfor
endfor

oldmay = may(sink)

may(sink) = may(sink) ∪ DeltaMay − {elements covered by definitions in sink}

DeltaMay = may(sink) − oldmay

for all successors, s, of sink
    call PMCPlus(src, s, DeltaMay)
endfor

end PMCPlus

```

Algorithm 3.7 Routine to Propagate Added Definitions to May Sets

```

procedure PMCMinus(src, sink, DeltaMay)

if  $\text{may}(\textit{sink}) - \textit{DeltaMay} = \text{may}(\textit{sink})$  then return

for all predecessors of sink, p, not equal to src
     $\textit{DeltaMay} = \textit{DeltaMay} - \text{may}(p)$ 
endfor

if  $\textit{DeltaMay} = \emptyset$  then return

if sink = loop header then do
     $\textit{loopmay} = \textit{loopmay} - \textit{DeltaMay}$ 
     $\textit{MayXIteration} = \text{true}$ 
    delete loop-carried dependences from DeltaMay
    return
endif
for all elements, ( $d : A^+$ ) of DeltaMay
    for all uses, u, in sink
        delete any loop-independent dependence from d to u
    endfor
endfor

 $\text{may}(\textit{sink}) = \text{may}(\textit{sink}) - \textit{DeltaMay}$ 

for all successors of sink, s
    call PMCMinus(sink, s, DeltaMay)
endfor

end PMCMinus

```

Algorithm 3.8 Routine to Propagate Deletion
of Definitions from May Sets

```

procedure PMustCPlus(src, sink, DeltaMust)
for all predecessors of sink, p,
    DeltaMust = DeltaMust  $\cap$  must(p)
endfor

if sink = loophdr then do
    if DeltaMust contained in loopmust then return
    loopmust = loopmust  $\cup$  DeltaMust
    MustXIteration = true
    return
    endif
else do
    DeltaMust = DeltaMust -  $A_{sink}^+$ 
    if DeltaMust contained in must(sink)
        then return
    must(sink) = must(sink)  $\cup$  DeltaMust
    endelse

for all successors of sink, s
    call PMustCPlus(sink, s, DeltaMust)
endfor

end PMustCPlus

```

Algorithm 3.9 Routine to Propagate Additions to Must sets

procedure PMustCMinus(*src*, *sink*, *DeltaMust*)

if *sink* = loop header **then do**
 loopmust = *loopmust* − *DeltaMust*
 MustXIteration = **true**
 return
endif

DeltaMust = *DeltaMust* − \mathcal{A}_{sink}^+

if *DeltaMust* = \emptyset **then return**

must(*sink*) = *must*(*sink*) − *DeltaMust*

for all successors of *sink*, *s*
 call PMustCMinus(*sink*, *s*, *DeltaMust*)
endfor

end PMustCMinus

Algorithm 3.10 Must Propagation Routines

in the loop not covered by their *must* sets must be checked for possible loop-carried dependence on the new definition. Third, if both the *loopmay* and *loopmust* sets change (a change in the *loopmust* set implies a change in the *loopmay* set) then all the dependences carried by the loop on the variable in the new definition must be recalculated. The flags *MayXIteration* and *MustXIteration* are set to true if *loopmay* and *loopmust* change, respectively.

Algorithm 3.6 updates the dependence graph in response to the addition of a variable definition. It uses algorithms 3.7, 3.8, and 3.9. Algorithms 3.7 and 3.9 propagate the addition of the new definition to the *may* and *must* sets, respectively, of the affected statements. A new definition can kill definitions that occur earlier in the program. Algorithm 3.8 propagates the removal of killed definitions from the *may* sets of affected statements of definitions.

Theorem 3.6 AddDef performs a correct data dependence update according to definition 3.4.

In order to prove this theorem we first prove the following three lemmas.

Lemma 3.2 Given

1. $may(j)$ for all statements j in program P ,
2. a definition u contained in statement s to form $P' = P + u$, and
3. a definition pair $(d : \mathcal{D}) \in may(x)$, $(d : \mathcal{D}) \notin may(x)$ as calculated by AllDeps on program P and P' , respectively,

PMCMinus will delete $(d : \mathcal{D})$ from $may(x)$.

Proof $(d : \mathcal{D}) \in may(x)$ as calculated by AllDeps on program P implies that there exists a path from d to x in P such that \mathcal{D} is not defined by any statement on the path. Because the definition pair $(d : \mathcal{D})$ is not contained in P' , all such paths include s . Suppose there is only one such path, $Q = d, q_1, \dots, q_k = s, \dots, q_n, x$. PMCMinus will

be called on statement s . PMCMinus calls itself on all the immediate successors, not equal to the header of the loop, of any node it visits until DeltaMay becomes empty. DeltaMay is passed from visited node to visited node. DeltaMay is reduced by the elements contained in $may(p_i)$, where p_i is a predecessor of the current visited node not equal to the predecessor from which DeltaMay was passed. Since there is only one path from d to x , at each q_i , $i \geq k$, there is no predecessor p such that $may(p)$ will contain $(d : \mathcal{D})$. Hence, PMCMinus will be called on x and DeltaMay will contain the definition pair $(d : \mathcal{D})$. This results in deleting $(d : \mathcal{D})$ from $may(x)$.

Suppose there is more than one path from s to x , $Q_i = s, q_{i,1}, \dots, q_{i,n_i}$. For each Q_i let k_i be the maximum such that there is only one path from s to q_{i,k_i} . We know that PMCMinus will propagate the change in may sets to q_{i,k_i} . q_{i,k_i} 's successor in Q_i , q_{i,k_i+1} , is a member of some other path, Q_j . Suppose PMCMinus reaches q_{i,k_i} first. Then the may information at q_{i,k_i} will be updated to remove $(d : \mathcal{D})$. When PMCMinus is called on q_{i,k_i+1} it will discover $(d : \mathcal{D})$ in may of its predecessor q_{i,k_i} . PMCMinus will not be called then any further at this point. However, PMCMinus, in its walk from s will eventually travel path Q_j . When PMCMinus is called on $q_{j,k_j+1} = q_{i,k_i+1}$, if Q_i and Q_j are the only paths from s to this point, PMCMinus will not find a predecessor with $(d : \mathcal{D})$ contained in its may set and PMCMinus will delete $(d : \mathcal{D})$ from $may(q_{j,k_j+1})$ and continue to q_{i,k_i} 's immediate successors. If there are more than two paths to q_{i,k_i} , then PMCMinus will halt at q_{i,k_i} and revisit again. Eventually, when the last path to this point has been walked, PMCMinus will make the deletion from the may set and continue. This same reasoning applies to every point on the paths from s to x which has more than one predecessor whose may set before the update might contain $(d : \mathcal{D})$. Thus, the propagation of the change will continue to x and the lemma is proved. \square

Lemma 3.3 Given

1. $may(j)$ for all statements j in program P ,

2. a definition u contained in statement s to form $P' = P + u$, and
3. definition pair $(d : \mathcal{D}) \notin \text{may}(x)$, $(d : \mathcal{D}) \in \text{may}(x)$ as calculated by AllDeps on program P and P' , respectively,

PMCPlus will add $(d : \mathcal{D})$ to $\text{may}(x)$.

Proof $(d : \mathcal{D}) \in \text{may}(x)$ as calculated by AllDeps on P' implies that there exists a path $Q = q_1, \dots, q_n$ from d to x such that no node on the path writes to \mathcal{D} . $(d : \mathcal{D}) \notin \text{may}(x)$ as calculated by AllDeps on P implies that $(d : \mathcal{D}) \notin \text{may}(q_i)$ for any member of Q . Otherwise, AllDeps would have propagated $(d : \mathcal{D})$ into $\text{may}(x)$. Also, d must be equal to s .

PMCPlus will propagate the change forward until some node y covers the definition being propagated forward or d already exists in $\text{may}(y)$. We have already established that neither of these actions can occur on the path Q . Hence, PMCPlus will add $(d : \mathcal{D})$ to $\text{may}(x)$. \square

Lemma 3.4 Given

1. $\text{must}(j)$ for all statements j in program P ,
2. a definition u contained in statement s to form $P' = P + u$, and
3. $u \notin \text{must}(x)$, $u \in \text{must}(x)$ as calculated by AllDeps on program P and P' respectively,

PMustCPlus will add u to $\text{must}(x)$.

The proof is similar to the proof for lemma 3.2.

With these three lemmas established, we now prove theorem 3.6.

Proof PMCMinus and PMCPlus both add loop-independent dependences as they update the *may* sets. If, during propagation of the change, they encounter a point

where the change would propagate to the next iteration, they set the flag *MayXIteration*. In this case we know that we must update *loopmay* and test for loop-carried dependences from the added definition. If *PMustCPlus* detects a point where *must* information crosses an iteration, it sets the flag *MustXIteration*. If this flag is set then we must update *loopmust* and retest loop-carried dependences everywhere in the loop. The update of *loopmay* and *loopmust* is done as in *AllDeps*. If *loopmust* is unchanged then we only calculate loop-carried dependences on the new definition as reflected in its addition to *loopmay*. The algorithm calculates loop-carried dependences identical to those calculated by *AllDeps*. Because the *loopmust* and *must* sets are identical, the only possible new loop-carried dependence have the new definition as their source. *AddDef* calculates these dependences in an identical fashion to *AllDeps*. If *loopmust* changes, then we recalculate all the loop-carried dependences in the loop. These dependence are calculated identically in *AddDef* and *AllDef*. Thus the theorem is proved. \square

Deletion of a Definition

Handling the deletion of a definition is similar to handling the addition of one. When a whole statement is deleted, the *may* and *must* sets of its successors must be recalculated from the sets of the deleted statement's predecessor and the statements themselves. The changes to the *may* and *must* sets must be propagated in a manner similar to the propagation of changes to these sets in response the addition of a definition.

Deleting a definition from a statement's *must* set, can allow previously killed definitions to propagate beyond the changed statement. For example, in the following,

$$\begin{aligned} A(I) &= \dots \\ A(I) &= \dots \\ &= A(I) \end{aligned}$$

deleting the second definition allows the first definition to propagate forward into the *may* set of the statement containing the use.

Thus, any definitions previously covered by a deleted definition must be propagated and possible dependences with these definitions as the source calculated. See algorithm 3.11.

Theorem 3.7 DelDef performs a correct data dependence update according to definition 3.4

Proof We first state the following lemma.

Lemma 3.5 Given

1. $\text{must}(j)$ for all statements j in program P ,
2. a definition u deleted from statement s to form P' , and
3. $u \in \text{must}(x)$, as calculated by AllDeps on program P and P' , respectively,

PMustCPlus will add u to $\text{must}(x)$.

The proof of the lemma is similar to that for lemma 3.2.

DelDef calls PMCPlus, PMCMinus, and PMustCMinus. Thus, by lemmas 3.3, 3.2, and 3.5, the *may* and *must* sets will be updated over the entire program. DelDef removes any dependence edges from the data dependence graph with their source equal to the deleted definition.

PMCPlus adds any dependences involving a definition previously covered by the deleted definition.

This completes the update. This is shown as follows. If AllDeps calculated a dependence $\langle u, v \rangle$ in P' and not P that is not covered in P by the deleted definition then u must reach v in P' . This implies the existence of a path in P' from u to v that does not include the deleted definition. But since the only change from P to P' is

```

program DelDef(s, var, def)
/* s is the statement containing the deleted definition.          */
/* var is the variable whose definition is being deleted          */
/* def contains the definition that was deleted.                  */

delete any dependences with def as the source

/* Assume predecessor(s) and successor(s)                      */
/* remain unchanged until after the dependence update.            */

p = predecessor(s) /* Only one predecessor if it contains a definition

DeltaMayMinus = may(s) - may(p)

DeltaMayPlus = may(p) - may(s)

DeltaMustMinus = def - must(p)

for all successors, succ, of s
    call PropMayChangeMinus(s, succ, DeltaMayMinus)
    if DeltaMayPlus ≠ ∅
        then call PropMayChangePlus(s, succ, DeltaMayPlus)
    if DeltaMustMinus ≠ ∅
        then call PropMustChangeMinus(s, succ, DeltaMustMinus)
    endfor

if MayChangeCrossesIteration
    then update loopmay
if MustChangeCrossesIteration
    then update loopmust

if loopmust changes
    then update carried dependences throughout loop body
if loopmay changes
    then update carried dependences where necessary

end DelDef

```

Algorithm 3.11 Updating Dependences Due to Deletion of a Definition

the deletion of the definition, this path must also exist in P . Since the definition does not cover u in P , AllDeps would calculate $\langle u, v \rangle$ in P . A similar argument applies to dependences present in P but not P' . \square

Addition of a Control Flow Edge

Changes in control flow affect data dependence at a statement by changing the set of definitions which can reach the statement. If a control flow edge is added from an existing node, u , to another node, v , containing statements x and w , respectively, and $\text{may}(w)$ did not already contain some element, e , of $\text{may}(x)$ which the statements in u , previous to w , do not cover, then e must be added to $\text{may}(w)$. If $\text{must}(w)$ contains some element e not contained in $\text{must}(x)$ and not due to a statement in v before w , then e must be deleted from $\text{must}(w)$.

The sink of the new control flow edge will actually be an empty header statement at the beginning of the target node containing the original target statement of the edge. From there the changes in dependence and the *must* and *may* sets will be propagated forward to the statements contained in the node. The propagation function of the statement that is the actual target of the new edge is the identity function. The *may* and *must* sets for this target, w , are quite easy to calculate. To form the *may* set for w , $\text{may}(w)$ is unioned with $\text{may}(x)$. To form the *must* set, $\text{must}(w)$ is intersected with $\text{must}(x)$.

These changes to w 's *may* and *must* sets must then be propagated forward to all the statements that p can reach. This is done exactly as in the case of an added definition, except that now many definitions are propagated. The changes are propagated from statement to statement until either the sets stop changing or all the statements within the loop have had their sets updated. Algorithm 3.12 contains the update of data dependences in response to the addition of a control flow edge.

```

program AddCF(e)
x is the source of e; w is the sink of e

for each variable in the program
    DeltaMay = may(x) − may(w)
    DeltaMust = must(w) − (must(w) ∩ must(x))

    for each successor of w
        call PMCPlus(x, w, DeltaMay)
    endfor

    for each element of DeltaMust
        call PMustCMinus(x, w, DeltaMust)
    endfor

    if MustXIteration
        then calculate all carried dependences in loop body
    else if MayXIteration
        then calculate carried dependences on the added definition

    endfor

```

Algorithm 3.12 Updating Data Dependence After
Control Flow Edge Addition

Theorem 3.8 AddCF performs a correct data dependence update according to definition 3.4.

Proof The only difference between P and P' relevant to the calculation of dependence information is that P' contains additional paths that include the control flow edge (s, t) . Since any control flow path appearing in P also exists in P' and there are no changes to the uses or definitions at any statement, for any statement w in P , any element of the set $may(w)$ must appear in $may(w)$ in P' . Thus, the changes to the *mayset* of any statement are limited to the addition of elements. Hence, changes to loop-independent dependences are limited to the addition of dependences.

An element \mathcal{X} appears in the set $must(w)$ only if a definition to the array elements represented by \mathcal{X} occurs on *every* path from the loop header to w . Since every path present in P' is present in P , if \mathcal{X} is an element of $must(w)$ in P' , then \mathcal{X} is an element of $must(w)$ in P . Similarly, the only possible changes to *loopmay* are additions and the only possible changes to *loopmust* are deletions. Hence, the only possible changes to loop-carried dependences are the addition of dependences.

Suppose the element $(x : \mathcal{X})$ is added to $may(w)$. This implies that there exists a path in P' from x to w involving (s, t) such that no statement on the path contains a definition covering \mathcal{X} . Hence, there exists such a path from x to s and such a path from t to w . The existence of the path from x to s implies that $(x : \mathcal{X})$ will be a member of $may(s)$. This implies that PMCPlus will be called with *DeltaMay* containing $(x : \mathcal{X})$. That a path exists from t to w such that $(x : \mathcal{X})$ is not covered implies that PMCPlus will eventually add this element to $may(w)$ and that PMCPlus will add a loop-independent data dependence from x to w . PMCPlus will also set *MayXIteration* if necessary.

If an element \mathcal{X} is deleted from $must(w)$ in P' , then there exists a path in P' including (s, t) from the loop header to w such that the array elements represented by \mathcal{X} are not defined. Hence, there exists such a path from the loop header to s and another from t to w . Hence, $must(s)$ will not contain \mathcal{X} and *DelDef* will call

PMustCMinus with *DeltaMust* containing \mathcal{X} . The path from t to w implies that PMustCMinus will delete \mathcal{X} from $must(w)$. (and calculate any new loop-carried dependences to w). PMustCMinus will continue and set MustXIteration if necessary.

Changes in loop-carried dependences arising from additional elements to *loopmay* or deleted elements from *loopmust* are calculated just as in AllDeps. Thus, the theorem is proved. \square

Deletion of a Control Flow Edge

Deletion of a control flow edge (x, w) requires recalculation of the *may* and *must* sets from the union of all the remaining predecessors of w . The changes to these sets are propagated in a manner similar to that for deletion of a definition, using the procedures PMCMinus, but now PMCMinus must be called for many definitions and many variables. Propagation of the changes in the sets to other statements occurs just as in the deletion of a definition. Algorithm 3.13 contains DelCF which updates data dependences in response to the deletion of a control flow edge.

Theorem 3.9 DelCF performs a correct data dependence update according to definition 3.4.

The proof is similar to that for theorem 3.8.

3.3 Generalizing the Algorithms

In the previous section, we developed a dependence analysis that incrementally calculated dependence information between pairs of accesses to arrays that consisted of a single subscripted reference. In the presence of call sites or nested loops, the analysis must deal with groups of statements or multiple executions of a single statement accessing multiple elements of an array. Consider the following example.

```

program DelCF( $x, w$ )
/*  $x$  is the source of the edge,  $w$  the sink of the edge.                                     */
if  $w$  has no predecessors then do
     $may(w) = \text{empty}; must(w) = \text{empty}$ 
    delete any dependences with  $w$  as sink
    return
endif

 $oldmay = may(w)$ 

for each variable in the program
     $may(w) = \cup may(p)$  for all predecessors,  $p$ , of  $w$ . (excluding  $x$ )
     $DeltaMay = oldmay - may(w)$ 

    if  $DeltaMay = \emptyset$ 
        then return

    for all successors,  $s$ , of  $w$ 
        call PMCMinus( $w, s, DeltaMay$ )
    endfor

    if MayXIteration
        then update loop-carried dependences where necessary

    endfor
end DelCF

```

Algorithm 3.13 Updating Data Dependence After
Control Flow Edge Deletion

```

DO I = 1, N
    CALL SUBR(A(1,I))
    DO J = 1, N
S1:        B(I,J) = F(A(J,I+1))
    ENDDO
ENDDO

```

```

SUBR(S):
DO K = 1, N
    S(K) = G(K)
ENDDO

```

The call to SUBR results in a definition to an entire column of an array (assuming that the array is stored in column major order). Since this column is not involved in the right hand side of S1, no true dependence exists, but we can only conclude that there is no dependence if we adequately represent the effect of the call site on the array A. The identical problem arises if the call to a subroutine is replaced by a loop defining only a single row of A. Some method to summarize the results of a call, or an inner loop, is required for our analysis to be correct in the presence of these constructs. ⁴

3.3.1 Extended Array Descriptors

We need a form of summary information that can represent a part of an array that may or must be referenced by a single reference or many references. This information can be used by the analysis algorithms to determine if the references represented by the this summary information result in dependences or if they cover other dependences. A number of schemes for providing summary information have been proposed [CK87,

⁴Previous dependence analysis schemes subsumed this problem in their independence tests while considering inner loops. This is incompatible with our goal of including the effect of covering in our dependence calculation because it is impossible to take advantage of the effect of covering without looking at the inner loops.

Cal87, Bal89], primarily for summarizing the effect of subroutine calls. We propose to use such a scheme for summarizing the effect of a loop.

Balasundaram has developed Data Access Descriptors(DADs) to be used for summarizing the activity of a call or a loop. We will describe a simplified form of these DADs known as *simple sections*[Bal89].

A simple section describes a portion of an array that is bounded by planes or hyperplanes parallel to the primary coordinate axes and to planes that are at 45 degrees to those axes. Balasundaram proved that a simple section of an n dimensional array can be bounded by at most $2n^2$ boundaries[Bal89].

The majority of access patterns for arrays will be either rectangular or triangular (along the major diagonal). These patterns can be represented exactly by DADs. Other shapes must be represented by the best conservative approximation. The conservative approximation for information that represents a use or definition that may occur is one that contains every element that is used or defined. Hence, the approximation for a definition contained in a *may* set will be a DAD containing every element in the represented shape. In contrast, the conservative approximation for information that represents a definition that must occur is one that contains only elements that are defined. Hence the approximation for a definition contained in a *must* set will be a DAD containing only elements in the represented shape.

The generalized algorithm treats an inner loop or a call site as a statement with definitions and uses summarized by DADs. Using the summarized information, we determine whether we can prove independence or covering. The operations performed on the DADs are set intersection, set union, and tests for set containment. These tests are described in Balasundaram's dissertation[Bal89]. We have already described how may information can be collected for inner loops; this is the calculation of *loopmay*. Others have treated this problem for call sites[Cal87, Co083].

In order to use the summary information to prove covering we must discover when the action of an inner loop, over all of its iterations, kills an entire array or part of an

array. This is an alternate statement of the problem of discovering must information about the effect of a loop on an array. In the next section, we describe a method for acquiring information about the region of an array killed in a loop. We then present the generalized form of the dependence update algorithm.

3.3.2 Covering by a Loop

Restricting ourselves to affine array expressions, an exact solution to the problem of array kills by a loop requires solving the same integer programming problem as required for an exact solution to dependence. Thus, the problem is NP-hard. We present, therefore, an approximation.

Within an inductive do-loop, kills to array sections occur through assignments to elements of the array. The subscript expressions of these assignments appear as functions of the induction variables of the loops surrounding the assignments. The only assignment statements of a loop body that can kill an array definition are the assignments that will be executed every time the loop body is entered. These assignments are found in the *loopmust* set for the loop. In order to make the problem tractable we restrict the subscript expressions we examine to be expressions which are affine functions of only a single induction variable. An array kill involving subscript expressions that are either not affine or involve more than one induction variable will not be discovered.

Theorem 3.10 Given an array A with vector denoted

$$A(i_1, i_2, \dots, *, \dots, i_n)$$

and a definition of A within a loop,

$$A(i_1, i_2, \dots, j, \dots, i_n),$$

the vector is killed by the definition if and only if j takes on all integral values from 1 to N .

This is illustrated in the following example.

```

DIM A(100)
DO 100 I =1, 50
    A(2I - 1) = ...
    A(2I) = ...
ENDDO

```

In this example, the entire array A is killed by the combined action of the two statements.

Generally, an array A is killed when one or more assignments to the array within a loop body collectively assign to the entire array or subarray over the range of the induction space. To discover these cases we examine the constant terms and coefficients of the induction variable in all the subscript expressions within the loop.

Consider the set of subscript expressions in a single dimension of an array,

$$\{a_i * I + b_i\} \text{ for } i = 1 \text{ to } m,$$

where I is the induction variable of the surrounding loop in which we are currently interested. We gather all the expressions with equal a_i 's and call this set a *comb*. The product of the induction variable coefficient, a_i , and the step of the induction variable is called the *length* of the comb. (Negative lengths are acceptable.) If the comb includes additive constants b_i for the integral range from some j to $a_i - 1 + j$, then the comb is *complete*. See figure 3.3. If a loop body contains a complete comb, then a single iteration of the loop defines a contiguous portion of the array of size at least equal to the length of the comb.

If the induction variable of the loop obtains values from $(1 - j)/length$ to $(N - length + 1 - j)/length$ (Where N is the size of the dimension of the array), for positive comb lengths, or from $(1 - j)/length$ to $(N + length - 1 - j)/length$ for negative comb lengths, then that entire dimension of the array is defined. See figure 3.3. Thus, from the loop bounds we can calculate the possible values of j in the comb completeness

```

DO I = 1,N,4
  A(I + 1) = ...
  A(I) = ...
  A(I + 2) = ...
  A(I - 1) = ...
ENDDO

```

-1	0	1	2
----	---	---	---

A Complete Comb

```

DO I = 1,N,4
  A(I + 1) = ...
  IF F(I) A(I) = ...
  A(I + 2) = ...
  A(I - 1) = ...
ENDDO

```

-1		1	2
----	--	---	---

An Incomplete Comb

Figure 3.3 Combs

test above. Algorithms 3.14 and 3.15 contain a batch and incremental algorithm, respectively, to find array kills.

Theorem 3.11 Within a loop, if a complete comb exists on an array A in subscript position k , and the loop body is executed for all integral values of the induction variable, then the dimension associated with the comb is killed by the loop.

Occasionally a loop, l , will define only part of an array. If a loop, m , executed after l , restricts its access to the part of the array defined in l , it will be useful to detect that l kills the part of the array that m uses. Using combs we can easily discover what part of the array was defined in l . If the representation of the summary information is capable of representing bound information, this information can be calculated and used in the dependence calculation.

```

procedure find_combs(lm, var, ivar):
/* lm is the set loopmust calculated by another phase of dependence analysis */
/*
/* var is the name of the array variable about which we are determining */
/* kill information. */
/* istep is the step induction variable of this inductive-do loop. */

```

partition *lm* into sets such that all the members of a set have equal coefficients of the induction variable.

define *Teeth*[*MaxCombLength*, *MaxCombLength*]

For each set, *s*, with coefficient, *a*, from smallest coefficients to largest coefficients

/*We know each set has a distinct constant additive term */

if size of *s* $\geq a$ **then do**

$l = a * istep$

 order the constant additive terms

 step through this ordered list. If there exist *l* adjacent numbers, then a comb is found

 let *j* be the smallest of these adjacent numbers

 lower bound of array defined by this comb is $a * lb + j$.

 upper bound of array defined by this comb is $a * ub + j + l - 1$

if bounds are insufficient to cover array **then** continue looking for combs

endif

endfor

Algorithm 3.14 Finding Array Kills

```

procedure FindKills(DeltaLoopMustPlus, DeltaLoopMustMinus)

```

```

for all elements, e, of DeltaLoopMustMinus

```

```

    a = coefficient of loop induction variable in e

```

```

    tooth = constant term in e

```

```

    if tooth is an element of comb of length istep * a

```

```

        then mark comb as invalidated.

```

```

    endfor

```

```

if no comb marked invalidated and kill exists

```

```

    then return

```

```

for all elements of DeltaLoopMustPlus, e

```

```

    a = coefficient of loop induction variable in e

```

```

    tooth = constant term in e

```

```

    if comb of length a * istep is marked invalidated

```

```

        then attempt to prove comb again, using tooth

```

```

    else attempt to prove new comb with length a * istep using
        all other elements of loopmust.

```

```

        if attempt is successful and bounds of new kill better than
            old bounds then retain comb and new kill

```

```

    endfor

```

```

if a complete kill existed before edit, and now does not

```

```

    then attempt to prove combs with members of loopmust with coefficients not
        represented in DeltaLoopMustPlus

```

```

end FindKills

```

Algorithm 3.15 Incremental Discovery of Array Kills

It is possible for combs of different lengths to combine to define a contiguous area of an array, but it will be difficult to prove covering in this more general case and we believe that these cases will rarely occur.

3.3.3 Nested Loops

Nested loops complicate subscript testing in two distinct ways. When testing two array references for independence, the test is made for either a loop-independent dependence or a loop-carried dependence carried at a particular *level*. Loops outside the loop carrying the dependence affect the calculation in a completely different way than loops within the loop carrying the dependence.

If the subscript contains an induction variable of a loop outside the loop carrying the dependence being tested, the value of this induction variable will be equal at the point of evaluation of the two subscript expressions and can be treated as a loop invariant symbol by the independence tests. However, the action of the outer loop increases the ranges of the two subscript expressions and makes intersection and dependence more likely. For instance, consider the following code fragment.

```

DO I = 1, 2
  DO J = 1, N
    A(2I+J)=...
    ...=A(I+J+N+1)
  ENDDO
ENDDO

```

A comparison of the minimum and maximum values of the two references inside a single iteration of the I loop finds that the minimum and maximum values of the subscript in the definition are $2I+1$ and $2I+N$, respectively, and for the use the minimum and maximum are $I+N+2$ and $I+2N+1$. For $I=1$, the references will not share a memory location. But for $I=2$, the references *will* share a memory location, $A(N+4)$, thereby causing a dependence. The effect of the loops outside the carrying

loop is taken into account in the independence tests given by Banerjee[Ban79]. Since covering is performed by statements or loops within the loop carrying the tested dependence, the presence of outer loops does not affect the covering test.

Loops within the loop carrying the dependence being tested complicate the problem in a different way. Independence tests must assume that the induction variables of the inner loop take on the entire range of their values. In the following code fragment the range space of the subscript expressions is shared over the space of the two loops at the inner level although the subscript expressions themselves are not equal. Thus, when the two *J* loops are executed, they share the same memory locations during a single iteration of the outer *I* loop. This results in a loop-independent dependence between the statements *S1* and *S2*.

```

                DO I = 1, N
                    DO J = 1, N
S1:                A(I,J)= ...
                    ENDDO
                DO J = 1, N-1
S2:                ...=A(I,J+1)
                    ENDDO
                ENDDO

```

In addition, an inner loop may create a cover as we saw in the previous section. The array kill information calculated by the previous section must be used to test for covering between dependences.

The complications due to nested loops are neatly accommodated in a generalized algorithm if we calculate dependences from the innermost loops to the outermost loops. As the algorithm moves its attention from a loop at level *k* to a loop at level *k*-1, the access descriptors contained in the *loopmay* and *loopmust* sets at level *k* go through a process called *translation* to convert them from DADs in terms of the induction variables of the *k* outer loops to DADs in terms of the induction variables

of the outer $k-1$ loops [Bal89]. The translated access descriptors contained in *loopmay* and *loopmust* are collected in sets called *Xmay* and *Xmust* respectively. Uncovered uses are likewise translated and placed in a set called *Xuses*.

At the entry of a loop, we store the summary information containing the translations of definitions and uses from within the loop. Thus, when a definition is added outside a loop l , when the change is propagated through l , we can perform the tests for independence and covering between the added definition and uses contained in l without visiting any statement in l .

AddDef, DelDef, AddEdge, and DelEdge begin by calculating dependences and updating the sets maintained for each statement and loop at the innermost level. If the information contained in *loopmay* or *loopmust* has changed as a result of the update at the innermost level, the additional elements of *loopmay* and *loopmust* will be translated and added to *Xmay* and *Xmust* at next higher level, and the translated versions of the elements deleted from *loopmay* and *loopmust* will be deleted from *Xmay* and *Xmust*. After the translation, the algorithms call themselves at the next outer loop level in order to propagate the changes beyond the loop. The inner loop is replaced in the analysis by a pseudo statement whose definitions will be translated *loopmay* and *loopmust* sets. This process continues until during the translation phase the information contained in *Xmay* and *Xmust* does not change. The routines to propagate the changes to *may* and *must* sets are unchanged from their earlier versions.

AddUse and DelUse act similarly. The two algorithms, after updating the dependences at the innermost loop, translate the changes to uses not covered by a definition inside the loop to the next outer level, and call themselves at the next level. This continues for each level containing the new use in AddUse and Del.

We present the generalized forms of AddUse and AddDef in algorithms 3.16 and 3.17, respectively. The generalizations of the other routines are similar.

```

program AddUse(s, var, use)
/* s is the statement location containing the new use
/* var is the variable involved in the new use
/* use holds the expression containing the use. e.g. A(I)
if s is a loop entry statement then
    add use to Xuses(s)
p = predecessor of s
for all elements, (u:A+), of may(p)
    call TestForLoopIndependent((u:A+), use)
endfor

for all elements, W, of must(p) do until matchfound
    matchfound = TestForContainment(W, use)
endfor

if not matchfound then do
    for all elements, W, of loopmust(s)
        M = min(M, FindThreshold(W, s))
    endfor

    for all elements, w, of loopmay(s)
        call TestForCarriedIndependence(w, use, M)
    endfor
    Xu = translation of use to next higher loop level
    Xs = statement(not an assignment) containing the entry to this loop
    call AddUse(Xs, var, Xu)
endif
end AddUse

```

Algorithm 3.16 Updating Dependences in
Response to Addition of a Use

```

program AddDef(s, var, def, mustflag)
/* s is the statement containing the new def
/* var is the variable involved in the new definition
/* def holds the added definition
if s is a loop entry statement
    then add def to Xdef(s)
p = predecessor(s)
oldmay = may(s); oldmust = must(s)
may(s) = may(p)  $\cup$  (s : def)
if mustflag
    then must(s) = must(p) - {any definitions covered by def} + def
DeltaMay = oldmay - may(s)
DeltaMust = oldmust - must(s)

if DeltaMay is not empty then do
    for all successors, r, of s
        call PropMayChange(s, r, DeltaMay)
    endfor
if DeltaMust is not empty then do
    for all successors, r, of s
        call PropMustChange(s, r, DeltaMust)
    endfor
endif

if MayXIteration then update loopmay
if MustXIteration then update loopmust

if loopmust changed, then do
    update all carried dependences on the array throughout the loop body
    Xs = innermost containing loop's entry statement
    Xd = translation of def to next higher level
    call AddDef(Xs, var, Xd, mustflag)
endif

if only loopmay changed, then do
    update carried dependence where necessary
    Xs = innermost containing loop's entry statement
    Xd = translation of def to next higher level
    call AddDef(Xs, var, Xd, mustflag)
endif

end AddDef

```

Algorithm 3.17 Update for Addition of a
Definition in Presence of Nested Loops

3.4 Calculation of Anti and Output Dependences

The algorithms presented in this chapter calculate an approximation to strong true dependences. Modifications to these algorithms are required to calculate other types of dependence.

The algorithms determine whether a path exists from a definition of an array element to a use of the array element that does contain a definition to that array element. An anti dependence exists if a path exists from a use of an array element to a definition such that the path does not contain a definition to the array element. Such a path exists in a program from a use to a definition if and only if such a path exists in the reverse program from the definition to the use. Hence, in order to calculate strong anti dependences, we can apply the same algorithms to the program, reversing the order of statements and basic blocks in the program. In addition, it will be necessary to calculate thresholds as though the iterations of the loops in the program occur from last to first. This has the effect of changing the sign on the thresholds.

Definitions are both the source and sink of output dependences. Since the addition and deletion of uses have no effect on output dependence, AddUse and DelUse are unnecessary for their calculation. Instead, the functions of AddUse and DelUse must be placed in AddDef and DelDef, respectively. the code inside AddUse and DelUse may be added without change except that the added or deleted definition takes the place of the use.

3.5 Complexity Analysis

The complexity analysis for AddUse is straightforward. The new use must be compared to all the elements of $may(p)$, $must(p)$, $loopmust$, and $loopmay$, where p is the predecessor of s . In addition, the use must be translated to the levels of all the loops containing the new use and the same comparisons must also be performed at those

levels. The time for the comparisons and the translations is bounded by the time required at the deepest level. If the use is contained in k loops and we assume k to be small, then the time required to update data dependence in response to the addition of an array use is $O(k \cdot (\|may(p)\| + \|must(p)\| + \|loopmust\| + \|loopmay\|))$.

DelUse is even easier to analyze. It merely deletes the dependences with the use as sink. Thus, its complexity can be expressed in terms of the out-degree of the data dependence subgraph, $O(\text{in-degree of } s)$.

AddDef propagates the new definition around the innermost loop everywhere that the *may* or *must* information changes, and when a use is present, compares the use against the new definition. Let the number of nodes reached by the definition at a particular level k be η_k . The propagation of changes to *may* and *must* sets within the innermost loop containing the new definition and the calculation of loop-independent dependences requires $O(\eta_k)$ operations. If the new definition crosses an iteration, then loop-carried dependences must be calculated throughout the loop. Let the number of uncovered uses of the array contained in the loop be $\|\eta_u\|$. The calculation of loop-carried dependences for a particular loop requires $O(\|loopmust\| + \|loopmay\| \cdot \|\eta_u\|)$ subscript comparisons. The definition is then translated to the next higher level and the process repeated. If we assume that the time required for a translation is bounded, it takes $O(\sum_k (\eta_k + \|loopmust_k\| + \|loopmay_k\| \cdot \|\text{uses}_k\|))$ to update data dependences in response to the addition of a definition enclosed in k loops.

To update in response to a definition deletion requires deleting dependences with their source at the deleted definition. In addition, the *may* and *must* sets containing the deleted definition must be recalculated. If the change propagates to the *loopmay* or *loopmust* set, then loop carried dependences on the array throughout the loop body must be recalculated as well. The time required to update the dependences is proportional to the out degree of the definition, which is bounded by the out degree of the data dependence subgraph. The number of *may* and *must* sets at level k containing the deleted definition will be η_k . Calculating loop-carried dependences

can require $O(\|loopmust\| + \|loopmay\|)$ subscript comparisons if the *loopmust* set changes. Thus, the time required to update in response to definition deletion is the same as for definition addition, $O(\sum_k \eta_k + (\|loopmust_k\| + \|loopmay_k\| \cdot \|uses_k\|))$.

The update in response to control flow edge addition involves calls to the *may* and *must* propagation routines. The update in response to the edit involves all the variables of the program. The changes to the *may* and *must* sets at the sink of the added edge resulting from the edit are calculated for each variable in the program. Then the propagation routines (PMCPPlus, PMustCMinus) are called to propagate the changes. Let V be the set of array variables in the program. Then the complexity of AddCF is $O((\sum_{v \in V} (2 \sum_k (\eta_k + \|loopmust_k\| + \|loopmay_k\| \cdot \|uses_k\|))))$. The update in response to control flow edge deletion is similar and results in the same complexity.

In the worst case, the time required for AddCF or DelCF is equal to the time required for executing the batch algorithm to calculate dependences for the entire program. However, the *may* and *must* propagation routines (PMCPPlus, PMCMinus, PMustCPlus, PMustCMinus) only visit statements in the program where the *may* or *must* information has actually changed as a result of an edit. Hence, DelCF and AddCF are optimal calculations of the changes in the *may* and *must* information for the data structure holding this information.

If a control flow edge is added so that the sink of the edge is close to the source, then it is likely that few variables, relative to the number of variables contained in the program, will have a difference between their *may* or *must* sets at the source and sink of the added or deleted edge. Thus, a “typical” edit can be expected to require the propagation of changes to this information for only a small number of variables.

Chapter 4

Symbolic Analysis for Subscript Testing

Both the independence tests and the tests for covering used in our dependence analysis require subscript expression pairs to be affine functions of the induction variables of containing loops. The covering test presented in the previous chapter requires that these functions be of an even more restrictive form, $(aI + b_1)$ and $(aI + b_2)$ where I is a loop induction variable and a and b_i are loop invariant.

Unfortunately, not all subscripts, even when they can be expressed in these forms, will be so written by the programmer. For instance, another variable might be used in place of the loop induction variable. Even when the subscripts are written in these forms, analysis must be done to recognize loop invariant expressions.

Thus, accurate dependence analysis on subscripted variables requires the results of a number of symbolic scalar analyses. Allen and Kennedy described the importance of symbolic scalar analysis for subscript testing and its implementation in PFC[AK84]. In this chapter we present new incremental algorithms to calculate the results of three symbolic scalar analyses. These scalar analyses are

1. Integer Expression Folding,
2. Loop Invariant Testing, and
3. Auxiliary Induction Variable Identification.

These analysis phases will use the dependence graph for scalar variables. The dependence graph for scalars can be incrementally built using Zadeck's techniques [Zad84]. Zadeck describes how to build def-use chains. For scalars, def-use chains are identical to true dependences. Zadeck's technique is easily converted to build

1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
27
28
29
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
160
161
162
163
164
165
166
167
168
169
170
171
172
173
174
175
176
177
178
179
180
181
182
183
184
185
186
187
188
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
220
221
222
223
224
225
226
227
228
229
230
231
232
233
234
235
236
237
238
239
240
241
242
243
244
245
246
247
248
249
250
251
252
253
254
255
256
257
258
259
260
261
262
263
264
265
266
267
268
269
270
271
272
273
274
275
276
277
278
279
280
281
282
283
284
285
286
287
288
289
290
291
292
293
294
295
296
297
298
299
300
301
302
303
304
305
306
307
308
309
310
311
312
313
314
315
316
317
318
319
320
321
322
323
324
325
326
327
328
329
330
331
332
333
334
335
336
337
338
339
340
341
342
343
344
345
346
347
348
349
350
351
352
353
354
355
356
357
358
359
360
361
362
363
364
365
366
367
368
369
370
371
372
373
374
375
376
377
378
379
380
381
382
383
384
385
386
387
388
389
390
391
392
393
394
395
396
397
398
399
400
401
402
403
404
405
406
407
408
409
410
411
412
413
414
415
416
417
418
419
420
421
422
423
424
425
426
427
428
429
430
431
432
433
434
435
436
437
438
439
440
441
442
443
444
445
446
447
448
449
450
451
452
453
454
455
456
457
458
459
460
461
462
463
464
465
466
467
468
469
470
471
472
473
474
475
476
477
478
479
480
481
482
483
484
485
486
487
488
489
490
491
492
493
494
495
496
497
498
499
500
501
502
503
504
505
506
507
508
509
510
511
512
513
514
515
516
517
518
519
520
521
522
523
524
525
526
527
528
529
530
531
532
533
534
535
536
537
538
539
540
541
542
543
544
545
546
547
548
549
550
551
552
553
554
555
556
557
558
559
560
561
562
563
564
565
566
567
568
569
570
571
572
573
574
575
576
577
578
579
580
581
582
583
584
585
586
587
588
589
590
591
592
593
594
595
596
597
598
599
600
601
602
603
604
605
606
607
608
609
610
611
612
613
614
615
616
617
618
619
620
621
622
623
624
625
626
627
628
629
630
631
632
633
634
635
636
637
638
639
640
641
642
643
644
645
646
647
648
649
650
651
652
653
654
655
656
657
658
659
660
661
662
663
664
665
666
667
668
669
670
671
672
673
674
675
676
677
678
679
680
681
682
683
684
685
686
687
688
689
690
691
692
693
694
695
696
697
698
699
700
701
702
703
704
705
706
707
708
709
710
711
712
713
714
715
716
717
718
719
720
721
722
723
724
725
726
727
728
729
730
731
732
733
734
735
736
737
738
739
740
741
742
743
744
745
746
747
748
749
750
751
752
753
754
755
756
757
758
759
760
761
762
763
764
765
766
767
768
769
770
771
772
773
774
775
776
777
778
779
780
781
782
783
784
785
786
787
788
789
790
791
792
793
794
795
796
797
798
799
800
801
802
803
804
805
806
807
808
809
810
811
812
813
814
815
816
817
818
819
820
821
822
823
824
825
826
827
828
829
830
831
832
833
834
835
836
837
838
839
840
84

expressions have been fully folded and their form simplified and put into normal form, the potential for successful application of independence and covering tests is greatly enhanced.

The algorithm for integer expression folding works as follows. Consider a subscript expression, s . For every variable v in s , we will determine whether another expression, e , can be used in place of v in s . If more than one true dependence¹, that is, a def-use link, reaches the use of v in s , and any of the right hand sides of these definitions are not identical, then no folding may take place for the use of v , since it is not possible to determine which definition of v should be folded into the use of v in s . Otherwise, the right hand side of the definition, d , which reaches s is examined, to determine whether it can be folded into s .

If d contains only integer variables, constants, and arithmetic operators (no function or procedure calls), then it can be folded into s if none of the variables involved in d can be redefined before their use in s . To determine this, for every variable x in d , we compare the reaching definitions of x at d with the reaching definitions of x at s . If they are identical, then the substitution can proceed. If they are not identical, then the definitions which reach s that do not reach d are marked "blocking".

In order to perform integer expression folding in response to a series of edits, we need to be able to invalidate folds as they become invalid and recognize new opportunities for folding as they occur. Obviously, if a definition used in a fold is changed, then the particular substitution is no longer valid, though the opportunity for a new fold might be created. This implies the need for a link from a folded definition to the point where it is substituted. We will mark as "folded" all definitions folded forward at the point of their original placement. A link will be placed to the point where the definition is substituted. These links will form a chain from the folded expression to all the shadow expressions into which the expression has been

¹A true dependence will exist from a special node, START, to all uses of a variable that can be reached prior to a definition within the containing procedure. Constants propagated into the procedure will be represented this way.

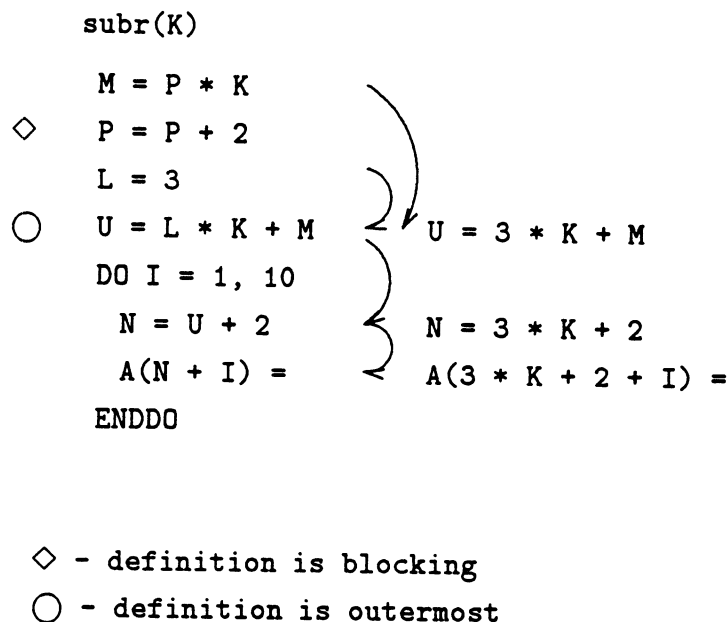


Figure 4.1 Shadow Expressions and Links

substituted. In order to recognize when an added definition invalidates a fold by making the reaching definitions set at the sink of the fold different from that at the fold's source, we will mark the reaching definition set for the substituted variable at the substituted use as "sink important", to represent the fact that the reaching definition set is important to a fold with its sink at this statement. Likewise, we will mark as "source important" the reaching definition sets of the variables that appear in a folded expression. This indicates that the reaching definition set of a particular variable is important to a fold whose source is at this statement. When one of these sets changes, by traveling the links backward, we can readily invalidate all the necessary folds.

In order to discover new opportunities for folding as they arise, we mark as "outermost" any definition whose left hand side has been folded forward and whose right hand side contain unfolded variables. These are the definitions at which it would benefit us to substitute forward something on the right hand side. If the true de-

pendences incident on one of these “outermost” marked definitions change, then we can check for possible new opportunities for folding into the right hand side of the definition. Definitions that are discovered to block a fold will be marked “blocking”. In addition to the mark, a link will be included from the blocking definition to the sink of the prevented fold. If the blocking definition is deleted, then we can try to perform the relevant fold.

The results of integer expression folding will be stored in shadow expressions at every location where a substitution has taken place. Figure 4.1 illustrates the links and shadow expressions that result from the application of our algorithm for a particular code fragment. The update to integer expression folding is driven by the additions and deletions of definition and uses and to changes to the reaching definitions sets. The algorithm we present comprises six programs:

- AddUse is called when a use is added to the right hand side of an assignment statement;
- DelUse is called when a use is deleted from the right hand side of assignment statement;
- AddDef is called when a definition (an assignment statement) is added;
- DelDef is called when a definition (an assignment statement) is deleted;
- AddCF is called when a control flow edge is added between two existing statements;
- DelCF is called when a control flow edge is deleted between two statements that remain in the modified program.

Theorem 4.1 Given a program P , the folded expressions for P , and the links and marks on reaching definitions in P and a use added to form program P' , AddUse in Algorithm 4.1 will form the correct folded expressions and links and marks for P' .

```

program AddUse(s,use)
/* s is the statement to which use has been added */

call ChangeDefUnfold(s)

if use is contained in a subscript or s is marked "outermost" or s is the sink of a fold
  then do
    let v be the variable contained in use
    call TryFold(v,s)
  endif

end AddUse

```

Algorithm 4.1 Fold Update for Added Use

```

program DelUse(s,use)
/* s is the statement from which use has been deleted. */

call ChangeDefUnfold(s)

call DelUseUnfold(s, use)
call TryFold(use)

end DelUse

```

Algorithm 4.2 Fold Update for Deleted Use

```

program AddDef(s)
/* s is an added assignment statement containing a scalar definition */
let v be the name of the variable on the left hand side of s
let l be a list of statements whose reaching definition sets of v have been modified

for every member, w, of l
    call ChangeReachingSetUnfold(v,w)
endfor

for every member, w, of l
    call ChangeReachingSetFold(v,w)
endfor

end AddDef

```

Algorithm 4.3 Fold Update for Added Definition

Proof Since the expression on the right hand side has been changed, any fold from *s* is invalidated. The addition of a use has no effect on the value of reaching definitions sets. Hence, from Definition 4.1 no fold is invalidated whose source is not at *s*. Folds from *s* are invalidated in AddUse by a call to ChangeDefUnfold. Thus, the marks, links, and expressions associated with any fold invalidated by the addition of a use are removed or undone by AddUse.

The definition whose right hand side contains the added use might be folded after the addition of the use. In order to determine this AddUse calls AddUseFold on the statement containing the added use. AddUseFold determines if the statement containing the added use is marked “outermost” and then tests the conditions contained in Definition 4.1 to see if the definition can be folded forward. Then, if the definition is marked “outermost”, AddUseFold tests if any definitions can be folded into the use. Thus, any folds created after the addition of a use are performed by AddUse.

□

```

program DelDef(s)
/* s is a deleted assignment statement containing a scalar definition */
let v be the left hand side of s
let l be a list of statements whose reaching definition sets of v have been modified

if s is marked as "folded" then do
    for every link, link from s
        let t be the sink of link
        if link is the only link to t
            then remove from the shadow expression at t the expression folded from s
            remove link
            /* Since s is deleted it is unnecessary to mark as not "folded". */
        endfor
    endif

for every member, w, of l
    call ChangeReachingSetFold(v,w)
endfor

if s is marked "blocking" then do
    use block link to find statement w containing use blocked from substitution by s
    call TryFold(v,w)
endif

end DelDef

```

Algorithm 4.4 Fold Update for Deleted Definition

```

program AddCF()
let  $l$  be a list of pairs of statements and variables whose reaching
    definition sets have been modified

for every member,  $(w,v)$ , of  $l$ 
    /* the reaching definitions set of variable  $v$  has changed at  $w$  */
    call ChangeReachingSetUnfold( $v,w$ )
endfor

for every member,  $(w,v)$ , of  $l$ 
    /* the reaching definitions set of variable  $v$  has changed at  $w$  */
    call ChangeReachingSetFold( $v,w$ )
endfor

```

Algorithm 4.5 Fold Update for Added Control Flow Edge

```

program DelCF()

let  $l$  be a list of pairs of statements and variables whose reaching
    definition sets have been modified

for every member,  $(w,v)$ , of  $l$ 
    /* the reaching definitions set of variable  $v$  has changed at  $w$  */
    call ChangeReachingSetUnfold( $v,w$ )
endfor

for every member,  $(w,v)$ , of  $l$ 
    /* the reaching definitions set of variable  $v$  has changed at  $w$  */
    call ChangeReachingSetFold( $v,w$ )
endfor

```

Algorithm 4.6 Fold Update for Deleted Control Flow Edge

```

procedure ChangeReachingSetUnfold(v,s)
/* The reaching definitions set of v at s has been changed */
if the reaching definition set of v at s is marked "sink important" then do
    for every expression e folded into shadow expression at s
        if e includes v then do
            remove e from shadow expression, replacing substituted variable
            mark s outermost /* Since shadow expression now contains an unfolded variable */
            for all sources, src, of links, l, for which use of v at s is the sink
                if l is only link out of src then do
                    mark definition at src as not "folded"
                    mark reaching set of v at src as not "source important"
                endif
            delete l
        endfor
    endif
endfor
endif
if the reaching definition set of v at s is marked "source important" then do
    for every sink sink of link l with source at s
        replace folded expression of l from shadow expression at sink with left hand side of s
        for every definition d in reaching definition of v at sink
            not present in reaching definition of v at s
            mark d as "blocking"
        endfor
    endfor
    mark reaching definition set of v at s as not "source important"
    mark definition at s as not "folded"
endif

```

Algorithm 4.9 Delete Folds for Changes in Reaching Sets

```

procedure ChangeReachingSetFold(v,s)

if s is marked "outermost" and v is unsubstituted and appears on right hand side of s
    then call TryFold(v,s)

```

Algorithm 4.10 Create Folds for Changes in Reaching Sets

```

program DelUseUnfold(use)
/* use is a particular use of a variable which has been deleted */
if use is not the sink of a fold then return

let v be the variable in use
let s be the statement containing use
if the deleted use is the only appearance of v in s
    then mark reaching definition set of v as not "sink important"
for every link l with sink at use
    let d be the source of l
    if l is the only link with source at d
        then
            mark d as not "folded"
            mark reaching definition of v at d as not "source important"
            for every folded use, u on right hand of d
                call DelUseUnfold(u)
            bf endfor
        endifor
    delete l
    endifor
end DelUseUnfold

```

Algorithm 4.7 Delete Folds for Deleted Use

```

program ChangeDefUnfold(s)
/* s is the location of a modified definition */
let v be the variable on the left hand side of s
for every link, l, for which s is source
    reverse fold at sink of l, replacing right hand side of s with v
endifor
mark s as not "folded"

end ChangeDefUnfold

```

Algorithm 4.8 Delete Folds for Changed Definition

```

procedure TryFold(v,use)
/* Test whether an expression can be folded for the uses of v in s */
let s be the statement containing use
let D be the set of sources of all the true dependences on v with sink at s
if right hand sides of all members of D are equal to a single expression, e, then do
  for every variable, x, in e
    test if reaching definitions of x at s and members of D are identical
    if test fails then do
      for {every definition d in reaching definition set at s
        that is not present in a reaching definition set at some member of D}
        mark d as "blocking"
        add block link from d to use of v in s
      endfor
      TestFailed = true
    endif
  endfor
if not TestFailed /* over all x */ then do
  fold e into uses of v in shadow expression at s
  place link from every element of D to s
  mark every member of D as "folded"
endif
endif

```

Algorithm 4.11 Attempt a Particular Fold

Theorem 4.2 Given a program P , the folded expressions for P , and the links and marks on reaching definitions in P and a use, u , contained in a statement s , that is deleted to form program P' , DelUse in Algorithm 4.2 will form the correct folded expressions and links and marks for P' .

Proof Since the expression on the right hand side has been changed, any fold from s is invalidated. The deletion of a use has no effect on the value of reaching definitions sets. Hence, from Definition 4.1, no fold is invalidated whose source is not at s or whose sink is not u . Folds from s are invalidated in DelUse by a call to ChangeDefUnfold. Thus, the marks, links, and expressions associated with any fold invalidated by the addition of a use are removed or undone by AddUse. Folds into u are removed by DelUseUnfold.

The definition whose right hand side contains the deleted use might be folded after the deletion of the use. If the definition is marked "outermost", DelUse calls TryFold to test if any definition can be folded into the use and perform the fold if so. Thus, any folds created after the addition of a use are performed by AddUse. \square

Theorem 4.3 Given a program P , the folded expressions for P , and the links and marks on reaching definitions in P and a use deleted to form program P' , AddDef in Algorithm 4.3 will form the correct folded expressions and links and marks for P' .

Proof When an assignment statement containing a scalar definition is added to P , reaching definition sets for the variable on the left hand side change at statements that are reached by the added definition. These are the only reaching definitions sets that can possibly change as a result of the added definition. ChangeReachingSetUnfold tests for all three conditions contained in Definition 4.1 which might no longer hold given a change in the reaching definitions sets at a statement. If a fold is found to be invalidated, ChangeReachingSetUnfold reverses the fold. Since ChangeReach-

ingSetUnfold is called for every statement where the reaching definition sets change, any fold that is invalidated by the added definition is reversed by AddDef.

New opportunities for folding arise at the same statements where the reaching definition sets change. ChangeReachingSetFold tests whether any fold involving the variable on the left hand side of the added definition can be performed at any of these statements and performs the fold if so. Inspection of Definition 4.1 shows that the conditions for an integer expression fold can be expressed solely in terms of the reaching definition sets at the source and sink of the fold. Thus, only a statement where a reaching definition set has changed can possibly be the source or sink of a new fold. Since all of these are inspected by ChangeReachingSetFold, any opportunities for folding are found.

Note that any new folding into the right hand side of the statement containing the new definition will be performed by calls to AddUse. □

Theorem 4.4 Given a program P , the folded expressions for P , and the links and marks on reaching definitions in P and a use deleted to form program P' , DelDef in Algorithm 4.4 will form the correct folded expressions and links and marks for P' .

Proof When a definition is deleted, it will be removed from any reaching definition set in which it appears. Thus, if two reaching definition sets containing the deleted definition are identical before the deletion of the definition, then they are identical afterward. Hence, no fold will be invalidated by a violation of the third condition in Definition 4.1. Condition 2 of Definition 4.1 cannot be invalidated by the deletion of a definition. Thus, the only way in which a fold can be invalidated by the deletion of a definition is by eliminating the only definition that reached the sink of the fold, implying that the deleted definition is the source of the fold. All folds invalidated by the deletion of the definition will be reversed by reversing the folds of the deleted definition in the shadow expressions in which it appears.

If a definition blocks a fold from occurring, then its deletion can result in the fold. Whether the deleted definition is marked “blocking” and whether this results in a new fold is tested in DelDef. The rest of the proof concerning new opportunities for folding is identical to the argument contained in the proof for Theorem 4.3. \square

Theorem 4.5 Given a program P , the folded expressions for P , and the links and marks on reaching definitions in P and a control flow edge added to form program P' , AddCF in Algorithm 4.5 will form the correct folded expressions and links and marks for P' .

Proof The addition of a control flow edge will affect reaching definition sets for any variable that is defined on all paths from s to t in P , if at least one exists, or any variable that is defined on a path to t from a node that does not reach s if no path from s to t exists in P . The same argument contained in the proof for AddDef leads to the conclusion that a fold can be invalidated only if the reaching definition sets change at either the source or the sink of the fold.

By noting the scalar variables and statements that specify a particular reaching definition set that is modified, we can use a call to ChangeReachingSetUnfold and ChangeReachingSetFold to test the statements where the reaching definition sets change to see if they are the source or sink of either an invalidated or new fold. This is done in the two for loops contained in AddCF. \square

Theorem 4.6 Given a program P , the folded expressions for P , and the links and marks on reaching definitions in P and a control flow edge deleted to form program P' , DelCF in Algorithm 4.6 will form the correct folded expressions and links and marks for P' .

The proof is identical to that for Theorem 4.1.

Complexity Analysis

The complexity analysis of the algorithms is straightforward. In response to a changed use, we trace and delete the links back to the definitions folded into the changed use. The time required for this is equal to the number of links that must be followed backward. These links must be deleted only if the definition at the head of the link is not folded forward into any other use. If a link is not deleted then it is not necessary to follow the link backward to the source of the link.

In response to a changed definition, we must perform a number of tasks. First, the uses into which this definition has been folded must be updated so that their shadow expressions reflect the changed definition. Depending on the nature of the change, this might require that the folds be undone or just a different right hand side be folded forward. Second, the changed definition might be one that is marked as blocking. If so, then it is necessary to find those expressions that can now be folded forward and fold them.

The time required to do this is proportional to the affected area, in the sense that no more work is required than the area of the change to the data structure holding the information. For a change to a use, the actual time required can be loosely bounded by the product of the in-degree of true dependences on a statement times the length of the longest chain of links. A change to a definition requires time proportional to no more than the out-degree of true dependences times the length of the longest chain.

4.2 Loop Invariant Testing

Subscript testing can benefit from the knowledge that a variable appearing in two different expressions has the same value in both expressions. Such a variable can be deleted from the two expressions without affecting the outcome of any test on the intersection of the ranges of the two expressions. A variable that always has the same value regardless of where it is encountered in a loop is called *loop invariant*.

Any variable not defined within a loop is, obviously, loop invariant with respect to that loop. These variables can be easily detected at their use by checking whether the use is the sink of any true dependence from within the loop. Assuming that all loops are dominated by a single node, the loop header, whether there exist more than one true dependence from outside the loop is irrelevant.

Occasionally a variable defined within a loop body will be invariant with respect to the loop, as illustrated by the following code fragment.

```
SUB(L,M)
DO I = 1, 10
    K = L*M
    A(K+I) = ...
ENDDO
```

In this example the variable **K** is equal to the same value through all its uses in all the iterations of the loop that follow its definition.

Implicit in the above definition is the notion that a loop invariant variable is invariant with respect to a particular loop in which it is contained. We would like to be able to identify the loops for which it is invariant. A variable is invariant for all the loops containing the innermost loop for which it is invariant. However, there may still be loops containing the variable, contained within these loops, for which the variable is not invariant. Consider the following code fragment.

```

DO I = 1, N
  L = 0
  DO J = 1, N
    DO K = 1, N
      A(I,J,K,L) = F(I,J,K,L)
    ENDDO
    L = L + 1
  ENDDO
ENDDO

```

L varies with the outermost and the next inner loop. However, L does not vary in the innermost loop.

Definition 4.2 A variable, v , is *loop invariant* with respect to a loop l if and only if either

1. v is not defined within the body of l or
2. every use of v within l is reached only by definitions of v with right hand side, e , that contain only loop invariant variables and do not contain v or any call to a subroutine.

Integer expression folding provides enough information to recognize that a variable is loop invariant. If a variable is truly loop invariant and it appears in a subscript expression before integer expression folding, then either no definition of the variable appears within the loop or, in the shadow expression corresponding to its use, the variable will have been replaced by its definition that appears within the loop. Since integer expression folding continues to attempt folding on folded expressions within subscripts, it follows that, after the completion of integer expression folding, no loop invariant variable that is defined within the loop will appear in a shadow expression of a subscript expression. Hence, after the completion of integer expression folding,

to test whether a variable appearing in a shadow expression of a subscript expression is loop invariant we test whether any definition from within the loop reaches the use of the variable in the shadow expression. If no such definition reaches the use of the variable in the shadow expression, the variable is loop invariant. This test can be performed on the fly during subscript testing.

Theorem 4.7 If a variable v is loop invariant with respect to a loop l , then, after integer expression folding, the shadow expression representing the use of v will not contain any variable defined within l .

Proof From the definition we know that if v is loop invariant, then either v is not defined within l , satisfying the theorem, or the use is reached by definitions with identical right hand sides, e . e contains only loop invariant variables not equal to v . Let the set of variables defined within l be V . The only variables that may appear in e are those that are elements of $V' = V - v$. Consider any of these variables, v' . The right hand side of the definition reaching the use of v' can contain only variables defined outside l or members of the set $V'' = V' - v'$. Eventually the only variables that may appear in the shadow expression are those that are defined outside l . \square

4.3 Induction Variable Identification

For dependence analysis to effectively deal with subscripted variables, it must identify every variable within a subscript expression that can be treated as an *auxiliary induction variable*. Auxiliary induction variables behave as induction variables but do not appear as the induction variable of a loop. The independence tests for array accesses use the effect of induction variables and auxiliary induction variables on the value of subscript expressions to prove the values of the subscript expressions are different over iterations of a loop.

```

K = 0
DO I = 1, 100
    K = K + 2
    A(K) = F(A(K-1))
ENDDO

```

In the above example, K is an auxiliary induction variable with step 2. In the subscript expression, the use of K can be replaced by $I*2$.

Generally speaking, we would like to detect cases where a variable v in a loop, can be replaced by an expression of the form $i * c + a$, where i is the induction variable of the loop and c and a are constants.

Definition 4.3 A variable, x , is an *identifiable auxiliary induction variable* (IAIV) for a loop if and only if

1. the entry to the loop is reached by a single definition of x and
2. during every iteration of the loop, x is assigned a value equal to itself plus a loop invariant value.

An obvious case of an IAIV would be a variable with a single definition in the loop such as

$v = v +/- \text{loop invariant expression.}$

These cases are easy to identify. A more difficult case is illustrated by the following code fragment.

```

DO
    I = J + 1
    J = I + 1
    A(J) = ...
    ... = A(I)
ENDDO

```

I and **J** are called *mutual induction variables*. Their definitions result in a step of 2 for both **I** and **J** during a single iteration. An algorithm that searches for assignments to variables with a right hand side equal to the left hand side plus or minus a loop invariant expression will miss auxiliary induction variables of this type.

An algorithm based on integer expression folding can discover these cases quite easily. Using the shadow expressions left by integer expression folding, the above code fragment becomes, after one step,

```

DO
    I = J + 1
    J = I + 1
    A(I + 1) = ...
    ... = A(I)
ENDDO

```

Next, integer expression folding attempts to fold forward an expression for **J** in the right hand side of the definition of **I**. This will not be successful because **J** is reached by two definitions: one loop independent from outside the loop, the other loop carried. The loop carried definition is an inductive expression for **I**, the left hand side of the statement where we are attempting the substitution. From this information and the knowledge that this inductive assignment occurs every iteration, we can identify **I** and **J** as auxiliary induction variables. In order to see that the inductive assignment occurs on every iteration we check that the statement containing the assignment is directly control dependent on the loop header. Thus, we can identify auxiliary loop induction variables during integer expression folding. While testing for expression folding, the algorithm tests for the formation of statements with an inductive form in the shadow expression of the right hand side. When such a statement is found, the algorithm tests the control dependence relation to see that the statement occurs every iteration. See algorithm 4.12.

Input: An assignment statement s whose right hand side (possibly a shadow expression) is being examined for possible opportunities for folding into it.

Output: Identification of whether the left hand side is an IAIV

let the left hand side of s be the variable I

let the right hand side of s (possibly a shadow expression) of the assignment statement be rhs

where v is a program variable

where c and d are loop invariant expressions

if rhs is of form $v + c$:

 then if v is the sink of one loop-independent dependence
 and one loop-carried dependence

 then if the loop carried dependence is from a definition of the form $I + d$

 then if an edge from the loop header to s is contained in the CDG

 then I is an IAIV

Algorithm 4.12 Finding IAIVs

Theorem 4.8 Given an IAIV I in a loop, algorithm 4.12 will identify I as an IAIV.

Proof From definition 4.3, we know that on every iteration of the loop I is assigned to a value equal to itself plus a loop invariant value. From theorem 4.7, we know that the variables in the expression forming this loop invariant value will be replaced in the shadow expression with expressions involving variables defined outside the loop. From definition 4.3, the other part of the original expression is I . Thus, the shadow expression will consist of I plus an expression involving variables defined outside of the loop. Hence, the test in algorithm 4.12 will succeed and I will be identified as an IAIV. □

Complexity Analysis

The only non-constant work done in algorithm 4.12 is the testing of whether c is a loop invariant value. The amount of work for this testing is bounded by the number of variables in the expression. This work is added to the overall complexity of the integer expression folding algorithm. If the l is the length of the longest chain of integer expression folding links present, and v is the maximum number of variables on the right hand side of an assignment statement, then the time required to perform and identify IAIVs is bounded by $O(l + (v + 1))$.

Chapter 5

Experiments and Results

It was the requirement for quick response time in an interactive programming tool that led us to postulate the need for incremental techniques of dependence analysis. As a step to confirming the practicality of the techniques, we analyzed the time required by the algorithms presented in this dissertation to perform a single update. These complexities appear as functions of characteristics of the programs being edited, the dependence graphs of the program, and the edit being made. See Table 5.2.

Unfortunately, these analyses alone are inadequate to confirm the practicality of the incremental algorithms. It is always possible to construct a program and an edit which will require a complete recalculation of the dependence information of the program. What is needed is a calculation of the performance of these algorithms when presented with the kind of programs and edits that they are likely to encounter.

Logs of editing sessions with programmers using interactive parallelism tools are not currently available, but examples of the large computationally intensive programs these programmers will be editing *are* available. We have used them to produce measurements of the characteristics of programs and edits which are important to the performance of our algorithms.

For our study we selected four programs from the Rice Compiler Evaluation Program Suite (RiCEPS). They are described in Table 5.1. RiCEPS is a collection of whole FORTRAN programs designed to provide examples of common programming techniques and coding styles. Thus, the four programs we have used to produce performance estimates of our algorithms possess the characteristics of programs that a user of our algorithms are likely to meet.

Programs	Routines	Lines of Code	Description
ONEDIM	12	659	Calculates the eigenfunctions and eigenenergies of the time independent Schroedinger equation for a one dimensional potential.
LINPACKD	10	796	Performs various linear algebra procedures and collects timing.
SPHOT	5	1200	Uses Monte-Carlo method to solve photon transport problem in a spherical geometry.
SIMPLE	7	1312	A 2-D Lagrangian code with heat diffusion.

Table 5.1 Programs from RiCEPS

5.1 Measurements

In order to determine which program characteristics are critical to the success of our algorithms, we turn to the time complexities shown in Table 5.2. Inspection of Table 5.2 reveals that the time required for updating dependence information in response to a program edit is particularly sensitive to a few characteristics of the program being edited, its control and data dependence graphs, and the particular edit. These characteristics are $\|\eta_s\|$, $\|\eta_t\|$, and $\|\eta\|$. The other factors appear only linearly and have natural limits to their sizes, such as the nesting level of a program or the size of the right hand side of an assignment statement. The only limit to $\|\eta_s\|$, $\|\eta_t\|$, and $\|\eta\|$ is the length of the program.

5.1.1 $\|\eta_s\|$ and $\|\eta_t\|$

$\|\eta_s\|$ and $\|\eta_t\|$ are values derived from the relative position in the control dependence graph of the source and sink of an edge added to the control flow graph of a program. One way to attempt to estimate the average values of $\|\eta_s\|$ and $\|\eta_t\|$ would be to measure $\|\eta_s\|$ and $\|\eta_t\|$ for each pair of nodes in the program. This would produce a measure of the length of paths in the control dependence graph. But this calculation

In the following, let

- $\|\eta_s\|$ = the number of nodes that reach s but not t in the CDG(G)
- $\|\eta_t\|$ = the number of nodes that reach t but not s in the CDG(G)
- d_i = the maximum in-degree of any node in η_s and η_t
- d_o = the maximum out-degree of any node in η_s and η_t
- d_i^d = the maximum in-degree of any node in the data dependence graph
- η = the set of nodes reached by an array definition
- k = the loop level of a use or definition

CFAdd	$\ \eta_t\ \cdot d_i \cdot d_o + \ \eta_s\ (\ \eta_s\ \cdot d_o + d_i \cdot d_o)$
CFDel	$\eta_s \cdot d_o$
AddUse	$k \cdot \text{may}(s) + \text{loopmust} + \text{loopmay} $
DelUse	$ d_i^d $
AddDef	$\sum_k \eta_k + (\ \text{loopmust}_k\ + \ \text{loopmay}_k\ \cdot \ \text{uses}_k\)$
DelDef	$\sum_k \eta_k + (\ \text{loopmust}_k\ + \ \text{loopmay}_k\ \cdot \ \text{uses}_k\)$
AddCF	$\sum_{v \in V} 2(\sum_k \eta_k + (\ \text{loopmust}_k\ + \ \text{loopmay}_k\ \cdot \ \text{uses}_k\))$
DelCF	$\sum_{v \in V} 2(\sum_k \eta_k + (\ \text{loopmust}_k\ + \ \text{loopmay}_k\ \cdot \ \text{uses}_k\))$

Table 5.2 Complexities of the Dependence Update Algorithms

would be correct only if it were equally likely for a control flow graph edge to be added between any pair of nodes in the control flow graph regardless of their relative positions.

The complete program provides a record of which edges were added to the control flow graph. In order to calculate more accurate values for η_s and η_t we derived from a finished program a series of edits that could be used to produce the program and calculated $\|\eta_s\|$ and $\|\eta_t\|$ for each of the edits.

We assumed that the series of edits would *directly* produce the program, that is, there would be no addition of edges which were deleted afterward. The number of edits required to produce the program were counted, and η_s and η_t for each of the edits were recorded.

In order to understand how η_s and η_t were measured, consider the control flow fragment from the eighth subroutine of SIMPLE shown in Figure 5.1. The series of edits that we assume were used to create this fragment begins with an edge from node 3 to itself. Nodes 4, 6, 8, and 14 were then added to this edge. We recorded the sizes of η_s and η_t associated with the addition of this edge as 0 and 0, respectively. Any edge added from a node to itself will produce η_s and η_t equal to the empty set. Another simple case in the sequence of edits used to create this fragment is the addition of the edge from node 25 to node 36. The sizes of η_s and η_t associated with this edge are 1 and 1, respectively. This is an example of an edge whose sink postdominates the source of the edge. In this case η_s and η_t will contain only s and t , respectively. Edges whose source and sink are the same node and edges whose sink postdominates the source of the edge are easy to identify from the control flow graph, and the sizes of their η_s and η_t can be immediately recorded. These edges account for ninety percent of the edge additions counted in our measurements.

A more complicated edge to examine is the edge added from node 8 to node 36. In the control dependence graph previous to the addition of this edge, node 8 is dependent on 6 which is dependent on 4 which reaches the sink of the edge, 36. Thus

η_s contains nodes 6 and 4 which results in a $\|\eta_s\|$ of 2. η_t contains only node 4 and hence $\|\eta_t\|$ is equal to 1.

This process was carried out by hand for all the subroutines in the programs we measured. This produced the distribution of values for η_s and η_t shown in Table 5.3. The distribution of values shows that only for rare cases will η_s or η_t be greater than two or three. In fact, for over ninety percent of the edits either adding or deleting a control flow edge, η_s and η_t will be equal to one or zero.

5.1.2 $\|\eta\|$

η_k is the set of statements at level k whose *may* and *must* sets must be modified in response to the program edit. η is the sum of η_k for all k loops surrounding the location of the edit. If the program edit is the addition of an array definition then $\|\eta\|$ is equal to the number of statements that the array definition reaches.

$\|\eta\|$ is a function of the added definition and the form of the program at the time that definition was added. Since so many of the statements that an array definition reaches follow the definition textually in the program, if we were to assume that the program was written from the beginning to the end, we would produce an average $\|\eta\|$ that was very low. So we chose instead to measure η for every array definition in the program as if the definition were the last to be added. This measurement exaggerates the size of η .

Program	$\ \eta_s\ $					$\ \eta_t\ $			
	0	1	2-3	4-7	> 8	0	1	2-3	> 4
ONEDIM	40	10	0	0	0	40	10	0	0
LINPACKD	18	10	0	0	0	18	10	0	0
SPHOT	14	85	6	10	2	14	101	1	1
SIMPLE	76	24	3	1	0	76	25	3	0
Total	148	105	6	10	2	148	146	4	1

Table 5.3 Distribution of values for η_s and η_t

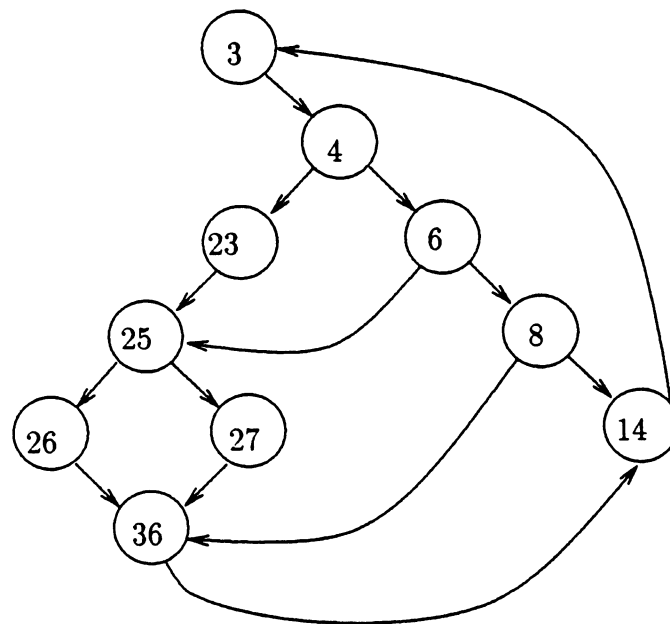


Figure 5.1 CFG fragment from SIMPLE

The count proceeded as follows. Flow sensitive interprocedural summary information of the type discussed in Chapter 3 was assumed. The number of statements a definition reached was indexed by the level of the reached statement. If a definition occurred at level l , then each statement at that level and within the same loop were counted as one. Statements enclosed in the same l level loop as the definition but nested at a deeper level than l were not counted; the loop at level l was counted. Statements at a level less than l were counted as one but at the higher level. For instance, consider the following code fragment from ONEDIM.


```

DO 500 I = 2, N, 1
    L = I - 1
    Z(N,L) = Z(L,L)
    H = D(I)
    IF (H .EQ. 0.0D0) GOTO 380
    DO 330 K = 1, L, 1
330      D(K) = Z(K,I) / H
    DO 380 J = 1, L, 1
        G = 0.0D0
        DO 340 K = 1, L, 1
340      G = G + Z(K, J) - G*D(K)
380 CONTINUE
500 CONTINUE
DO 510 I = 1, N, 1
510 Z(N,I) = 0.0D

```

The definition of Z at level one reaches an assignment and a GOTO statement inside its own loop at that level. In addition it reaches two loops at this level. Thus, η_1 in this code fragment is equal to four. At level 0 the definition reaches the loop which has 510, following its own. Since this loop kills the definition of Z , it can reach no further. Thus η_0 is equal to 1. The sum of these is three. This is the value recorded in Table 5.4. The distribution of $\|\eta\|$ is shown for all the definitions in the programs studied.

5.2 Implications and Further Arguments

When examining these results we must remember that the incremental methods are designed as alternatives to batch algorithms to perform the same task.

The sizes of η_s and η_t from Table 5.3 indicate that almost all updates to the control dependence graph can be done much more quickly using our incremental technique

Program	0	1	2-3	4-7	8-15	16-31	32-63	64-127	128-255
ONEDIM	7	16	25	35	18	5	0	0	0
LINPACKD	7	6	7	10	37	5	0	1	0
SPHOT	2	1	0	0	0	24	25	0	0
SIMPLE	0	2	8	13	25	28	28	57	46
Total	16	25	40	58	80	62	53	58	46

Table 5.4 Distribution of values for η

than a batch algorithm. Thus, the empirical results indicate the practicality of those techniques.

The sizes of η indicated in Table 5.4 indicate that in some cases the addition of a definition must be propagated widely around the program. This makes an incremental method less appealing. But most of the time an edit involves the definition of only a single array variable. Except where a use or definition of the variable is encountered and subscript testing is required or a meet operation must be performed at the end of alternate branches of control, the propagation of the changes to the *may* and *must* sets of the variable requires only the copying of pointers. This operation can be performed within a small number of machine instructions. Subscript testing, on the other hand, requires a much greater amount of work. In the incremental algorithm the only subscript testing that occurs involves the variable involved in the added or deleted definition. In contrast, the batch version requires subscript testing over the entire program for *all* the array variables appearing in the program. Thus, the incremental version can still be significantly faster in practice.

For an edit involving the addition or deletion of a control flow edge, this argument is less compelling. But another argument may be made. Suppose that few array kills appear in the program. Then when a control flow edge is added it will be likely that all the array definitions from the source of the new edge will already appear in the *may* sets at the sink of the new edge. Thus it will be unnecessary to propagate any changed information about this variable and the algorithm terminates quickly.

On the other hand, suppose many array kills appear in the program. In this case, while changes must be propagated, they must only be propagated until a kill to the changed information is encountered. Since a kill is likely to be encountered, the algorithm again terminates quickly. Thus, even the addition or deletion of a control flow edge could benefit from the use of incremental techniques. Further research is required to accurately assess the value of the incremental techniques for updating data dependences in response to control flow changes.

Chapter 6

Related Work

6.1 Dependence Analysis

6.1.1 Parallelism Detection

Work on automatic parallelization techniques began in the 1960's. Researchers during this time explored techniques for parallelization at granularities both below and above the statement level. During this period, relatively less work was done exploring parallelism between loop iterations. In 1973, Baer published a survey [Bae73] where he summarized the work that was done at all levels. In this survey he discussed the work done by Bernstein, Russell, Muraoka, and others in calculating data dependence and automatically using this information to look for parallelism between statements. The work by some of the same researchers in detecting parallelism between iterations of DO-loops is discussed as well.

Lamport's paper [Lam74] was one of the first successful treatments of the automatic transformation of sequential DO-loops to parallel form. This paper contains one of the first mentions of an iteration space. He identified two forms of parallel calculations on arrays within do-loops. The Coordinate Method involves splitting or distributing the individual statements in a loop and then performing the calculation on each member of the array independently. This kind of parallel calculation is more commonly known as SIMD parallelism or vectorization. The other method, which he called the Hyperplane method, is applicable to multiprocessors and is identical to what is known as the wavefront method discussed by others in [Kuh80, Wol82, Mur71].

Included in this paper is a very simple test on subscript expressions for identifying when transformations from sequential to one of the parallel forms are safe.

Towle explored the analysis of control and data dependence and its use in parallelizing transformations in his Phd thesis [Tow76]. Kuck et. al. [Kuc78, KKP+81] discussed the value of dependence analysis in optimizing programs for parallel architectures and characterized dependences as anti, output, flow, and input.

Covering was described by Kuhn in his Phd thesis [Kuh80]. Kuhn's methods calculated a dependence relation that resembled closely our concept of strong dependence. He described array accesses by convex sets and performed a complex calculation on these sets to arrive at his relation. His methods were tested in Paraphrase but were eventually abandoned in favor of Banerjee's tests. Banerjee [Ban79] first published a range test for dependence analysis in the presence of subscripted variables when the subscript expressions are affine.

Wolfe [Wol82] developed a scheme to annotate dependence information with direction vectors and then showed how to use the direction vectors to determine the correctness of various optimizations. Allen [All83] characterized dependences as carried or loop independent. This distinction is the key to optimizing for concurrent architectures, since it is the loop carried dependences that inhibit parallelization for these machines.

In 1983, Ferrante, Ottenstein, and Warren presented the Program Dependence Graph [FO83, FOW84], which they claim to be a dependence representation sufficient to represent the program itself. In their control dependence graph they use region nodes to consolidate dependence on identical conditions. They present a calculation of control dependence based on post dominator information. This control dependence relation is minimal in the sense that a statement is control dependent only on those statements whose execution and result can imply the execution of the first statement. This minimality corresponds closely to that of the strong dependence relation defined in chapter 3.

Brandes [Bra88] discusses the desirability of what he calls *direct dependences* in automatic parallelization. A direct dependence is a dependence in a weak dependence graph that is not equal to the composition of any combination of other direct dependences. That is, it is not equal to any transitive edge. His method for calculating these dependences is to prune transitive edges after he has calculated the entire weak dependence graph. He has made the observation that this pruned dependence graph is more useful for a user to see than the full transitive dependence graph calculated by traditional methods. This notion of direct dependence in some ways approximates our definition of strong dependence. The distinction is that strong dependences represent all possible flows of value when more than one execution path is available, where the direct dependences do not. We think strong dependence is the more natural and intuitive notion.

6.1.2 Intermediate Analysis

The intermediate analysis required to effectively apply independence tests in PFC has been discussed by Allen and Kennedy in [AK84]. The analysis required to identify loop invariant expressions was first developed to support optimizations intended to move code out of loops so that it would be executed less frequently [Coc70]. The results of this loop invariant analysis are used in their analysis of loop induction variables in a manner similar to that discussed in Chapter 4 [AK87]. In PFC, the induction variables are not merely identified but actually substituted into the code being analyzed. For this reason they call their procedure Induction Variable Substitution.

6.2 Programming Environments

At Rice, the Rn programming environment is geared toward numerical applications. One of its primary goals is exploring the use of extensive interprocedural information in the compilation of scientific programs. The heart of this system is a database

that keeps information about how programs are composed of individual procedures and interprocedural information for each procedure. The environment includes a procedure editor that calculates intraprocedural information [CKT85]. Allen and Kennedy [AK85] proposed that Rn evolve into a parallel programming environment called ParaScope. The methods and techniques presented in this dissertation are directly aimed at fulfilling the requirements for the planned parallel programming environment.

FORGE is a parallel programming environment that allows the user to direct the editor to perform parallelizing transformations on his behalf. Between the transformations, dependence information is calculated incrementally. These incremental updating techniques are limited to a small set of well-defined program changes and do not apply to arbitrary changes made by a user.

SUPERB is an interactive parallelizing tool built to support programming on the SUPRENUM project [KBGZ88]. Its goals are similar to those of ParaScope with the exception that arbitrary editing changes are not allowed. SUPERB makes use of incremental techniques to update the dataflow information in response to user directed transformations of his program.

6.3 Incremental Techniques

Incremental compilation was first investigated as a method for making better use of scarce computational resources available for compilation. The goal was to avoid repeating an entire compilation in response to relatively small program changes. Schemes using the statement as the basic unit of change were developed. Nonetheless, typically programmers were still limited to recompiling an entire procedure at a time [Loc65]. As computing resources became less scarce, these techniques fell out of favor, and work on incremental analysis and compilation ceased.

When interactive programming environments came into vogue, incremental techniques regained attention. However, their requirements were somewhat changed. In

particular, the new structure or syntax-directed editors provided a use for techniques that could limit recompilation to a finer granularity, such as the statement or sub-statement level.

Ryder's original technique [Ryd83] for incremental data flow analysis is one of the first in this new body of work. It is limited in that it does not allow for changes in control flow. The entire data flow problem must be re-solved in response to any change in control flow. One of Ryder's contributions was the recognition that worst case complexity analysis of incremental algorithms in terms of the size or number of changes is misleading. Ryder and Carroll [RC86] have more recently presented an incremental method of interval analysis that does not suffer from the earlier limitation of being unable to deal with control flow changes.

Zadeck [Zad84] developed another method of data flow analysis, called the Partitioned Variable Technique (PVT), that is both more general than Ryder's original work and has the fastest algorithmic complexity of any known solution for partitionable data-flow problems. While it is limited in the type of data flow problem that it can calculate directly, it is quite capable of handling the calculation of def-use chains for scalar variables. We have proposed to use Zadeck's techniques for the calculation of data dependence on scalars.

Reps [Rep82] presented an optimal time algorithm for updating attribute trees in a syntax-directed editor. Reps showed that his algorithm was optimal by proving that, for any change to his input, within a constant factor his algorithm only looked at parts of the graph that changed in response to the change in input. Since these changed parts of the graph require change by any algorithm updating the attribute tree, within a constant factor, he did only as much work as any correct algorithm.

Pollock has more recently developed techniques to be used for incremental optimization of intermediate code [Pol86]. She addresses not only the problem of performing the analysis required for optimization of programs but also the problem of managing an optimized program in an incremental programming environment.

Chapter 7

Conclusions and Future Work

Computer science is, essentially, an engineering discipline. Its goal is to provide tools useful for calculating the solution to some problem. Hence, the validation of the importance of any work in computer science must necessarily lie in how the results of that work enable the solution to some problem important to society outside of computer science.

If basic science is the collection of facts about the world in the absence of any clear idea of how these facts can be used, then basic engineering is the design of tools without any clear idea what they will be used to build. It is the paradox of basic research that its importance can never be confirmed at the time it occurs.

We are fortunate in computer science that for the foreseeable future there exist problems requiring ever faster computers and these faster computers and the harder problems on which they are applied will require more effective programming tools to program them. This thesis has developed a part of what may become a tool for programming these faster, parallel, computers.

In chapter 2 we presented a definition of control dependence that, while equivalent to the most common one presently in use[FOW87], is more easily used and thought about. We developed and presented a form of the control dependence graph which contains not only the control dependence relation but the postdominator relation as well. This allowed us to develop algorithms for updating the control dependence relation without reference to any data structures other than the control dependence graph and the control flow graph of the program.

In chapter 3 we discussed the notion of data dependence as it is calculated and applied today. We saw the difficulties presented by this form of the data dependence relation and defined the strong data dependence relation to make clear the distinction between the two notions of the relation. We showed how an approximation for strong dependence can be calculated efficiently in a batch algorithm and then presented algorithms to update the relation incrementally.

In chapter 4 we addressed some of the problems in efficiently performing the subscript testing necessary for the algorithms in chapter 3. These required that information about the equivalence of scalar values appearing in subscripts be calculated. After presenting an incremental method for integer expression folding we developed novel methods of identifying loop invariant variables and auxiliary induction variables that were able to take advantage of the work done by integer expression folding so that only one phase is required to perform all three of these scalar optimizations.

In chapter 5 we looked at the time complexities of the algorithms presented in chapters 2 and 3 and determined which characteristics of an update were most likely to adversely affect the time required for updating the dependence information in response to a typical edit. These characteristics were measured on real FORTRAN programs. We found that the time required for control dependence update will be nearly constant for most edits. The time for updating data dependence will be greater than this but should still be only a fraction of the time required for calculating the relation in batch.

Further work requires the construction of an interactive programming environment implementing these algorithms. Thus, the ease of implementing the algorithms can be tested during the design and construction of the environment and their performance in updating can be measured during actual use.

Any piece of basic research in engineering must wait to be validated by the presence of its results in a later piece of work. We present this work in the hope that it will find a place in the researches and implementations that follow.

Bibliography

- [ABKP86] J. R. Allen, D. Bäumgartner, K. Kennedy, and A. Porterfield. PTOOL: A semi-automatic parallel programming assistant. In *Proceedings of the 1986 International Conference on Parallel Processing*. IEEE Computer Society Press, August 1986.
- [AK84] Randy Allen and Ken Kennedy. Automatic translation of fortran programs to vector form. Technical Report Rice COMP TR84-9, Rice University, July 1984.
- [AK85] Randy Allen and Ken Kennedy. Programming environments for supercomputers. Technical Report Rice COMP TR85-18, Rice University, March 1985.
- [AK87] J. R. Allen and K. Kennedy. Automatic translation of FORTRAN programs to vector form. *ACM Transactions on Programming Languages and Systems*, 9(4):491-542, October 1987.
- [AKPW83] J.R. Allen, Ken Kennedy, Carrie Porterfield, and Joe Warren. Conversion of control dependence to data dependence. In *Conference Record of the Tenth ACM Symposium on Principles of Programming Languages*, January 1983.
- [All83] J. R. Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations*. PhD thesis, Rice University, April 1983.
- [AU77] A. Aho and J. Ullman. *Principles of Compiler Designing*. Addison-Wesley, 1977.
- [Bae73] J.L. Baer. A survey of some theoretical aspects of multiprocessing. *ACM Computing Surveys*, 5(1):31-80, March 1973.
- [Bal89] Vasanth Balasundaram. *Interactive Parallelization of Numerical Scientific Programs*. PhD thesis, Rice University, April 1989.
- [Ban79] U. Banerjee. *Speedup of Ordinary Programs*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1979.

- [Bra88] Thomas Brandes. The importance of direct dependences for automatic parallelization. In *Proceedings of the International Conference on Supercomputing*, 1988.
- [Cal87] David Callahan. *A Global Approach to Dependence Analysis*. PhD thesis, Rice University, 1987.
- [CCH+87] A. Carle, K. Cooper, R. T. Hood, K. Kennedy, L. Torczon, and S.K. Warren. A practical environment for Fortran programming. *Computer*, October 1987.
- [CCKT86] David Callahan, Keith D. Cooper, Ken Kennedy, and Linda Torczon. Interprocedural constant propagation. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 152-161, June 1986.
- [CK87] D. Callahan and K. Kennedy. Analysis of interprocedural side effects in a parallel programming environment. In *Proceedings of the First International Conference on Supercomputing*. Springer-Verlag, Athens, Greece, 1987. Available as Rice University, Department of Computer Science Technical Report TR87-56, July 1987, To appear: *Journal of Parallel and Distributed Computing*.
- [CKT85] Keith D. Cooper, Ken Kennedy, and Linda Torczon. The impact of interprocedural analysis and optimization in the rn programming environment. Technical Report Rice COMP TR85-27, Rice University, December 1985.
- [Coc70] John Cocke. Global common subexpression elimination. *SIGPLAN Notices*, pages 20-24, July 1970.
- [Coh73] William L. Cohagen. Vector optimization for the asc. In *Proceedings of the Seventh Annual Princeton Conference on Information Sciences and Systems*, 1973.
- [Coo83] Keith Cooper. *Interprocedural Data Flow Analysis in a Programming Environment*. PhD thesis, Rice University, 1983.
- [FO83] Jeanne Ferrante and Karl Ottenstein. A program form based on data dependency in predicate regions. In *Conference Record of the Tenth ACM Symposium on the Principles of Programming Languages*, January 1983.
- [FOW84] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. Technical Report RC 10543, IBMTJW, May 1984.

- [FOW87] J. Ferrante, K. J. Ottenstein, and J. D. Warren. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems*, 9(3):319–349, July 1987.
- [KBGZ88] Ulrich Kremer, Heinz-J. Bast, Michael Gerndt, and Hans P. Zima. Advance tools and techniques for automatic parallelization. In *Proceedings of the Second International Symposium Colloquium*, 1988.
- [KKP⁺81] D.J. Kuck, R.H. Kuhn, D.A. Padua, B. Leasure, and M. Wolfe. Dependence graphs and compiler optimizations. In *Conference Record of the Eighth ACM Symposium on the Principles of Programming Languages*, January 1981.
- [Kuc78] David J. Kuck. *Computers and Computations*, volume 1. John Wiley and Sons, 1978.
- [Kuh80] Robert Henry Kuhn. *Optimization and Interconnection Complexity For: Parallel Processors, Single-Stage Networks, and Decision Trees*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, February 1980.
- [Lam74] Leslie Lamport. The parallel execution of do loops. *CACM*, 17(2):83–93, February 1974.
- [Loc65] Kenneth Lock. Structuring programs for multiprogram time-sharing on-line applications. In *Proceedings AFIPS Fall Joint Computer Conference*, pages 457–475, 1965.
- [Mur71] Y. Muraoka. *Parallelism Exposure and Exploitation in Programs*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, February 1971.
- [Pol86] Lori L. Pollock. *An Approach to Incremental Compilation of Optimized Code*. PhD thesis, University of Pittsburgh, 1986.
- [RC86] Barbara Ryder and Martin D. Carroll. An incremental algorithm for software analysis. In *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, 1986.
- [Rep82] Thomas Reps. Optimal-time incremental semantic analysis for syntax-directed editors. In *Conf. Record of Ninth ACM Symposium on Principles of Programming Languages*, 1982.

- [Ryd83] Barbara Ryder. Incremental data flow analysis. In *Conference Record of the Tenth ACM Symposium on the Principles of Programming Languages*, 1983.
- [Tow76] Ross Albert Towle. *Control and Data Dependence for Program Transformations*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, March 1976.
- [Wol82] M. J. Wolfe. *Optimizing Supercompilers for Supercomputers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, October 1982.
- [Zad84] K. Zadeck. Incremental data flow analysis in a structured program editor. In *Proceedings of the SIGPLAN 84 Symposium on Compiler Construction*, June 1984.