

AN OPERAND DEPENDENCE GRAPH METHOD
FOR CODE OPTIMIZATION

By

JONATHAN M. ASURU

Bachelor of Science
University of Lagos
Lagos
1979

Master of Science
Oklahoma State University
Stillwater, Oklahoma
1985

Submitted to the Faculty of the
Graduate College of the
Oklahoma State University
in partial fulfillment of
the requirements for
the Degree of
DOCTOR OF PHILOSOPHY
May, 1990

C O P Y R I G H T

by


Jonathan Maduabughci Asuru

May, 1990

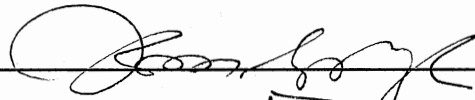
1375558

AN OPERAND DEPENDENCE GRAPH METHOD
FOR CODE OPTIMIZATION

Thesis Approved:

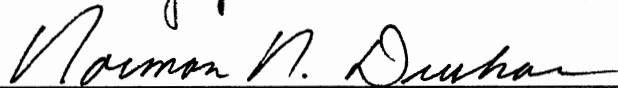


Thesis Adviser



M. Samadzadeh-H.

Joyce S. Triske



Dean of the Graduate College

PREFACE

Real programs tend to have the following features:

(1) code redundancies are concentrated in basic blocks and in loops; (2) redundant statements involve mostly lexically identical expressions; and (3) each distinct program statement is used in few sections of a program. These characteristics imply that high quality object code can be produced by applying code improvement procedures to small segments of a program rather than to an entire program.

This study addresses a well known problem in computer science - the optimization of a compiler generated intermediate code to produce an equivalent code with less redundant statements. There are two aspects to this work: (1) the development of a region relative code improvement technique for programs with structured control flow graph; and (2) the unification of common subexpression, code hoisting, and code sinking optimization problems. The purpose of the study is to develop a one-pass intermediate code optimization method with the capability to recognize both local and global redundancies in a program region.

The methods developed for redundant statement detection are (1) representation of statements with an operand dependence graph, (2) modelling of variable reaching

definitions with operand version numbers, and (3) extension of the notion of partial redundancy to include statements on disjoint control flow paths. Algorithms for many code optimization procedures and their worst case time complexity bounds are presented.

I wish to express my sincere gratitude to individuals who assisted me directly to bring this endeavor to fruition. My special thanks goes to Dr. George E. Hedrick, my major adviser for his direction and encouragement. I will like to thank my other committee members, Dr. J. Friske, Dr. K. M. George, and Dr. M. E. Zamadzadeh for their helpful suggestions and comments on the draft of the thesis. I will also express my appreciation to Margaret Brown for typing part of this manuscript. I am very grateful to my wife, Agatha and to my daughter, Awuri for their support, patience, and encouragement.

TABLE OF CONTENTS

Chapter	Page
I. INTRODUCTION	1
The Problem	1
Literature Review	4
Operand Dependence Graph Based Method	11
II. INTERMEDIATE CODE FORM	15
Introduction.	15
Operand Rank and Operand Order.	15
Structure of a DST Entry.	16
Array Indexing Representation	18
Structured Variable Transformation.	21
Summary	22
III. CONTROL STRUCTURE ANALYSIS	23
Introduction.	23
Structured Program Flow Graph	23
Ordering of Nodes	27
Path Covers	33
Path Analysis and Code Optimization	45
Path Analysis Information Representation.	47
Summary	52
IV. VARIABLE DEFINITION ANALYSIS	53
Introduction.	53
Operand Version Numbering	54
Properties of Operand Version Numbers	59
Summary	66
V. INTRAPROCEDURAL ALIAS ANALYSIS	67
Introduction.	67
Pointer Aliasing in C	68
Pointer to Structure Transformation	72
Alias Analysis Procedure.	74
Summary	80

Chapter	Page
VI. OPERAND DEPENDENCE GRAPH	81
Introduction.	81
What is an Operand Dependence Graph	81
Work Lists.	83
Essential Node Information.	84
Operand Dependence Graph Construction	86
Summary	95
VII. DETECTION OF FEASIBLE OPTIMIZATIONS.	96
Introduction.	96
Redundant Statement Elimination	96
Code Hoisting Optimization.	102
Code Sinking.	111
Constant Folding.	115
Loop Optimization	122
Complexity of Code Optimization	133
Summary	140
VIII. SIMPLE RECURRENCE LOOP OPTIMIZATION.	141
Simple Recurrence Array Reference	141
Simple Recurrence Elimination	148
Simple Recurrence Analysis.	149
Loop Unrolling and Simple Recurrence.	153
Summary	155
IX. SUMMARY, CONCLUSION, AND RECOMENDATIONS.	156
Summary	156
Useless Code Elimination.	157
Improvements.	158
Further Studies	159
REFERENCES	161

LIST OF TABLES

Table	Page
I. Fork-width and Join-width of the Nodes in a Program Flow Graph.	41

LIST OF FIGURES

Figure	Page
1. Distinct Statement Table Representation of a Program Fragment.	19
2. Single Exit Structured Flow Graph	25
3. Predominance Tree Post-dominance Tree	27
4. While-do Loop to Do-while Loop Conversion	31
5. Conditional Control Environment Cover Checking Algorithm	42
6. Node Path Cover Analysis Algorithm.	43
7. Flow Graph and Forward Reachability Sets.	51
8. Flow Graph Showing Definitions and References of a Variable.	60
9. Variable Definitions and Reference Analysis Data Structure	61
10. Alias Processing in a Basic Block	70
11. Node Listing Generator Algorithm.	75
12. Alias Computation Algorithm	76
13. C Program	79
14. C Program After Structure Member and Pointer to Structure Transformations	79
15. Contents of ALIAS-OUT and IND-ASSIGN After Alias Analysis.	80
16. Partial Redundancy Involving Disjoint Statement Instances	90
17. Partial Redundancy Involving Statements on Common Execution Paths	91

Figure	Page
18. A Redundant Statement	99
19. A Partially Redundant Statement not Tested for Redundancy.	100
20. A Partially Redundant Expression Which Fails Full Redundancy Test	101
21. An Inter-region Redundancy.	102
22. Hoistable Code.	104
23. A Non-Hoistable Code.	106
24. Code Hoisting Algorithm	107
25. Code Hoisting Candidate in a Nested Conditional Structure	109
26. Code Sinking Example.	113
27. Simple and Non-Simple Constant Expression	116
28. Constant Folding Example.	119
29. Constant Folding Procedure.	120
30. Evaluation Points Determination Procedure	121
31. Loop Invariant Detection Procedure.	124
32. Template for Defining Fields of a Sequence Tree Node.	128
33. ODG Segment Showing Loop Induction Variables.	129
34. Sequence Tree of Induction Variable Family.	130

CHAPTER I

INTRODUCTION

The Problem

Compilers for high-level languages employ general schemes when translating a source program into machine code. The code produced is usually inefficient with respect to both code size and to running time when compared to an input/output equivalent hand-written assembly program. Algorithms for improving the quality of compiler generated code have been developed [2, 6, 12, 27, 36]. However, the algorithms are applied separately causing these code improvement algorithms to interact creating a phase ordering problem. In order to generate very efficient code, a compiler may have to apply these algorithms several times.

The usual approach to compiler intermediate code optimization consists of two separate steps. In the first step, a compiler removes the inefficiencies in each basic block by detecting and eliminating common subexpressions, evaluating expressions with constant operands, and by deleting useless instructions. These block specific optimizations constitute the local code optimization step.

In the second step, a compiler increases the window of instructions examined for improvement by combining statements from many basic blocks. The program optimization techniques which combine many basic blocks are called global program optimization algorithms. To perform this step of the optimization, a compiler requires global information about both definitions and uses of program variables. Global information gathering is called global data flow analysis. There is no single data flow analysis technique that can capture all the information that an optimizing compiler uses for global code improvement. As a result, a compiler performs separate flow analysis for each global code optimization problem.

For instance, in order to eliminate globally redundant expressions among basic blocks, a compiler determines the expressions available at the entry and exit points of each block. If an expression computed in a block, B, is found in the pool of available expressions, then a compiler can delete the expression from block B. The search for available expressions takes time; there is no mechanism to avoid useless searches. Repeated scanning of the intermediate code increases the running time of optimizing compilers.

A characteristic of code improvement methods based on information propagation is that different techniques are used to detect redundant statements. For instance, value numbers [12] and directed acyclic graphs [2] are employed

to implement local code optimization algorithms, while bit vectors and equivalence relations are used for global code optimization problems. The use of different implementation techniques to model a code optimization procedure (feasible at both the local and global levels) increases both the size and complexity of optimizers. In order to reduce the size of optimizing compilers, only a small number of the well known code improvement transformations are applied in many compilers.

This study develops an intermediate code optimization method for structured program flow graphs using a directed graph representation of program statements. A structured program flow graph is a program flow graph with the following properties.

1. there is no jump into the middle of a conditional structure;
2. there is no jump into the middle of a loop;
3. every loop has a unique point outside that loop to which control transfers upon loop termination;
4. there is no overlap of control structures;
5. every conditional structure has a common join (the next statement executed after control leaves a conditional structure); and
6. every backward jump to the beginning of a loop is contained in that loop.

The goals of this study include:

1. to develop a uniform method for characterizing

redundancy and a uniform mechanism for specifying the data flow and control flow constraints for the various optimization problems;

2. to develop an optimization procedure which can detect and eliminate a number of local and global redundancies in a single pass; and
3. to keep the cost of optimization proportional to the actual number of potential redundancies.

Literature Review

Basic Block Optimizations

Aho and Ullman [2] describe an elegant method for improving straight line sections of a program based on representing the instructions of a block with a directed acyclic graph (DAG). The optimizations performed with a DAG include common subexpression elimination, dead code elimination, scalar propagation, and constant folding [1]. The leaf nodes of a DAG represent initial values, while interior nodes of a DAG contain the operation symbols and identifiers for storing the results of operations. An advantage of the DAG method is that block specific information used in global data flow analysis problems are determined by traversing the DAGs of a block. However, the DAG method can improve individual basic block only.

Cocke and Schwartz [12] present the value number method for optimizing a basic block. Their algorithm associates

value numbers to expressions and variables used within a block, such that variables and expressions having the same value are assigned a common value number. The data structure for value numbering is a hash table of available expressions in a block.

Recursive Descent

Wulf, et al. [39] describe a method which integrates parsing with the detection of feasible optimizations in a program. Detection of redundancies is possible during parsing because the syntax of the source language, BLISS does not allow goto statements. As a result, control environments (basic blocks, conditional structures, and loops) are well defined. Feasible optimizations such as linear code motion (code hoisting and code sinking), common subexpressions, and loop invariants are identified and marked without eliminating the redundancies.

Two distinct approaches are used to recognize redundancies: (1) a congruent expressions table contains equivalence classes of lexically identical expressions which compute the same value; and (2) an ordering relation defined on basic block statements partitions each block statements into three subsets called prologue, epilogue, and postlogue. At the join of a conditional control structure, the mutual intersection of prologue sets of linear blocks in that conditional structure yields hoistable code and the mutual intersection of postlogue sets identifies sinkable code.

The notable feature of the recursive descent approach is that it demonstrates that most of the common code optimization procedures can be performed in one pass over an intermediate code without examining the entire program. The problem with recursive descent is that it cannot be applied to programming languages with the goto construction.

Data Flow Analysis

Data flow analysis is the most widely used technique for eliminating global redundancies in compiler generated code. Global program optimization by data flow analysis consists of two separate steps: analysis and optimization. In the analysis step, a system of data flow equations for the type of code optimization problem is solved to obtain information reaching the beginning and end of each flow graph node. The optimization step uses the information obtained from flow analysis to remove redundancies (if any) from each flow graph node (basic block).

For each global optimization problem, the intermediate text is scanned twice (once during analysis and once during optimization). Since the intermediate code is usually maintained in secondary storage, enormous time is spent on I/O. An optimization procedure may be applied several times in order to discover more redundancies.

Each data flow analysis procedure has $O(N^2)$ complexity, where N is the number of nodes in a program flow graph. Because the data flow analysis step is the dominant cost,

most research on data flow analysis based methods is focused on reducing the number of iterations. There are three approaches to reducing the number of iterations.

One iteration reduction scheme uses a data flow analysis procedure which converges to a fixed point in only a few iterations. Properties such as reducibility [19] and topological ordering [18] are used to determine the order in which information is propagated. Flow analysis algorithms based on interval analysis [5] and the iterative analysis of Hecht and Ullman [18] are representative methods.

Another approach to improving the efficiency of data flow analysis procedure is incremental flow analysis [34]. Incremental flow analysis avoids complete recalculation of data flow sets after each optimization procedure by isolating the region of a program affected by an optimization. The concept is attractive, but the process is complex. Incremental flow analysis is an on-going research and the procedure is not understood well enough to be included in compilers. Incremental flow analysis increases the complexity of an optimizer and also requires more storage for data flow analysis bit vectors.

The third iteration reduction approach is ordering of optimization procedures to avoid negative phase ordering problems. Phase ordering problems exist because the optimization procedures are not completely independent. One optimization procedure may create a redundancy eliminated by another optimization procedure. By suitable ordering of

optimization procedures, many redundant computations can be eliminated in a single application of a code optimization procedure.

A major problem with data flow analysis based methods is that the cost of redundancy detection is dependent on program length and number of nodes in a flow graph but not on the number of potential redundancies in a program. This makes data flow analysis technique unsuitable for optimization procedures where loop unrolling is performed, as both the number of statements and the number of basic blocks increase.

Another problem with data flow analysis is that it assumes every distinct statement is equally likely in each basic block. Hence, equal length data flow bit vectors are used in each basic block to represent data flow information. Basic blocks usually have few statements which means the data flow bit vector for a block is usually sparse.

A third problem is that two sets of optimization procedures are implemented with data flow analysis based methods, one set of algorithms remove local (intrapblock) redundancies and the other set of procedures eliminate global redundancies.

Global Value Numbers

Rosen, Wegman and Zadeck [33] develop a method which extends the value number method to eliminate global redundancies. The sequence of steps necessary to optimize a

program using their technique include

1. convert a program to static single assignment (SSA) form;
2. assign ranks to computations;
3. move computations backward and forward;
4. eliminate redundant computations in rank order;
5. apply question propagation to move computations out of a loop; and
6. reconvert program to original non-SSA form.

The global value number approach is an improvement over data flow analysis in that it applies the same mechanisms (movement of computations and question propagation) to detect global redundancies. However, the global value number method has the following drawbacks:

1. too many variables are created during SSA transformation;
2. uses too many tables (there is one hash table for each flow graph node and a moveable computation table for each edge of the directed acyclic graph of a flow graph) to hold intermediate code statements;
3. redundant statements with different ranks cannot be detected in the same optimization pass; and
4. the algorithm has a worst case time complexity $O(N^3)$, where N is the number of nodes in a flow graph.

Program Dependence Graph

Ferrante, Ottenstein, and Warren developed an intermediate program form called program dependence graph (PDG) for applying various intermediate code improvement procedures to a program [15]. There are four steps in the construction of a PDG:

1. construction of the DAG representation of each basic block;
2. computation of reaching definition information for each variable used in a basic block;
3. linking of each use of a variable with that variable's possible definition points. These definition-use edges constitute the data dependence edges; and
4. linking of each statement with the predicate(s) which control the execution of that statement.

After constructing a PDG, many compiler optimizations are carried out by a graph walk of the relevant sections of a program, but only one optimization procedure can be performed at a time.

One advantage of the PDG is that data flow information update is performed directly on the dependence graph after each optimization procedure. However, the PDG requires more space than data flow analysis methods and incurs considerable cost when searching for feasible optimizations in an intermediate code. Moreover, since the PDG is an

intermediate form, it cannot be easily integrated into compilers employing common intermediate forms.

Operand Dependence Graph Based Method

A code optimizer should have two important attributes: (1) the time complexity of redundancy elimination should be proportional to the number of potentially redundant statements in a program; and (2) the optimizer can perform several optimization procedures in one pass over an intermediate code. These two characteristics, if present in a code optimizer will improve the efficiency of a code optimizer. The second attribute reduces the number of passes over the intermediate code during code optimization phase of program compilation. To the best of the author's knowledge, the BLISS optimizing compiler [39] is the only optimizer that performs most local and global code optimizations in one optimization pass. However, the time complexity of redundancy detection is not proportional to the number of redundant statements.

The work presented in this dissertation develops code optimization technique in which both of these attributes are present. A number of approaches are developed and combined to produce the desired qualities at a moderate cost. These approaches include rigorous control structure analysis, use of operand version numbers, definition of a unifying concept of partial redundancy, and the representation of the intermediate code of a program with a factoring graph called

the operand dependence graph.

Control structure analysis involves the identification of loop and non-loop sections of a program, the computation of forward reachability, predominance, and post-dominance relations, and the assignment of path weights called fork width and join width to each node of a program flow graph. The predominance relation among flow graph nodes is used to define a topological ordering on flow graph nodes such that processing the nodes in topological order preserves the precedence constraint imposed by control flow. Control structure analysis is described in detail in chapter III.

Program operands (identifiers and constants) are assigned version numbers which distinguish instances of the same operand. A version number is a nonnegative integer assigned to an operand at a reference or at a definition point. The version numbers of operands are propagated through forward edges (non-looping arcs) of a flow graph in a manner similar to reaching definitions, but without setting up and solving a system of data flow equations. When the nodes of a flow graph are processed in topological order, the version number of a program operand is monotonically increasing along any forward control flow path. Version number related issues are discussed in chapter IV.

The search for redundant statements in an intermediate code is confined to optimization regions. An optimization region is either a loop with all the loops nested within

that loop or an acyclic structure preceding or following a loop. This restriction is adequate to detect most redundancies in a program since most optimizations are performed in basic blocks and in program loops. Each program region is represented with a directed graph -- the operand dependence graph.

The operand dependence graph structure exposes lexically identical expressions which should be analyzed for various forms of redundancy, such as common subexpression elimination, code hoisting, and code sinking. A feature of operand dependence graph which enhances the detection of potential redundancies is the fact that lexically identical expressions and operands from different basic blocks (whether equivalent in value or not) can be represented with the same graph node. In this way, the operand dependence graph factors out those statements which may be redundant in a program. The operand dependence graph facilitates the detection of loop invariant statements and the recognition of loop induction variables because the loop statements which may be loop optimization candidates are connected with graph edges. Chapter VI describes issues related to operand dependence graph representation.

The operand version numbering technique and the operand dependence graph representation technique are used to define a concept of partial redundancy which includes both statements on a common execution path and statements on disjoint execution paths. With this unifying concept of

partial redundancy, common subexpressions, code hoisting, and code sinking candidates are identified with the same mechanism, thus avoiding separate search procedures to identify candidate statements for each type of optimization problem. In chapter VII, individual code optimization procedures are discussed and chapter VIII describes a technique for removing simple recurrences in a loop.

The main contributions of this study are:

1. use of operand version numbers to model variable reaching definitions;
2. use of the path cover concept to check path constraints of various code optimization problems;
3. use of a uniform mechanism to detect common subexpressions, hoistable code, and sinkable code; and
4. a method for eliminating simple linear recurrence array references from sequentially executed loop.

CHAPTER II

INTERMEDIATE CODE FORM

Introduction

The operand dependence graph based code optimization method can be integrated into a compiler employing common intermediate forms such as quadruples, triples, statement trees, and/or directed acyclic graphs (DAGs). Because of the implementation strategy for an operand dependence graph, a program's intermediate code is represented as a sequence of distinct statement table (DST) locations. A DST is a hash table of distinct intermediate code statements in a program. The sequence of DST locations specify the intermediate program in sequential execution order.

Operand Rank and Operand Order

Without defining an order for operands of intermediate code operations, the number of distinct intermediate code statements in a program may increase when an order is not specified for operands of commutative operations. For instance "a + b" and "b + a" will be treated as distinct if distinctness is based solely on lexical patterns. The

approach used to avoid entering equivalent statements into the DST is to assign a rank to each operand (constant, temporary, or declared variable).

The rank of an operand is a unique number assigned to an operand by some ranking rule. A rank assignment rule adopted for the operand dependence graph is to assign a number which reflects the order in which tokens are entered into a symbol table. Operands of commutative intermediate code operations are rearranged so that operands are in increasing rank order. This operand ordering rule preserves the result of numerical computations. A source level expression such as $a + b + c$ will be translated into one of these equivalent computations $(a + b) + c$, $(b + a) + c$, $c + (a + b)$, or $c + (b + a)$.

Structure of a DST Entry

An intermediate code statement in the DST consists of four fields:

1. A generalized n-tuple which represents an abstract language operation.
2. An ordered set of dependent operands.
3. Temporary name generated for that statement if the statement is an expression.
4. Operand dependence graph node for the statement.

A layout of the fields of an intermediate code statement in a DST is shown below.

N-TUPLE		D-OPERAND		TN		ODG-NODE
---------	--	-----------	--	----	--	----------

N-TUPLE: Abstract language operation.

D-OPERAND: Dependent operands set.

TN: Temporary name.

ODG-NODE: Operand dependence graph node.

An n-tuple ($n \geq 0$) consists of an operator name and a list of the n operands on which a specified operator will be applied. The use of a generalized n-tuple makes it possible to represent operations with more than two operands (such as procedure calls) with a single statement.

The dependent operands field (D-OPERAND) contains the set of variables which affects that n-tuple. Let S be an n-tuple of the form θ, O_1, \dots, O_n , where θ is an operation symbol and O_1, \dots, O_n are the n operands for the operation θ . The algorithm for computing the elements of D-OPERAND follows.

D-OPERAND = \emptyset ;

If O_i is a source variable or a temporary, then

D-OPERAND = D-OPERAND \cup $\{O_i\}$

The C program fragment below serves as an example to illustrate the operand dependence graph based intermediate code form.

```
float a, b, c, x1, x2, temp;
...
if(b * b - 4 * a * c > 0) {
    temp = sqrt(b * b - 4 * a * c);
    x1 = (-b + temp) / (2 * a);
    x2 = (-b - temp) / (2 * a); }
```

A listing of the variables and constants in increasing rank order is a, b, c, x1, x2, temp, 4, 0, 2. The sequence of three address statements corresponding to the fragment is

```

s1:  t1 = b * b
s2:  t2 = 4 * a
s3:  t3 = t2 * c
s4:  t4 = t1 - t3
s5:  if t4 > 0 goto s7
s6:  goto s23
s7:  t1 = b * b
s8:  t2 = 4 * a
s9:  t3 = t2 * c
s10: t4 = t1 - t3
s11: t5 = sqrt t4
s12: temp := t5
s13: t6 = -b
s14: t7 = t6 + temp
s15: t8 = 2 * a
s16: t9 = t7 / t8
s17: x1 := t9
s18: t6 = -b
s19: t10 = t6 - temp
s20: t8 = 2 * a
s21: t11 = t10 / t8
s22: x2 := t11
s23: ...

```

The distinct statement table and the sequence of distinct statement table indexes corresponding to the intermediate code is depicted in Figure 1.

Array Indexing Representation

An array object is represented with two components: a base and an indexing vector. Array base is the constant component of the expression for calculating an element's address, while an indexing vector specifies the stride for each dimension of an array.

	N-TUPLE	D-OPERAND	TN	ODG-NODE
0.	* b, b	{b}	t1	
1.	* a, 4	{a}	t2	
2.	* c, t2	{c, t2}	t3	
3.	- t1, t3	{t1, t3}	t4	
4.	> t4, 0 goto s7	{t4}		
5.	goto s23	{}		
6.	sqrt t4	{t4}	t5	
7.	:= temp, t5	{t5}		
8.	- b	{b}	t6	
9.	+ temp, t6	{temp, t6}	t7	
10.	* a, 2	{a}	t8	
11.	/ t7, t8	{t7, t8}	t9	
12.	:= x1, t9	{t9}		
13.	- t6, temp	{t6, temp}	t10	
14.	/ t10, t8	{t10, t8}	t11	
15.	:= x2, t11	{t11}		

Sequence of intermediate code statements = 0, 1, 2, 3, 4, 5, 0, 1, 2, 3, 6, 7, 8, 9, 10, 11, 12, 8, 13, 10, 14, 15.

Figure 1. Distinct Statement Table Representation of a Program Fragment

If A is an n-dimensional array declared with dimensions d_1, \dots, d_n , stored in row major order and if $A[s_1] \dots [s_n]$ is an element specification, where s_1, \dots, s_n are subscript expressions, then the address of $A[s_1] \dots [s_n]$ is given by the expression

$$\text{addr}(A) - (L_1 * d_2 * \dots * d_n * w + \dots + L_n * w) \quad (1)$$

$$+ s_1 * d_2 * \dots * d_n * w + \dots + s_n * w \quad (2)$$

where $\text{addr}(A)$ is the address of the first byte of the storage area for elements of A; w is the amount of storage required to store one element of A; and L_1, \dots, L_n are the lower bounds on subscript values for the respective dimensions. The base component of A is given by expression

(1) and the indexing vector for A is the sequence of constant factors of terms in expression (2) and is defined as

$$(d_2 * \dots * d_n * w, d_3 * \dots * d_n * w, \dots, w). \quad (3)$$

Expression (2) for array element address calculation is a vector dot product operation. Thus, (2) can be rewritten as

$$(d_2 * \dots * d_n * w, \dots, w) \cdot (s_1, \dots, s_n) \quad (4)$$

To model array indexing operation by means of vector dot product, two intermediate code operators are introduced. The first operator called INDEX denotes the dot product operation and takes as operands an indexing vector and a subscript vector. The second operator CREATE-VECTOR converts a sequence of subscript expressions for an array element into a subscript vector.

Suppose A and B are arrays stored in row major order. A and B are indexing equivalent if A and B have the same indexing vector. In a formal sense, two arrays A and B are indexing equivalent if

1. A and B have the same number of dimensions;
2. Element size of A equals element size of B;
3. Only the size of their first dimensions may differ.

Indexing equivalent arrays reference a common indexing vector. A unique tag which serves as the name of an indexing vector is assigned to each distinct indexing vector in a procedure.

Application to Loop Optimization

In procedural languages, array dimensions once specified are invariant throughout the life time of a procedure invocation. Therefore, the indexing vector component of a dot product operation is invariant in a procedure.

The optimization of a program loop requires multiple passes over the body of a loop in order to obtain the information necessary for performing loop specific code improvements such as loop invariant motion and strength reduction of loop induction variables. Since most sources of loop code improvements are due to linearizing subscript expressions of multi-dimensional arrays, modelling array indexing operations with vector dot product exposes most loop invariants and induction variables without searching loop statements.

Structured Variable Transformation

To accommodate structured variables in an operand dependence graph, the fields of each structured variable are renamed with internal unique names generated by a compiler. A source level reference to a structure member in a statement is translated to reference the unique internal name for that field. An implementation strategy for mapping source level field names to internal names is a synonym table. Each structured variable has its own synonym table. By renaming the members of a structured variable, the scalar

components of a structure become amenable to data flow analysis.

Summary

The intermediate code form for an operand dependence graph based program representation consists of a distinct statement table for the distinct intermediate code statements in a program and a sequence of distinct statement table indexes which specifies the intermediate code program. Array indexing operation is represented as a dot product operation and a renaming transformation is applied to structured variables to simplify the handling of structure members.

CHAPTER III

CONTROL STRUCTURE ANALYSIS

Introduction

The optimization of compiler generated code depends upon the accurate knowledge of the control structure within a program. For a "gotoless" language such as Bliss [39], the control structure of a program can be deduced from the programming language syntax. When processing a program written in a language which permits "goto" statement, control flow analysis is necessary to identify the control structures of a program. In this section, control structure analysis issues relevant to an operand dependence graph based code optimization method are discussed.

Structured Program Flow Graph (SPFG)

The first step in control flow analysis is the construction of a program flow graph from the set of linear (basic) blocks of a program's intermediate code.

Definition 1. A program flow graph is a connected rooted directed graph, $G = (N, A, r)$, where N is a finite set of basic blocks (also called nodes), A is a subset of $N \times N$, and r is the initial basic block (the block where program

execution begins). A directed edge (n_1, n_2) connects two nodes n_1 and n_2 , if n_2 can be executed immediately following the execution of the last statement of n_1 .

Let n be a node of $G = (N, A, r)$, the immediate successors of n , denoted $\text{succ}[n]$, is defined as $\{ x \mid (n, x) \in A \}$. Similarly, $\{ y \mid (y, n) \in A \}$ form the immediate predecessor set of n , denoted $\text{pred}[n]$.

Definition 2. A reducible program flow graph is a program flow graph $G = (N, A, r)$, such that the backedges of G are unique[19].

Definition 3. A structured program flow graph is a reducible flow graph in which every loop has a unique loop exit.

Definition 4. A single exit program flow graph is a structured program flow graph $G = (N, A, r, e)$, where N, A , and r are as defined above, and e is a unique node such that there exists a path from every node to e .

Henceforth in this work, any reference to a flow graph implies a single exit structured program flow graph (SESPFG). Figure 2 is an example of a SESPFG.

Dominance and Forward Reachability Relations

A depth first search procedure as described in [22] is applied to a program flow graph to identify its backedges. Each node, n of a flow graph is labeled with a unique positive number, $d \leq |N|$, such that the label on n is the reverse of the order in which n is last visited during

depth first search. The unique label, d assigned to a node is called the depth first number (DFN) of that node. For each node, n , of a SESPFG, $G = (N, A, r, e)$, let $DFN[n]$ denote the depth first number of n .

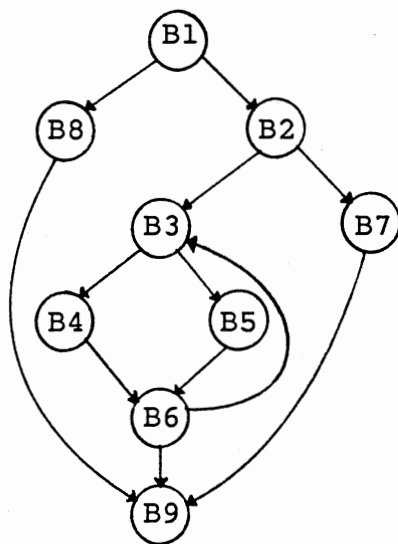


Figure 2. Single Exit Structured Flow Graph

Two fundamental properties [19] of reducible flow graphs are (1) backedges are unique and (2) if (b, h) is a backedge, then $DFN[h] \leq DFN[b]$.

Let $B = \{ (b, h) \mid DFN[b] \geq DFN[h] \}$ be the set of backedges of a program flow graph.

Definition 5. The acyclic program flow graph of a reducible flow graph is $G' = (N, A - B, r, e)$.

Definition 6. Suppose $G = (N, A, r)$ is a program flow graph and suppose further that d and n are any two nodes of G . d predominates n if and only if (iff) every path from the initial node r to n always includes d .

Definition 7. If G is a SESPPFG, $G = (N, A, r, e)$, then the graph $R(G) = (N, E, e, r)$ is a reverse flow graph of G , where $E = \{ (x, y) \mid (y, x) \in A \}$.

Definition 8. Let n and p be nodes of a PFG, $G = (N, A, r, e)$. Node p post-dominates n iff p is a predominator of n in $R(G)$.

If p post-dominates n in G , then whenever control transfers to n , control eventually will transfer to p . Post-dominance information of a flow graph is used to determine the exit or join point of a control structure.

Both predominance and post-dominance relations can be represented with dominance trees. The predominance and post-dominance trees of the flow graph in Figure 2 are shown in Figure 3.

Definition 9. Suppose n_1 and n_2 are any two nodes of a PFG, $G = (N, A, r, e)$. Node n_2 is forward reachable from n_1 if either:

1. there is a path which includes n_1 in the acyclic flow graph G' of G , from the initial node r to n_2 ; or
2. n_1 is inside a while-do control structure and n_2 is forward reachable (by condition 1) from the exit node of the while-do loop containing n_1 .

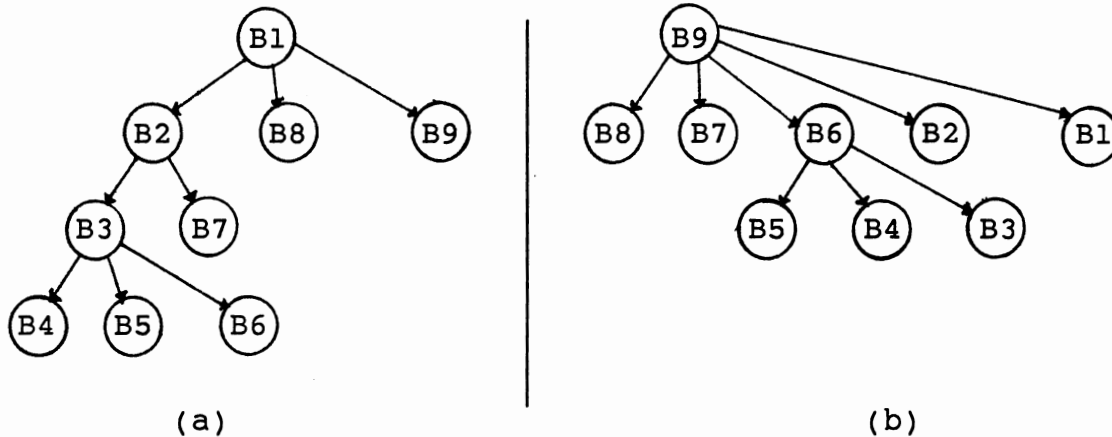


Figure 3. (a) Predominance Tree
(b) Post-dominance Tree

Forward reachability as defined is a reflexive, transitive, and antisymmetric relation. Thus, forward reachability is a partial order. If n_2 is forward reachable from n_1 , then n_1 and n_2 lie on some common execution path.

Suppose n_1 and n_2 are distinct nodes of a structured program flow graph $G = (N, A, r, e)$, then n_1 and n_2 are disjoint if n_1 is not forward reachable from n_2 and n_2 is not forward reachable from n_1 .

Ordering of Nodes

Code optimization based on the operand dependence graph technique depends on the identification of a processing order for the nodes of a program flow graph. A suitable node processing order must preserve any precedence constraint imposed by control flow. The node processing order developed for the operand dependence graph is called

predominated-inverse-post-dominated (PIPD) order.

Let $T(G)$ be the dominator tree of a program flow graph, $G = (N, A, r, e)$, such that the children of each parent node are ordered from left to right by increasing depth first number. The PIPD ordering of flow graph nodes is the preorder traversal listing of $T(G)$.

Let n_1 and n_2 be distinct nodes of a flow graph in which the nodes are listed in a PIPD order. Some characteristics of PIPD order are:

1. If n_1 predominates n_2 , then n_1 precedes n_2 in a PIPD order listing of nodes.
2. If there is a forward path from n_1 to n_2 , then n_1 precedes n_2 in a PIPD order listing of nodes.
3. The exit node, e of a single exit flow graph is the last node in a PIPD order listing of nodes.
4. The initial node r of a flow graph is the first node in a PIPD order listing of nodes.
5. a node and its predominees are contiguous in a PIPD order listing of nodes.
6. If n_2 post-dominates n_1 , then n_2 succeeds n_1 in a PIPD order listing of nodes.

The first four properties are due to the antisymmetric and transitive properties of the predominance and forward reachability relations. The fifth characteristic is a property of preorder traversal of trees. In a preorder tree traversal, the root of a subtree is visited first, then the children of that subtree are visited next in a left to right

order. The sixth property is due to the preorder traversal of predominance tree and the ordering of children nodes in a predominance tree.

Definition 10. Suppose n is a node of a program flow graph. The index (position) of n in a PIPD order listing of nodes is called the linear order number (LON) of n .

Assuming an indexing origin of one, if each node in a PIPD order listing of nodes is replaced with its LON, the resulting list is a sequence of first $|N|$ positive integers in increasing order. Thus, PIPD order is a total (linear) ordering. From now on, any reference to linear (total) order in the text refers to PIPD order.

Loop Identification

Linear ordering of nodes simplifies the task of finding loop sections of a structured program flow graph. In a structured program flow graph, a loop has a unique entry (header) node. A loop header node is a loop node that predominates every other node of a loop. Since a node and its predominees are contiguous in a linear order listing of nodes, the flow graph nodes constituting the body of a loop are contiguous.

Suppose $B(G)$ is the set of backedges of a reducible flow graph G . Suppose further that (b, h) is an element of $B(G)$. Define $[h] = \{x \mid (x, h) \text{ is in } B(G)\}$. The last element of $[h]$ is the element with the largest linear order number. The immediate post-dominator of the last element of

$[h]$ is the exit node of the loop whose header is h .
 Given a back edge (b, h) and $[h]$ of a structured program flow graph G , let the last element of $[h]$ be g . Suppose the LONs of h and g are S and T respectively. The loop region whose header node is h is the set $\{ n \mid S \leq \text{LON}[n] \leq T \}$, where $\text{LON}[n]$ represents the linear order number of n . The nodes in $[h]$ are called looping nodes.

Each node of a structured program flow graph is assigned a region tag subject to the following constraints:

1. The region tag of a node that does not belong to any loop is zero;
2. In a nested loop, the nodes which constitute the body of an inner loop have the same region tag; and each inner loop has a distinct region tag;
3. If a loop has no inner loops, the header node and the nodes belonging to that loop have the same region tag;
4. The region tags of nodes in a parent loop are less than the region tag of any loop contained in that parent.

Flow Graph Transformation

After program loops have been identified, any loop whose program flow subgraph has the structure shown in Figure 4(a) is transformed to the subgraph of Figure 4(b). This transformation converts a while-do loop into a do-while loop without changing a program's semantics.

The while-do loop to do-while loop conversion both increases the number of movable loop invariant statements

and ensures that moved loop invariants are executed only when a loop's body is executed. Following loop transformation, the predominance and post-dominance trees are updated to include the new nodes and edges added to a flow graph. A new PIPD ordering is obtained from the modified predominance tree.

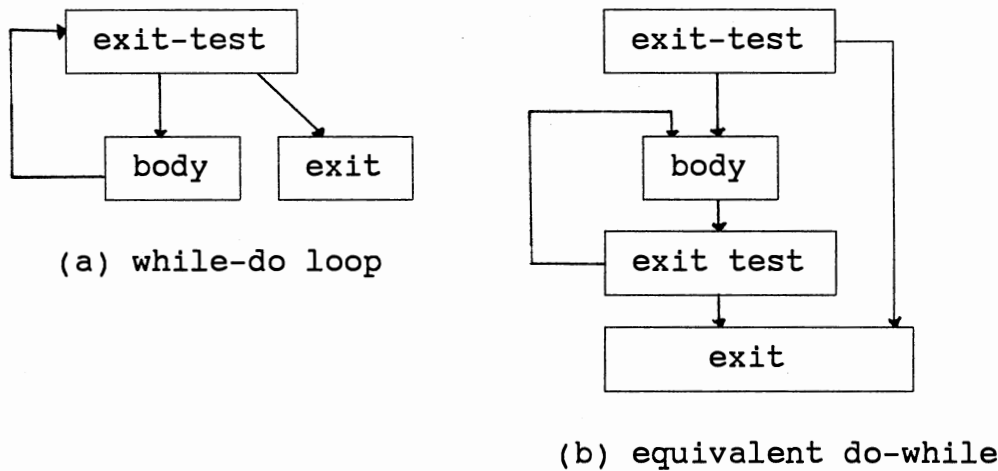


Figure 4. While-do Loop to Do-while Loop Conversion

Node Classification

To keep track of transitions from one control environment to another control environment, flow graph nodes are typed. Five types of nodes are distinguished: join of a conditional control structure, loop header, loop exit, end of loop marker, and ordinary node. A node is classified a

join node, if

1. it has at least two immediate predecessors in the DAG of a flow graph;
2. it is not a loop header;
3. it is not a loop exit node; and
4. the immediate predecessors are disjoint.

A flow graph node is a loop header node if it is a destination of a backedge, and a node is a loop exit if it is the unique node to which control transfers upon loop termination. An end of loop marker node is the first non-loop node following the last node of a program loop in a PIPD ordering of nodes. If a node is both a loop exit and an end of loop marker, then that node is classified as end of loop marker. If a node is not a join, loop header, end of loop marker, or loop exit, then that node is an ordinary node.

During the processing of intermediate code statements, special operations are initiated when certain node types are encountered. For instance, when a join node is about to be processed, any potentially hoistable or sinkable code in the preceding control environment is analyzed for forward code motion or backward code motion optimization. If the next flow graph node to be processed is an end of loop marker, loop specific optimizations such as loop invariant statement detection, loop invariant code motion, and loop induction variable simplification are performed before continuing with statement processing. By assigning type tags to flow graph

nodes, the necessary code improvement operations can be initiated at a control environment boundary.

Path Covers

Code optimization problems can be grouped into two categories based on their path constraints. One class of code optimization problems requires the information of interest to be present along all paths leading to a point. The second class of problems require that the information of interest occur in at least one path to a point. Forward reachability information of a flow graph node is sufficient to check the path constraint of class two problems. In this section, the notion of a path cover is introduced as an approach for checking all-path data flow constraint directly.

Definition 11. Let $\{n_1, \dots, n_k\}$ ($k \geq 1$) be a subset of the nodes of a flow graph. Suppose m is a flow graph node such that there exists a forward path from each node in $\{n_1, \dots, n_k\}$ to m . $\{n_1, \dots, n_k\}$ is a path cover for m , if every path from the initial node of a flow graph to m must include a node from $\{n_1, \dots, n_k\}$.

This type of path cover is called a node path cover problem. When $k = 1$, $\{n_1\}$ is a path cover for m if $n_1 = m$ or n_1 is a predominate of m .

In Figure 2 (page 25), the subset of nodes which are path covers for node B_9 are $\{B_1\}$, $\{B_2, B_8\}$, $\{B_3, B_7, B_8\}$, $\{B_6, B_7, B_8\}$, $\{B_4, B_5, B_7, B_8\}$, and $\{B_9\}$.

Definition 12. Let f be a fork node of a program flow graph DAG and let m be the immediate post-dominator of f .

Define $SCOPE[f, m] = \{ n \mid f \text{ predominates } n \text{ and } n \neq m \}$.

The set $SCOPE[f, m]$ specifies the scope of a conditional control structure whose header node and join node are f and m , respectively. The definition of $SCOPE[f, m]$ excludes the join node m from $SCOPE[f, m]$. Every node in $SCOPE[f, m]$ with the exception of f is control dependent on f . That means the execution of any node in $SCOPE[f, m] - \{f\}$ depends on the truth value of the predicate at f . Based on this definition of a conditional control environment, the subsets $\{B3, B4, B5\}$, $\{B2, B3, B4, B5, B6, B7\}$, and $\{B1, B2, B3, B4, B5, B6, B7, B8\}$ are control environments of the flow graph in Figure 2.

Definition 13. Suppose $\{n_1, \dots, n_k\}$ is a subset of nodes of a program flow graph. A common predominator of $\{n_1, \dots, n_k\}$ is a node which predominates every node in that set. Suppose f is a common predominator of $\{n_1, \dots, n_k\}$. Node f is the least common predominator of $\{n_1, \dots, n_k\}$ if every common predominator of the nodes in the set also predominates f .

Definition 14. Suppose $\{n_1, \dots, n_k\}$ is a subset of nodes of a program flow graph. A common post-dominator of $\{n_1, \dots, n_k\}$ is any node which post-dominates every node in the set. Suppose j is a common post-dominator of $\{n_1, \dots, n_k\}$. Node j is the least common post-dominator of the nodes in the set if every post-dominator of $\{n_1, \dots, n_k\}$

post-dominates j .

Definition 15. Suppose $\{n_1, \dots, n_k\}$ is a subset of the nodes belonging to some conditional control structure C . Let f be the least common predominator of $\{n_1, \dots, n_k\}$. Suppose the immediate post-dominator of f is m . The set $\{n_1, \dots, n_k\}$ is a conditional environment cover for the conditional control structure C if every path from f to m must include a node from $\{n_1, \dots, n_k\}$.

The concept of a conditional control environment cover provides a method for checking node path covers. To reduce the amount of computation involved in node path cover analysis, the lemmas below are used.

Lemma 1. Suppose $\{n_1, \dots, n_k\}$ is a subset of the nodes of a flow graph. Let m be a flow graph node such that there exists a forward path from each node in $\{n_1, \dots, n_k\}$ to m . Let the least common predominator of $\{n_1, \dots, n_k\}$ be f . If $\{n_1, \dots, n_k\}$ is a path cover for m , then f predominates m .

Proof. Node f is a path cover for each node in $\{n_1, \dots, n_k\}$. If f does not predominate m , then there exists at least one forward path from the initial node of a flow graph to m which does not pass through f . Therefore $\{n_1, \dots, n_k\}$ is not a path cover for m . Hence, if $\{n_1, \dots, n_k\}$ is a path cover for m , then the least common predominator of the nodes in the covering set predominates m . □

Lemma 2. Suppose $\{n_1, \dots, n_k\}$ is a subset of the nodes of a flow graph. Let m be some flow graph node for which m is

forward reachable from each node in the subset to m . Let the least common post-dominator of $\{n_1, \dots, n_k\}$ be j . If $\{n_1, \dots, n_k\}$ is a path cover for m , then j predominates m .

Proof. Every path from each node of $\{n_1, \dots, n_k\}$ to m must pass through j because j is a common join of paths originating from the nodes in the subset. If $\{n_1, \dots, n_k\}$ is a path cover for m , then j is also a path cover for m . Suppose $\{n_1, \dots, n_k\}$ is not a path cover for m , then there exists a forward path from the initial node of a flow graph to m which does not pass through any node in the subset. Therefore, if $\{n_1, \dots, n_k\}$ is a path cover for m , the least common post-dominator of $\{n_1, \dots, n_k\}$ is a predominator of m . □

By lemmas 1 and 2, $\{B_4, B_5\}$ is not a path cover for node B_9 (Figure 2) because B_9 can be reached from B_8 and B_7 without passing through B_4 or B_5 . Notice that $\{B_3\}$, $\{B_4, B_5\}$, and $\{B_3, B_4, B_5\}$ are path covers for B_6 . The subset $\{B_3, B_4, B_5\}$ is a union of the covering sets $\{B_3\}$ and $\{B_4, B_5\}$. This example illustrates that it is not necessary to examine every node in a potential path cover set in order to deduce whether a subset of nodes is a path cover for some node or control environment. Every input set to a path cover problem has an equivalent subset of essential nodes called a minimal set.

Definition 16. Let $\{n_1, \dots, n_k\}$ be a subset of the nodes of a flow graph. A minimal set for $\{n_1, \dots, n_k\}$ is a

subset $\{b_1, \dots, b_s\}$ of $\{n_1, \dots, n_k\}$ such that if n_i ($1 < i < k$) $\in \{b_1, \dots, b_s\}$, then $\{b_1, \dots, b_s\}$ does not contain any node n_i predominates or n_i post-dominates.

Lemma 3. Suppose $\{b_1, \dots, b_s\}$ is a minimal set for $\{n_1, \dots, n_k\}$. The subset of nodes $\{b_1, \dots, b_s\}$ is a control environment cover iff $\{n_1, \dots, n_k\}$ is a control environment cover.

Proof. Obvious from the definition of a minimal set. Lemmas 1 and 2 state necessary conditions for a set of nodes to cover every path to a given node, while lemma 3 states that the path cover problem can be decided with a smaller set containing non-redundant elements. □

Path Cover Analysis

The notion of a path cover has been defined without an effective procedure for deciding whether a set of nodes is either a path cover for a given node or a control environment. In order to specify a precise method for determining path covers, the concept of fork-width and join-width of a node are introduced.

Definition 17. Let f be a fork node of the DAG of a program flow graph and let d be the immediate post-dominator of f . The path-width of the conditional structure $\text{SCOPE}[f, d]$ is the number of acyclic paths from f to d .

McCabe[25] states that the number of independent paths (cyclomatic complexity) of a structured program is the number of predicates plus one. In a structured program flow

graph, a conditional structure induces a subflow graph on a program flow graph. If McCabe's cyclomatic complexity measure is applied to a control structure with loop backedges removed (DAG of control structure), then the cyclomatic complexity of that control structure is equal to the path-width of that control structure. The path-width of a control structure gives the number of alternate paths in that control structure.

Definition 18. Let f be a fork node in the DAG of a flow graph and let j be the immediate post-dominator of f . The fork-width of f is a positive number, σ with the following constraint:

1. there exist σ nodes predominated by f ;
2. the σ nodes form a minimal set for the conditional environment $\text{SCOPE}[f, j]$; and
3. every path from f to j must include one of the σ nodes.

Definition 19. Suppose j is a node with two or more immediate predecessors in a flow graph DAG and d is the immediate predominator of j . The join-width of j is a positive integer δ , such that

1. there exist δ nodes predominated by d ;
2. the δ nodes form a minimal set for the conditional environment $\text{SCOPE}[d, j]$; and
3. every path from d to j must include one of the δ nodes;

The value σ and the value δ are related to the path-width of the conditional environment $\text{SCOPE}[f, j]$ or $\text{SCOPE}[d, j]$ as the case may be. This fact is stated in the next lemma.

Lemma 4. Suppose f is a fork node in the DAG of a program flow graph. The fork-width of $f \leq$ path-width of the conditional structure originating at f .

Proof. Let σ be the fork-width of node f . σ is the cardinality of a minimal set of the control structure whose head is f .

$\sigma \geq 1$ because the path-width of a fork node ≥ 1 . Suppose $\sigma > 1$. Then the minimal set contains mutually disjoint nodes. Since fork nodes create disjointedness, there are at most q nodes in the minimal set, where q is the number of fork legs in the control structure headed by f . Each fork leg is an alternate path to the join of a conditional structure. Hence, $\sigma \leq q \leq$ the number of acyclic paths from f to the join of f . Therefore, the fork-width of $f \leq$ the path-width of the control structure originating at f . \square

The fork-width (join-width) of a fork node (join node) is not unique. To ensure that a deterministic value is calculated for the parameters σ and δ , the following computation rule is adopted:

1. The fork-width and join-width of a nested control structure should be evaluated in deepest to shallowest order;
2. Fork-width of a node with a unique immediate successor in the DAG of a flow graph is one;
3. Join-width of a node with a unique immediate predecessor in the DAG of a flow graph is one;
4. Let f be a fork node of a flow graph DAG and let j be the

immediate post-dominator of f . Suppose $\text{FWD-SUCC}[f, \text{pred}[j]]$ denotes $\{n \mid n \in \text{pred}[j] \text{ and } (f \text{ predominates } n \text{ or } n \in \text{succ}[f])\}$.

The fork-width of f is the sum of the join-widths of the nodes in $\text{FWD-SUCC}[f; \text{pred}[j]]$.

5. Let j be a node with two or more immediate predecessors in a flow graph DAG. The join-width of j is the sum of the join-widths of the nodes in $\text{pred}[j]$.
6. The join-width of the initial node of a flow graph is zero.
7. The fork-width of the exit node of a single exit flow graph is zero.

The result of applying this computation rule to the flow graph in Figure 2 is depicted in Table I. Before computing the fork-widths and join-widths of nodes, while-do control structures must be transformed to do-while control structures.

TABLE I

Fork-width and Join-width of the
Nodes in a Program Flow Graph

Node	Fork-width	Join-width
B1	4	0
B2	3	1
B3	2	1
B4	1	1
B5	1	1
B6	1	2
B7	1	1
B8	1	1
B9	0	4

Let $\text{FORK-WIDTH}[n]$ and $\text{JOIN-WIDTH}[n]$ represent the fork-width and join-width of node n , respectively. Figure 5 is an algorithm to determine if a given set of nodes of some conditional control structure is a path cover for that conditional control structure and the algorithm in Figure 6 determines whether a subset of nodes is a path cover for some node m .

ALGORITHM 1: Control Environment Cover Algorithm.

Input. A minimal set, S containing a subset of the nodes of a program flow graph;

The arrays FORK-WIDTH and JOIN-WIDTH holding the fork-widths and join-widths, respectively of a flow graph;

The DAG of a program flow graph;

Predominance and Post-dominance relations of a program flow graph.

Output. TRUE if S is a conditional environment cover; FALSE otherwise.

Method. First partition S into subsets corresponding to subconditional structures (steps 1 and 2). Then check if each subset of S is a conditional environment cover (step 3). Finally, check if the subconditional structures combined is a cover for the conditional environment S describes.

1. Partition S into distinct subsets C_1, \dots, C_z , such that members of each subset have the same immediate post-dominator.

2. For each C_i compute h_i as follows:

 If $|C_i| = 1$, then $h_i :=$ the element of C_i ;

 Else

$h_i :=$ least common predominator of nodes in C_i ;

 endfor

 If $z > 1$, then begin

$F :=$ least common predominator of $\{h_1, \dots, h_z\}$;

$J :=$ immediate post-dominator of F ;

 end

 Else

$F := h_1$; $J :=$ immediate post-dominator of h_1 ;

 endif

 PATH-COVER := TRUE;

3. For each C_i do

 If $|C_i| > 1$, then begin

$PW := 0$;

 For each n in C_i do

 if n is a fork node of flow graph DAG, then

$PW := PW + \text{FORK-WIDTH}[n]$;

 else $PW := PW + \text{JOIN-WIDTH}[n]$;

 endfor

 if $PW \neq \text{FORK-WIDTH}[h_i]$, then begin

 PATH-COVER := FALSE;

 exit loop;

 end

 end

(Continued from page 42)

```

Else begin
  let d be the immediate post-dominator of hi;
  if d != J, then
    if FORK-WIDTH[hi] != JOIN-WIDTH[d], then begin
      PATH-COVER := FALSE;
      exit loop;
    end
  end
end
endfor

4. if PATH-COVER = TRUE, then begin
  if z > 1, then begin
    SUCC := U FWD-SUCC[hi; pred[J]]
           i = 1, ..., z
    if FWD-SUCC[F; pred[J]] != SUCC, then
      PATH-COVER := FALSE;
    end
  end
end

```

Figure 5. Conditional Control Environment
Cover Checking Algorithm

ALGORITHM 2. Node Path Cover Algorithm.
Input. Same as Algorithm 1.
Output. Same as Algorithm 1.
Method. Combines Algorithm 1 and lemmas 1 and 2.

```

PATH-COVER := FALSE;
if k = 1, then begin
  if (nl = m) or (nl predominates m), then
    PATH-COVER := TRUE;
  end

else begin
  perform steps 1 and 2 of Figure 5;
  if F does not predominate m or
    J does not predominate m, then
    PATH-COVER := FALSE;
  else perform steps 3 and 4 of Figure 5;
end

```

Figure 6. Node Path Cover Analysis
Algorithm

As an example of path cover analysis, Figures 5 and 6 will be applied to determine whether $\{B4, B5, B7\}$ is a conditional control environment cover and a path cover for node B9 with respect to the flow graph in Figure 2. First $\{B4, B5, B7\}$ is subjected to conditional environment cover analysis. The values of various variables at the end of each step of Figure 5 are shown below.

Step 1:

```
C1 = {B4, B5}; h1 = B3;
C2 = {B7};     h2 = B7;
```

Step 2:

```
F = B2; J = B9;
PATH-COVER = TRUE;
```

Step 3:

```
for C1 PW = FORK-WIDTH[B4] + JOIN-WIDTH[B5]
           = 2 = FORK-WIDTH[h1 = B3]
```

Step 4:

```
pred[J = B9] = {B6, B7, B8}
SUCC = FWD-SUCC[h1 = B3; pred[J = B9]]
      U FWD-SUCC[h2 = B7; pred[J = B9]]
      = {B6} U {B7} = {B6, B7}
FWD-SUCC[F = B2; pred[J = B9]] = {B6, B7}
FWD-SUCC[F; pred[J]] = SUCC
```

Since the value of PATH-COVER is TRUE at the end of step 4, $\{B4, B5, B7\}$ is a path cover for the conditional control environment enclosing $\{B4, B5, B7\}$.

To decide if $\{B4, B5, B7\}$ is a path cover for B9, Figure 6 is used. After performing steps 1 and 2 of Figure 5, the statement `PATH-COVER := FALSE` in Figure 6 is executed next. At this statement, the value FALSE is assigned to PATH-COVER because lemmas 1 and 2 are not satisfied. Hence $\{B4, B5, B7\}$ is not a path cover for B9.

Path Analysis and Code Optimization

The control flow constraints of some code optimization problems can be determined by using either a conditional environment or a node path cover test procedure. In this subsection, the path constraints of common code optimization problems are formulated as path cover problems.

Common Subexpression Elimination

Suppose E is an expression such as $x + y$ at some program point, q . The instance of E at q is redundant if (1) $x + y$ always is computed before control transfers to point q ; and (2) no statements between the previous evaluations of $x + y$ and q has a side effect on either x or y . Suppose p_1, \dots, p_k are the most recent program points with previous instances of expression E reaching point q , then condition (1) is satisfied if $\{p_1, \dots, p_k\}$ is a node path cover for the point q .

Code Hoisting

Suppose E is an expression evaluated in some disjoint blocks $\{B_1, \dots, B_k\}$ of a conditional control structure, C . Let F be the fork node where the conditional structure C originates. E can be factored out of $\{B_1, \dots, B_k\}$ and placed in the fork node, F if (1) every path originating from F must include a node from $\{B_1, \dots, B_k\}$; and (2) the instances of E in $\{B_1, \dots, B_k\}$ use the same value of the

source operands. The first condition is a conditional control environment path cover problem.

Code Sinking

Let $\{B_1, \dots, B_k\}$ be a set of disjoint nodes of a conditional structure, C . Suppose there is an instance of some assignment statement S in each of B_1, \dots, B_k . Let m be the merge point of the conditional structure. The statement S can be factored out of $\{B_1, \dots, B_k\}$ if (1) every path from the immediate predominator of m to m must pass through a block in $\{B_1, \dots, B_k\}$; (2) the variable assigned to in statement S is not referenced in any statement following S in blocks B_1, \dots, B_k ; (3) no statement in a block which succeeds a B_i ($1 \leq i \leq k$) in the conditional structure containing $\{B_1, \dots, B_k\}$ references the variable assigned to in S ; and (4) the source operands of S are not modified by statements following S in blocks B_1, \dots, B_k .

The first condition is satisfied if $\{B_1, \dots, B_k\}$ is a path cover for the conditional environment enclosing $\{B_1, \dots, B_k\}$.

Loop Invariant Code Motion

Let $G = (N, A, r, e)$ be a flow graph in which any while-do control structure has been transformed to a do-while control structure. Suppose E is an expression whose source operands are invariant in some program loop, L . Let p be the point

where E is located in L and suppose the exit gates of L are $\{B_1, \dots, B_k; k \geq 1\}$. E can be moved out of L if p is a common predominator of $\{B_1, \dots, B_k\}$.

Path Analysis Information Representation

The path analysis questions prevalent in an operand dependence graph based code improvement system are:

- Q1. does node n_1 predominate n_2 ?
- Q2. Does node n_1 post-dominate node n_2 ?
- Q3. Is node n_1 forward reachable from node n_2 ?
- Q4. Are nodes n_1 and n_2 disjoint?
- Q5. Is the subset $\{n_1, \dots, n_k\}$ of flow graph nodes a path cover some node m ?
- Q6. Is the subset $\{n_1, \dots, n_k\}$ of nodes a path cover for control environment described by $\{n_1, \dots, n_k\}$?
- Q7. What is the least common predominator of the subset $\{n_1, \dots, n_k\}$ of node?
- Q8. What is the least common post-dominator of the subset $\{n_1, \dots, n_k\}$ of nodes?

There are a number of data structures which are suitable representations for these path problems. For instance, dominance trees and two dimensional tables can be used to answer predominance and post-dominance related questions. The search time for a dominance tree representation is of logarithmic order, while the search time for a two dimensional table implementation is $O(1)$. However, a two dimensional table incurs a quadratic space

complexity. Both search time efficiency and storage space efficiency must be considered in selecting a data structure.

Each node of a flow graph has an associated path record which consists of the following fields:

1. linear order number of node;
2. post dominance number of node;
3. set of predominees of node;
4. set of post-dominees of node;
5. fork-width of node;
6. join-width of node;
7. set of forward reachable node;
8. immediate predecessors of node;
9. immediate successors of node.

To reduce the storage required for the set type fields of path record, the properties of predominance and post-dominance relations are exploited.

Let $POST-TREE[G]$ represent the post-dominance tree of a flow graph, $G = (N, A, r, e)$. Suppose the children of a non-leaf node in $POST-TREE[G]$ are ordered from left to right by decreasing linear order number. A preorder listing of $POST-TREE[G]$ is the reverse sequence of the preorder listing of the predominance tree of G . The position (index) of a node in a preorder listing of $POST-TREE[G]$ is called the post dominance number (PDN) of that node. Suppose the linear order number of some node n is x , then the post dominance number of n is $|N| - x + 1$.

In a preorder tree traversal, the root of a subtree and the descendants of that root are contiguous. Hence, when node names either are replaced by LONs in a preorder traversal of predominance tree or are replaced by PDNs in a preorder traversal of a post-dominance tree, the root of that subtree and the descendants of that root form a finite sequence of consecutive positive integers. Therefore, an ordered pair of positive integers is sufficient to specify the subset of predominees (post-dominees) of a node.

Suppose n is a node of a flow graph. The subset of nodes n predominates is specified as $[x, y]$, where x is the LON of n and y is the LON of the last descendant of n in the preorder listing of the predominance subtree rooted at n .

$$[x, y] = \{ t \mid x \leq t \leq y \}.$$

Similarly, the subset of nodes n post-dominates is specified as $[u, v]$, where u is the PDN of n and v is the PDN of the last descendant of n in the preorder listing of the post-dominance subtree rooted at.

Suppose n is a flow graph node whose predominance interval is $[x, y]$. Let m be some flow graph node whose LON is q . n predominates m if

$$x \leq q \leq y.$$

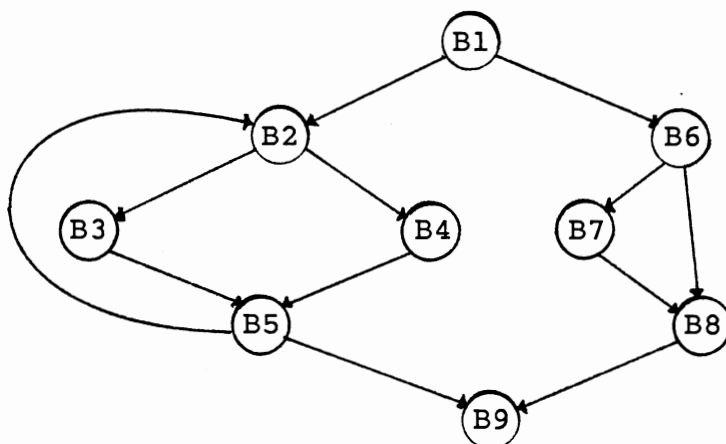
The same relationship holds if x , y , and q are post-dominance numbers and $[x, y]$ represents post-dominance interval of n . The use of intervals requires two comparisons to determine if n predominates (post-dominates)

m and storage for $2|N|$ interval numbers.

Forward Reachability Set Representation

The question "is the node n_1 forward reachable from node n_2 ?" is the most frequently asked question in an operand dependence graph based code improvement technique. Unfortunately forward reachability set of every flow graph node may not be represented with a single ordered pair of LONs. The reason is because "if-then-else" conditional structure introduce disjoint true and false branches. Figure 7 illustrates how conditional control structures affect the contiguity of forward reachability sets.

In Figure 7, nodes B1 and B2 corresponds to if statements with explicit then and else parts, while node B6 corresponds to an if statement with then part only. Notice that for the if statement with no else part the forward reachability of the successors of B6 are contiguous. On the otherhand, the forward reachability sets of the successors of B2 are not contiguous. Also, there are two break points in the forward reachability set of B3. This is due to nesting an if-then-else statement within another if-then-else statement.



Node Processing Order = B1 B2 B3 B4 B5 B6 B7 B8 B9

<u>Node</u>	<u>Forward Reachability Sets</u>								
B1	B1	B2	B3	B4	B5	B6	B7	B8	B9
B2		B2	B3	B4	B5				B9
B3			B3		B5				B9
B4				B4	B5				B9
B5					B5				B9
B6						B6	B7	B8	B9
B7							B7	B8	B9
B8								B8	B9
B9									B9

Figure 7. Flow Graph and Forward Reachability Sets

If there are $x \geq 0$ break points in a node's forward reachability set, then $x + 1$ ordered pairs of LONs are required to specify that node's forward reachability set. For example, the forward reachability set of B3 is $\{ [3, 3], [5, 5], [9, 9] \}$. Determining if a node is forward reachable from B3 requires three LON interval searches. To provide quick response to forward reachability question a hybrid representation scheme is proposed. If a node's

forward reachability set is contiguous, then an ordered pair of LONs should be used to specify that node's forward reachability set. If a break point exists in the forward reachability set of a node, then a bit vector of $|N|$ bits (one bit per node) should be used to specify that node's forward reachability set.

Suppose a program has no if-then-else or case conditional structure, the forward reachability set of every basic block is contiguous. For such a program, there is no need to perform either code hoisting or code sinking optimization since there are no parallel blocks.

Summary

Control structure analysis is the processing of a program flow graph to derive structure information about a flow graph. The information extracted from a flow graph include pre-dominance, post-dominance, and forward reachability relations between nodes of a flow graph. Predominance and post-dominance relations information are used to (1) define a topological order on nodes and to identify the extent of control structures.

The concept of a path cover for a control structure is introduced to unify "all-path" code improvement problems. Path cover analysis is implemented by assigning path weights called fork-width and join-width to flow graph nodes.

CHAPTER IV

VARIABLE DEFINITION ANALYSIS

Introduction

An assignment of a value to a variable invalidates computations performed with previous values of that variable along control flow paths leading to a new definition point. The association of each variable referencing statement with the set of statements which could define the value of that variable at a use point has been implemented using use-definition chains and definition-use chains[1].

In order to construct the use-definition chains or definition-use chains, reaching definitions data flow analysis system of equations is solved. The use of reaching definition information to detect feasible global code optimizations induces additional processing after applying an optimization procedure. Post optimization processing includes the recomputation of reaching definitions following any optimization procedure which moves code or eliminates statements. The code optimization technique developed in this work, confines redundant statement detection to program regions. Since inter-region redundancies are not removed,

computations outside a region do not affect the detection of redundant code within that region.

This chapter describes the method developed to handle variable definition and variable reference analysis in an operand dependence graph based code optimization technique. The method is based on the fact that a definition of a variable creates a new version (instance) of that variable. Instead of linking a variable's definition point with the statements which may reference that value, a unique number called a version number is associated with a definition instance.

Operand Version Numbering

Before processing the initial node of a program flow graph, the version number of every variable and constant is initialized to zero. Let v be any program variable and let $VN[v]$ denote the current version number of v . Suppose S is a statement of the form $v := \text{exp}$ (exp is some expression). After S is processed, $VN[v]$ is incremented by one. Any statement T for which there exists a forward path from S to T but before another definition statement for v can reference the version of v created at S .

Suppose B is a basic block (flow graph node) and suppose v is a variable. Let $RVNTOP[v,B]$ denote the subset of versions of v which can reach the top of B via forward edges. The top of B means the point preceding the first statement in block B . A definition of v in some block P can

reach the top of another block B if

1. B is forward reachable from P;
2. that definition of v in P is the last definition of v in block P;
3. there are no definitions of v in any block between P and B for which there exists a forward path to B.

A version number marking the definition of the variable v in some block P can reach the top of another block B if that definition satisfies conditions (1), (2), and (3). If v is referenced in B before being defined, then the value of v prior to any definition of v in B is one of the definitions of v represented by the reaching versions set $RVNTOP[v,B]$. If v is assigned a value in B before any statement which references v in B, then the value of v at any point in B is the definition of v closest to that point.

The statements in a block may change the value of a subset of a program's variables when that block is executed. To describe the effect of a block on the set of variables in a program, an ordered triple of version numbers called block mutation record (BMR) is maintained for each variable. The first component of BMR is called the initial block version history (IBVH); the second member of BMR is designated entry block version number (EBVN); and the third component is called block exit version number (BEVN). The initial block version history of v is a representative for the instances of v which initially reaches the top of a block and is defined as the largest version number in the set $RVNTOP[v,B]$

(for some block B). The entry block version number is the current version number of v at the top of B.

The block exit version number of v is the current version number of v at any point in block B.

Let $BMR[v,B]$ represent the block mutation record of the variable v in block B. Suppose $BMR[v,B] = (x, y, z)$. If v is not defined in B, then $y = z$ at every point in B. On the otherhand, if v is assigned value in B, then $z > y$ following the first statement which defines v because $VN[v]$ is incremented each time a value is assigned to v . Therefore, the relation $z \geq y$ must be true for each variable at every point in a block.

At the top of B, the value of x is the highest version number in the reaching versions set $RVNTOP[v,B]$. Since x is a version number, $x \leq y$ at the top of B. Hence, initially the relation

$$x \leq y \leq z$$

holds. If v is defined in B, then after the definition statement the version number of v is incremented and x and z are set to the new version number for v . Thus, if v receives a new value in a block, then at the end of that block, the relation

$$y < x = z$$

must be true.

By examining the version numbers of a variable's BMR at the end of a block, it is possible to tell whether a variable is invariant in that block. The strength of the

version number concept is that it is oblivious to the actual program statements which alter the values of variables. The expression $z - y$ gives the number of times a variable is defined in a basic block.

Operand Version Propagation

Suppose v is a variable and S is a statement which references the value of v in an operation. The value of v used in statement S is one of the definitions of v which could reach the program point containing S . Which instance of v is used in S depends on the execution path taken to reach S . To compute the definitions reaching each node of a program flow graph, definitions reaching both the top and bottom of a node are required. With the version number approach, the definitions reaching the end of a node is determined from the values of the initial block version history and block exit version number components of variables block mutation records. Suppose B is a basic block whose block mutation record of some variable v at the end of B is (x, y, z) . The components of the block mutation record for B will satisfy one of the conditions

$$x \leq y \leq z \quad (R1)$$

$$y < x = z \quad (R2).$$

If condition (R1) is true, then the definitions of v reaching the end of B is the same as the definitions of v reaching the top of B . If condition (R2) is satisfied, then the definition of v reaching the end of B is the last

definition of v in B .

Let $RVNTOP[v,B]$ and $RVNBOT[v,B]$ denote the versions of v reaching the top and end of B , respectively. Suppose $pred'[B]$ is the set of immediate predecessors of B in the DAG of a program flow graph, then the equation

$$RVNTOP[v,B] = \bigcup_{P \in pred'[B]} RVNBOT[v,P] \quad (R3)$$

computes the versions of v reaching the top of B .

The above equation propagates versions of a variable along forward paths. If a program contains a loop, the reaching versions equation may produce incomplete solution. However, because code optimization is relative to a program region, it is not necessary to propagate reaching version numbers through loop backedges. Since variable reaching version numbers are propagated along forward paths (no cycles), the reaching versions analysis problem is computable in one iteration of a flow graph.

Version Analysis Implementation Strategy

Three types of information are required to analyze the definitions and usage patterns of program variables. Two of the four required items of information -- block mutation record and reaching version numbers already have been described. The other information item is a version creation point table (VCPT). The VCPT of a variable describes the set of program points where that variable may be defined and referenced in an intermediate code program. A VCPT entry is

a triple consisting of a definition descriptor, a version reference string, and a value class. The definition descriptor component consists of an intermediate code statement identifier, a basic block enclosing intermediate code statement, and the version number assigned to that definition. The version reference string for a version of a variable is a sequence of basic blocks where that instance of a variable may be used in an operation. The number of repetitions of a basic block in a version reference string is equal to the number of block statements where that version is referenced. The value class field indicates whether that version of a variable is a constant or not.

The flow graph in Figure 8 is used to illustrate the concept of operand version numbers for the variable v . Figure 9 shows the version creation point table, block mutation record, reaching versions sets, and version reference string for each version of v when the flow graph nodes are processed in the order B_1, B_2, \dots, B_9 .

Properties of Operand Version Numbers

To derive some properties of operand numbers, it is assumed that the statements of a block are processed in sequential execution order and that blocks are processed in topological order. A fundamental property of operand version number is monotonicity.

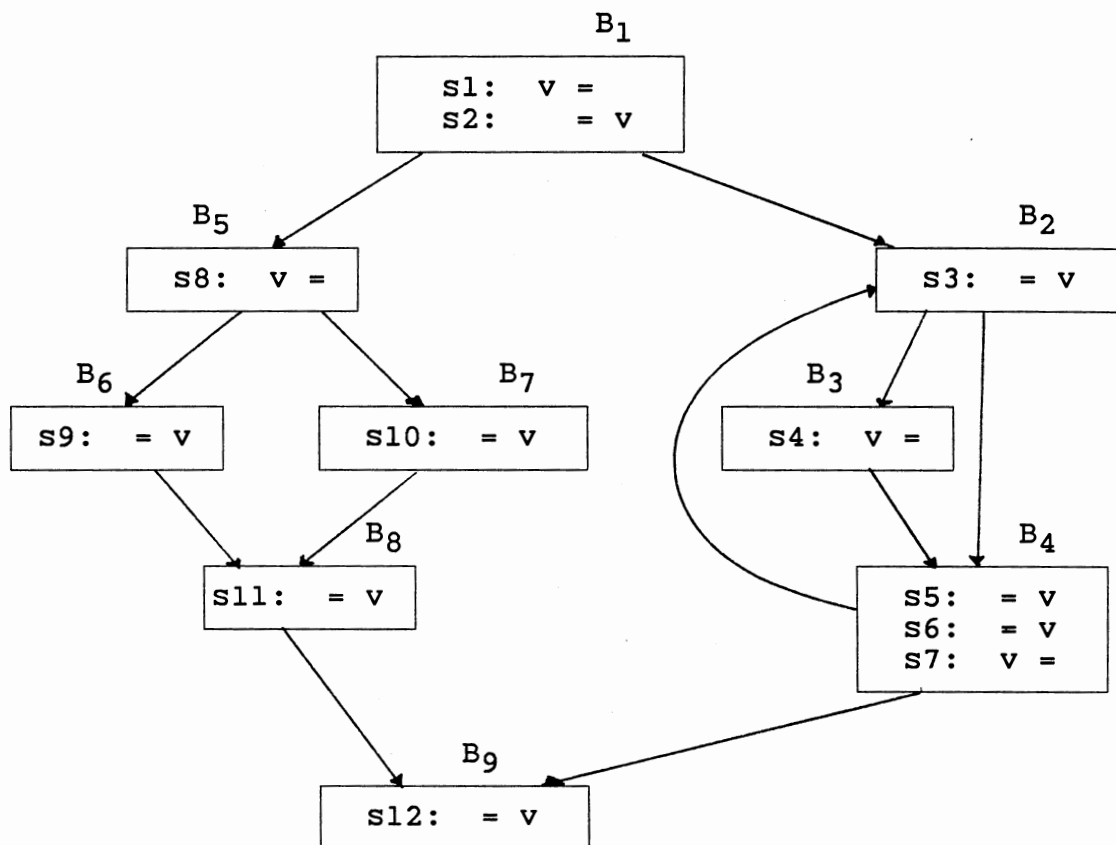


Figure 8. Flow Graph Showing Definitions and References of a Variable

Lemma 5. For any program variable v , the version number of v is monotonic between every pair of program points.

Proof. Let B_m and B_n be any two distinct blocks such that B_m precedes B_n in topological order. Let the version number of v at the end of B_m and at the top of B_n be X_m and X_n , respectively. If there exists some statement S which lies in some node B_j between B_m and B_n which may alter the value of v , then after S is processed $VN[v]$ is incremented. Thus X_m does not reach the top of B_n . Hence,

Statement	Basic Block	Version Number	Reference String
s1	B ₁	1	B ₁ B ₂ B ₄ B ₄
s4	B ₃	2	B ₄ B ₄
s7	B ₄	3	B ₉
s8	B ₅	4	B ₆ B ₇ B ₈ B ₉

(a) Version Creation Point Table and Version Reference String for 'v'.

```

RVNTOP[v,B1] = (0)
RVNTOP[v,B2] = (1)
RVNTOP[v,B3] = (1)
RVNTOP[v,B4] = (1, 2)
RVNTOP[v,B5] = (1)
RVNTOP[v,B6] = (4)
RVNTOP[v,B7] = (4)
RVNTOP[v,B8] = (4)
RVNTOP[v,B9] = (3, 4)

```

(b) Forward Reaching Definition of 'v'

Basic Block	Entry BMR	Exit BMR
B ₁	(0, 0, 0)	(1, 0, 1)
B ₂	(1, 1, 1)	(1, 1, 1)
B ₃	(1, 1, 1)	(2, 1, 2)
B ₄	(2, 2, 2)	(3, 2, 3)
B ₅	(1, 3, 3)	(4, 3, 4)
B ₆	(4, 4, 4)	(4, 4, 4)
B ₇	(4, 4, 4)	(4, 4, 4)
B ₈	(4, 4, 4)	(4, 4, 4)
B ₉	(4, 4, 4)	(4, 4, 4)

(c) Block Mutation Record of 'v' at Entry and Exit Points of Each Basic Block of Flow Graph

Figure 9. Variable Definition and Reference Analysis Data Structures

$$X_n > X_m. \quad (1)$$

If there does not exist a statement S which may have a side effect on v between B_m and B_n , then X_m reaches the top of block B_n in which case

$$X_n = X_m. \quad (2)$$

Combining (1) and (2) $\implies X_n \geq X_m$.

Therefore, the version number of a variable between any pair of program points is monotonic. \square

Lemma 6. Suppose v is a variable and B is a node of a program flow graph. Let the $BMR[v, B] = (X_1, X_2, X_3)$. If $X_3 = X_2$ at the end of B, then v is invariant in B.

Proof. $X_3 > X_2 \implies$ B contains a statement which changes the value of v.

$X_3 = X_2 \implies$ value of v is the same at every point in B.

Therefore, v is invariant in B if $X_3 = X_2$ at the end of B. \square

Lemma 7. The BMR of any constant is (0, 0, 0).

Proof. Obvious. \square

Definition 20. Let p be a statement point in some block B of a program flow graph. Suppose the variable v is a source operand of a statement at the point p. The version history of v at the point p is the greatest version number of v reaching the point p.

Lemma 8. Let B_1 and B_2 be two disjoint nodes of a program flow graph such that B_1 and B_2 are in the same program region. Let v be a program variable referenced in B_1 and B_2 at the points P_1 and P_2 , respectively. Suppose further that the version history of v at P_1 is X_1 and the version history

of v at P_2 is X_2 . If $X_1 = X_2$, then the instances of v reaching P_1 in B_1 and the instances of v reaching P_2 in B_2 are the same.

Proof. Since B_1 and B_2 are disjoint, there is no forward path from either B_1 to B_2 or from B_2 to B_1 . Therefore, no value of v computed in either B_1 or B_2 can reach the other through a forward path. Suppose $X_1 = X_2$. Then the value of v used at P_1 is computed outside B_1 and the value of v used at P_2 is defined outside B_2 . There must exist some flow graph node F such that (1) there is a forward path from F to B_1 ; (2) there is a forward path from F to B_2 ; and (3) F is the least common pre-dominator of B_1 and B_2 . F exists since a program flow graph is both connected and rooted. The version number of v does not change between the end of F and P_1 (monotonicity of version number). Similarly the version number of v does not change between the end of F and P_2 . Hence, the value of v at the end of F equals the value of v at both P_1 and P_2 . Therefore, $X_1 = X_2$ implies the value of v at the disjoint points P_1 and P_2 are the same. \square

Lemma 9. Let $\{B_1, \dots, B_k; k \geq 2\}$ be a set of disjoint nodes of a program flow graph. Suppose J is a common post-dominator of $\{B_1, \dots, B_k\}$ and suppose further that $\{B_1, \dots, B_k\}$ cover every path to J . Let v be a program variable whose block exit version number at each B_i ($1 \leq i \leq k$) is Y_i . If the version number of v at the top of J is an element of the set (Y_1, \dots, Y_k) , then the value of v

is invariant between the end of each B_i and J .

Proof. Suppose the version number of v at the top of node J is Z . Then $Z \geq Y_i$; $1 \leq i \leq k$ (monotonicity of version number). Since the nodes are processed according to a topological order, node J is processed after B_1, \dots, B_k have been processed. Moreover, any node forward reachable from a B_i but precedes node J is processed before node J . If $Z = Y_i$ (for some i), then by lemma 6, v is invariant in every node between B_i ($1 \leq i \leq k$) and node J . Therefore, if the version number of v at the top of J is a member of $\{Y_1, \dots, Y_k\}$, then v does not change in value between the end of each node B_i and node J . \square

Lemma 10. Suppose $R[H]$ is a loop region of a program flow graph with header node, H and whose looping nodes are B_1, \dots, B_k ((B_i, H) is a backedge). Let v be a variable and let X_h be the version number of v at the top of H . Suppose Z_1, \dots, Z_k are the version numbers of v at the end of B_1, \dots, B_k , respectively. The variable v is invariant in $R[H]$ if $X_h = Z_1 = \dots = Z_k$.

Proof. Version number of v is incremented in L if there is a statement which may alter the value of v in L . If $X_h = Z_1 = \dots = Z_k$, then v is not assigned any value in L . Therefore, v is invariant in L if $X_h = Z_1 = \dots = Z_k$. \square

Theorem 1. For a structured program flow graph, code optimization by regions does not require the computation of a fixed point for the set of reaching definitions in a program.

Proof. In a region relative code optimization procedure, only the redundancies within a region are removed when that region is processed. By the definition of a structured program flow graph in Chapter I, either a control structure is completely nested within another control structure or it is distinct. Suppose R is a cyclic region of a structured program flow graph. We consider two possible cases: (1) R does not have an inner loop; or (2) R has an inner loop.

Without loss of generality, suppose R corresponds to the high-level control structure

```
while (C) do S; end (1)
```

The statement sequence

```
if (C) then
  S; (2)
endif
while (C) do S; end
```

is equivalent to (1). In (2), the first iteration of (1) is peeled off.

Suppose R does not have any inner loops, then the if statement in (2) does not contain any loops. Let C_{if} and C_{while} represent the conditional expression C in the if and while statements of (2), respectively. Similarly, let S_{if} and S_{while} denote the S in the if and while statements of (2), respectively. Suppose $CSE(C)$ and $CSE(S)$ are the common subexpressions in the conditional expression C and the statement sequence S , respectively. Then

$$\begin{aligned} \text{CSE}(C_{\text{if}}) &= \text{CSE}(C_{\text{while}}) \\ \text{CSE}(S_{\text{if}}) &= \text{CSE}(S_{\text{while}}) \end{aligned} \quad (3)$$

(3) implies that intraloop common subexpressions is not affected by both repeated execution of a loop and by definitions reaching a loop from outside that loop. Therefore, only definitions generated within a loop influence the detection of the common subexpressions contained in that loop.

If R is a nested loop, then there can be two types of common subexpressions in an inner loop: intraloop (within an inner loop) and interloop (between an inner loop and an outer loop). An expression, E , located in an inner loop of a nested loop is redundant with respect to computations of an outer loop, if the value of E is invariant in that inner loop. Determining whether a loop statement is invariant is accomplished by checking the definitions in that loop. Therefore, the elimination of intraregion redundancies of a structured program flow graph can be done without propagating reaching definitions through loop backedges. \square

Summary

This chapter introduced the concept of version numbers to simulate variable reaching definitions. The version numbers of a variable is monotonic between program points when the nodes of a program flow graph are processed according to a total ordering.

CHAPTER V

INTRAPROCEDURAL ALIAS ANALYSIS

Introduction

Redundant statement detection depends on the accurate knowledge of potential definition points of program variables. If every variable is assigned value through direct assignment statements and read statements, then variable definition analysis is straight forward. However, some programming languages contain constructs that create memory aliases (that is two or more names referring to the same location). Memory aliasing can inhibit some optimizations when alias analysis is not included in a global code optimizing compiler.

There are two levels of alias analysis commonly called intraprocedural and interprocedural alias analysis. Intraprocedural alias analysis gathers memory aliasing relationship within a single procedure, while interprocedural alias analysis solves the aliasing problem for a collection of procedures making up a program. This section describes a method for handling pointer variables in an operand dependence graph program representation. The technique presented is suitable for a single procedure only.

The intraprocedural alias analysis presented is based on the following assumptions.

1. The source language does not permit label variables and memory overlap;
2. The procedure being analyzed does not call another procedure;
3. The procedure being analyzed does not have a procedure parameter in its formal parameter list.

The presentation of pointer analysis is based on the C language.

Pointer Aliasing in C

The purpose of pointer alias analysis is to determine the subset of variables which may be affected by indirect assignment through a pointer and indirect reference through pointer dereferencing. To correctly determine the aliases in a C program, the indirection level (the number of *'s pre-pended to a variable in a declaration statement) of a variable must be considered. In the C language, a pointer at indirection level L can be used to access data objects at indirection level $(L - 1), \dots, 0$. Pointer dereferencing is specified by pre-pending a number of *'s to a pointer in an expression.

Suppose B is a node of a program flow graph. Let $\text{IND-ASSIGN}[B]$ represent sequence of variables whose values may be modified in B through a pointer. Let $\text{ALIAS-IN}[B]$ and $\text{ALIAS-OUT}[B]$ contain the set of possible aliases at the top

of node B and at the bottom of node B, respectively. The top of node B is the point before the first statement in node B and the bottom of node B is the point following the last statement of node B.

For each node B of a flow graph, the statements processed to generate elements of IND-ASSIGN[B], ALIAS-IN[B], and ALIAS-OUT[B] are statements of the following forms:

P1. $p = \&q$

P2. $p = q$

P3. $p = (*)^k q$

P4. $(*)^k p = \&q$

P5. $(*)^k p = (*)^k q$

In each of the statement forms, p is a pointer variable and q is either a pointer or an ordinary variable depending on the context.

An element of ALIAS-IN[B] or ALIAS-OUT[B] is a triple $(p, v, v.IL)$. An alias triple $(p, v, v.IL)$ describes the fact that the pointer p is an alias for the variable v declared with v.IL levels of indirection. If v is not a pointer variable, then v.IL is zero.

The representation for an element of IND-ASSIGN[B] is a triple (SID, p, v) , where SID is the identity of the intermediate code statement with an indirect assignment to the variable v; p is the pointer through which an indirect assignment is made; and v is a variable which may be affected by the indirect assignment through the pointer p.

Figure 10 specifies the operations performed for each of the statement forms P1, ..., P5 while processing basic block statements.

Procedure block-alias(B)

Parameter.

B: A basic block.

For each statement S in B do

If S is of the form $p = \&q$, then begin

(1) Delete from ALIAS-OUT[B] all triples whose first component is P.

(2) For each triple of the form $(q, x, x.IL)$ such that $x.IL > 0$

add $(p, x, x.IL)$ to ALIAS-OUT[B].

(3) add $(p, q, q.IL)$ to ALIAS-OUT[B].

(4) If p is a pointer to a structure and q is a structure variable, then for each structure member M of q do add $(p, SM, M.IL)$ to ALIAS-OUT[B];
/* SM is synonym for q.M */

end

Else if S is of the form $p = q$, then begin

/* p and q are pointer variables */

(1) Delete from ALIAS-OUT[B] all triples whose first component is p.

(2) For each triple of the form $(q, x, x.IL)$ add $(p, x, x.IL)$ to ALIAS-OUT[B].

end

Else if S is of the form $p = (*)^k q$, then begin

/* p and q are pointer variables */

(1) Delete from ALIAS-OUT[B] all triples of the form $(p, x, x.IL)$.

Let p.IL denote the indirection level of p.

(2) For each triple of the form $(q, x, x.IL)$ such that $x.IL < p.IL$

add $(p, x, x.IL)$ to ALIAS-OUT[B];

(3) p is a pointer to a structure, then for each triple of the form $(p, x, x.IL)$

add $(px, x, x.IL)$ to ALIAS-OUT[B];

/* px is a pointer created to replace $p \rightarrow x$ */

end


```

Else if S is of the form  $(*)^k p = \&q$ , then begin
  (1) Let  $L := p.IL - k$ ;
  (2) for each triple of the form  $(p, x, x.IL)$ 
      such that  $x.IL > L$  do
    delete  $(x, y, y.IL)$  from ALIAS-OUT[B];
    add  $(SID, p, x)$  to IND-ASSIGN[B];
    add  $(x, q, q.IL)$  to ALIAS-OUT[B];
    if  $q.IL > 0$ , then begin
      for each triple of the form  $(q, v, v.IL)$ 
        in ALIAS-OUT[B] do
          add  $(x, v, v.IL)$  to ALIAS-OUT[B];
          add  $(p, v, v.IL)$  to ALIAS-OUT[B];
        endo
      end
    else if q is a structured variable, then begin
      if x is a pointer to structure, then
        for each member M of q do
          add  $(x, SM, SM.IL)$  to ALIAS-OUT[B];
          /* SM is synonym for q.M */
        endo
      end
    endif
  endo
  add  $(p, q, q.IL)$  to ALIAS-OUT[B];
end

Else if S is of the form  $(*)^k p = (*)^j q$ , then begin
  (1) Let  $L := p.IL - k$ 
       $R := q.IL - j$ ;
  (2) for each triple of the form  $(p, x, x.IL)$ 
      such that  $x.IL = L$  do
    for each triple of the form  $(q, y, y.IL)$ 
      such that  $y.IL = R$  do
      add  $(SID, p, x)$  to IND-ASSIGN[B];
      if  $x.IL > 0$ , then begin
        delete every triple of the form
           $(x, z, z.IL)$  from ALIAS-OUT[B];
        add  $(x, y, y.IL)$  to ALIAS-OUT[B];
        add  $(p, y, y.IL)$  to ALIAS-OUT[B];
        if  $y.IL > 0$ , then
          for each triple of the form  $(y, u, u.IL)$ 
            in ALIAS-OUT[B] do
              add  $(x, u, u.IL)$  to ALIAS-OUT[B];
              add  $(p, u, u.IL)$  to ALIAS-OUT[B];
            endfor
          end
        endfor
      end
    endfor
  endfor
end
end block-alias.

```

Figure 10. Alias Processing in a Basic Block

Pointer to Structure Transformation

Let T be a structured type whose members are M_1, \dots, M_n . Suppose further that no M_i ($i = 1, \dots, n$) is a structured type. Suppose p is a pointer to a structure of type T . Then the expression $p \rightarrow M_i$ (for some i) selects field M_i of structure T . Pointer to structure transformation converts a program with expressions of the form $p \rightarrow M$ to an equivalent program without expressions of the form $p \rightarrow M$.

Pointer to structure transformation involves two steps. The first step creates an equivalent pointer variable for each structure member for which there exists a variable of type pointer to some structure. In the second step of the transformation, statements and expressions are inserted to replace expressions of the form $p \rightarrow M$ with the unique pointer variable created for the structure member M . The pointer to structure transformation procedure steps are:

1. for each variable, p declared as pointer to some structure of type T , generate a sequence of unique names; one name for each member of structure of type T .
2. Suppose ps_1, \dots, ps_n are the names generated in step 1. Let the members of structure T be M_1, \dots, M_n .
For $i = 1, \dots, n$ do
 Declare ps_i a pointer to type of M_i .
3. For each statement of the form $p = \&v$ for some variable v of type T do

Insert the statements

$ps_i = \&vM_i$ ($i = 1, \dots, n$) below $p = \&v$,

(where vM_i is the synonym created for the member $v.M_i$ in a structure member renaming transformation).

4. For each expression of the form $p = \text{expr}$ ($\text{expr} \neq \&v$, where v is a structured variable) do

Insert the statements

$ps_i = \text{expr} + \text{disp}[M_i]$ ($i = 1, \dots, n$) below $p = \text{expr}$,

(where $\text{disp}[M_i]$ is the displacement of the member M_i within structure T).

5. Replace each expression of the form $p \rightarrow M_i$ with $*ps_i$, (where ps_i is the simpler pointer variable created for field M_i of structure T).

6. If p is a formal parameter of a procedure, then insert the statements

$ps_i = p + \text{disp}[M_i]$ ($i = 1, \dots, n$) before the first

executable statement of a procedure's statement sequence.

After applying pointer to structure transformation to a procedure, every pointer object is either a pointer to a scalar or a pointer to an array. A pointer to an array object is assumed to point to every array element. The pointer to structure transformation will slightly increase the number of statements in an intermediate code.

Alias Analysis Procedure

Alias analysis consists of three steps, the first of which is a pointer to structure transformation. The other two steps are node listing generation and the actual computation of alias information for each node.

A node listing for alias analysis is a sequence $NL = (B_1, \dots, B_t)$ of nodes of a program flow graph such that

1. the nodes in a control structure form a subsequence of NL ;
2. a subsequence of nodes for a control structure are in linear order;
3. if S is a subsequence of nodes constituting a program loop, then (S, S) is a subsequence of NL ;
4. if B_1 and B_2 are any two nodes of a flow graph such that B_1 predominates B_2 , then B_1 precedes B_2 in the first subsequence of NL with B_1 and B_2 .

A node listing which satisfies conditions (1) - (4) has a maximum length of $O((d + 1)|N|)$, where d is the maximum depth of a loop and $|N|$ is the number nodes of a program flow graph. The factor $|N|$ for the size complexity of a node listing is based on a worst case assumption that the number of nodes in a loop is $O(|N|)$. The factor $(d + 1)$ is derived from property 3 of a node listing. The sequence of flow graph nodes for a loop region is duplicated in a node listing to ensure that the alias information reaching the

top of a program loop from points outside a loop and from points within that loop as a result of repeated execution of loop code are included in loop alias computation. A node listing for iterative data flow analysis of a reducible flow graph requires a node listing of length $(d + 2)|N|$ to converge [16]. Therefore, a node listing which satisfies constraints (1) - (4) saves at least one iteration. Figure 11 is a procedure for generating node listing.

Algorithm 3: Node Listing Generator Algorithm

Input.

NODE: a linearly ordered set of nodes of a program flow graph with loop region information.

Output.

The sequence NL of flow graph nodes satisfying of the characteristics of a node listing.

Method.

Append flow graph nodes to NL in linear order. If a loop header node is encountered then append the sequence (S,S) to NL, where S is a sequence of nodes for the body of that loop.

Procedure Node-list()

```

NL := ∅;          /* a global variable for node listing */
x := 1;          /* linear order number (LON) */
while x < |N| do
  NL := NL U NODE[x]; /* NODE[x] is node whose LON = x */
  if NODE[x] is a loop header, then
    x := Region-list(x);
  else x := x + 1;
endwhile
end Node-list.

```

(Algorithm 3 continued from previous page)

```

Procedure Region-list(x: loop header)

  y := LON of end of loop marker node of loop whose head is
      x;
  n := x + 1;
  while n < y do
    NL := NL U NODE[n];
    if NODE[n] is a loop header, then
      n = Region-list(n); /* process nested loop */
    else n := n + 1;
  endwhile
  for n := x to y do /* duplicate loop subsequence */
    NL := NL U NODE[n];
  endfor
  NL := NL U NODE[y]; /* append end of loop marker node */
  return(y + 1);
end Region-list.

```

Figure 11. Node Listing Generator Algorithm

The algorithm for computing alias information of each node is specified in Figure 12. Since the length of a node listing is at most $(d + 1)|N|$, the alias computation algorithm has $O((d + 1)|N|*L + t)$ complexity, where t is the time required for pointer to structure transformation, and L is the number of statements in a program. The t term can be eliminated if pointer to structure transformation is done during parsing. Aliasing information obtained from Figure 12 is a transitive closure of the alias relation in a program.

Algorithm 4: Alias Analysis Algorithm.

Input.

A program flow graph, $G = (N, A, r)$;
 NL: a node listing.

Output.

ALIAS-IN[B], ALIAS-OUT[B], and IND-ASSIGN[B] for each node B of a program flow graph.

Method.

First initialize ALIAS-IN[B], ALIAS-OUT[B], and IND-ASSIGN[B] of each node, B of G to \emptyset . Then sequentially process the nodes in NL. For each element, B of NL, examine statements of B in sequential execution order. If a statement of B is one of the pointer forms P1-P5, perform the operations specified in Figure 10 for that statement form. Terminate alias computation when every element in NL has been processed.

```

For each block B of a flow graph do
  ALIAS-IN[B] := ALIAS-OUT[B] :=  $\emptyset$ ;
  IND-ASSIGN[B] :=  $\emptyset$ ;
endfor
For x := 1 to |NL| do /* NL is a node listing */
  B := NL[x]; /* xth element of node list */
  ALIAS-IN[B] = U ALIAS-OUT[C]
                C  $\in$  pred[B]
  ALIAS-OUT[B] := ALIAS-IN[B]
  IND-ASSIGN[B] :=  $\emptyset$ ;
  Apply Figure 10 on block B;
endfor

```

Figure 12. Alias Computation Algorithm

An Example

The C program in Figure 13 will serve as an example to illustrate the alias analysis technique developed in this chapter. Because the program contains the structured variable `cord`, structured variable renaming transformation must be applied first. Suppose the (member, synonym) pairs created for variable `cord` is (`x_axis`, `cord_x_axis`), (`y_axis`, `cord_y_axis`), and (`next`, `cord_next`). Structure variable

renaming transformation replaces the expression `cord.x_axis` with `cord_x_axis` and the expression `cord.y-axis` with `cord-y-axis` in statement `s8`.

Next, the pointer to structure transformation is performed on the program. Let `px_axis`, `py_axis`, and `pnext` be the sequence of pointers created to replace uses of the pointer `p`. Suppose `hdx_axis`, `hdy_axis`, and `hdnext` are the replacement pointers for the pointer variable `hd`. After the two structure related transformations, the original C program is transformed to Figure 14. Since the program is a single basic block, the node list is a single node.

Finally, aliasing information is derived from the program. Figure 15 show the contents of `ALIAS-OUT[B]` and `IND-ASSIGN[B]`. `ALIAS-IN[B]` is empty. The result of alias analysis indicate that `px_axis` and `hdx_axis` are aliases for `cord_x_axis` which is equivalent to `cord.x_axis`. Similarly, `py-axis` and `hdy_axis` are aliases for `cord_y_axis` which shares the same location with `cord.y-axis`. With the alias information known, a code optimizer can discover that statements `s6`, `s7`, and `s8` compute the same expression.


```

main() {
    struct point {
        short x_axis;
        float y_axis;
        struct point *next;
    };
    struct point cord, cord_array[50], *hd, **pp1, *p;
    float sum1, sum2, sum3;
    s1: p = &cord;
    s2: pp1 &p;
    s3: hd = *pp1;
    s4: p -> x_axis = 5;
    s5: p -> y_axis = 25.0;
    s6: sum1 = p ->x_axis + p ->y_axis;
    s7: sum2 = hd ->x_axis + hd ->y_axis;
    s8: sum3 = cord.x_axis + cord.y_axis;
}

```

Figure 13. C Program

```

main(){
    struct point {
        short x_axis;
        float y_axis;
        struct point *next;
    };
    struct point cord, cord_array[50], *hd, **pp1, *p;
    float sum1, sum2, sum3;
    short cord_x_axis, *px_axis, *hdx_axis;
    float cord_y_axis, *py_axis, *hdy_axis;
    struct point *cord_next, **pnext, **hdnext;
    s1: p = &cord;
    s1.1: px_axis = &cord_y_axis;
    s1.2: py_axis = &cord_y_axis;
    s1.3: pnext = &cord_next;
    s2: pp1 = &p;
    s3: hd = *pp1;
    s4: *px_axis = 5;
    s5: *py_axis = 25.0;
    s6: sum1 = *px_axis + *py_axis;
    s7: sum2 = *hdx_axis + *hdy_axis;
    s8: sum3 = cord_x_axis + cord_y_axis;
}

```

Figure 14. C Program After Structure Member and Pointer to Structure Transformations

```

ALIAS-OUT = {
  s1: (p, cord, 0), (p, cord_x_axis, 0),
      (p, cord_y_axis, 0), (p, cord_next, 1),
  s1.1: (px_axis, cord_x_axis, 0),
  s1.2: (py_axis, cord_next, 0),
  s1.3: (pnext, cord_next, 1),
  s2: (ppl, p, 1), (ppl, cord, 0),
      (ppl, cord_x_axis, 0), (ppl, cord_y_axis, 0),
      (ppl, cord_next, 1),
  s3: (hd, cord, 0), (hd, cord_x_axis, 0),
      (hd, cord_y_axis, 0), (hd, cord_next, 1),
      (hdx_axis, cord_x_axis, 0),
      (hdy_axis, cord_y_axis, 0),
      (hdnext, cord_next, 1)
}

IND-ASSIGN = {
  (s4, px_axis, cord_x_axis), (s5, py_axis, cord_y_axis)
}

```

Figure 15. Contents of ALIAS-OUT and IND-ASSIGN
After Alias Analysis

Summary

An alias analysis method which handles both pointers at many levels of indirection and structured types has been described. A pointer to structure transformation is applied to convert pointers to structures to pointers to simpler types (scalars and arrays). A node listing derived from a total ordering of the nodes of program flow graph is used to propagate alias information.

CHAPTER VI

OPERAND DEPENDENCE GRAPH

Introduction

An operand dependence graph (ODG) is a directed graph which exposes the subset of program statements to be subjected to code redundancy analysis. The feasible optimizations detectable with an operand dependence graph include common subexpression elimination, code hoisting, code sinking, loop invariant motion, and strength reduction of loop induction variables. With the exception of alias analysis, the construction of an ODG does not require prior computation of other flow analysis information.

What is an Operand Dependence Graph?

An operand dependence graph is a directed graph representation of the statements in a program region of a structured program flow graph. There is one control node for each distinct operand as well as one control node for each distinct intermediate code statement in an ODG. Each distinct instance of a distinct statement and each distinct instance of a distinct operand is represented with an instance node. The set of statement instance nodes for a

distinct intermediate code statement are linked together with the control node for that statement. Similarly, the set of instances of a distinct operand and the control node for that operand are linked together. The version histories of operands are used to distinguish between operand and statement instances.

An operand dependence graph node describing an instance of a distinct statement is decorated with the flow graph nodes (having a copy of the statement instance represented at a node) and an instance signature.

Definition 21. Suppose S is a distinct intermediate code statement. Let v_1, \dots, v_n denote the distinct variable source operands in statement S in lexical order. Let p be some program point with an instance of statement S . For each source variable operand v_i , let h_i represent the version history of v_i at program point p . The instance signature of the statement S at point p is the n -tuple (h_1, \dots, h_n) .

Two instances of a distinct statement with the same instance signatures are said to be similar. In an ODG, instances of the same statement with different instance signatures are represented with different graph nodes.

The operand dependence graph of one program region is distinct from the operand dependence graph of another program region. Within a program region, graph nodes are connected by two types of edges. The first edge type called instance link connects a control node of a distinct graph object

(statement or operand) and the instance nodes of that object, and the second type of edge called data link connects operation instance nodes and operand instance nodes.

In a formal sense an operand dependence graph representation of a program is a triple $Z = (C, I, E)$, where C is a set of control nodes for the distinct statements and the distinct operands in a program region; I is a sequence of instance nodes of distinct statements and operands; and E is a sequence of edges from the ordered pairs $C \times I$ and $I \times I$.

Work Lists

Code optimization and operand dependence graph construction proceed simultaneously. Some redundancy analysis cannot be performed on some set of statements until the complete control environment surrounding those statements is seen. Such statements are buffered for later analysis. Five buffers are maintained during graph construction to hold program objects (statements and operands) which require further processing. These buffers and the type of information they contain are described below:

CHQ: is a queue of statements which contains statement instances to be analyzed for code hoisting optimization.

CSS: is a stack of statements to be checked for code sinking

optimization.

LIQ: is a queue of loop invariant statements.

BEIQ: is a queue of potential basic loop induction expressions.

LAVQ: is a queue of loop active variables and constants.

An element of CHQ, CSS, LIQ, or BIEQ is of the form (CN, IN), where CN is a statement control node and IN is a statement instance node. A LAVQ entry is an ordered pair of operand control node and operand instance node of constants and variables active (referenced or defined) in a loop. When processing a nested loop, the contents of LAVQ is saved at the beginning of an inner loop and restored after the body of an inner loop has been processed. The restoration of the LAVQ of a parent loop is a union of the saved LAVQ of a parent loop and the LAVQ of an inner loop.

Essential Node Information

There are four node types in an operand dependence graph intermediate program representation; two of which are control nodes of distinct program objects (operations and operands) and the remaining two are instance nodes of distinct program objects. A control node contains a detailed description of either a distinct statement or a distinct operand, while an instance node contains information necessary for detecting potentially redundant code.

The set of basic fields for each node type are described below.

The field names for statement control node are

- OC: Operation class (store, arith, procedure call, string, logical, flow control, etc);
- OP: Operation name;
- VC: Value class for operation computing a value (float, integral, boolean, string, pointer, condition code, etc);
- IN: Current number of distinct instances of a statement in program region;
- ILH: Instance list head;
- SON: Source operands control nodes;
- DON: Dependent operands control nodes;
- DEST: Destination operand control node;
- CSS_FLAG: An array of flags (one flag per loop nesting level) to indicate that an instance of statement has been pushed into the CSS stack;
- CHQ_FLAG: An array of flags to indicate when an instance of statement has been entered into CHQ queue;
- BIEQ_FLAG: An array of flags indicating that an instance of statement is in BIEQ queue.

An operation instance node contains the following fields:

- SCN: Control node of statement instance;
- NIP: Next instance pointer;
- OPDS: Operand instance nodes;
- SIN: Statement instance number;
- DEST: Destination operand instance node;
- IOB: Sequence of basic blocks with a non redundant copy of statement instance;
- SIG: Statement instance signature (vector of operands version histories);

LI_FLAG: Flags to indicate whether statement instance is a loop invariant (one flag per nesting level).

The components of an operand control node include

DNT: Data node type (array, scalar variable, constant, statement label, and procedure name);

VN: Number of different instances of operand;

VC: Operand value class (float, integral, char, boolean, string, etc);

ILH: Head of instances list;

LAVQ_FLAG: Array of loop active variable and constant flags (one flag per loop nesting level).

Lastly, the essential field names of an operand instance node are

NIP: Next instance pointer;

IVN: Operand instance version number;

SUCC: Sequence of data link successor nodes;

LI_FLAG: Flags to indicate whether operand instance is invariant in a loop (one flag per loop nesting level).

Operand Dependence Graph Construction

To construct an operand dependence graph representation of a program region, these general rules must be followed:

1. Process the flow graph nodes in linear order;
2. Process the statements of each flow graph node in sequential execution order;
3. Before processing the statements in the initial node of a program flow graph, initialize the block mutation record of every variable to (0, 0, 0); and at the beginning of a program region create empty instances of the various auxiliary data structures (LAVQ, CSS, CHQ, etc);

4. Non-redundant instances of lexically identical statements in different flow graph nodes with the same statement instance signature are represented on a common statement instance node;
5. If the next flow graph node to be processed is a join node of a conditional control structure, then examine the CHQ queue and the CSS stack for statements. If there exists a statement in either CHQ or CSS, then apply the necessary code motion detection algorithm on the instances of a code motion candidate;
6. If the next flow graph node to be processed is an end of loop marker, then analyze the statements in that loop to recognize loop invariant statements and induction variables;
7. A statement instance is represented by drawing directed edges (one edge per distinct source operand) from operand nodes to the operation instance node for a statement. The operand instance node from which a directed edge is drawn to link an operation instance node is the version history of that operand with respect to that statement instance. If an operation produces a result, then a directed edge leaves an operation instance node and enters the operand instance node representing the result operand.
8. If there exists a previous instance of the next statement to be added to an ODG, then analyze the forward reachability relation between the previous statement

- instances and the new statement to determine the type of code redundancy analysis to apply on the duplicate statement instances.
9. After adding a statement which defines a program variable to an ODG, assign a new version number to the variable assigned to, add a version record into that variable's version creation point table, and update the block mutation record for that variable.
 10. If a variable is a call-by-reference parameter of a procedure call, then assign a new version version number to that variable following the call statement and make the necessary entries into the version creation point table and block mutation record of that variable.
 11. If a statement is an indirect assignment through a pointer, then after processing that statement, assign new version numbers to every variable that could be affected by the indirect assignment. Enter a new version record into the version creation point table of each affected variable and update the block mutation record of that variable.

Detection of Partial Redundancies

One distinguishing feature of an operand dependence graph program representation is that the detection of candidate statements for common subexpression elimination, code hoisting, or code sinking redundancy analysis does not require separate graph traversals. The concept of partial

redundancy is used to identify lexically identical statements to be subjected to redundancy analysis in a program region.

Definition 22. Suppose S is a distinct program statement and suppose S_p is an instance of statement S at some program point p . Let v_1, \dots, v_n be the sequence of the distinct variable source operands in statement S . Suppose S_p is to be added to an ODG and suppose further that (h_1, \dots, h_n) is the instance signature of the statement instance S_p . The statement instance S_p is partially redundant with respect to previous instances of statement S if for each source variable v_i , there exists a directed edge from version h_i of variable v_i , $1 \leq i \leq n$, to a previous instance node of statement S .

There is one major difference between partial redundancy as defined here and as defined by Morel and Renvoise [27]. The difference is that partial redundancy with respect to an ODG includes statements on disjoint execution paths as well as statements on a common execution path, while Morel and Renvoise restrict partial redundancy to statements on a common execution path. This distinction makes it possible to detect both common subexpression and code motion candidates with the same mechanism in an operand dependence graph. In Figure 16, the instance of $a + b$ in block B3 is partially redundant with respect to the instance in block B2 if block processing order is B1, B2, B3, B4 and $a + b$ in node B4 is partially redundant with respect to the

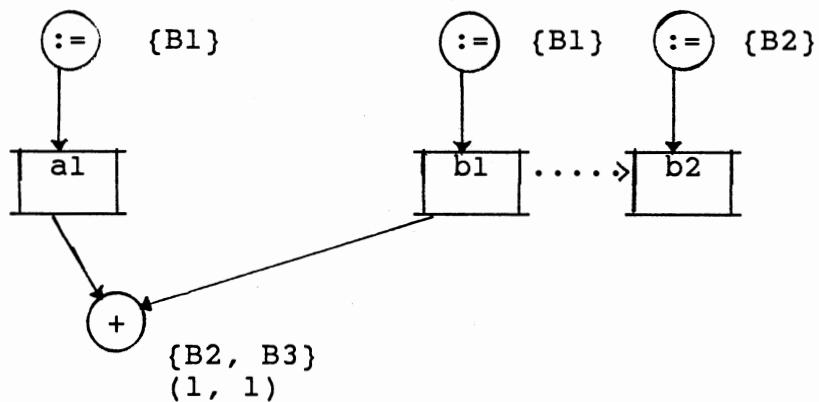
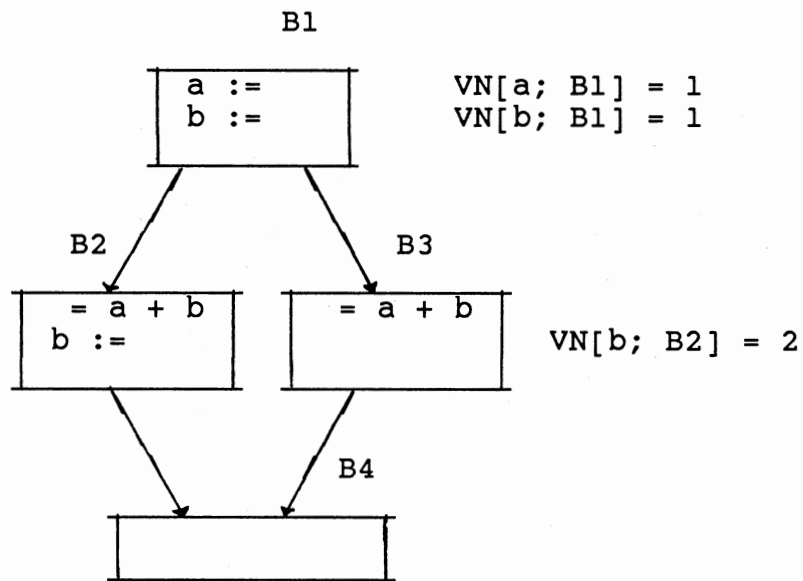


Figure 16. Partial Redundancy Involving Disjoint Statement Instances

- [] Operand Instance Node
- Operation Instance Node
- Data Edge
- ...> Instance Link
- { } Instance Occurrence Blocks
- () Instance Signature

instances of $a + b$ in blocks B2 and B3. Figure 17 is an example of partial redundancy in which the previous instances of a statement and the partially redundant instance do not have the same instance signature.

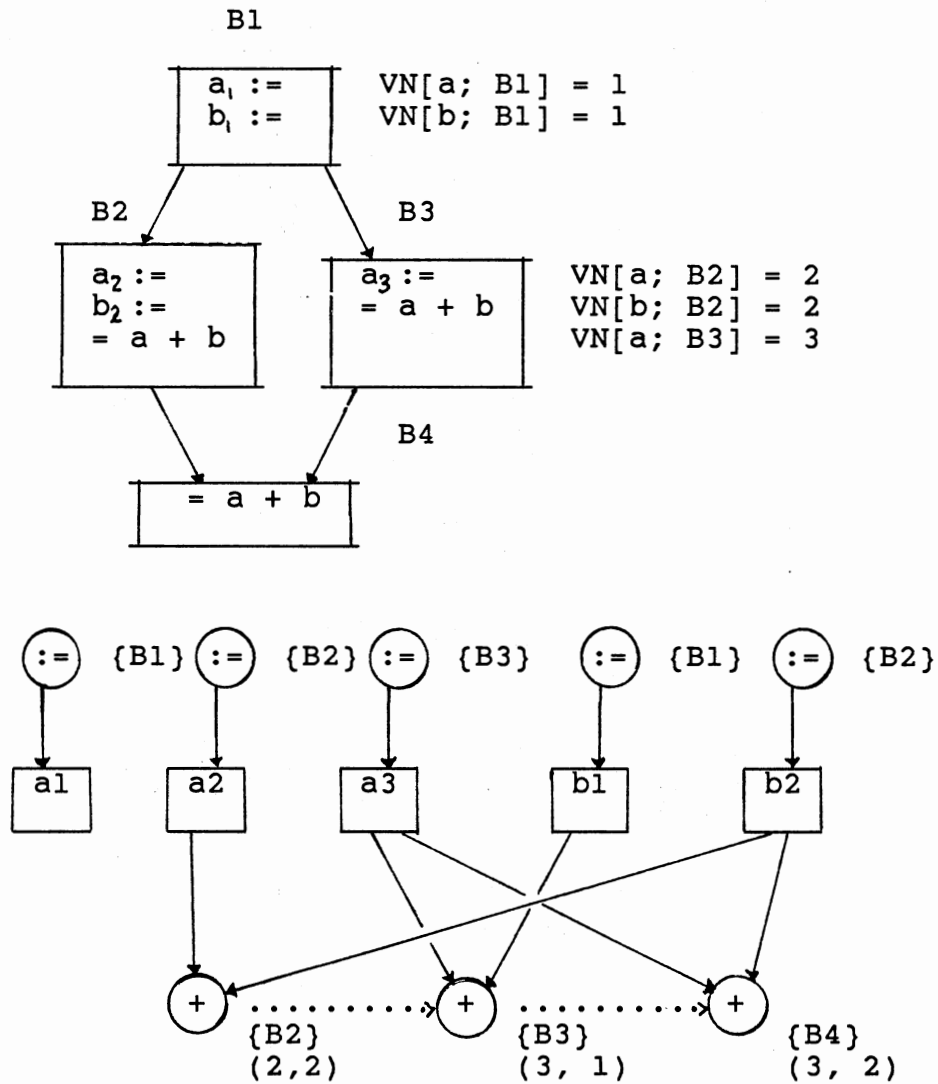


Figure 17. Partial Redundancy Involving Statements on Common Execution Paths

If a partially redundant statement has the same instance signature with a previous instance of the same statement, then that partially redundant statement is either analyzed for common subexpression elimination or for code hoisting optimization. The particular analysis to be performed depends on the forward reachability relation between the basic block containing the most recent previous instance of that statement and the basic block with the partially redundant instance. The most recent previous instance of a statement is the instance which occurs in a basic block closest to the basic block with a partially redundant instance in block processing order. If a partially redundant statement and the most recent previous instance of the same statement are in disjoint blocks, then the statements are candidates for code hoisting analysis, otherwise a partially redundant statement is checked for common subexpression elimination.

When a partially redundant statement does not have identical instance signature with any previous instance of the same statement, then that partially redundant statement is a common subexpression elimination candidate. The notion of partial redundancy does not expose every code sinking candidate. For instance, in Figure 17, the instances of $a + b$ in blocks B2 and B3 are candidates for code sinking optimization but neither is partially redundant with respect to the other. However, the search procedure for lexically identical statements discovers all code sinking candidates.

Before a statement is added to an operand dependence graph, the ODG_NODE field of a distinct statement record in the distinct statement table is checked to determine if a previous instance of that statement exists in the current graph segment. The ODG_NODE field of a statement in the distinct statement table holds the address of a statement's control node in an ODG. Thus, if the ODG_NODE field of a distinct statement contains a null value, then a previous instance of that statement has not been encountered in the program region.

If the contents of the ODG_NODE field of a distinct statement record is not a null value, then the list of previous instances are searched to determine if current statement is partially redundant. The ILH (instances list head) of a distinct statement's control node contains the node address of the first instance of a statement in an ODG. For each previous statement instance node visited, the instance signature of that previous instance is compared with the instance signature of current statement. If there is a match of instance signatures, then the forward reachability relation between the basic block with the most recent previous instance and the basic block containing the current instance is used to decide which redundancy analysis procedure to apply. If there is no match of instance signatures, but each element of the instance signature of the current statement is an element of the instance signature of some previous instance of that statement, then

that current statement instance is a partially redundant common subexpression.

If a current statement is not partially redundant with respect to previous instances of the same statement, then the redundancy detection procedure which can be applied to the statement instances is that for code sinking. To determine if code sinking redundancy analysis is necessary, the instance occurrence block field of previous instances of statement are examined for a previous statement instance which lies on a parallel control flow path relative to current statement instance. Partially redundant common subexpression candidates are subjected to common subexpression redundancy analysis procedure as soon as they are discovered, while code hoisting and code sinking candidates are placed in either a CHQ queue or CSS stack until a conditional control environment join is reached.

Operand Dependence Graph Characteristics

Operand dependence graph program representation provides two ways of searching a dependence graph. The instance links (edges connecting instances of the same object) is used to search distinct program objects (operand or statement) with multiple instances without visiting nonrelated nodes.

The other means of searching an operand dependence graph is by visiting nodes through data links (edges connecting operand nodes and statement instance nodes).

When the only means of visiting statement nodes is by following data links as in [15], the detection of common subexpression, code hoisting, and code sinking candidates incur considerable cost as both single instance and multiple instance statements are examined. Moreover, separate graph traversal must be performed for each optimization problem.

A feature unique to operand dependence graph is that a statement instance node can represent several instances of the same statement from different basic blocks of a program region. The capability to represent several instances of a statement which are not necessarily equivalent in value with one graph node enhances the detection of partial redundancies. In other dependence graphs, equivalent values are the only shared nodes.

Summary

An operand dependence graph is a directed graph representation of the statements belonging to a program region. Graph nodes are connected with two types of edges called instance link and data link. Instance links connect instances of a graph object (statement or operand) and its control node while data links connect statements to their source and destination operands. Instance links provide a fast means of searching lexically identical statements for partial redundancies without examining unrelated graph nodes.

CHAPTER VII

DETECTION OF FEASIBLE OPTIMIZATIONS

Introduction

This chapter addresses the issue of recognizing feasible optimizations while building an operand dependence graph. Each redundancy detection problem can be specified with three types of constraints. These constraints have been classified into information flow (path problem), variable reaching definitions, and variable reference constraints. The techniques for checking these constraints for the various intermediate code optimization problems in an operand dependence graph program representation are presented in the next several sections.

Redundant Statement Elimination

Redundant statement elimination involves the detection and removal of a statement instance for which a previous active instance of that statement exists along every control flow path leading to that statement. A statement instance S_p , at some point p , in a program is redundant if

1. S_p is partially redundant with respect to some previous instances of S ;

2. the set of previous instances of S which renders S_p partially redundant is a path cover for the point p ; and
- 3a. either the set of basic blocks where the previous instances of S are located and the basic block where S_p is located have the same region tag; or
- 3b. the set of basic blocks with previous instances of S are located in an outer loop of a nested loop and S_p is a loop invariant of an inner loop.

Condition (1) is detected at the time a statement instance is added to an operand dependence graph. To check the second condition, the minimal set of the set of flow graph nodes with active previous instances of S reaching the point p along a forward path is formed, then the minimal set is subjected to path cover test with respect to the point p . The third condition is checked by comparing the region tag of the point p and the region tags of previous statement instances which renders S_p partially redundant.

The code fragment below exemplify the necessity of the third (3a and 3b) constraint.

```

while (q < 5) {
    = p + q
    ...
    while (p > 0) {
        ...
        = p + q
        ...
        p =
    }
    ...
}

while (q < 5) {
    = p + q
    ...
    while (x > 0) {
        ...
        = p + q
        ...
    }
    ...
    q =
}

```

In the left code segment, $p + q$ in the inner loop is not redundant with respect to the instance in the outer loop. The situation is different in the second fragment because $p + q$ in the inner loop is a loop invariant. As a result, this second instance of $p + q$ is redundant with respect to the first instance. Because path cover test is likely to involve more operations than comparing region tags, condition 3(a) should be checked before checking the path cover condition. When condition 3(a) does not apply, redundancy analysis of a partially redundant statement should be suspended until the rest of the body of an inner loop has been processed.

The notation $(a + b)_i$ denotes the instance of $a + b$ in flow graph node B_i and b^j means after the assignment to b is executed, the version number of the variable b is j . In Figure 18, $(a + b)_4$ is redundant with respect to the instances $(a + b)_2$ and $(a + b)_3$. Now consider $(a + b)_4$ in Figure 19. By the definition of partial redundancy, $(a + b)_4$ will not be subjected to redundancy test even though in the general sense of partial availability $(a + b)_4$ is partially available along the flow graph edge (B_3, B_4) . The operand dependence graph has this capability to avoid some redundancy checks that are bound to fail. In the case of Figure 20, $(a + b)_4$ is identified as partially redundant, but fails the path cover test because there is no active previous instance of $(a + b)$ along the edge (B_2, B_4) .

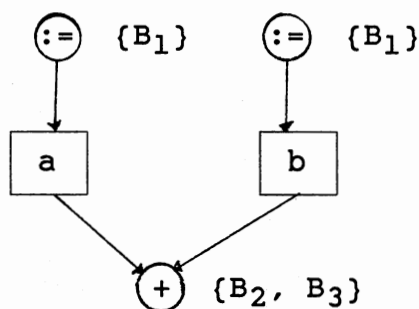
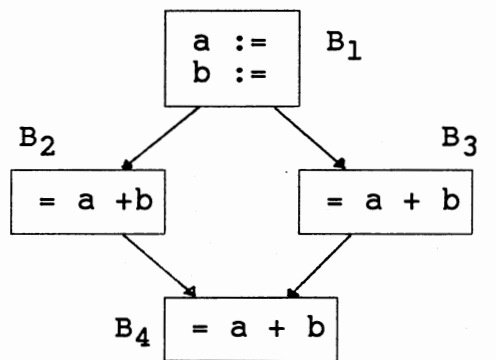


Figure 18. A Redundant Statement

Figure 21 illustrates the type of redundant computation missed by an operand dependence graph. The problem is due to the definition of a program region. Because node B_3 is a loop, the flow graph is split into three segments $\{B_1, B_2\}$, $\{B_3\}$, and $\{B_4\}$. Since redundant statement elimination is restricted to a program region, $(a + b)_4$ cannot be detected as a common subexpression. This shortcoming of the region relative optimization technique will not affect the execution time of the optimized code since most of the useful optimizations are performed in program loops.

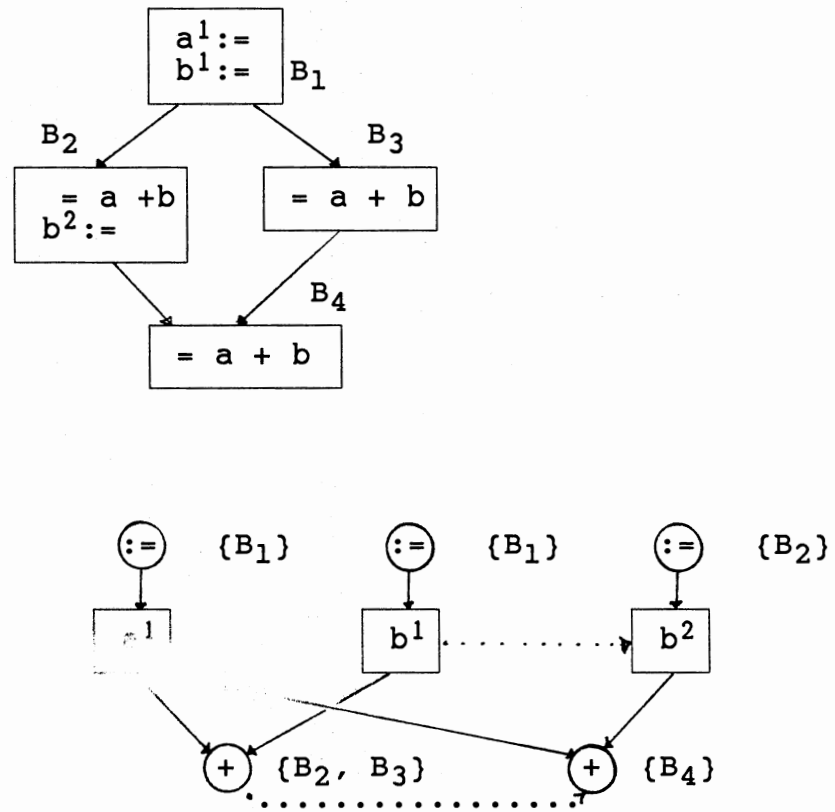


Figure 19. A Partially Redundant Statement
Not Tested For Redundancy

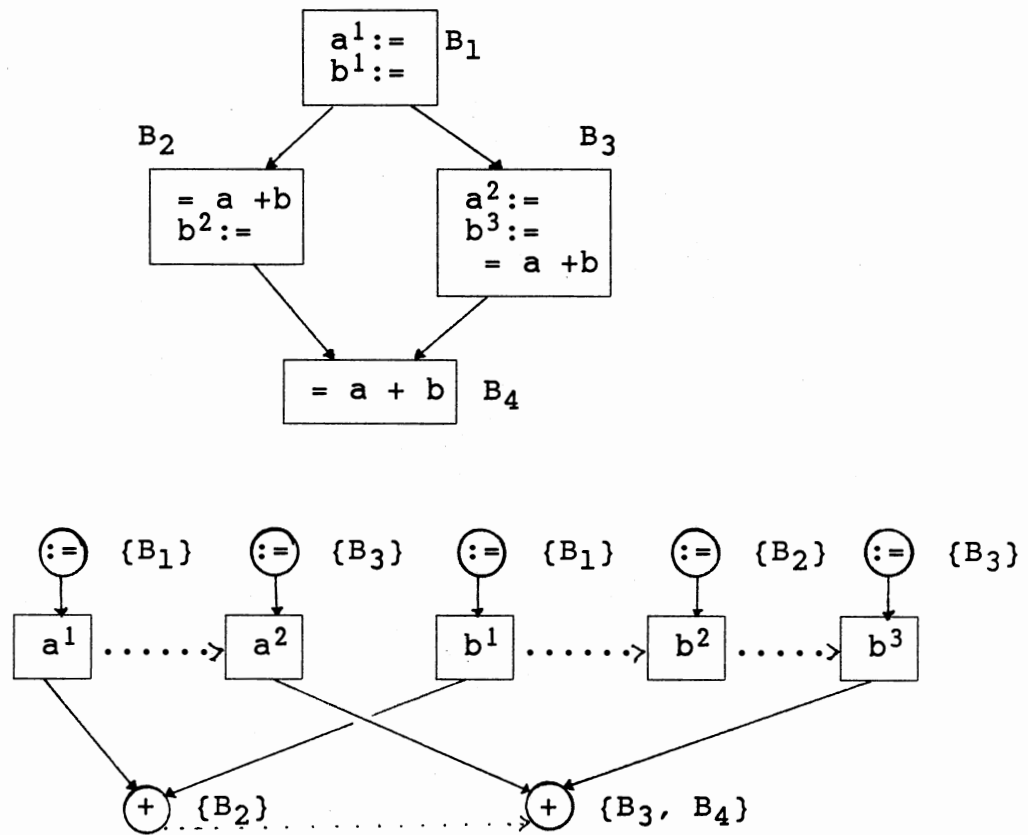


Figure 20. A Partially Redundant Expression Which Fails Full Redundancy Test

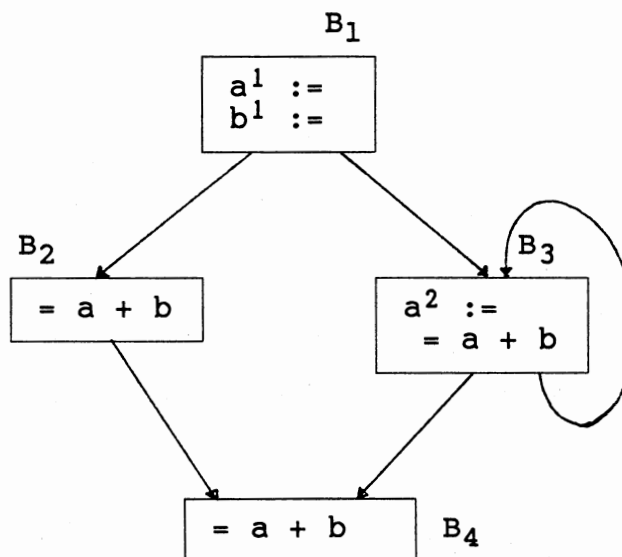


Figure 21. An Inter-region Redundancy

Code Hoisting Optimization

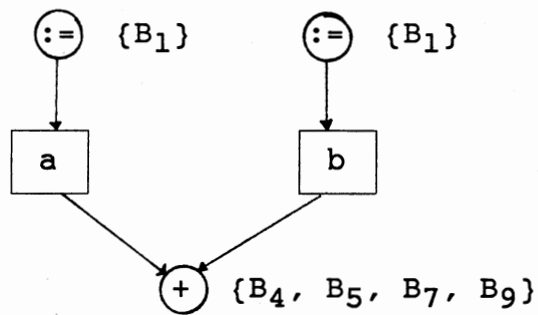
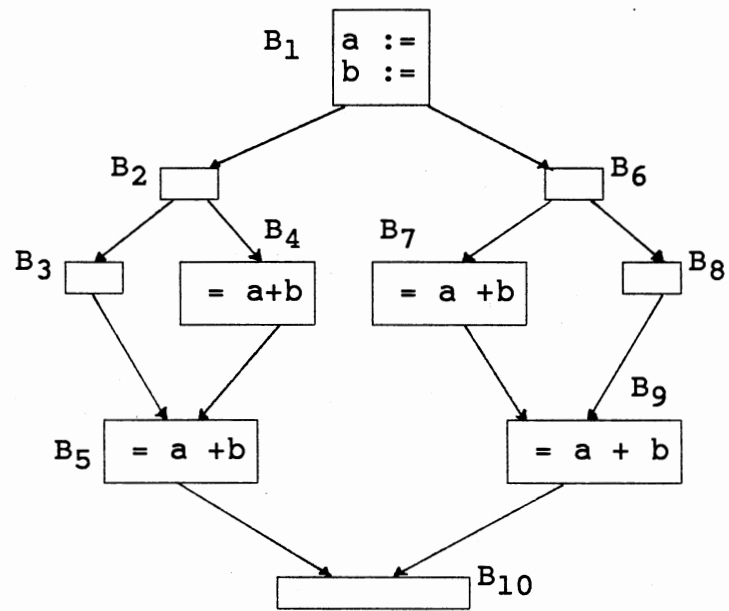
Code hoisting optimization or backward code motion is applied to lexically identical expressions which occur on disjoint control flow paths of a conditional control structure. A set of lexically identical expressions in a conditional control structure with *then* and *else* parts can be moved to the fork of that conditional structure if (1) the expression instances compute the same value; and (2) there exists an instance of that expression on every forward path originating from the fork of a conditional control structure to the join of that conditional structure.

In the context of an operand dependence graph program representation, the two conditions are met when the

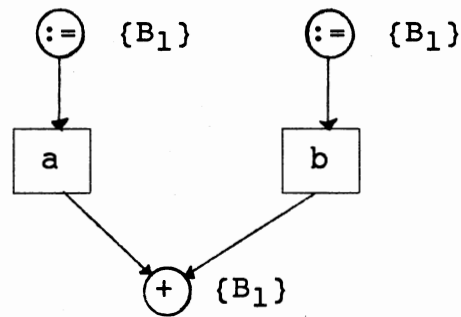
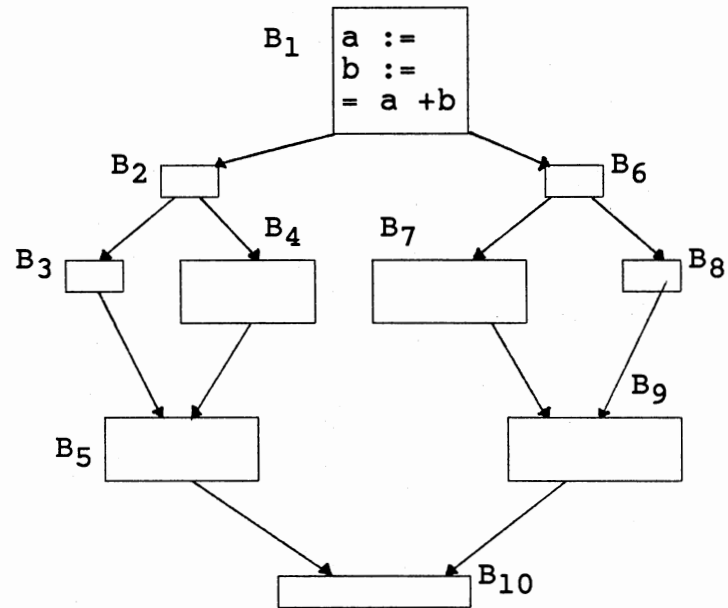
expression instances are represented with the same statement instance node; a subset of the flow graph nodes with a copy of that expression are mutually disjoint; all the mutually disjoint flow graph nodes with a copy of that expression have the same region tag; and the disjoint flow graph nodes with an instance of that expression constitute a control environment path cover for the conditional structure.

Requiring all the flow graph nodes with an instance of a backward motion candidate to have the same region tag guards against moving an expression from an inner loop to an outer loop in a nested loop structure.

Figures 22 and 23 provide examples of permissible and non permissible code hoisting optimization. In Figure 22(a), the instances $(a + b)_5$ and $(a + b)_9$ lie on every path from B_1 (fork of conditional structure) to B_{10} (join of conditional structure); both $(a + b)_5$ and $(a + b)_9$ compute the same value; and $(a + b)_5$ and $(a + b)_9$ are in the same control environment. The copies of $(a + b)$ in the conditional structure can be removed by placing a copy of $(a + b)$ in B_1 . After code hoisting optimization, Figure 22(a) is transformed to Figure 22(b). Although the flow graph nodes with instances of $(a + b)$ in Figure 23 are disjoint and occur on both legs of the fork node B_1 , $(a + b)$ cannot be moved to B_1 because the copies do not belong to same control environments. $(a + b)_2$ is in an outer loop, while $(a + b)_3$ occurs in an inner loop.



22(a) Flow Graph and ODG Before Code Hoisting



(b) Flow Graph and ODG After Code Hoisting

Figure 22. Hoistable Code

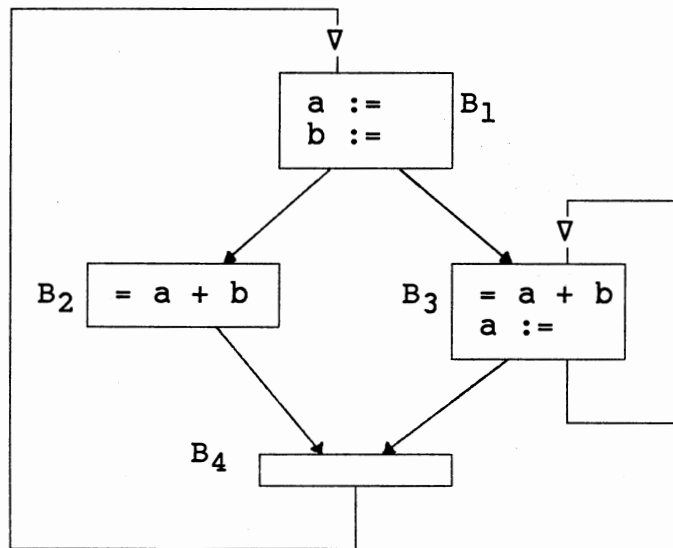


Figure 23. A Non-Hoistable Code

The algorithm for code hoisting is specified in Figure 24. First, post-dominance information of the join of a conditional structure is used to select the instances of a code hoisting candidate in the same control environment as that of a join node. Then the minimal set of the selected instances are computed and subjected to conditional structure path cover analysis.

Suppose the flow graph of Figure 25 is applied to the code hoisting algorithm (Figure 24). Initially $S = \{B_4, B_8, B_9\}$, of flow graph nodes with instances of $a + b$ is computed and tested for conditional environment path cover test with respect to the fork node B_1 . The set S fails the path cover test because there are no copies of $a + b$ along the paths passing through B_2 and B_6 . The set S is then partitioned

into the subsets $C_1 = \{B_9, B_8\}$ and $C_2 = \{B_4\}$ to process the nested conditional structures. The partition C_1 is a conditional control environment cover for the conditional control structure comprising $\{B_7, B_8, B_9\}$. Therefore, the instances of $a + b$ in B_8 and B_9 can be deleted by placing a copy of $a + b$ in node B_7 . Since nodes B_7 and B_4 do not have the same immediate predominator, node B_7 is in its own partition. The second loop of the code hoisting algorithm terminates as each of the remaining partitions has only one element.

Algorithm 5. Code Hoisting Algorithm

Input.

J: Join node of a conditional structure;
 CHQ: list of code hoisting candidates;
 Program flow graph with predominance and post-dominance information;
 Operand dependence graph;
 Intermediate code program.

Output.

Possibly modified operand dependence graph;
 Possibly transformed intermediate code program.

Auxiliary.

S: set of flow graph nodes with copy of code hoisting candidate.

Method.

For each code motion candidate statement x , select the instances of x post-dominated by the join node J . Compute the minimal set of the selected instances and check if the minimal set of instances lie on every path of the parent conditional structure. If the minimal set of instances of x pass the path cover test, then move x to fork of parent structure. Else analyze any substructure for possible code motion.

```

While CHQ is not empty do
  Turn-off CHQ FLAG;
  g = dequeue(CHQ); /* ODG node for statement */
  S = {B | B is a flow graph node with a copy of

```

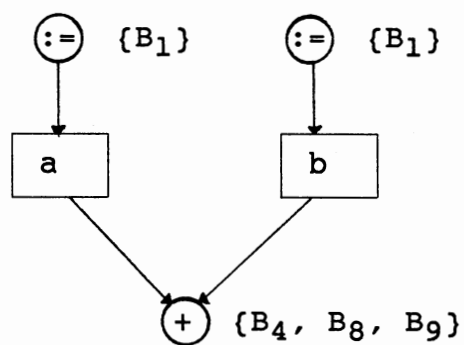
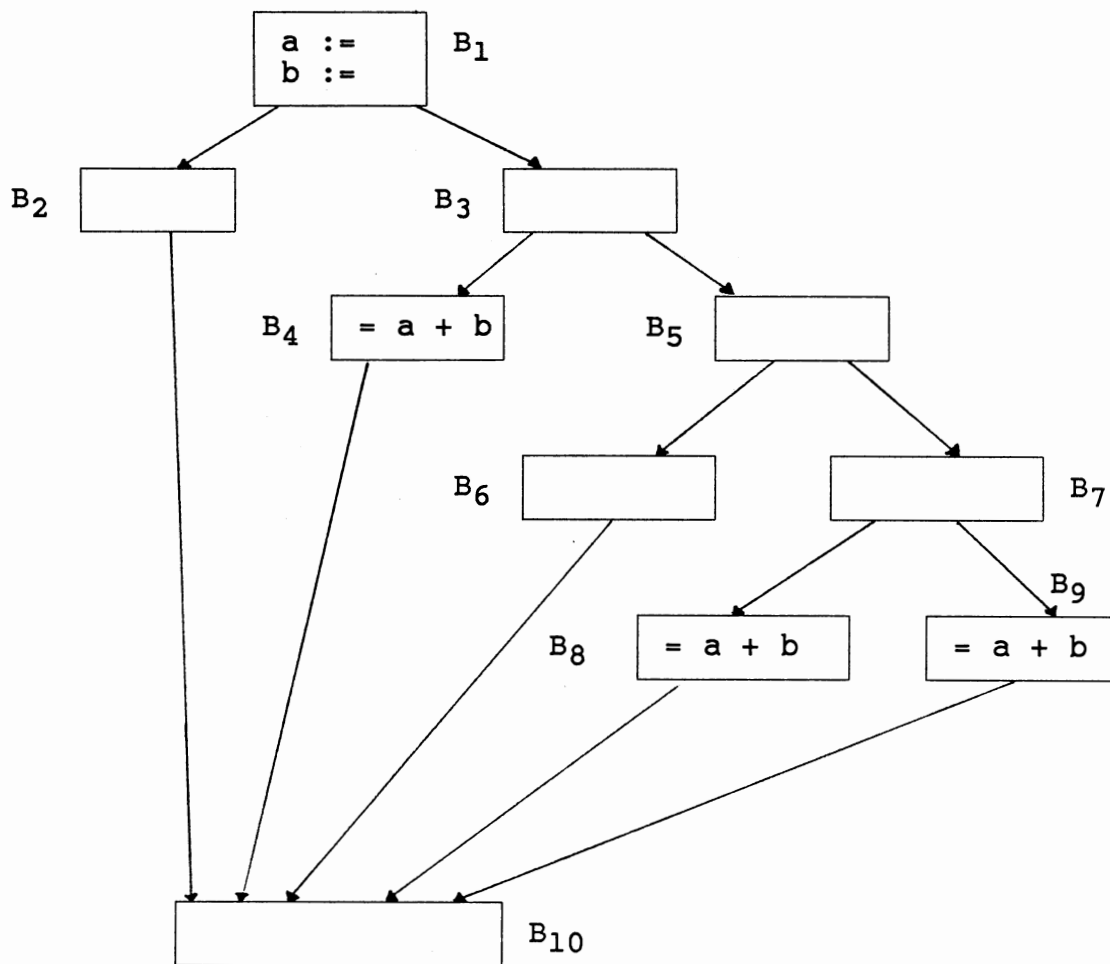
```

        statement at ODG node g; J post-dominates B;
        and region tag of B = region tag of J});
S = minimal set of S;
If S is a singleton, then stop;
If S is a path cover for conditional structure,
then begin
    Let x represent the expression at node g of ODG;
    f := least common predominator of nodes in S;
    Delete each flow graph node n, such that n has a
    copy of x and node f predominates n from the
    instance list of ODG node g and delete statement
    x from flow graph node n;
    Create a copy of x in flow graph node f;
    Insert flow graph node f in the instances list
    of ODG node g;
    End.
Else begin
    Partition S into disjoint subsets  $C_1, \dots, C_z$ ,
    such that each  $C_i$  contain flow graph nodes with
    the same immediate predominator.
    While there exists a partition, P of S such that
     $|P| > 1$  do
        If partition P is a control environment cover,
        then begin
            Let x be the expression at ODG node g;
            f := least common predominator of nodes in
            P;
            Delete each flow graph node n, such that n
            has a copy of x and node f predominates n
            from the instance list of ODG node g and
            delete statement x from flow graph node n;
            Create a copy of x in flow graph node f;
            Insert flow graph node f in the instances
            list of ODG node g;
            Place flow graph node f in a partition;
        Endif.
        Delete partition P from set of partitions;
    Endwhile.
    End.
Endif

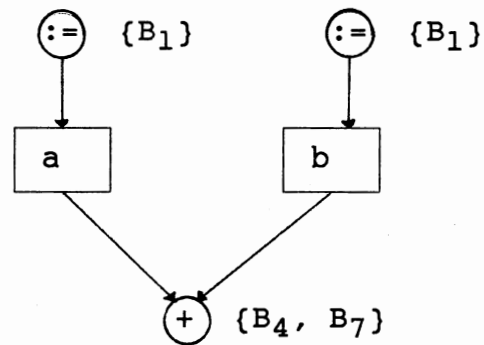
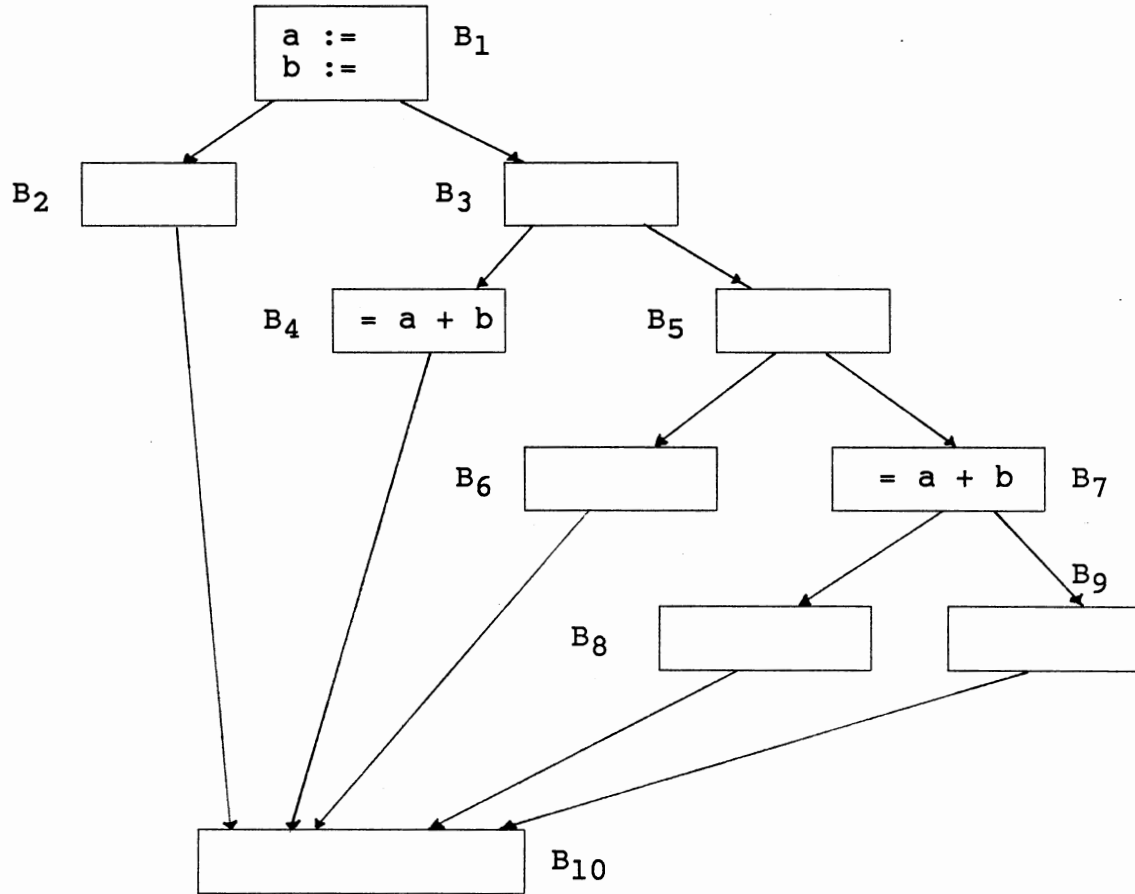
```

End Code-hoisting Algorithm.

Figure 24. Code Hoisting Algorithm



25(a) Control Structure Before Code Hoisting



(b) Control Structure After Code Hoisting

Figure 25. Code Hoisting Candidate in a Nested Conditional Structure

Code Sinking

Code sinking optimization procedure (forward code motion) moves common code in the *then* and *else* branches of a conditional structure to the *join* of that conditional structure. Let *S* be a distinct program statement and let *C* be some conditional control structure whose join node is *J*. Suppose there exists instances of *S* in the *then* and *else* branches of *C*. The instances of *S* in *C* can be removed and replaced with a single instance at the top of the join node *J* if

1. the set of points with instances of *S* in *C* constitute a path cover for *J*;
2. The statements following the last instance of *S* on each forward path to the join node do not have a side effect on any source operands of *S*;
3. Any results produced by the last instances of *S* is not referenced in the conditional structure; and
4. The join node *J* and every node in the conditional structure *C* have the same region tag.

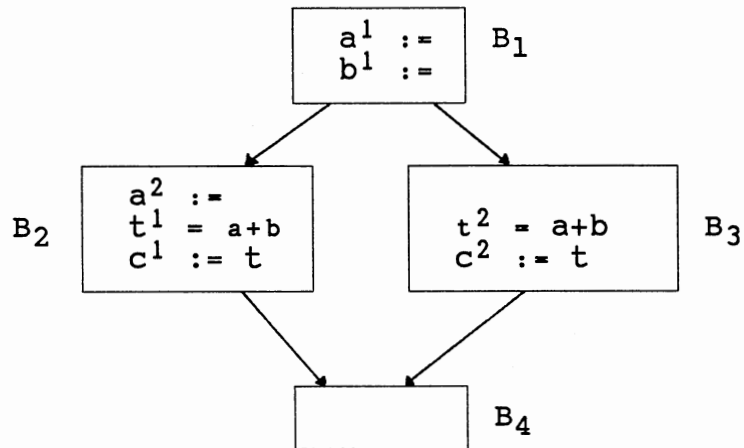
The first two conditions a forward code motion candidate must satisfy are identical to those for global common subexpressions. The third condition prevents moving a statement which may generate the input data used in another statement and the fourth constraint ensures that the statement instances are located in the same control environment. To check the third constraint, the effects of

a procedure call and pointer aliasing must be considered. For a procedure call, the results of a call statement includes the value returned by a procedure and the call-by-reference parameters which may be modified in a called procedure.

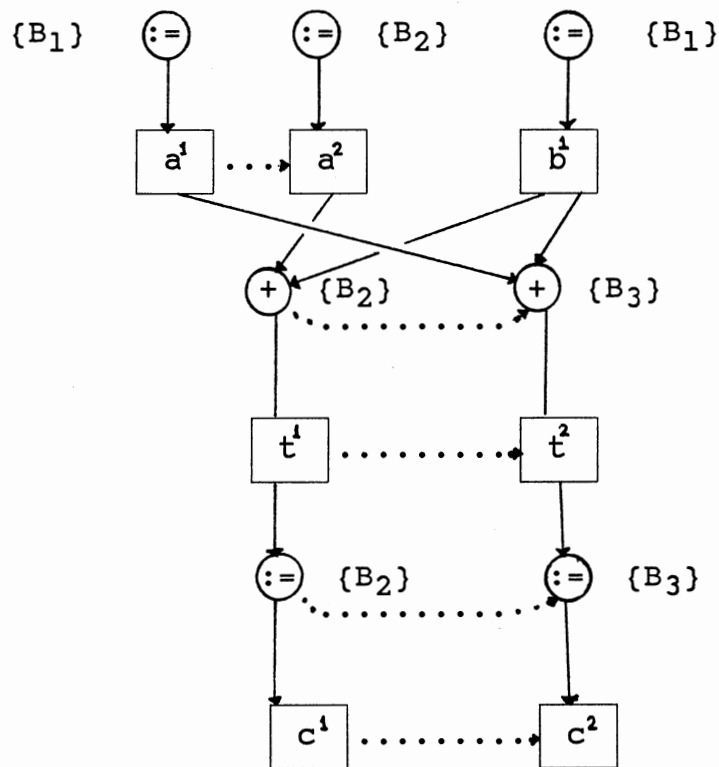
In Figure 26, the statements $t_1 = a + b$ and $c := t_1$ qualify for code sinking optimization if the statements are analyzed in reverse execution order. However, if the statement $t_1 = a + b$ is analyzed first, then only $c := t_1$ can be moved since t_1 is referenced in the second statement. To recognize $a + b$ as a sinkable expression, the statements of the conditional structure must be rescanned after moving the second statement. Rescanning is avoided by using a stack to hold forward code motion candidates. The last-in-first-out property of a stack ensures that code sinking candidates are processed in reverse statement execution order.

Since the first two constraints a code sinking candidate must satisfy are identical to those for global common subexpressions, common subexpression elimination procedure can be used as part of a code sinking procedure. A dummy instance of a code sinking candidate is created and made to appear to originate from the join of a conditional structure. If the dummy instance is fully redundant with respect to the instances in a conditional structure, then conditions (3) and (4) can be checked to complete the test for code sinking. This approach reuses the procedure for

redundant common subexpressions detection.



26(a) Code Sinking Candidates



26(b) Operand Dependence Graph
Before Code Sinking

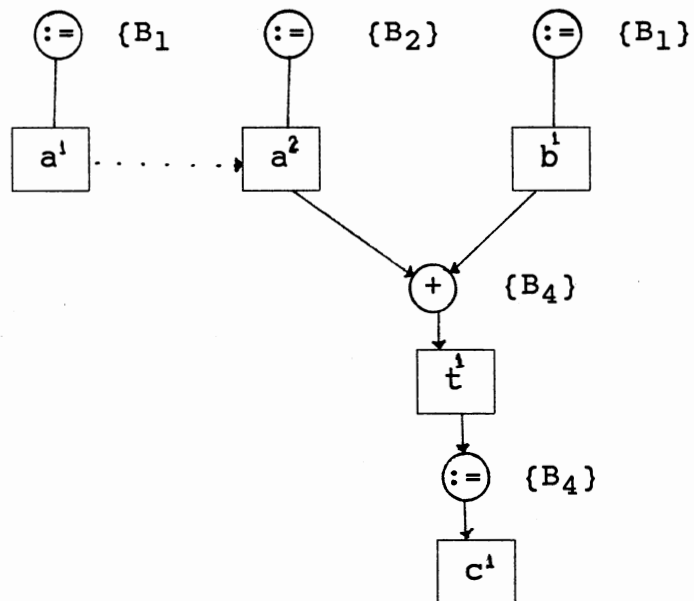
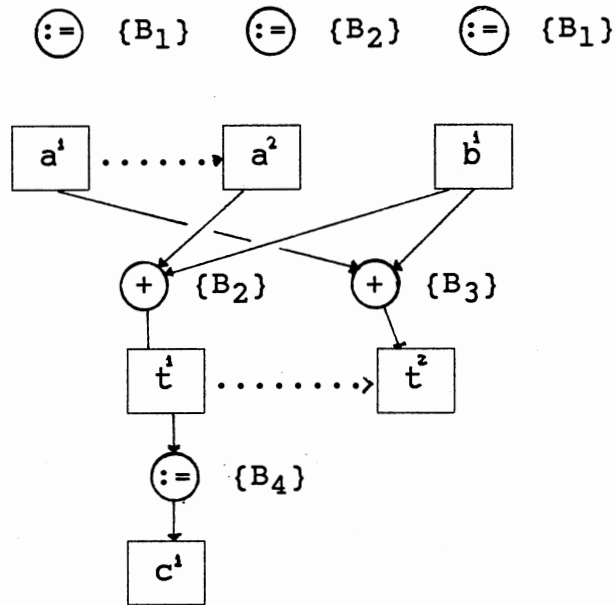


Figure 26. Code Sinking Example

The sequence of steps for analyzing a code sinking candidate are

1. create a dummy instance S_j (J is a join node) of S and make S_j appear to be located in the join block J . Use S_j to search the operand dependence graph. If S_j is not partially redundant, then goto step 6.
2. Apply redundant statement elimination algorithm on S_j . If S_j is not fully redundant, then goto step 6; If there exists only one previous instance of S which renders S_j redundant, then goto step 6.
3. If there is a data link (edge) from the destination operand node of any of the previous instances which render S_j redundant to some statement node in the conditional structure C , then goto step 6.
4. If the region tags of the instances in C and the region tag of instance S_j are not identical, then goto step 6.
5. Delete all the instances of S which render S_j redundant from the operand dependence graph and from the basic blocks where they are located; Insert the dummy instance S_j into the operand dependence graph; Make S_j the first statement of the join node J .
6. Turn off CSS-FLAG in the control node for S ;
Stop.

Constant Folding

Constant folding is compile-time evaluation of an expression whose operand values are constants. To perform

constant folding, a compiler must detect the sections of a program where a variable takes on constant values. The constant folding procedure described in this section is based on an exhaustive evaluation scheme. In exhaustive evaluation program analysis, an expression is evaluated with the set of possible operand values which may be used to execute that expression at run-time. Figure 27 illustrates the limitation of what Kildall[21] calls simple constant folding technique. The classical constant propagation framework cannot discover that the expression $a + b$ in Figure 27(b) is the unique constant 3 because constant propagation is applied to expressions whose variable operands have unique constant values at an expression evaluation point.

The information for recognizing potentially constant valued expressions are obtained from the dependent operand field of a statement record in the distinct statement table

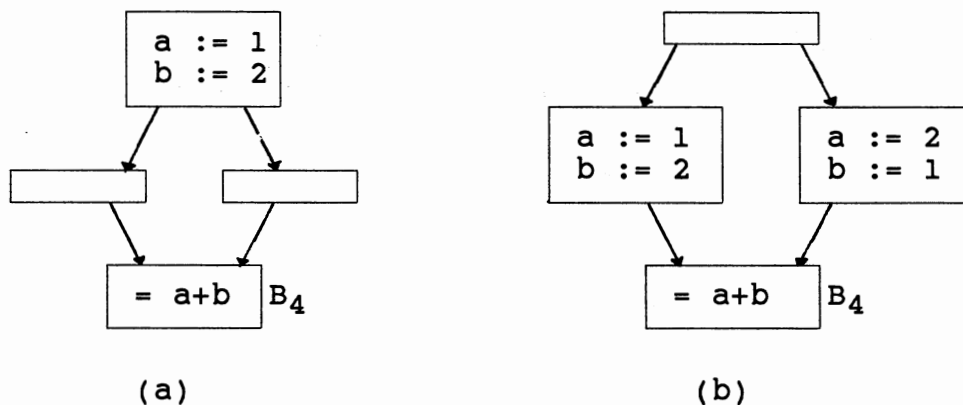


Figure 27. Simple and Non-Simple Constant Expression

and from the field in a variable's version record which indicates whether an instance of a variable is a constant or not. If the dependent operand field of an arithmetic or a logical expression is empty, then the operands used in that expression are symbolic constants. Expressions with all symbolic constant operands are folded immediately.

In typical programs, few expressions have constant values. Applying constant propagation to an entire program as is performed in [21, 37] is unnecessary because only loop invariant expressions, nonloop expressions, and expressions with symbolic constants can have constant values. To reduce the cost of constant folding, these classes of statements are the only ones analyzed.

Constant folding is performed with other optimizations while the operand dependence graph representation of a program region is being built. The versions of each operand reaching an expression's evaluation point is determined and from this information, the constant folding procedure enters a simple mode or an exhaustive evaluation mode. A simple analysis mode is entered when each operand of an expression has unique reaching definition at an expression evaluation point. The constant folding procedure makes one expression evaluation in a simple mode. When any operands of an expression has multiple reaching definitions, the constant folding procedure goes into an exhaustive evaluation mode.

To perform exhaustive analysis of a constant folding candidate, the flow graph nodes where to evaluate a constant

folding candidate are determined first. An evaluation point is the first flow graph node on a distinct forward path to an expression evaluation point, such that at least one of the variable operands has a unique definition which reaches the end of that flow graph node. The constant folding candidate is not moved to any of the evaluation points, rather the evaluation points serves to identify the specific operand instances which may be used to execute that expression at run-time. After identifying the evaluation points, the values of the operands at those points are substituted for the operand values and evaluated. If the result of the operation is the same at each of the evaluation points, then the expression can be folded.

To apply constant folding to the expression $a + b$ in Figure 28, the flow graph nodes with the reaching definitions for a and b at the top of node B_7 are determined from the reaching version numbers.

$$\text{REACHING-DEF}[a; B_7] = \{B_2, B_3, B_5\}$$

$$\text{REACHING-DEF}[b; B_7] = \{B_1, B_4, B_5\}$$

Since neither a nor b is defined in node B_7 , the reaching definitions for a and b are split into two subsets and propagated along the immediate predecessors of node B_7 . Thus $\text{REACHING-DEF}[a; B_7]$ is split into $\{B_2\}$ and $\{B_3, B_5\}$, while $\text{REACHING-DEF}[b; B_7]$ is split into $\{B_1\}$ and $\{B_4, B_5\}$. The subsets $\{B_2\}$ (for a) and $\{B_1\}$ (for b) are propagated to B_2 and the other two subsets are propagated to B_6 .

At node B_2 , there is a unique definition of *a* reaching the end of B_2 . Hence, node B_2 is an evaluation point. The definitions of *a* and *b* reaching the end of B_6 are further split into two subsets and propagated to the predecessor nodes B_4 and B_5 where there are definitions of either *a* or *b* and the process terminates. The nodes B_2 , B_4 , and B_5 are the points to evaluate the expression $a + b$. The value of $a + b$ at all the three points is 3.

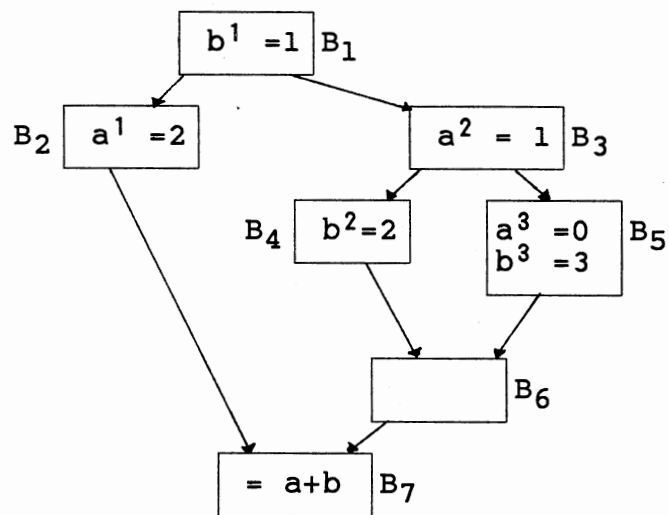


Figure 28. Constant Folding Example

Algorithm 6. Constant Folding Algorithm

Input.

OP: Operation symbol.
 RD_a: Reaching definitions of first operand.
 RD_b: Reaching definitions of second operand.
 FN: Flow graph node where potentially constant expression is located.
 OPD_PAIRS: Possible Operand Combinations at run-time

Output.

VAL: Value of expression if expression is a unique constant.
 FLAG: Status flag indicating whether expression is a constant.

Auxiliary.

OPD_PAIRS: Pairs of flow graph nodes specifying the possible operand instances to be used to evaluate a constant folding candidate.

Method.

Determine the possible operand instances which could be used to compute an expression at execution time. Then perform the operation on the possible operand values and compare the results.

```

Step1. If |RDa| = |RDb| = 1, then begin
    fold expression;
    If OP is a logical expression for a
        conditional jump, then modify flow graph;
    End
Step2. Else begin /* enter exhaustive mode */
    N := 0; /* number of elements in OPD_PAIRS */
    OPD_PAIRS := ∅;
    eval_pairs(OPD_PAIRS, RDa, RDb, FN, N);
    FLAG = TRUE;
    VAL := value of expression using OPD-PAIRS[0];
    While N ≠ 0 do
        If VAL ≠ value of expression using
            OPD_PAIRS[N], then begin
            FLAG := FALSE;
            N := 0;
            End
        Else N := N - 1;
        Endif
    Endo
    If FLAG = TRUE, then begin
        Replace expression with VAL;
        If OP is a logical expression for
            conditional jump, then modify flow graph;
        Endif
    End
Endif
End Constant_fold

```

Figure 29. Constant Folding Procedure

```

Procedure eval_pairs (e_pairs, rd_a, rd_b, fgn, n)

  e_pairs : possible definition instances to be used to
            execute a constant folding candidate.
  rd_a :   definitions of first operand reaching the
            flow graph node fgn.
  rd_b :   definitions of second operand reaching the
            flow graph node fgn.
  fgn :   flow graph node currently checked as an
            evaluation point.
  n :     counts the number of elements in e-pairs.
  Ya, Yb : auxillary storage for subsets of rda and
            rdb respectively.

  step1.  If |rd_a| = 1, then begin
            For each x ∈ rd_b do
              e_pairs[n] := (fgn, x);
              n := n + 1;
            Endo
            End

            Else if |rd_b| = 1, then begin
              For each x ∈ rd_a do
                e_pairs[n] := (x, fgn);
                n := n + 1;
              Endo
            End

            Else /* at least two definition instances of
                  each operand reach node fgn */
              begin
                For each flow graph dag predecessor, p
                  of fgn do
                    Ya := {x ∈ rd_a | p is forward reachable
                          from x};
                    Yb := {x ∈ rd_b | p is forward reachable
                          from x};
                    eval_pairs(e_pairs, Ya, Yb, p, n);
                  Endo
                End
              Endif
            End Eval-pairs.

```

Figure 30. Evaluation Points Determination Procedure

Loop Optimization

Operand dependence graph is very amenable to the detection of loop improvement candidates. A fundamental step in loop optimization is the recognition of loop invariant variables. With loop invariant variables known, loop invariant code motion and induction variable simplification optimizations can be performed.

Loop Invariant Statement Detection

A loop optimization mode is entered when the next flow graph node to be processed is an end of loop marker node. The operand dependence graph constructor places constants, variables referenced or assigned to in a loop in the auxiliary storage LAVQ. A variable in LAVQ is invariant if that variable's version number at the top of a loop's header node and at the top of that loop's end of loop marker node are equal.

A statement in a loop, L is invariant in L if every source operand of that statement is (1) a constant; (2) a loop invariant variable; or (3) value of an expression computed from operands of type (1) and (2). In terms of operand dependence graph, a statement is a loop invariant if every path to an operation instance node through data links originates from class (1) and class (2) operands.

To discover the set of invariant statements in a loop using an operand dependence graph, the graph section

representing the statements in that loop are searched breadth-first. After recognizing the loop invariant statements in L , each invariant is analyzed for loop invariant motion optimization. The condition for moving an invariant statement out of a loop is that the point where that statement is located in a loop must predominate every exit gate of a loop.

The set LAVQ of loop active operands is partitioned into segments to efficiently manipulate nested loops. A segment of LAVQ consists of all the constants and variables active in a loop. A stack of segment pointers point to the base of each LAVQ segment. The top element of the stack of segment pointers identifies the LAVQ segment for the current loop. When control leaves an inner loop, the LAVQ segments for the inner loop and the parent loop are merged (in a union operation) to obtain the LAVQ segment for the parent loop.

The loop invariant detection procedure (Figure 31) traverses a particular path as long as a visited node is marked invariant. By visiting the operand dependence graph nodes breadth-first, the invariance of a lower level node can be determined from predecessor nodes.

Algorithm 7. Loop Invariant Code Motion Algorithm

Input.

LAVQ-SEG: Loop active operands.
 HDR: Loop header node.
 ODG of program region.
 Intermediate code program.

Output.

Possibly modified ODG of program region.
 Possibly modified intermediate code program.

Auxiliary.

LIQ: Loop invariant statements.
 WORKLIST: Sequence of ODG nodes to be traversed.

Method.

Mark the ODG node of each operand in LAVQ_SEG whose version number at the top of the loop header node and at the top of the end of loop marker node is the same "invariant" and enter that operand into WORKLIST. Then traverse the ODG segment of program loop breadth-first, starting at ODG nodes in WORKLIST. Mark each operation node whose operand nodes are marked invariant "invariant", append operation node to WORKLIST, and append marked statement into LIQ. When WORKLIST is empty, apply loop invariant code motion test to each statement in LIQ.

```

For each operand, v ∈ LAVQ-SEG do
  If version number of v at top of HDR = current version
    number of v, then begin
      place v into WORKLIST;
      mark the ODG node for v "invariant";
    endif
  endo
LIQ = ∅;
While WORKLIST is not empty do
  n := first(WORKLIST); /* removes current first item
                        from WORKLIST */
  For each data link successor, s of n in loop do
    If s is not marked "invariant", then begin
      If s is an operand node and the data link
        predecessors of s are marked "invariant", then
        mark s "invariant" and append s to WORKLIST;
    end
    Else begin /* s is an operation node */
      If every data link predecessor of s is marked
        "invariant", then begin
        If operation at s is not assignment, then
          begin (mark s "invariant"; append s to LIQ;
                append s to WORKLIST;)
        end
        Else begin /* operation is an assignment */
          If destination operand is assigned to
            once, then (mark s "invariant", append s
              to LIQ, append s to WORKLIST;)
        end
      end
    end
  end
end

```

```

        endif
    end
endif
endfor
endwhile
For each statement,  $s \in \text{LIQ}$  do
    If the location of  $s$  predominates every exit gate in
        loop, then move  $s$  to loop pre-header;
endfor
End (Loop_invariant Code Motion).

```

Figure 31. Loop Invariant Detection Procedure

Induction Variable Optimization

An induction variable is a variable whose values form an arithmetic or geometric progression while control remains in a loop. Knowing the first term and common difference (arithmetic) or common ratio (geometric) of a progression, the successive terms of that series can be generated. A tree structure called sequence tree is used to represent induction variables.

A sequence tree is a representation for the set of induction variables dependent on a single basic induction variable. The root of a sequence tree is a basic induction variable and the other tree nodes are induction variables derived from the value of that root induction variable. A sequence tree node (except for the root node), s is a child of another sequence tree node, p if the value of p is referenced directly in the intermediate code statement for computing the value of s .

A sequence tree node is a 5-tuple (ind-var, init-val, step, type, children), where ind-var, init-val, and type fields are the name, initial value, and type (arithmetic or geometric progression), respectively of an induction variable. The step field is a common difference or common ratio depending on the value of type field and the children component are pointers to subtree nodes.

Basic Induction Variable Detection. A variable v is a basic induction variable if v is initially live on entry to a loop and v is assigned to in a loop through a distinct statement of the form $v := v \pm d$, where d is a constant or a loop invariant. The operand dependence graph constructor identifies potential induction expressions and places them in BIEQ (basic induction expression queue). An element of BIEQ is a statement of the form $t = v \pm d$, where v is a non-temporary.

Determining whether v is a basic induction variable involves three checks: (1) the temporary t is assigned to v ; (2) the operand d is either a constant or a loop invariant; and (3) no other loop statement may alter the value of v . If the expression $t = v \pm d$ and the variable v pass the three checks, then v is made the root of a sequence tree with the children field initially set to null.

Other Induction Variables. Having identified a basic induction variable, the next step is to find other induction variables which derive their values from that basic induction variable are determined by depth-first search of

an ODG. Suppose v is an induction variable (basic or nonbasic). The data link successors of v in the operand dependence graph are visited depth-first looking for induction variables. Graph search along a path continues as long as any new operation node visited is a statement of the form

$$t = v \pm d;$$

$$t = v * d; \text{ or}$$

$$t = d^v$$

(where d is a loop invariant or a constant). If a statement is one of these three forms, then a sequence tree node is created for t and the address of the tree node for t is added to the children list of tree node v . The field values of a sequence tree node for each type of induction variable are filled using the template in Figure 32. As an example, consider the loop code below.

```

        i := 3
        CMP i, 1000
        BGT $L2

$L1:    t1 = i * 4
        t2 = i - 1
        t3 = t2 * 4
        t4 = INDEXEDLOAD f, t3    /* f[i - 1] */
        t5 = i - 2
        t6 = t5 * 4
        t7 = INDEXEDLOAD f, t6    /* f[i - 2] */
        t8 = t4 + t7             /* f[i - 1] + f[i - 2] */
        INDEXEDLOAD f, t1, t8    /* f[i] = '' */
        t9 = i + 1
        i := t9
        CMP i, 1000
        BLE $L1

$L2:    ...

```

```

Statement form:  $t = v \pm d$ 

  child = create_tree_node(); /* returns tree node */
  child.ind var := t;
  child.init_val := parent.init_val  $\pm$  d;
  if constant(parent.init_val, d), then
    fold(child.init_val);
  child.step := parent.step;
  child.type := 'AP'; /* arithmetic progression */
  parent.children := parent.children U {child};

End /*  $t = v \pm d$  */

Statement form:  $t = v * d$ 

  child = create_tree_node();
  child.ind var := t;
  child.init_val := parent.init_val * d;
  if constant(parent.init_val, d), then
    fold(child.init_val);
  child.step := parent.step * d;
  if constant(parent.step, d), then
    fold(child.step);
  child.type := 'AP';
  parent.children := parent.children U {child};

End /*  $t = v * d$  */

Statement form:  $t = \text{power}(d, v)$ 

  child := create_tree_node();
  child.ind var := t;
  child.init_val := power(d, parent.init_val);
  if constant(parent.init_val, d), then
    fold(child.init_val);
  child.step := power(d, parent.step);
  if constant(parent.step, d), then
    fold(child.step);
  child.type := 'GP'; /* geometric progression */
  parent.children := parent.children U {child};

End /*  $t = \text{power}(d, v)$  */

constant( $x_1, \dots, x_n$ ) = true if each  $x_i$  is a constant;
                        = false otherwise.

```

Figure 32. Template for Defining Fields of a Sequence Tree Node

The operand dependence graph segment relevant to identifying the induction variables in the example loop is shown in Figure 33. The statements $t_2 = i - 1$; $t_5 = i - 2$; and $t_9 = i + 1$ are entered into BIEQ as loop statements are represented on the ODG, but only $t_9 = i + 1$ meets the conditions for basic induction expression. Next the data links for node i are traversed depth-first to locate other induction variables dependent on i . Figure 34 is a sequence tree representation of the induction variables in the example loop.

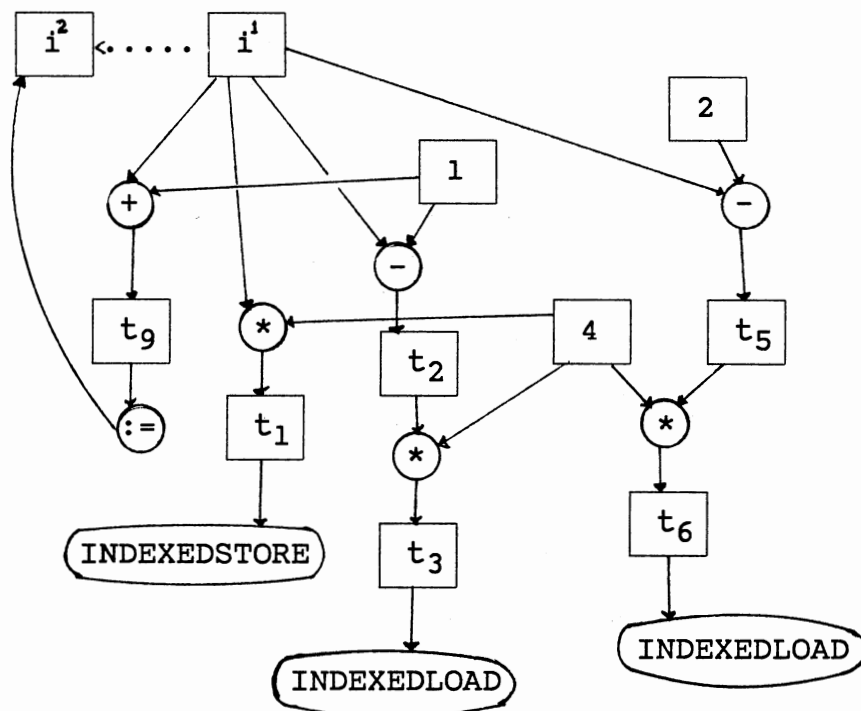


Figure 33. ODG Segment Showing Loop Induction Variables

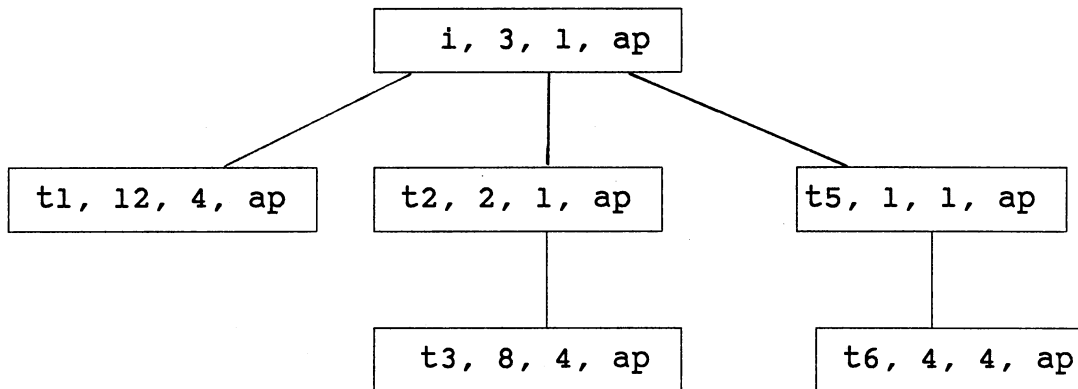


Figure 34. Sequence Tree of Induction Variable Family

Induction Variable Prunning. After constructing a sequence tree for a class of related induction variables, the next step is to identify and eliminate non-essential induction variables from a loop. This step is called induction variable prunning. An induction variable is prunnable if that induction variable is referenced only in statements which compute other induction variables (except basic induction variables). The subset of induction variables examined for prunning are those represented at the internal nodes of a sequence tree.

In the running example, the statements $t2 = i - 1$ and $t5 = i - 2$ qualify for prunning because $t2$ is used only in the expression $t2 * 4$ and $t5$ is referenced in the expression $t5 * 4$. Before eliminating a non-essential induction variable, the initial and step values of that variable must

be preserved since the initial and step values of a child sequence tree node are evaluated from the initial and step values of a parent node.

If both the initial and step values of a non-essential induction variable are constants, then the values are already preserved since they are substituted directly to compute the initial and step values of descendant sequence tree nodes. However, if either the initial value, the step value, or both of a pruning candidate is not a constant, then the statement to compute a nonconstant item (initial or step value) must be added to a loop's pre-header node. After inserting the necessary statements in a loop pre-header, a non-essential induction variable can be removed. Returning to the example loop, the statements $t_2 = i - 1$ and $t_5 = i - 2$ can be deleted from both the loop and the sequence tree.

Induction Variable Simplification. Induction variable simplification is essentially a strength reduction procedure. Strength reduction is applied to induction variables of the form $t = v * d$ or $t = d^v$, where v is an induction variable. Suppose the statement $t = v * d$ is an induction expression. Then the expression $v * d$ is replaced by introducing two new statements, one in a loop pre-header and the second in a loop's body.

In a loop's pre-header block, the statement
 $t = t.\text{init_val}$
 (where $t.\text{init_val}$ is the expression or constant contained in

the initial value field of the sequence tree node for t).
 Let u be the basic induction variable from which the value of t is derived. Below each instance of the statement $u = u \pm c$ in a loop, introduce the statement $t = t \pm t.step$ ($t.step$ is the value of the step field of the sequence tree node for t).

Similarly, if an induction expression is of the form d^v , the statements $t = t.init_val$; and $t = t * t.step$ are inserted at a loop's pre-header and in a loop's body (at the appropriate points), respectively. After induction variable pruning and induction variable simplification steps, the example loop is transformed to the version below.

```

t1 = 12
t3 = 8
t6 = 4

$L1: t4 = INDEXEDLOAD f, t3
     t7 = INDEXEDLOAD f, t6
     t8 = t4 + t7
     INDEXEDSTORE f, t1, t8
     t9 = i + 1
     i := t9
     t1 = t1 + 4
     t3 = t3 + 4
     t6 = t6 + 4
     CMP i, 1000
     BLE $L1

$L2: ...

```

Extracting the induction variables and representing them with sequence trees has several advantages. One advantage is that unnecessary temporaries and statements are

not introduced into the intermediate code. In the induction variable optimization scheme described in [6, 11], too many temporaries are created and requires additional constant propagation, scalar propagation, and useless code elimination passes to clean up the code.

A second advantage is that after the graph search to detect the set of induction variables, the remaining steps of the induction variable optimization procedure do not involve further graph walk. Even in the one graph traversal, only the relevant parts of a loop (those nodes connected to basic induction variables) are visited. With the exception of program dependence graph[15] based approach, other methods scan every loop statement.

Thirdly, because the sequence tree is not embeded in the operand dependence graph, it can be easily integrated into any compiler. Ottenstein's[28] method is efficient when the intermediate code representation is the program dependence graph, but it requires prior applications of constant folding, common subexpression elimination, scalar propagation, etc. There is no such ordering with the operand dependence graph based method.

Complexity of Code Optimization

The main distinguishing characteristics of operand dependence graph based program analysis are (1) code optimization is confined to control structures (straight line code segments, if-then-else, and loop); (2) individual

statements are analyzed for a particular type of redundancy; (3) program analysis and optimization are combined in one step; (4) minimal data flow information (reaching version numbers) is used to detect redundant statements; and (5) optimization decision is based the section of a program already processed.

Information used to detect feasible code optimizations are usually propagated through control flow paths. Any program has a finite number of parallel paths (paths without a common join) determined by the structure of "if" and case statements in a program. The maximum number of parallel paths in a program is bounded by the number of alternate control flow paths in the conditional structures of that program. This bound denoted π is called the data flow width of a program flow graph. A statement instance at some point, p is redundant if there is an active previous instance of that statement on each distinct forward path leading to p . Thus, to determine if a particular statement instance is redundant, at most π previous instances of that statement are examined.

Lemma 11

If h is the number of expressions analyzed for code hoisting and π is the data flow width of a program flow graph, then the complexity of code hoisting optimization is $O(h*\pi)$.

Proof. The number of disjoint instances of any distinct statement with a common instance signature in a

conditional structure is at most π . The code hoisting algorithm first tries to move code to the highest level of a conditional structure. If that attempt fails, then the algorithm performs code motion bottom-up (if the conditional statement is nested). At most both the highest level motion and the nested if analyses are performed on a code motion candidate. In the highest level code motion analysis, $O(\pi)$ disjoint instances are processed during path cover test.

There are $(\pi - 1)$ "if" statements in a conditional structure with π data flow width. Thus, there are $(\pi - 2)$ "if" statements nested within an if structure with $(\pi - 1)$ "if" statements. The nested if analysis part examines two disjoint instances per nested "if" statement giving a cost of $O(2*\pi - 4) = O(\pi)$. Therefore, the time complexity for applying code hoisting optimization on π disjoint instances of a statement is $O(\pi + \pi) = O(\pi)$. □

Lemma 12

If r is the number of partially redundant expressions analyzed for common subexpression elimination and π is the data flow width of a program flow graph, then the cost of common subexpression elimination is at most $O(r*\pi)$.

Proof. There are at most π previous instances of a partially redundant statement which reach a common join. It takes $O(\pi)$ to determine whether π program points is a path cover for some other point. Thus, the complexity for analyzing r partially redundant statements for full

redundancy is $O(r*\pi)$. □

Lemma 13

Let s be the number of code sinking candidates subjected to code sinking optimization analysis and let π be the data flow width of a program flow graph. Then the complexity of code sinking analysis is $O(s*\pi)$.

Proof. Code sinking optimization and common subexpression elimination have the same path cover constraint. Replacing r in lemma 12 with s reduces lemma 12 to lemma 13. □

Lemma 14

Suppose f is the number of expressions analyzed for constant folding and π is the data flow width of a program flow graph. Then the cost of constant folding optimization is $O(f*\pi)$.

Proof. The constant folding procedure evaluates a constant folding candidate with the operand values defined on each of the possible paths control may transfer to the point where a folding candidate is located. The number of different potential definitions of a variable reaching any program point is at most π . This implies the number of expression evaluations to determine whether an expression is a unique constant is $O(\pi)$. f different expressions will require at most $O(f*\pi)$ expression evaluations. □

Lemma 15

Let n be the total number of loop statements over all program loops. The cost of loop optimization is $O(n)$.

Proof. At most $O(n)$ statements are visited during loop invariant statements detection and $O(n)$ statements are visited during search for induction variables giving a total of $O(n + n) = O(n)$. □

Theorem 2

Optimizations performed with the operand dependence graph are safe.

Proof. A code optimization procedure is safe if nonredundant statements are not eliminated and the optimized version of a program does not induce any run-time errors not present in the unoptimized code. Three factors ensure the safety of any transformation applied to a program in an operand dependence graph based implementation:

(1) predominated-inverse-post-dominated ordering of flow graph nodes guarantees that a node is not processed until the flow graph nodes which may compute values referenced in that block have been processed. Thus, the proper reaching definitions are always used to detect redundancies within a program region. (2) Path cover constraint is enforced for all code optimization problems and loop invariant motion is conservative. (3) Code motion related optimizations are not performed until either a join or an end of loop marker node is seen. These factors prevent premature optimizations.

Hence, code improvement transformations applied using operand dependence graph are safe. □

Theorem 3

Let P be an unoptimized version of an intermediate code program and let P' represent the optimized version after applying operand dependence graph intermediate code improvement procedure on P . Suppose P'' is the resulting program after running P' through the optimizer. If useless code elimination is not applied on P' and the regions of P are preserved in P' , then $P'' = P'$.

Proof. Since the regions of P are the same for P' , any inter-region redundancy in P will not be identified in P' . Because lexically identical expressions use the same temporary name to store the value of that distinct expression, redundancies involving expressions using intermediate results are recognized in the same optimization pass. Therefore, no new intrasegment redundancies are discovered in P' .

The induction variables detected in P' are the basic induction variables and "strength reduced" induction variables of P . These set of induction variables will not require further simplification. Lastly, statements moved to a loop pre-header node will not contain common subexpressions since intra-region common subexpressions are eliminated before loop invariant code motion and induction variable optimizations. Therefore, running P' through the

optimizer does not change P'. □

Discussion

Global program analysis by data flow analysis technique is an $O(N^2)$ process, where N is the number of nodes in a flow graph. Data flow information is represented with bit vectors and most of the vectors are usually sparse because basic blocks are relatively short. If N is large, then the $O(N^2)$ cost becomes expensive. Redundancy analysis cost with the operand dependence graph is dependent on two factors; nesting depth of conditional statements and the number of partially redundant statements in a program.

Studies indicate that most optimizations occur in basic blocks [8] and in inner loops of nested loops. This implies that for many programs, the number of global redundancies will be small. Let L be the length of a program (number of statements) and let D be the number of distinct statements in a program. Then $(L - D)$ is the number of statements which may be analyzed for code hoisting, common subexpression elimination, and code sinking. The cost of applying duplicate statement reduction optimizations to $(L - D)$ statements using an operand dependence graph is $O(\pi * (L - D))$. In the worst case, $(L - D)$ is $O(N)$ and π is $N/2$, resulting in $O(N^2)$ process. For many programs, π and $(L - D)$ will be small in which case the cost of redundancy elimination is almost linear.

Summary

The concepts (path cover, variable version numbers, operand dependence graph, and partial redundancy) developed in earlier sections are tied together to detect various forms of redundancies in an intermediate text. A uniform concept of partial redundancy is used to detect common subexpressions, hoistable code, and code sinking candidates. The cost of program improvement is proportional to the product of the data flow width of a program flow graph and the number of partially redundant statements in a program. An induction variable optimization procedure which uses sequence trees to hold induction variables before committing to introducing new statements is developed.

CHAPTER VIII

SIMPLE RECURRENCE LOOP OPTIMIZATION

Simple Recurrence Array Reference

Many numerical algorithms contain recurrent loops. A recurrent loop is a repetition structure in which a value computed in some iteration, i is referenced in a later iteration, j ($j > i$). Vectorizing compilers[7, 22] employ elaborate algorithms to detect the presence of a recurrence in array references to determine when to generate vector code. However, sequential code compilers do not include recurrence analysis in its suite of optimization procedures. This section presents a method for improving sequentially executed loop with simple linear recurrence array references.

A loop, L is a simple linear recurrence loop if L has an array in which at least two distinct elements of that array are accessed in every iteration, such that at least one of the elements accessed in the i th iteration of L is also referenced (indexed load operand) in the $(i+1)$ th iteration. An example of a simple recurrence loop is

```
for (i = 3; i ≤ 1000; i++)
    f[i] = f[i-1] + f[i-2];
```

The value of $f[i-2]$ in the next iteration is the value of

$f[i-1]$ in the current iteration. Similarly, the value of $f[i-1]$ in the next iteration is the value of $f[i]$ in the current iteration.

Array element access is usually more expensive than the access of simple values because of the extra code generated to map an element selection expression to a memory location. Simple recurrence optimization is the reuse of an array element value accessed in the i th execution of a loop and referenced in the $(i + 1)$ th execution of that loop without reloading that element from the array storage.

An array to be analyzed for simple recurrence optimization should possess the following characteristics:

1. all the subscript expressions used to specify element locations are induction variables;
2. the initial and step values of each induction variable are known constants; and
3. at least two distinct elements of that array are accessed in a loop and one of the accesses is an indexedload operation.

Suppose A is an array accessed with two subscript expressions, e_1 and e_2 which satisfy properties (1) and (2) and suppose further that e_1 is an indexedload operand on A . The reference $A[e_1]$ is a simple recurrence optimization candidate with respect to the array element $A[e_2]$ if

(1) either both e_1 and e_2 are increasing sequences of the same type or both e_1 and e_2 are decreasing sequences of the same type; (2) step value of $e_1 =$ step value of e_2 ;

(3) the difference between the initial values of e_2 and e_1 is equal to the step value of e_1 ; (4) $A[e_1]$ and $A[e_2]$ are accessed in every loop iteration; and (5) no statement which lies on an acyclic path originating at the loop header block to any point where $A[e_1]$ is referenced for the first time in a loop may store into $A[e_1]$.

Let H be the header block of a loop, and let $\{B_1, \dots, B_k\}$ be the set of loop blocks such that $\{B_1, \dots, B_k\} \times H$ are back edges. A loop statement, s is executed in every iteration if any forward path from H to each of the blocks $\{B_1, \dots, B_k\}$ contains an instance of s . Condition (4) for simple recurrence loop optimization is satisfied if each forward path from the header block H to every node in the set $\{B_1, \dots, B_k\}$ has statements which access both $A[e_1]$ and $A[e_2]$.

A simple procedure for determining whether the elements $A[e_1]$ and $A[e_2]$ are both accessed in every iteration consists of the following steps.

Step 1. If loop is a single node loop, then condition is satisfied;

Else perform steps 2-4.

Step 2. Compute the sets

$$\text{ACCESS_POINT}[A, e_1] = \{ n \mid A[e_1] \text{ is accessed in block } n \}$$

$$\text{ACCESS_POINT}[A, e_2] = \{ m \mid A[e_2] \text{ is accessed in block } m \}$$

Step 3. Compute the composite forward reachability sets

$$FR_1 = \bigcup_{n \in \text{ACCESS_POINT}[A, e_1]} FWR[n]$$

$$FR_2 = \bigcup_{m \in \text{ACCESS_POINT}[A, e_2]} FWR[m]$$

(where $FWR[x]$ = forward reachability set of node x)

Step 4. Calculate $FR_1 \cap FR_2$

$$\text{Compute } P = \bigcup_{B_i \text{ is a looping node.}} \text{pred}[B_i]$$

Condition (4) is satisfied if

$$P \subseteq FR_1 \cap FR_2$$

and

$$\text{either } FR_1 \cap FR_2 = FR_1$$

$$\text{or } FR_1 \cap FR_2 = FR_2$$

Element Update Constraint

The problem is given an array, A and a subscript expression, e for some element of A referenced in a loop; can the location $A[e]$ be modified before being referenced? To answer this question, these sequence of steps are followed.

The first step is to determine whether there is a store into any element of A prior to the reference $A[e]$. This is accomplished by comparing the version history of A at the point of the indexedload operation with the version number of A at the top of the loop header block. If they are equal, then there is no store into $A[e]$. Suppose the two numbers are not equal, then the relationship between the

location assigned to and the location referenced is determined next.

If the array A is a call-by-reference parameter and interprocedural alias analysis information is not available, then it is assumed that any element of A can be assigned to. Another worst case assumption is made if the subscript expression for an indexedstore is not an induction variable.

When the subscript operand for an indexedstore operation is a loop induction variable, but either the initial value of the induction variable or the step value of the induction variable is not a numeric constant, the destination of the indexedstore is taken to be any element location. This situation is illustrated in the loop below.

```

for (i = 3; i ≤ 1000; i++) {
    f[i-d] = 5;
    f[i] = f[i-1] + f[i-2];
}

```

The subscript expression $i - d$ is an induction variable, but the value of d is not known. If the value of d is one or two, then $f[i-1]$ or $f[i-2]$ cannot be a simple recurrence reference.

Suppose the subscript expression for the indexedstore into A is an induction variable whose initial and step values are known constants, then the indexedstore subscript and the indexedload subscript are subjected to mathematical analysis to determine if there is a loop iteration in which both subscripts are equal. Let s and e represent the indexedstore subscript and indexedload subscript,

respectively. The analysis involves equating the formula for calculating s and e and then solve the resulting equation for integer solutions. If an integer solution exists, then the array reference $A[e]$ cannot be a simple recurrence array reference.

In order to find solutions to the sequence equation, the sequence type of s and the sequence type of e must be considered. Three equation classes are distinguished based on the sequence type of s and e : (1) both s and e are arithmetic sequences; (2) both s and e are geometric sequences; and (3) s and e are of different types.

The n th term, t_n of an arithmetic progression is given by the formula

$$t_n = \alpha + (n-1)d, \quad (n, d \geq 1)$$

where α is the first term (initial value), and d is the common difference (step value). For a geometric progression, the n th term g_n is given by

$$g_n = \beta r^{n-1} \quad (n \geq 1, r > 1)$$

where β is the first term, and r is the common ratio.

Suppose both subscripts are arithmetic progressions.

Let

$$\alpha_1 + (n-1)d_1$$

and

$$\alpha_2 + (n-1)d_2$$

be the formulas for their n th terms. The equation

$$\alpha_1 + (n-1)d_1 = \alpha_2 + (n-1)d_2$$

has an integer solution if

$$(d_2 - d_1) \text{ divides } (\alpha_1 - \alpha_2),$$

$$(\alpha_1 - \alpha_2) = (d_2 - d_1) = 0,$$

or

$$(\alpha_1 = \alpha_2 \text{ and } n = 1).$$

For the case where both subscripts are geometric progressions, let

$$\beta_1 r_1^{n-1}$$

and

$$\beta_2 r_2^{n-1}$$

be the formulas for the n th term of s and e , respectively.

The equation

$$\beta_1 r_1^{n-1} = \beta_2 r_2^{n-1}$$

has an integer solution if

$$(\beta_1 = \beta_2 \text{ and } n = 1),$$

$$(\beta_1 - \beta_2) = (r_1 - r_2) = 0, \text{ or}$$

$$(\beta_1 \text{ divides } \beta_2) \text{ and } (r_2 \text{ divides } r_1)$$

When one of the subscripts is a geometric sequence and the other is an arithmetic sequence, the equation

$$\alpha + (n-1)d = \beta r^{n-1}$$

is solved.

A solution to this equation exists if the following conditions are simultaneously satisfied:

$$\alpha + d = \beta r$$

$$\beta \text{ divides } (\alpha + d)$$

$$r \text{ divides } (\alpha + d)$$

$$d \text{ divides } \alpha$$

$$d \text{ divides } \beta$$

If there does not exist an integer solution to the formed equation, then the value of $A[e]$ is not computed in the same loop iteration it is referenced.

Simple Recurrence Elimination

Given that an array reference $A[e_1]$ is a simple recurrence reference with respect to another element access $A[e_2]$, the reference $A[e_1]$ can be eliminated from a loop by performing the following steps in the sequence presented. Let t_1 denote the temporary into which the element $A[e_1]$ is loaded in an indexedload operation and let t_2 denote the value of $A[e_2]$.

Step 1.

Move the indexedload operation

$$t_1 = A[e_1]$$

to the loop pre-header, but below the statement which computes the initial value of e_1 in the pre-header.

Step 2.

Let H be the header node of the loop in question.

For each block, B such that (B,H) is a backedge do

If B is a conditional block, then begin

If $A[e_1]$ is an operand of the comparison operation for the conditional jump, then begin

(1) Create a new temporary t_3 ;

(2) Just before the compare operation introduce the copy statement $t_3 = t_1$;

(3) Substitute t_3 for t_1 in the comparison statement;

Endif

Just before the compare statement in block B , introduce the copy statement $t_1 = t_2$

End

Else just before the unconditional jump statement in B , introduce the copy statement $t_1 = t_2$

Endo

Step 3.

If the only use of e_1 in the loop after performing step1 and step2 is in the induction statement of the form

$$e_1 = e \pm c, \text{ or}$$

$$e_1 = e_1 * c,$$

then eliminate the induction statement.

In many structured programs, a loop has only one backedge in which case the copy statement $t_1 = t_2$ is introduced in one loop block. Except when $A[e_1]$ is an operand of a compare statement in a block that contains the looping statement, simple recurrence array reference elimination reduces the number of statements in a loop. Even when the value of $A[e_1]$ is used to determine loop termination, two scalar copy statements will execute faster than indexedload and induction variable update operations. Moreover, if some of the scalar operands t_1 , t_2 , and t_3 are placed in registers, loop execution time will be significantly improved.

Simple Recurrence Analysis

There are two stages of simple linear recurrence analysis. The first stage identifies array variables in a loop to be tested for simple recurrence optimization, while in the second stage simple recurrence test are performed on subscripts of the arrays selected in stage one.

In the first step, array variables accessed with one subscript or not referenced in an expression are removed from consideration. Also, arrays accessed with non induction variable subscript or accessed with induction variables whose initial values or step values are not known constants are removed from the list of test candidates.

An array variable not disqualified in the first phase has at least two distinct subscripts which select elements

of that array. Suppose A is an array for simple recurrence test. The second phase of the recurrence test proceeds as follows. Let $S = \{s_1, \dots, s_k\}$, $k \geq 2$, be the set of subscripts used to access A in a loop.

Step 1.

Partition S into two subsets S_a and S_g , such that

$$S_a = \{s \in S \mid s \text{ is an arithmetic sequence}\}$$

$$S_g = \{s \in S \mid s \text{ is a geometric sequence}\}.$$

Step 2.

If S_a is nonempty and S_a has at least two elements, then partition S_a into the subsets I and D , where

$$I = \{i \in S_a \text{ and } i \text{ is an increasing sequence}\}$$

$$D = \{d \in S_a \text{ and } d \text{ is a decreasing sequence}\}.$$

Step 3.

If I has at least two elements, then arrange the elements of I in increasing initial subscript value order. For each pair of adjacent elements s_1, s_2 in I , where s_1 precedes s_2 in I check the following conditions:

- (1) s_1 is an indexedload subscript.
- (2) Step value of $s_1 =$ step value of s_2 .
- (3) Second value of $s_1 =$ initial value of s_2 .
- (4) $A[s_1]$ and $A[s_2]$ are both accessed in every iteration.
- (5) Any indexedstore operation into A which precedes the reference $A[s_1]$ may not store into $A[s_1]$.
- (6) If the above conditions are satisfied, then apply

simple recurrence elimination procedure on $A[s_1]$.

Step 3b. If D has at least two elements, then arrange the elements of D in decreasing initial value order. For each adjacent elements of s_1, s_2 in D , where s_1 precedes s_2 in D , repeat the six steps performed for I .

Step 4.

If S_g has at least two elements, then substitute S_g for S_a and repeat step2 and step3.

The fibonacci-like loop

```
for (i = 3; i ≤ 1000; i++)
    f[i] = f[i-1] + f[i-2];
```

serves as an example to illustrate the simple recurrence removal procedure. First the array f qualifies for simple recurrence test. The subscripts $i, i - 1$, and $i - 2$ are all arithmetic sequences and their initial values and step values are constants.

At the beginning of the second phase of the analysis, the set $S = \{i, (i - 1), (i - 2)\}$ is formed. In Step1, S is partitioned into the subsets

$$S_a = S$$

$$S_g = \emptyset$$

At step2, S_a is further partitioned into I and D . The D subset is empty because the subscripts $i, i - 1$, and $i - 2$ are all increasing sequences. The initial values of $i, i - 1$, and $i - 2$ are 3, 2, and 1, respectively, and the step values of $i, i - 1$, and $i - 3$ are 1, 1, and 1, respectively. Ordering the subscripts in I in increasing initial value

order produces the sequence

$$I = \{ (i - 2), (i - 1), i \}$$

Next, the adjacent pair $(i - 2, i - 1)$ are subjected to the five tests in step3. Since the pair $(i - 2, i - 1)$ pass the five tests, simple recurrence elimination procedure can be applied to the element $f[i - 2]$. Let t_1 and t_2 be the temporaries for storing the values of $f[i - 2]$ and $f[i - 1]$, respectively. The statement

$$t_1 = f[1]$$

is introduced outside the loop and inside the loop the statement

$$t_1 = t_2$$

is inserted at the end of the loop code. After this transformation the loop code becomes

```
t1 = f[1]
for (i = 3; i ≤ 1000; i++) {
    t2 = f[i-1];
    f[i] = t2 + t1;
    t1 = t2;
}
```

The remaining pair $(i - 1, i)$ also passes the simple recurrence test and $f[i - 1]$ is moved out of the loop. The final code after eliminating the recurrence array references $f[i - 1]$ and $f[i - 2]$ is

```
t1 = f[1];
t2 = f[2];
for (i = 3; i ≤ 1000; i++) {
    t3 = t2 + t1;
    f[i] = t3;
    t1 = t2;
    t2 = t3;
}
```

Loop Unrolling and Simple Recurrence

Ordinarily loop unrolling improves the execution time of a program loop by reducing the number of times the termination condition is tested, but usually at the expense of a larger loop code. Under certain circumstances a loop can be unrolled to transform a non simple linear recurrence loop to an equivalent loop with simple recurrence as the example below illustrates.

```
for (i = 3; i ≤ 1000; i++)
    A[i] = A [i-2] + 5;
```

This loop is not a simple linear recurrence loop because when $i = 4$, $i - 2 = 2 \neq 3$. If the loop is unrolled twice as in

```
for (i = 3; i ≤ 1000; i+=2){
    A[i] = A[i-2] + 5;
    A[i+1] = A[i-1] + 5;
}
```

then the array references become simple recurrence references. When $i = 5$, $A[i-2]$ is $A[3]$ ($A[3]$ is defined in the first iteration) and $A[i-1]$ is $A[4]$ ($A[4]$ is assigned to in the first iteration). Applying simple recurrence array reference elimination procedure to the unrolled loop results in the loop code below.

```
t1 = A[1];    /* A[i-2] */
t2 = A[2];    /* A[i-1] */
for (i = 3; i ≤ 1000; i+=2) {
    t3 = t1 + 5;
    A[i] = t3;
    t4 = t2 + 5 ;    (*)
    A[i+1] = t4;
    t1 = t3;
    t2 = t4;
}
```

The condition under which loop unrolling should be applied to induce simple recurrence relation is when all the conditions for simple recurrence loop are satisfied except for the third constraint. To determine how many times to unroll a loop, the formula for calculating the n th term of the indexedload subscript is used.

Suppose the subscript expressions which failed the simple recurrence test are e_1 and e_2 and suppose further that e_1 precedes e_2 in the sorted order. Assuming e_1 and e_2 are arithmetic progressions, the equation

$$a_1 + (n - 1)d = a_2$$

is solved to determine the term (n) of the sequence e_1 which equals the first term (a_2) of the sequence e_2 . The minimum number of times to unroll a loop to create the maximum number of simple recurrences is $n - 1$. Unrolling the loop n or more times does not increase the number of simple recurrences, however, it does induce common subexpressions among some of the array references.

Coming back to the example loop

```
for (i=3; i ≤ 1000; i++)
    A[i] = A[i-2] + 5;
```

$e_1 = i - 2$; $e_2 = i$.

The formula for the n th term of e_1 is

$$1 + (n - 1)$$

(where $a_1 = 1$, $d = 1$)

and the first term of e_2 is 3.

Solving the equation

$$1 + (n - 1) = 3$$

yields $n - 1 = 2$ or $n = 3$. Thus, if the loop is unrolled twice, there will be two simple recurrences.

Suppose the example loop is unrolled three times, then the resulting code is

```

for (i=3; i ≤ 1000; i+=3) {
    A[i] = A[i-2] + 5;
    A [i+1] = A[i-1] + 5;
    A[i+2] = A[i] + 5;
}

```

Notice that in the third statement of the unrolled loop, the element $A[i]$ is referenced creating a common subexpression with respect to the first statement, but the number of simple recurrences is still two.

Summary

An optimization procedure for simple recurrence loops is developed for improving sequentially executed loops. Simple recurrence detection is a special case of general detection of recurrences in a loop. Simple recurrence array reference optimization replaces indexedload operations involving arrays with scalar copy statements. The improvement in loop execution time comes from the elimination of the statements which map subscript expressions to array elements memory addresses.

CHAPTER IX

SUMMARY, CONCLUSION, AND RECOMMENDATIONS

Summary

This study has demonstrated that an operand dependence graph is a viable alternative to current methods of compiler code improvement and that common subexpression elimination (local and global), code motions, and loop optimizations can be performed in a single optimization pass in a region relative code optimization scheme. An operand dependence graph representation of a program is not sufficient to detect feasible optimizations, but it does play a very important role - that of highlighting potentially redundant statements. Control flow and variable definition information (reaching version numbers) are then used to decide complete redundancy. In this way, blind searches for feasible optimizations can be reduced.

The concepts of variable version numbers, path cover, conditional environment cover, and partial redundancy are developed to unify common subexpression, code hoisting, and code sinking optimization problems which traditionally required different data flow analysis steps. The fact that commonalities exist between these code optimization problems

has a positive impact on their implementation; these optimization procedures can be implemented with a common set of program modules, thereby reducing the size of a code optimizer.

Useless Code Elimination

Useless code cannot be performed in a one-pass optimization procedure when program statements are processed in normal program execution order. To be able to detect redundant assignment to a variable, the definitions and uses of that variable must be known. Complete variable definition information is not available during operand dependence graph construction. In order to add useless code detection capability to an operand dependence graph, one of two approaches can be used; (1) analyze a program in two passes or (2) process the program flow graph in reverse topological order.

With a two-pass optimization approach, the other optimizations are performed in one pass. Then in the second pass, each variable assignment operation is examined to determine if that instance of a variable may be referenced in some other statement. Processing the flow graph regions in reverse topological order is attractive because useless code elimination and the regular optimizations can be performed in a one pass. However, there is a small price to pay - constant folding opportunities may be missed. In my view, useless code detection should be placed in the

jurisdiction of a code generator because of the strong interaction between register allocation and useless code.

Improvements

The partitioning of a control flow graph into control regions needs improvement. Currently, all the nodes of a program region either belong to a loop structure or an acyclic structure. A program region partitioning scheme which combines both acyclic and loop structures may enhance the detection of some common subexpressions currently classified as inter-region common subexpressions. Moreover, larger program regions translate to fewer number of distinct statement table initializations during the construction of an operand dependence graph of a program region.

Operand dependence graph based code optimization restricts redundant computation detection to lexically identical expressions. The limitation of this approach is that any redundancy induced by value equivalence cannot be detected. Redundancy recognition by value equivalence is more general than textual equivalence, but a value equivalence technique must maintain equivalence classes of equal variables. I am not sure the addition of an equivalent variables determination procedure to an operand dependence graph will significantly improve code quality, since most redundancies are introduced in loops where array references are linearized. Maybe using value numbers instead of version numbers will identify both types of

redundancies. What impact value numbers will have on operand dependence graph construction rules I do not know, but the idea is worth exploring.

Further Studies

The application of operand dependence graph is restricted to structured program flow graphs. Although the majority of real programs have structured flow graphs, there are still some programs with non-reducible control structures. It is not known whether an operand dependence graph will be effective for non-reducible flow graphs without some major modifications. I am inclined to believe that at least reaching version numbers will have to be pre-computed as in reaching definitions to obtain a conservative data flow information.

The known code optimization procedures lack adaptive capability. A program without any redundancy and a program with redundancies will go through the same code improvement stages. There is no mechanism for avoiding fruitless searches. An operand dependence graph has elementary adaptive mechanism. For instance, first instance of each distinct intermediate code statement is not subjected to any of the duplicate code (common subexpressions, code hoisting, and code sinking) redundancy tests. From a theoretical standpoint, the question "does program P in its current form have any redundancies?" is unsolvable, but are there heuristics that can tell when to avoid optimization passes

that will not improve object code efficiency? An adaptive capability will be useful in programming environments where programs under development are constantly modified.

A performance comparison of operand dependence graph based approach with other (data flow analysis, program dependence graph, and global value numbers) methods should be investigated to gather various statistics such as optimizer code size, running times, storage requirements for data structures, code size of optimized code, and running times of optimized code. This could be a master's thesis project.

REFERENCES

1. AHO, A. V., SETHI, R., and ULLMAN, J. D. *Compilers: Principles, Techniques, and Tools*. Addison Wesley 1986.
2. AHO, A. V. and ULLMAN, J. D. Optimization of straight line code. *SIAM J. Comput.* 1, 1(Mar. 1972), 1-19.
3. ALLEN, F. E. Program optimization. *Annual Review in Automatic Programming* 5, (1969), 239-307.
4. ALLEN, F. E., CARTER, J., FABRI, J., FERRANTE, J., HARRISON, W. H., LOEWNER, P. G., and TREVILLYAN, L. H. The experimental compiling system. *IBM. J. Research and Development* 24, 6(June 1980), 695-715.
5. ALLEN, F. E., and COCKE, J. A program data flow analysis procedure. *Communications ACM* 19, 3(March 1976), 137-147.
6. ALLEN, F. E., COCKE, J., and KENNEDY, K. Reduction of operator strength. *Program Flow Analysis: Theory and Practice*, eds (Muchnick and Jones), 1981, 79-101.
7. ALLEN, J. R. Dependence Analysis for Subscripted Variables and its Application to Program Transformation. Ph. D. thesis, Dept. Mathematical Sciences, Rice University, Houston, April 1983.
8. ASURU, J. M. and HEDRICK, G. E. A directed graph for intermediate program representation. *Third Workshop on Applied Computing*, Stillwater, Oklahoma, Mar. 1989, 29-33.
9. BIEMAN, J. M. Measuring Software Data Dependence Complexity. Ph.D. thesis, University of Louisiana, Computer Science Dept., April, 1984.
10. CARTER, L. R. An Analysis of Pascal Programs and Several Basic Block Optimizations. Ph.D. thesis, Dept. of Computer Science, University of Colorado, Boulder, May, 1980.
11. COCKE, J. and KENNEDY, K. An algorithm for reduction of operator strength. *Communications ACM* 20, 11(Nov. 1977), 850-856.

12. COCKE, J. and SCHWARTZ, J. T. *Programming Languages and Their Compilers: Preliminary Notes, Second Revised Version*. Courant Institute of Mathematical Sciences, New York (1970).
13. COUTANT, D. S. Retargetable high-level alias Analysis. *13th ACM Symposium on Principles of Programming Languages*, (Jan. 1986), 110-118.
14. CYTRON, R., LOWRY, A., and ZADECK, K. Code motion of control structures in high-level languages. *13th ACM Annual Symposium on Principles of Programming Languages*, (Jan. 1986), 70-85.
15. FERRANTE, J., OTTENSTEIN, K. J., and WARREN, J. D. The program dependence graph and its use in optimization. *ACM Trans. Prog. Lang. System* 9, 3(July 1987), 319-349.
16. GESCHKE, C. M. Global Program Optimization. Ph. D. thesis, Dept. of Computer Science, Carnegie-Mellon University, 1972.
17. HECHT, M. S. *Flow Analysis of Computer Programs*. Elsevier North-Holland, 1977.
18. HECHT, M. S. and ULLMAN, J. D. A simple algorithm for global data flow analysis problems. *SIAM J. Comput.* 4, 4(1975), 519-532.
19. HECHT, M. S. and ULLMAN, J. D. Characterizations of reducible flow graphs. *J. ACM* 21, 3(1974), 367-375.
20. JAIN SUNEEL, and THOMPSON CAROL. An efficient approach to data flow analysis in a multiple pass global optimizer. *Proceedings of the SIGPLAN'88 Conference on Programming Language Design and Implementation*, (Atlanta, Georgia, June, 1988), 154-163.
21. KILDALL, G. A. A unified approach to global program optimization. *ACM Symposium on Principles of Programming Languages*, (Boston, MA, Oct. 1973), 194-206.
22. KUCK, D. J., KUHN, R. H., LEASURE, B. and WOLFE, M. The structure of an advanced vectorizer for pipelined processors. *Proceedings of IEEE Computer Society Fourth International Computer Software and Applications Conference*, (Chicago, Oct. 1980).
23. LENGAUER, R. and TARJAN, R. E. A fast algorithm for finding dominators in a flow graph. *ACM Trans. Prog. Lang. and Syst.* 1, 1(1979), 121-141.

24. LOWRY, E. S. and MEDLOCK, C. W. Object code optimization. *Communications ACM* 12, 1(Jan. 1969), 13-22.
25. MCCABE, T. J. A complexity measure. *IEEE Trans. Soft. Eng. SE-2*, 4(Dec. 1976), 308-320.
26. METCALF, M. *Fortran Optimizations*. Academic Press, 1982.
27. MOREL, E. and RENVOISE, C. Global optimization by suppression of partial redundancies. *Communication ACM* 22, 2(Feb. 1979), 96-103.
28. OTTENSTEIN, K. J. *A simplified framework for reduction in strength*. Computer Science Technical Report, TR 85-4d, Michigan Technological University, (June 1988).
29. OTTENSTEIN, K. J. Data-flow Graphs as an Intermediate Program Form. Ph. D. thesis, Computer Sciences Dept., Purdue University, Lafayette, Indiana, Aug. 1978.
30. PITTMAN, T. J. Practical Code Optimization by Transformational Attribute Grammars Applied to Low-level Intermediate Code Trees. Ph. D. thesis, University of California, Santa Cruz, 1985.
31. REIF, J. H. and LEWIS, H. R. Symbolic evaluation and the global value graph. *4th ACM Symposium on Principles of Programming Languages and Systems*, 104-118, Jan. 1977.
32. REIF, J. H. and TARJAN, R. E. Symbolic program analysis in almost linear time. *SIAM J. Comput.* 11, 1(Feb. 1982), 81-93.
33. ROSEN, B. K., WEGMAN, M. N., and ZADECK, F. K. Global value numbers and redundant computations. *15th ACM Symposium on Principles of Programming Languages*, Jan. 1988, 12-27.
34. RYDER, B. G. and PAUL, M. C. Incremental data flow analysis. *ACM Trans. Prog. Lang. Syst.* 10, 1(Jan. 1988).
35. SITES, R. L. The compilation of loop induction expressions. *ACM Trans. Prog. Lang. Syst.* 1, 1(July 1979), 50-57.
36. ULLMAN, J. D. Fast algorithms for the elimination of common subexpressions. *Acta Informatica* 2, 3(July 1973), 191-213.

37. WEGMAN, M. N. and ZADECK, F. K. Constant propagation with conditional branches. *12th ACM Symposium on Principles of Programming Languages*, Jan. 1985, 291-299.
38. WEIHL, W. E. Interprocedural data flow analysis in the presence of pointers, procedure variables and label variables. *Conference Record, 7th ACM Symposium on Principles of Programming Languages*, (Jan. 1980), 83-94.
39. WULF, W. et al. *The Design of an Optimizing Compiler*. American-Elsevier 1975.

VITA ²

Jonathan M. Asuru

Candidate for the Degree of
Doctor of Philosophy

Thesis: AN OPERAND DEPENDENCE GRAPH METHOD FOR CODE
OPTIMIZATION

Major Field: Computing and Information Sciences

Biographical:

Personal Data: Born in Rumuoro-Ogbakiri, Rivers,
Nigeria, September 25, 1954, the son of Walson and
Lavender Asuru.

Education: Bachelor of Science degree in Computer
Science from University of Lagos, Nigeria in June,
1979; Master of Science degree in Computing and
Information Sciences from Oklahoma State
University in December, 1985; Completed the
requirements for the Doctor of Philosophy degree
at Oklahoma State University in May, 1990.

Professional Experience: Teaching Assistant and System
Manager, School of Physical Sciences, University
of Port Harcourt, Nigeria, November, 1980 to
August, 1982; Teaching Assistant, Department of
Computing and Information Sciences, Oklahoma State
University, August, 1984 to July, 1989. Assistant
Professor, Department of Computer Science,
University of Tennessee at Chattanooga, August,
1989 to present.

Professional Organization: Member of the Association
for Computing Machinery (ACM).