EVALUATION OF PROGRAMS AND PARALLELIZING COMPILERS USING DYNAMIC ANALYSIS TECHNIQUES

BY

PAUL MARX PETERSEN

B.S., University of Nebraska–Lincoln, 1986
M.S., University of Illinois at Urbana–Champaign, 1989

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1993

Urbana, Illinois

©Copyright by
PAUL MARX PETERSEN
1993

EVALUATION OF PROGRAMS AND PARALLELIZING COMPILERS USING DYNAMIC ANALYSIS TECHNIQUES

Paul Marx Petersen, Ph.D.
Department of Computer Science
University of Illinois at Urbana–Champaign, 1993
D. Padua, Advisor

The dynamic evaluation of parallelizing compilers and the programs to which they are applied is a field of abundant opportunity. Observing the dynamic behavior of a program provides insights into the structure of a computation that may be unavailable by static analysis methods.

A program may be represented by a dataflow graph generated from the dynamic flow of information between the operations in the program. The minimum parallel execution time of the program, *as it is written*, is the longest (critical) path through the dynamic dataflow graph. An efficient method of finding the length of the critical path is presented for several parallel execution models. The inherent parallelism is defined as the ratio of the total number of operations executed to the number of operations in the critical path. The effectiveness of a commercial parallelizing compiler is measured by comparing, for the programs in the Perfect Benchmarks, the inherent parallelism of each program, with the parallelism explicitly recognized by the compiler.

The general method of critical path analysis produces results for an unlimited number of processors. Upper and lower bounds of the inherent parallelism, for the case of limited processors, can be derived from the processor activity histogram, which records the number of concurrent operations during each time period.

Stress analysis is a derivative of critical path analysis that determines the locations in a program that have the largest contribution to the critical path. Inductions are a computation that introduce an internal stress. A specific method is presented which measures the effects of removing the serializing effects of inductions on the inherent parallelism.

Dependence analysis is crucial to the effective operation of parallelizing compilers. Static and dynamic evaluation of the effectiveness of compile-time data dependence analysis is presented, the evaluation compares the existing techniques against each other, and against the theoretical optimal results. Special attention is paid to the dependences which serialize interprocedural parallelism. In addition to evaluating the static dependence analysis techniques, a method for finding dynamic dependences is presented that includes a record of the dependence distances that were present during an execution.

I might not have finished this ordeal called a Ph.D. if it were not for the love, support, and encouragement I received from the following people: my parents Marx Petersen and Elaine Petersen, Mary's parents Kenneth and Ruth Anne Schoen, my sisters Dawn, Carla and Kari, and my loving wife Mary, who went through most of the trials and tribulations along with me (and sometimes wondering if it would ever end). I dedicate this thesis to these wonderful human beings.

ACKNOWLEDGMENTS

I would like to express my sincere appreciation to my advisor, David Padua, for his patience, guidance, insight, and encouragement which made the completion of this thesis possible. I will always be extremely grateful for the many hours of stimulating conversation he and I have had over the past 6 years. His ability to quickly grasp a concept and suggest avenues of experimentation have been invaluable. Our many conversations have enriched my life in uncountable ways. Additionally, my thanks go to Professor Padua for having me as his student, and encouraging me to attend the University of Illinois. And most of all, I express my thanks for his willingness to keep reading my thesis.

I am also grateful for David Kuck, Constantine Polychronopoulos, Daniel Reed, and L.V. Kale for taking time out of their busy schedules to serve on my thesis committee.

There are several people I would like to thank for the help that they have given me during the past 6 years. I thank my friends and colleagues at the University, especially my officemates from the compiler group who provided an unending source of stimulating discussions ranging from compilers, to politics, to pizza and beverage. Additionally, many thanks are due to my friends in and out of Urbana-Champaign for their constant encouragement and support.

I would also like to thank the individuals in the Department of Computer Science and the Center for Supercomputing Research and Development who have aided me in my studies. I thank the secretaries at CSRD, and Merle Levy for proof-reading many papers and my thesis. I also would like to acknowledge the financial, technical, and secretarial support provided by the Center for Supercomputing Research and Development (CSRD). I am very grateful to have been able to work at CSRD, an excellent research environment.

The work presented in this thesis would not have been possible without the support of Kuck and Associates, Inc. for supplying a version of KAP which was usable for my experiments. Also, I would like to thank the people at NCSA for allowing many experiments to be run on the IBM RS/6000 workstation cluster, particularly David Garver who grasped the implications of my work and helped expedite the paperwork.

Most importantly, I would like to thank my family, and particularly my parents, for their constant aid throughout my life. With unlimited patience and love, my mother, through a life of example, taught me the joys of learning. Whose own curiosity and love for learning kindled a spark in my life. I would like to thank my father, whose own joy of life and conviction that one can succeed at anything helped inspire my commitment to enhance my education, for his love, support, and encouragement throughout my educational career.

Finally, I do not think that I would have survived this process without the continuous support of my wife, Mary. She always believed in me and helped me keep body and soul together through the struggles of this Ph.D.

Thank you all.

TABLE OF CONTENTS

CHAPTER PAGE

1 INTRODUCTION . . . 1
- 1.1 The Grand Plan . . . 1
- 1.2 Understanding the Intent of a Program . . . 2
- 1.3 Amdahl's Law . . . 4

2 BACKGROUND . . . 6
- 2.1 Dependence Analysis . . . 6
- 2.2 Simple and Approximate Dependence Tests . . . 8
  - 2.2.1 Constant Test . . . 8
  - 2.2.2 GCD and Generalized GCD Tests . . . 9
  - 2.2.3 Banerjee's Inequalities . . . 10
  - 2.2.4 Banerjee Infinity Test . . . 11
- 2.3 Integer Programming Based Dependence Tests . . . 12
  - 2.3.1 Simplex Based Integer Programming Test . . . 12
  - 2.3.2 Omega Test . . . 13

3 THE DELTA PROGRAM MANIPULATION SYSTEM . . . 14
- 3.1 Introduction . . . 14
- 3.2 The SETL Language . . . 15
- 3.3 Organization of Delta . . . 16
- 3.4 Using Delta to Instrument Fortran Programs . . . 17
- 3.5 Delta Function Definitions for Instrumentation . . . 20
  - 3.5.1 Routine Instrumentation . . . 22
- 3.6 Possible Enhancements . . . 25
- 3.7 Conclusion . . . 25

4 CRITICAL PATH ANALYSIS . . . 26
- 4.1 Introduction . . . 26
- 4.2 Overview of the Evaluation Method . . . 27
- 4.3 Implementation . . . 28
  - 4.3.1 Measuring the Sequential Execution Time of the Parallelized Program . . 30
  - 4.3.2 Parallel Constructs Produced by KAP/Concurrent . . . 31

4.3.3 Measuring the Parallel Execution Time of the Parallelized Program . . . 32
4.3.4 Shadow Variables and Operation-Level Implicit Parallelism 34
4.3.5 Shadow Variables and Loop-Level Implicit Parallelism 35
4.3.6 Declaration of Shadow Variables 36
4.3.7 Subroutine and Function Calls 38
4.3.8 Control Dependence 38
4.4 Experimental Evaluation of KAP/Concurrent 39
4.4.1 Speedup for the Perfect Benchmarks 39
4.4.2 Speedup for Numerical Recipe Kernels 42
4.5 Conclusion 44

5 PARALLEL ACTIVITY MEASUREMENTS 46
5.1 Introduction 46
5.2 Overview of the Evaluation Method 46
5.3 Histogram Generation 47
5.3.1 Histogram Buckets 47
5.4 Histogram Generation from Loop Iterations 48
5.4.1 Natural Interval Storage 49
5.4.2 Heuristics for Interval Measurements from Loop Iterations 50
5.5 Average Parallelism for Limited Processors 50
5.5.1 Upper Bounds of Parallel Execution Time 51
5.5.2 Lower Bounds of Parallel Execution Time 51
5.5.3 Using Larger Time Quanta 54
5.5.4 Algorithms for Larger Time Quanta 55
5.5.5 Limiting the Resources Used by the Execution 55
5.6 Average Parallelism Results 59
5.6.1 Perfect Benchmarks Results 59
5.7 Conclusion 61

6 STRESS ANALYSIS 70
6.1 Introduction 70
6.2 Overview of the Evaluation Method 71
6.3 Results 72
6.3.1 Stress Analysis Examples 76
6.4 Conclusion 76

7 INDUCTION VARIABLES 79
7.1 Introduction 79
7.2 Overview of the Evaluation Method 81
7.2.1 State Space for Variable Identification 82
7.2.2 Use of Transition Tables 85
7.2.3 Enhancement to the Model 86
7.2.4 Deficiencies of this Approach 88
7.3 Implementation 89
7.3.1 Invariant Variable Recognition 90
7.3.2 RPN Statement Form 91
7.3.3 Example Instrumentation 93

7.3.4 Inductions on Array Elements 95
7.3.5 Interprocedural Induction Variables 95
7.3.6 Correctness of the Upper Bound 96
7.4 Experimental Results from the Perfect Benchmarks 97
7.5 Conclusion 98

8 STATIC DEPENDENCE ANALYSIS 99
8.1 Introduction 99
8.2 Description of the Experiments 100
8.2.1 Classification of Unanalyzable Potential Dependences 102
8.3 Experimental Results 103
8.4 Conclusion 104

9 DYNAMIC DEPENDENCE EVALUATION 108
9.1 Introduction 108
9.2 Overview of the Evaluation Method 108
9.2.1 The Ideal Parallel Machine 109
9.3 Data Dependence 109
9.3.1 Examples of Data Dependence from Programs 111
9.4 Critical Path Parallelism 112
9.4.1 Types of Parallelism 114
9.4.2 Dynamic Dependence Evaluation 114
9.4.3 Instrumentation for Dependence Evaluation 115
9.4.4 Constrained Execution 116
9.4.5 Serialization Caused by Variable Dependence Distance 120
9.5 Static Results 120
9.6 Effectiveness of Data Dependence Tests 123
9.7 Observations from the Perfect Benchmarks 123
9.7.1 Evaluation and Classification 125
9.7.2 Intraprocedural and Interprocedural Parallelism 125
9.7.3 Correlation with Interprocedural Parallelism 126
9.7.4 Future Enhancements 126
9.8 Conclusion 127

10 DYNAMIC DEPENDENCE ANALYSIS 128
10.1 Introduction 128
10.2 Overview of the Evaluation Method 129
10.2.1 Strategy for Dynamic Analysis 129
10.2.2 Dependence Distances 131
10.3 Uses of the Dynamic Dependence Analysis Method 132
10.4 Results from the Perfect Benchmarks 140
10.5 Conclusion 140

11 CONCLUSIONS AND FUTURE DIRECTIONS 145
11.1 General Observations 145
11.1.1 Dynamic Evaluation and Analysis 146
11.2 Observations on the Perfect Benchmarks 147
11.2.1 Poor Performance by KAP/Concurrent 147
11.2.2 Mediocre Performance by KAP/Concurrent 148
11.2.3 Good Performance by KAP/Concurrent 149
11.3 Future Directions 149

BIBLIOGRAPHY 150

VITA 154

Chapter 1

INTRODUCTION

Parallel processing evokes grand dreams of unlimited performance, of MIPS and MFLOPS. With graceful coordination each processor dutifully shares the work. No time is wasted; each processor acknowledges the requirements of the others and fulfills their needs as they become known. The execution pathway remains wide and uncluttered, enabling every processor to contribute equally in reducing the execution time.

But then the dream turns to a nightmare. Roadblocks appear along the way to delay and frustrate. Squabbles arise and tempers flare as each processor valiantly tries to obtain the resources required for its continued existence. Idle time and sloth run rampant as the pathway is constricted so that only one task may proceed.

The nightmare arises in part from not knowing precisely how information is shared. At first the program is an unfathomable collection of interrelated computations. Each computation seemingly is entangled with every other, allowing only a single thread of execution to continue. Through the illumination of the program with the available tools, structures begin to form, and independent computations arise. But the tools do not provide sufficient illumination to peer into the depths of the program, and portions remain obscured.

The tools that are available today combine to form the current state of parallelizing compilers. Researchers are forging ahead, extending the scope and increasing the illumination of which each tool is capable. Improvements in interprocedural analysis, dependence analysis, symbolic manipulation, and aliasing all point towards a greater understanding of the program and fewer portions that remain unknown.

The traditional method of restricting the analysis to static methods, disallowing feedback from the programs execution, may be unduly restraining progress in parallel processing. This thesis embarks upon a journey, charting unexplored areas and outfitting the programmer with tools for dispelling the darkness and bringing the landscape into view. The methods presented here not only provide additional information about the program's semantics, but also serve as a measure with which to compare existing static analysis techniques.

1.1 The Grand Plan

Static analysis determines what a program is capable of doing. The static analysis techniques used by parallelizing compilers give conservatively correct answers to semantic analysis problems. The static methods must consider every possible result of a specific program configuration. Dynamic analysis determines what a program actually does. It does not consider possibilities that never occurred during the program's execution. In this way, it can generate optimistic

answers that are still correct for a particular input data set. It is the subtle distinction between static and dynamic analysis that makes it possible to use the latter to measure the effectiveness of the static analysis methods.

The focus of this thesis is on discovering how dynamic analysis can be applied to enhance the understanding of program behaviors and the effects of parallelizing compilers on those programs. These two subjects are closely interrelated, and understanding one can lead to increased understanding of the other.

For example, parallelizing compilers rely on data dependence analysis to determine the structure of a program and to decide how the computations can be mapped onto a machine. Data dependences calculated dynamically may be more precise (i.e., give additional information) than those computed statically. The more precise dependence information may be a subset of the dependences that can exist in some execution of the program, however, it is still useful for two reasons. The first is hand tuning of the program through manual transformations. These transformations can be performed to extract the inherent parallelism. Manual transformations are necessary because the dynamically computed dependence information is no longer conservative and thus cannot be automatically applied without human intervention. The second usage of more precise data dependence calculations is to use the information to evaluate the existing static dependence tests. One can compare the dynamic data dependence calculations with the existing dependence calculations used in the parallelizing compiler to show where the existing techniques are lacking. It is the greater understanding of the program's semantics that solves both problems; knowing where to apply the transformations, and knowing when the static analyses are inadequate.

Parallelizing compilers are required to perform many analysis tasks in order to understand the intent of the programmer who specified the task to be accomplished by the program. The best static analysis techniques available are employed by the compiler in this quest. As innovations are developed, they are incorporated into the parallelizing compiler to improve its performance. Normally this constant improvement in parallelizing compilers would be a source of problems in evaluating the compilers. The parallelizing compiler may already be using the best available static techniques. It may be impossible to create an improved technique, within the same framework, against which to compare the methods used in the parallelizing compiler. Here we have an advantage, because we step outside the static framework and include dynamic analysis.

1.2 Understanding the Intent of a Program

Figure 1.1 shows the relationship among the experiments reported in Chapters 3 to 10. The relationship represented is one of specialization. The descendants of an experiment or method in Figure 1.1 either rely on the results of the parent or compute a more precise result than the parent. The figure shows that all the experiments rely on the Delta Program Manipulation System, described in Chapter 3, for support services.

Delta is designed to be a prototyping environment for constructing source-to-source instrumenting systems and parallelizing compilers for Fortran. Static dependence analysis is an intrinsic component of Delta. An experiment describing the effectiveness of static dependence analysis is described in Chapter 8.

A dynamic variation of the static dependence analysis is described in Chapter 10. This method of dynamic dependence analysis is performed through an instrumenting transformation

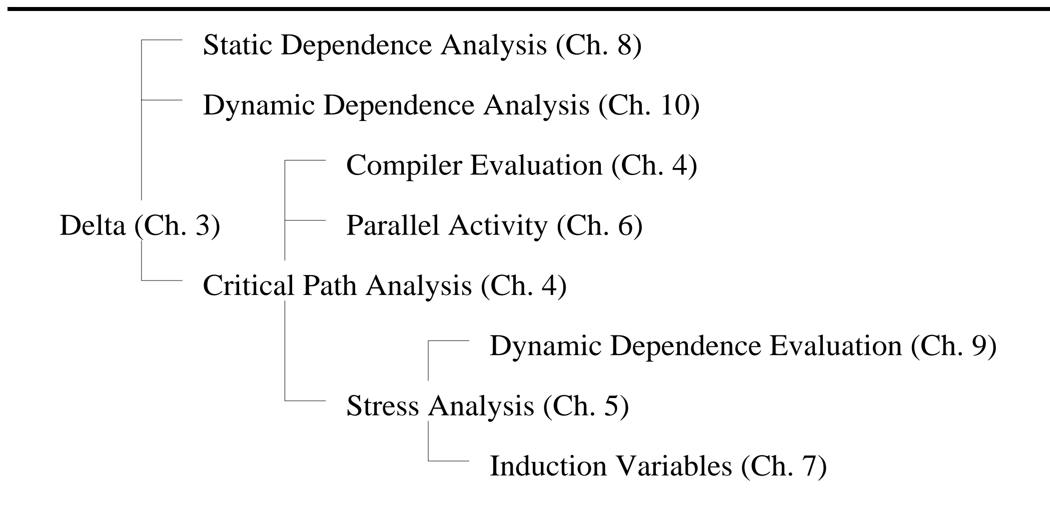

Figure 1.1: Relationship of experiments

written as a Delta function. One possible use of dynamic dependence analysis is to gain an understanding of the information that is passed between loop iterations, without regard to subroutine boundaries. Dynamic dependence analysis generates a report of the *flow*- and *output*-dependences categorized by the loop that carries the dependence. The information generated by the dynamic dependence analysis is more optimistic than the static analysis, since it reports only the dependences that occurred, but it is more precise because it can also report interprocedural dependence information that is lacking in many existing parallelizing compilers. Along with the existence of a dependence, the distances of the carrying loop are recorded. The information reported by dynamic dependence analysis is useful in isolating the variables that are shared across different loop iterations. Once the shared references are properly synchronized, the loop can run concurrently.

Chapter 4 describes the principle of critical path analysis, a dynamic analysis technique upon which the rest of the experiments are built. Critical path analysis is a technique that allows the minimum parallel execution time to be calculated. This is accomplished by scheduling, in a multiprocessor with an unlimited number of processing elements, each operation in the program as early as possible, subject to all existing data and control dependences. It is assumed that the only data dependences that need to be considered are the *flow*-dependences. All other data dependences are ignored under the assumption that a good compiler can remove them.

Also in Chapter 4, the execution time of automatically parallelized programs is simulated under the same assumptions as used by the critical path analysis. The comparison of the average parallelism, obtained by the parallelizing compiler, with the inherent parallelism allows the effectiveness of an existing parallelizing compiler to be determined.

Another use of the data generated by critical path analysis is described in Chapter 5. The average parallelism (or speedup) numbers generated by critical path analysis are with respect to an unlimited number of processors. If the number of processors is restricted, then the performance will decrease. Chapter 5 describes an experiment to generate bounds on the average parallelism under the constraint of limited processing resources.

Other forms of dynamic analysis can be built on top of critical path analysis. Each form is constructed as a tool to measure some aspect of the program. The most general tool developed is that of stress analysis. Chapter 6 describes the concept of stress analysis and demonstrates an implementation of a tool to measure the internal stress in a program. Any dependence which differentially introduces a delay into a program's execution causes stress. Stress analysis does not differentiate between the causes of the stress; it only indicates where the stress occurred.

Stress analysis can be categorized as a tool for determining important segments of the program's critical path. In its pure form, stress analysis measures a program against itself. It does not consider the effects of any outside influence such as the transformations performed by a parallelizing compiler. In particular, the stress analysis tool can measure two types of stress: (a) stress induced by data dependence and (b) stress induced via an ordering constraint. Recurrences are an example of stress induced by a data dependence. Stress that is induced via an ordering constraint may be removed by changing the lexical order of the statements. This type of stress is found when a control dependence has a greater influence than a data dependence.

Additional constraints must be added to the stress analysis technique to create other tools that measure against the background of a parallelizing compiler. This approach is applied to dependence evaluation in Chapter 9. Dynamic dependence evaluation compares the static data dependence analysis with the true dependences that are encountered at run-time. This is another form of stress analysis, where the differential of the data dependence arcs act as the forces causing stress in the program.

An induction variable is any variable defined inside a loop whose value at any point in time is a function of the loop index and the loop invariant values. The results of stress analysis in Chapter 6 include implicitly the effects of induction variables but also include many other effects. The general technique of stress analysis locates only the source of the stress and does not predict the effects of removing the cause of the stress. Chapter 7 describes a tool that modifies the general technique of critical path analysis to not only report the location and cause of stresses induced by induction variables, but also to calculate an upper bound on the effect of removing the calculation of the variable.

As with most of the other experiments, induction variable evaluation has two uses. It can be used to understand the program in isolation, by locating all existing induction variables, or it can be used to evaluate a parallelizing compiler. Assuming that the parallelizing compiler has eliminated all the induction variables that it was capable of eliminating, then any induction variables remaining must have been missed by the parallelizing compiler.

1.3 Amdahl's Law

When one talks about the prospects of parallel processing, it is important to determine if any potential exists at all. Amdahl's Law [Amd67] states that the parallel portion of a program dictates the potential speedup. Equation (1.1) describes the relationship between the parallel composition of the program and the speedup. The limit as the number of processors approaches infinity is shown in Equation (1.2):

$$S_p = \frac{T_1}{T_p} = \frac{T_1}{T_1\left[(1-F_p)+\frac{F_p}{p}\right]} \tag{1.1}$$

$$\lim_{p \to \infty} S_p = \frac{1}{1 - F_p} \quad (1.2)$$

where S_p is the speedup on p processors, T_1 and T_p are the sequential and parallel execution times, and F_p is the fraction of parallel code in the program.

Thus, for a fixed F_p, the speedup is constant. The job of the parallelizing compiler is to restructure the program to maximize the fraction of the program (F_p) that can be executed in parallel. It is the intent of this thesis to provide methods for increasing our understanding of program analysis tools in the hope of improving performance.

However, the assumption that F_p is constant may not be realistic for some problems [GMB88]. As an example, if we **assume** that the sequential portion of the program grows at αN^k and the parallel portion grows at βN^{k+1}, then the resulting fraction of parallelism can be expressed as:

$$F_p = \frac{\beta N^{k+1}}{\alpha N^k + \beta N^{k+1}} \quad (1.3)$$

$$= \frac{N}{N + \alpha/\beta} \quad (1.4)$$

Taking the limit, from Equation (1.6), as the number of processors approach infinity gives Equation (1.7).

$$\lim_{p \to \infty} S_p = \frac{1}{1 - F_p} \quad (1.5)$$

$$= \frac{1}{1 - \frac{N}{N+\alpha/\beta}} \quad (1.6)$$

$$= \frac{\beta}{\alpha} N + 1 \quad (1.7)$$

If the parallel component of the execution time grows at a rate $O(N)$ faster than the rate of growth of the sequential component, then the potential speedup will also grow at a rate of $O(N)$ as demonstrated by Equation 1.7.

Therefore, all is not lost. Even if the parallelizing compiler is unable to change the fraction of parallel computation in the code, we may be able to change the fraction of parallel computation by changing the data-set size. Parallel processing wins both ways: by using a parallelizing compiler to discover more parallel computation and through larger problems when the parallelism scales with the data-set size.

Chapter 2

BACKGROUND

2.1 Dependence Analysis

A data dependence is said to exist if the order in which two statements are executed affects the results of the statements. The simplest case arises when the interaction is due to scalar variables. For example, the following two consecutive statements

S_1: `A = B + C`
S_2: `D = E + F`

could be executed in any order, or in parallel, without affecting the result of the program because they access different variables, and therefore do not interact. However, the statements

S_1: `A = B + C`
S_2: `D = A + 1`

must be executed in the order specified by the program because S_2 uses a value computed by S_1. In these cases, we say that there is a dependence from S_1 to S_2, which indicates that S_2 must execute after S_1 to guarantee correctness.

A dependence exists from S_1 to S_2 if S_1 accesses a variable modified by S_2 as illustrated by

S_1: `A = B + C`
S_2: `B = E + F`

or if both statements modify the same variable:

S_1: `A = B + C`
S_2: `A = E + F`

The dependence relation is antisymmetric. A dependence from S_1 to S_2 does not imply the existence of a dependence from S_2 to S_1. A necessary condition for a dependence to exist

from a statement S_1 to a statement S_2 is that an execution of S_1 interacts with an execution of S_2, and that the execution of S_1 precedes the execution of S_2 in the sequential version of the program.

Detecting dependences due to scalar variables is relatively simple. However, the study of data dependence when the interaction involves arrays is more complicated. Most of the work published on dependence analysis (including the tests considered in this chapter) focus on statements with array references and assume that the two statements to be analyzed are both inside the same (multiply-nested) `DO` loop. It is also possible to assume, without loss of generality, that the loops are normalized; that is, the loop's lower limit is zero and the step is one.

Consider the loop:

$$
\begin{array}{ll}
 & \texttt{DO}\ I_1 = L_1, U_1 \\
 & \quad \ldots \\
 & \qquad \texttt{DO}\ I_d = L_d, U_d \\
S_v: & \qquad\quad X(f_1(I_1,\ldots,I_d), f_2(I_1,\ldots,I_d), \ldots, f_n(I_1,\ldots,I_d)) = \ldots \\
S_w: & \qquad\qquad \ldots = X(g_1(I_1,\ldots,I_d), g_2(I_1,\ldots,I_d), \ldots, g_n(I_1,\ldots,I_d)) \\
 & \qquad \texttt{END DO} \\
 & \quad \ldots \\
 & \texttt{END DO}
\end{array}
$$

Here, X is an n-dimensional array, and f_i and g_i are functions from $\mathbf{Z}^d$ to $\mathbf{Z}$.

To decide whether there is a dependence from S_v to S_w it is necessary to determine whether there are two executions of S_v and S_w such that

(1) The execution of S_v takes place in iteration $\bar{I}' = (I'_1, I'_2, \ldots, I'_d)$

(2) The execution of S_w takes place in iteration $\bar{I}'' = (I''_1, I''_2, \ldots, I''_d)$

(3) $\bar{I}' \leq \bar{I}''$

(4) $f_i(\bar{I}') = g_i(\bar{I}'')$ for all $(1 \leq i \leq n)$ and for some $I'_1, I'_2, \ldots, I'_d, I''_1, I''_2, \ldots, I''_d$ within the loop limits specified in the program.

The conditions that determine a dependence from S_w to S_v are the same except that condition (3) is replaced by (3′): $\bar{I}'' < \bar{I}'$.

The problem of determining dependence is sometimes decomposed into several subproblems, one for each possible ordering relationship between the components of the vectors $\bar{I}'$ and $\bar{I}''$. For example, if $\bar{I}', \bar{I}'' \in \mathbf{Z}^2$, the condition $\bar{I}' < \bar{I}''$ can be decomposed into four cases:

$$
\begin{array}{lcl}
I'_1 < I''_1 & \text{and} & I'_2 < I''_2 \\
I'_1 < I''_1 & \text{and} & I'_2 = I''_2 \\
I'_1 < I''_1 & \text{and} & I'_2 > I''_2 \\
I'_1 = I''_1 & \text{and} & I'_2 < I''_2
\end{array}
$$

In general, these cases are specified using direction vectors which are of the form $\Psi = (\psi_1, \ldots, \psi_d)$ where each ψ_k is one of $<$, $>$, or $=$ and represents the ordering relation between

I'_k and I''_k. In the case of the previous example, the direction vectors are $(<,<)$, $(<,=)$, $(<,>)$, and $(=,<)$.

There is a *potential dependence* for each pair of array references within the body of the loop if at least one of them is on the lefthand side of an assignment statement. For each potential dependence it is necessary to invoke a dependence test to determine whether or not an actual dependence exists. When the test determines that no actual dependence exists, it *breaks* the potential dependence.

A potential dependence involves *coupled subscripts* if it can only be broken by simultaneously considering the subscripts in a multidimensional array reference. Of the tests described in this chapter, only the generalized GCD and the integer programming tests are able to handle coupled subscripts. Other dependence tests have been developed specifically to handle the coupled subscript case and are described in the literature [LYZ89, WT92].

The next two sections describe in detail the data dependence tests considered in this thesis. Section 2.2 lists the simple and approximate data dependence tests. These tests are used because they are quick and break most of the potential dependences. Section 2.3 lists data dependence tests based on two variations of integer programming. These dependence tests can be used when the approximate techniques cannot break the dependence.

2.2 Simple and Approximate Dependence Tests

When f_i and g_i are linear functions and the loop limits are known at compile-time, the data dependence problem described above can be solved exactly through integer programming techniques. However, for reasons of efficiency it is usually solved through one of the approximate solution techniques described below. Also, some of the tests (constant, GCD, and Banerjee infinity) can be applied even when the loop limits are not known. All the techniques discussed here require that f_i and g_i be linear functions of the iteration vector $\bar{I}$, and that all the coefficients and constant terms in these functions be known at compile-time. Alternatively, research into symbolic dependence analysis [Hag90] has shown that it may be possible to extend the reach of dependence analysis to cover subscripts whose coefficients are not compile-time constants.

The goal of all the approximate techniques described below is to break potential dependences. Except for the generalized GCD test, these techniques work one subscript at a time. A potential dependence is broken only if the test shows that, for some i between 1 and n, there are no index vectors $\bar{I}'$ and $\bar{I}''$ that satisfy the equation $f_i(\bar{I}') = g_i(\bar{I}'')$. Doing the test independently for each subscript is conservative because the system of equations may not have a solution even if each equation does. The constant and GCD tests do not use direction vectors. The other tests, all derivatives of Banerjee's test, are applied once for each direction vector of a potential dependence.

2.2.1 Constant Test

The constant test has been singled out for special consideration because it has an important characteristic: it can prove dependence. If all the subscripts in the two array references are loop invariant and have the same value, then there will always be a data dependence for all potential direction vectors. If any pair of corresponding subscripts are constant and different, then there will never be a data dependence regardless of the values of any other subscript.

Let A and B be references to the same scalar variable, perhaps via equivalence. Given two statements S_1 and S_2 in a loopnest, where a scalar variable A is written to in S_1 and a scalar variable B is read from in S_2, it is easy to determine if a dependence exists. The first step is to determine whether control can flow from S_1 to S_2. If it is impossible for the definition of A to reach the use of B, then there is no dependence. If the memory locations referred to by A and B are identical and control can flow from S_1 to S_2 then there is a dependence. The two locations can be the same for two reasons, either the references A and B refer to the same name in the same lexical scope, or A and B are statically aliased or equivalenced.

The constant test for array references is a simple extension of the scalar dependence test. Assume we have statements S_1 and S_2 (where control flows from S_1 to S_2) that have a write to the array $\mathrm{X}(p_1, \ldots, p_n)$ in S_1 and a read from the array $\mathrm{X}(q_1, \ldots, q_n)$ in S_2. If a pair of subscripts p_k and q_k are invariant in the loopnest and it is statically determinable that $p_k \neq q_k$, then the two references are proved independent.

If it is proven that for all k in $[1 \ldots n]$, $p_k = q_k$, then we have proven that the two array references are dependent. Simple examples of this test are to compare the array references `X(1)` with `X(2)` proving independence, and the array references `X(M+1,N)` with `X(M+1,N)`, where `M` and `N` are loop invariant, proving dependence.

2.2.2 GCD and Generalized GCD Tests

The greatest common divisor (GCD) test establishes an existence criterion for the solution to integer diophantine equations. This test states that if the greatest common divisor of the coefficients of the equation divide the constant term, then a solution exists. Conversely, if it does not divide the constant term, then no solution can exist. The following two examples illustrate this test.

$X(2*I)$ and $X(2*J+1)$	$X(2*I)$ and $X(6*J+4)$
$2*I-2*J=1$	$2*I-6*J=4$
$\gcd(2,2)=2$	$\gcd(2,6)=2$
2 does not divide 1	2 divides 4
No dependence	Undecided

In the first column it is determined that the equation generated from the subscripts is inconsistent and has no solution. Thus, it is impossible for a dependence to exist between the two array references because they can never access the same memory location. However, in the second column, we have shown that a solution exists to the dependence equation. If this solution lies in the bounds of the iteration space of the enclosing loops, then dependence is proven. Otherwise, if the solution lies outside the bounds of the iteration space, then no dependence exists. In either case, checking the loop bounds is outside the scope of the GCD test.

The generalized GCD method is described in [Ban88] and is an extension of the GCD method that considers all subscripts in a multiply dimensioned array simultaneously. To illustrate this algorithm, consider the model loopnest shown in Section 2.1. The condition for the existence of a data dependence is that $f_k(\bar{I}') = g_k(\bar{I}'')$ for all k in $[1 \ldots m]$ for some value of $\bar{I}'$ and $\bar{I}''$ in the iteration space. Since only linear functions are considered, we can rewrite $f_k(\bar{I}') = A_{k0} + \sum_{i=1}^{d}(A_{ki}I_i')$ and we can rewrite $g_k(\bar{I}'') = B_{k0} + \sum_{i=1}^{d}(B_{ki}I_i'')$ where A_{ki} and B_{ki} are the ith coefficient of the kth equation. Therefore, the requirement that $f_k(\bar{I}') = g_k(\bar{I}'')$ can then be expressed as $\sum_{i=1}^{d}(A_{ki}I_i' - B_{ki}I_i'') = (B_{k0} - A_{k0})$.

The final equation is actually a system of linear diophantine equations, one for each subscript position (i.e., value of k). A requirement that must be satisfied for the existence of an integer solution is that the greatest common divisor of the coefficients of the lefthand side must divide the righthand side for all equations. A solution exists iff $\gcd(A_{k1}, A_{k2}, \cdots, B_{k1}, B_{k2}, \cdots)$ divides $(B_{k0} - A_{k0})$.

If an index space $\bar{x}$ is defined as $(\bar{I}' \ \bar{I}'')$, then the equations can be rewritten in the form $\bar{x}A = \bar{b}$ where A is the collection of the coefficients of each equation, and $\bar{b}$ are the righthand sides of the equations. The matrix A can always be factored into a unimodular matrix U and an echelon matrix D (with element $d_{11} > 0$) such that $UA = D$ [Ban88]. The determinant of a unimodular matrix is either $+1$ or -1, and an echelon matrix is upper triangular.

The system of diophantine equations described by $\bar{x}A = \bar{b}$ has a solution iff there exists an integer vector $\bar{t}$ such that $\bar{t}D = \bar{b}$. Since D is an echelon matrix, the solution to $\bar{t}D = \bar{b}$ is much easier to determine than that of the original problem.

The generalized gcd test can only prove the independence of two array references. If a solution exists to the set of diophantine equations, we must assume dependence, even though we have no guarantee that the solution belongs to the iteration space of the loopnest.

2.2.3 Banerjee's Inequalities

The mathematical basis for Banerjee's inequalities is derived from the *Intermediate Value Theorem* that states: Let $\mathcal{F}$ be a continuous real valued function in $\mathbf{R}^n$. Let $B_{\min}, B_{\max}$ denote any two values of $\mathcal{F}$ on a connected set $\Re \subset \mathbf{R}^n$, and suppose that $B_{\min} \leq c \leq B_{\max}$. Then the equation $\mathcal{F}(x) = c$ has a solution $x \in \Re$. Conversely with $B_{\min}$ and $B_{\max}$ the minimum and maximum values respectively of $\mathcal{F}$ in $\Re$, if $B_{\min} \leq c \leq B_{\max}$ is not true, then no solution can exist and independence has been proven. Data dependence tests based on this formulation of Banerjee's inequalities cannot prove dependence since only the existence of a real valued solution has been proven which does not always imply the existence of an integer valued solution.

Banerjee's test is defined as follows [WB87]. Given functions f and g and a direction vector Ψ, it must be shown whether $f(\bar{I}') = g(\bar{I}'')$ can hold for any $\bar{I}'$, $\bar{I}''$ under the constraints of the direction vector Ψ and the loop bounds. In the following, the symbols A_k and B_k are the coefficients of the subscripts being analyzed, i.e., $f(\bar{I}') = A_0 + \sum A_k I'_k$ and $g(\bar{I}'') = B_0 + \sum B_k I''_k$. The dependence equation can be stated as follows:

$$\sum_{k=1}^{d} (A_k I'_k - B_k I''_k) \quad = \quad B_0 - A_0$$

For each value of k and corresponding direction ψ_k ($\psi_k \in \{<, >, =\}$), we find a lower and upper bound such that:

$$LB_k^{\psi_k} \leq A_k x_k - B_k y_k \leq UB_k^{\psi_k}$$

under the assumption that x_k and y_k are any real values subject to:

$$\begin{array}{ccc} L_k \leq & x_k, y_k & \leq U_k \\ x_k & \psi_k & y_k \end{array}$$

Summing these bounds results in the inequality:

$$\sum_{k=1}^{d} LB_k^{\psi_k} \leq \sum_{k=1}^{d}(A_k x_k - B_k y_k) \leq \sum_{k=1}^{d} UB_k^{\psi_k}$$

or, equivalently

$$\sum_{k=1}^{d} LB_k^{\psi_k} \leq B_0 - A_0 \leq \sum_{k=1}^{d} UB_k^{\psi_k}$$

If either

$$\sum_{k=1}^{d} LB_k^{\psi_k} > B_0 - A_0 \quad \text{or} \quad \sum_{k=1}^{d} UB_k^{\psi_k} < B_0 - A_0$$

hold, then the functions cannot intersect under the constraints of the direction vector. Clearly, this formulation of Banerjee's test requires that all loop limits be known at compile-time.

When the loop bounds are all constant, the test is called Banerjee rectangular. When some of the loop bounds are a function of outer loop indices, the test is called Banerjee trapezoidal. For Banerjee trapezoidal, the bounds are no longer limited to $L_k \leq I_k \leq U_k$, but to the more general bounds

$$p_{k0} + \sum_{j=1}^{k-1} p_{kj} I_j \leq I_k \leq q_{k0} + \sum_{j=1}^{k-1} q_{kj} I_j$$

The constants p_{kj} and q_{kj} define the limits in the loop nest. Other than the more complicated equations involved, the trapezoidal variation is identical to the rectangular test.

Traditionally the most widely studied dependence tests are those based on Banerjee's inequalities. Banerjee [Ban88] presents a formalization of these ideas. The basis for this dependence test is the ability to compute the bounds for any linear function $\mathcal{F}$.

In Banerjee's test, each subscript of a multidimensional array reference is evaluated independently. Thus, this test is not able to determine independence correctly when analyzing coupled subscripts. Coupling occurs when the dependence can only be analyzed fully by simultaneously considering all subscripts in the array references.

2.2.4 Banerjee Infinity Test

For program parallelization the dependence tests must be conservative. They must report all dependences and assume dependence if independence cannot be proven. Another way of stating this criterion is that the true dependence arcs for a given iteration space are always a subset of the arcs present in any iteration space that is a superset of the given iteration space. Thus, increasing the iteration space does not violate the conservative criteria.

Given an integer i in the subset of the integers allowable by the Fortran language, define $\mathcal{M}$ to be an arbitrarily large integer such that the relationship $-\mathcal{M} < i < \mathcal{M}$ holds. We expand this notation to include any arbitrary linear function $f(i)$ involving integers legal in Fortran programs, of the integer i so that the relationship $-\mathcal{M} < f(i) < \mathcal{M}$ is true.

For any normalized loop where the loop index is bounded by $0 \leq I \leq N$ with the assumption that $0 \leq N$, we can rewrite this constraint as $0 \leq I \leq \mathcal{M}$. Since the iteration space $[0 \ldots N]$ is a subset of the iteration space $[0 \ldots \mathcal{M}]$, any dependence arcs found over $[0 \ldots N]$ will also be found in the iteration space $[0 \ldots \mathcal{M}]$.

The infinity test is the same as the rectangular Banerjee test if one assumes that all loops are normalized, to have a lower bound of 0 and an upper bound of $\mathcal{M}$. The appropriate changes are made in the Banerjee rectangular algorithm to accomplish this.

2.3 Integer Programming Based Dependence Tests

Data dependence analysis of linear array references is equivalent to deciding if there is an integer solution to a set of linear equalities and inequalities [SM89]. Therefore, it makes sense to consider using integer-programming-based methods in the experiments. Mathematical programming whether linear or integer requires the use of an objective function. The goal is either to minimize or maximize an objective function. Dependence analysis is interested in the existence or non-existence of a solution. Thus, the choice of the objective function has little impact, but may alter the order in which the iteration space is searched.

The integer programming problem can be stated as: Does there exist $\bar{x}$ such that $A\bar{x} = \bar{b}$, $B\bar{x} \geq 0$, $\bar{x} \geq 0$, for integer $\bar{x}$. In this formulation, the matrix A refers to the coefficients of the equalities, and the matrix B refers to the coefficients of the inequalities describing the bounds of the iteration space.

General integer programming techniques have many advantages over the approximations mentioned in the previous sections. The ability to consider simultaneously all subscripts of an array reference allows this dependence test to analyze coupled subscripts correctly. Affine loop bounds are incorporated naturally into the inequality matrix. Execution constraints such as covering conditionals can be introduced into the dependence equations.

In the literature, it has been reported that the linear programming approximation to the full integer programming algorithm is sufficient in most cases [MHL91].

2.3.1 Simplex Based Integer Programming Test

Several methods are available to solve the integer programming problem. One method of implementation is the Branch and Bound algorithm. This algorithm works by first solving the real valued linear programming problem using the simplex method. The solution is checked to see if it is integral. The first non-integer component (x_i) is selected and is used to create two new integer programming problems with new constraints. The first problem is the same as the original problem with the additional constraint $x_i \leq \lfloor \text{value of } x_i \rfloor$. The second problem is the same as the original problem with the additional constraint $x_i \geq \lceil \text{value of } x_i \rceil$. This constraint process generates a binary tree of problems that repeatedly divide the iteration space.

In effect, the Branch and Bound algorithm does an exhaustive search of the iteration space; however, it optimizes the search. Any region of the iteration space that does not have at least a real valued solution will not be searched for an integer solution. Once an integral solution is found, the process stops and reports success. The implementation of the Branch and Bound algorithm reported here requires that all variables have non-negative values, and that the iteration space is bounded.

2.3.2 Omega Test

The Omega Test [Pug91] is an extension of the Fourier-Motzkin linear-programming algorithm allowing integer constraints on the solution vector. In addition to supporting the full capabilities of integer-programming, the Omega Test also permits the systematic handling of unknown additive terms. Consider the subscripts `X(I+N)` and `X(I')` where $1 \leq I, I' \leq N$ (`I` and `I'` are lexically the same variable). The Omega Test supports the addition of `N` as an unknown constrained variable. After the addition of the loop limit, we find that the system of equations is inconsistent since $I+N \neq I'$ for all `I`, `I'` in [`1...N`].

We will use the Omega Test as the final data dependence test in the experiments. Currently, the Omega Test is the most accurate data dependence test with a practical implementation in existence.

Chapter 3

THE DELTA PROGRAM MANIPULATION SYSTEM

3.1 Introduction

The ability to investigate a new concept rapidly can enhance the productivity of a researcher. By allowing new ideas to be quickly prototyped, the researcher can exhibit the feasibility of hypothetical ideas and demonstrate their practicality. This chapter describes a prototyping tool that can be used to explore concepts in restructuring compilers.

We believe that providing compiler writers with a prototyping tool will accelerate the development of more robust compilers and new applications for the compiler techniques. In addition, such a tool can serve as an open experimental laboratory to augment the textual curriculum in compiler construction.

The initial implementation of the Delta Program Manipulation System (Delta) [Pad89] is targeted to the Fortran 77 source language. The core of Delta is an open system of functions embedded in the SETL language [Sny90, Lev89]. A Delta user composes these functions to create various kinds of "value-added" Fortran 77 pre-processors.

The inspiration for Delta came in part from the symbolic algebra environments that have evolved over the years. Systems such as Macsyma, Reduce, Maple and Mathematica create an environment to interactively explore mathematical equations. A user enters an equation into the system, applies a transformation, and is immediately presented with the result. This instant feedback allows the method of experimentation used to be incrementally altered depending on the result from the previous operation.

Delta has been designed to perform the same role as a symbolic algebra system, but for a different problem domain. A Fortran compilation unit will replace the equation as the elemental object in Delta. Function application is the method used in Delta for applying transformations to each compilation unit.

Another source of inspiration derives partly from Knuth's dream of programming by means of languages+program manipulating editors. It is possible to think of Delta as a programmable editor that performs semantic editing of Fortran compilation units.

Source-to-source transformations are a common way of adding functionality to a compilation unit. In the case of Delta, all transformations work with Fortran 77 plus annotations. With the addition of dynamic memory allocation to the target language, it is possible to express all other transformations as annotations to the source language. Many transformations are

possible at the source code level, including some common scalar optimizations and source annotations to indicate vectorization or parallelization. Instrumenting a compilation unit can also be viewed as a program transformation. This chapter will illustrate the application of Delta to an instrumentation task.

3.2 The SETL Language

SETL was developed at the Courant Institute [SDDS86] for the purpose of facilitating program development using standard mathematical expressions to operate on the intrinsic data types. In SETL and the examples that follow, the '$' symbol will be used to start a source comment that extends to the end of the line. The elemental data types in SETL include booleans, unrestricted integers, reals, character strings and functions. The symbol `OM` is defined to represent an undefined value. It is returned as an error condition by operations that have an undefined result and generally can be used to indicate an unknown or undefined value. The compound

```
Tuples: [1, 4, 9, 16];                                $ First four perfect squares.
Sets:   {2, 7, 3, 5};                                 $ First four prime numbers.
Maps:   {["one", 1], ["two", 2], ["three", 3]};       $ Function from names to numbers.
```

Figure 3.1: Example SETL datatypes

data types in SETL, as shown in Figure 3.1, are the ordered collection called a tuple, and a unique unordered collection called a set. Sets and tuples may contain any datatype as elements. A special kind of set is called a map. A map is a set of 2-tuples whose first elements are unique. A map can behave like a discrete function from the first element of each tuple to the corresponding second element. Maps are used in Delta to represent databases such as symbol tables, statement tables, and property lists of various program constructs.

SETL is an imperative language whose control constructs follow the Algol family of languages. However, SETL has also inherited many properties from Lisp. Functions in SETL are first class objects, arguments are passed by value, and data are stored in dynamic heap allocated memory with garbage collection.

SETL supports multiple assignment using a tuple as the lefthand side of an assignment statement. The statement `[a, b] := [1, 2];` is similar to the two statements `a := 1; b := 2;`. However, notice that the righthand side is evaluated in its entirety before the lefthand side; thus, to swap two variables the statement `[x, y] := [y, x];` would be sufficient.

The current implementation of Delta is written in a restricted subset of two dialects of SETL. The syntax of the language implemented in the ISETL2 [Lev89] interpreter is used to create the source code except where similar constructs are not available in the SETL2 [Sny90] compiler. Thus, the language used is the intersection of the capabilities available in the implementations of ISETL2 and SETL2. Other features of SETL that are needed in this chapter will be described as they are used.

3.3 Organization of Delta

Delta has two main components. The first is a scanner, which translates a sequence of syntactically correct Fortran 77 compilation units into a sequence of SETL data objects referred to in this section as program objects. The second component of Delta is a collection of SETL functions that operate on components of these translated compilation units (or program objects).

The organization of the Delta program object, which represent a program compilation unit (subroutine, function, block data, or main program), is a nested collection of SETL data objects. The Delta program object is designed to be self-documenting and verbose (even wordy), at the great expense of efficiency. To accomplish the documentation, field names in the program object are represented as character strings. The top-level structure of this object is a map from the domain {`"symtab"`, `"routine_type"`, `"initial_statement"`, `"expression"`, `"statements"`} onto their values. Other optional fields may also be present at the top-level as required to represent features of the Fortran source code.

Each value of the top-level structures in the program object belong to a set of possible values. The field `"routine_type"` is a member of the set {`"ROUTINE"`, `"FUNCTION"`, `"PROGRAM"`, `"BLOCKDATA"`}. The `"initial_statement"` field contains a statement tag that names the lexically first statement in the (otherwise unordered) statement table. The `"symtab"` field is a database containing the program's symbol table. This database is implemented as a map from each identifier onto a map describing the properties of that identifier. The `"expression"` field is a tuple of expression nodes containing explicit trees for the expressions represented in the original Fortran source code. The last required field is the `"statements"` database. The `"statements"` database is a map from statement tags onto maps describing the properties of each statement.

Statement tags are an arbitrary character string. The scanner creates tags of the form `"Sn"` where `n` ranges from 1 to the number of statements in the compilation unit. Therefore, `"S1"`, `"S2"`, and `"S34"` are all examples of statement tags.

```
[{["op","INTEGER_CONSTANT"],["value",123],["type","INTEGER"]}, $ expression (1)
 {["op","INTEGER_CONSTANT"],["value",321],["type","INTEGER"]}, $ expression (2)
 {["op","+"],["type","INTEGER"],["args",[1,2]]}]               $ expression (3)
```

Figure 3.2: Exploded form of expression `123+321`

```
[{["op","+"],["type","INTEGER"],["args",
  [ {["op","INTEGER_CONSTANT"],["value",123],["type","INTEGER"]},
    {["op","INTEGER_CONSTANT"],["value",321],["type","INTEGER"]}]]}]
```

Figure 3.3: Imploded form of expression `123+321`

Two forms of expressions are maintained inside Delta. We can refer to these forms as the "exploded" form (Figure 3.2), stored in the expression table, and the "imploded" form (Figure 3.3), used by most Delta functions. Delta maintains the ability to operate on both forms for the flexibility that each form provides.

SETL has no primitive pointer datatype. Therefore, SETL cannot copy an explicit pointer to an object and then modify the object referenced by the pointer. Because of this lack, in the "exploded" form (Figure 3.2) the expression table indices are used as a form of pointer. The children of an expression node are indicated by their corresponding indices in the expression table. The "imploded" form (Figure 3.3) represents the same information as that contained in the "exploded" form except that instead of pointing to the node's children, the children are nested within the expression.

3.4 Using Delta to Instrument Fortran Programs

Instrumenting a source file to obtain program statistics is a common way of measuring a program's behavior. In this chapter, we discuss the instrumentation of a Fortran program. The instrumentation described next is a simple form of execution analysis that summarizes the time spent in each loop nest. More complex instrumentation methods are discussed in Chapters 4 through 10.

One common use of program instrumentation is to collect an execution profile. The particular execution profile considered in this chapter is the time spent in the lexical loop nests of a program. Determining the execution time of a loop nest is important because those loops that consume the largest portion of the execution time are prime candidates for optimization. It is possible to consider all loops in a program, but instrumenting inner loops may add significant overhead to the execution without providing any significant information. Thus, for this example we will only compute the execution time of the outermost loop. To simplify the instrumentation library, all instrumentation intervals (i.e., lexically outermost loops) are uniquely identified by an integer. A mapping from these integers to a line in the source code is required to generate an understandable report. The instrumentation revolves around an interval file that contains this mapping of interval numbers to source code locations. The interval file is generated during the instrumentation process, and it is used during the program's execution.

We now describe an example of how the final instrumentation will look. The instrumentation algorithm is described in the next section. Consider a program consisting of two files, `test.f` and `subs.f`. The file `test.f` shown in Figure 3.4 contains the main program `TEST`. Figure 3.5 lists the two subroutines, `IDENT` and `MATMUL`, that reside in the file `subs.f`. This example program initializes two matrices `B` and `C` to the identity matrix, multiplies the matrices, places the result in the matrix `A`, and outputs the result.

The Fortran parser used by Delta normalizes the input source code to minimize the syntactic differences between programs. In this example the `DO` loops are converted to the more modern `DO-ENDDO` form, all variables are explicitly declared, and the loop nesting structure is properly indented.

The instrumentation library we describe consists of four subroutines. The first initializes the interval structure for the program. A call to `INIT_INTERVALS(`*interval-file*`)` initializes and loads the name of each interval that will be encountered during the program's execution. The next subroutine, `EXIT_INTERVALS(`*summary-file*`)`, writes the summary of the execution information for each loopnest to *summary-file*. The other two subroutines, `START_INTERVAL(`n`)` and `END_INTERVAL(`n`)`, delineate the beginning and end of each outermost `DO` loop. A different integer n is assigned to each interval. Figures 3.6 and 3.7 reflect the result after instrumentation. The symbol $\Rightarrow$ is used to mark the statements that were added as part of the instrumentation.

```
      PROGRAM TEST
      PARAMETER (N=10)
      REAL A(N,N), B(N,N), C(N,N)
      CALL IDENT(B,N)
      CALL IDENT(C,N)
      CALL MATMUL(A,B,C,N)
      DO 1 J = 1, N
1     PRINT *, (A(I,J), I=1,N)
      END
```

Figure 3.4: The original source file `test.f`

```
      SUBROUTINE MATMUL(A,B,C,N)
      REAL A(N,N), B(N,N), C(N,N)
      DO 2 I = 1,N
      DO 2 J = 1,N
      X = 0.0
      DO 3 K = 1,N
3     X = X + B(I,K) * C(K,J)
2     A(I,J) = X
      RETURN
      END
```

```
      SUBROUTINE IDENT(A,N)
      REAL A(N,N)
      DO 4 I = 1, N
      DO 5 J = 1, N
5     A(I,J) = 0.0
4     A(I,I) = 1.0
      RETURN
      END
```

Figure 3.5: The original source file `subs.f`

```
        PROGRAM TEST
        EXTERNAL IDENT, MATMUL
        REAL A, B, C
        INTEGER I, J, N
        PARAMETER (N=10)
        DIMENSION A(N, N), B(N, N), C(N, N)
⇒       CALL INIT_INTERVALS('LOOPS.intvl')
        CALL IDENT(B,N)
        CALL IDENT(C,N)
        CALL MATMUL(A,B,C,N)
⇒       CALL START_INTERVAL(1)
        DO J = 1, N, 1
           PRINT *, (A(I,J),I=1,N)
        ENDDO
⇒       CALL END_INTERVAL(1)
⇒       CALL EXIT_INTERVALS('LOOPS.sum')
        STOP
        END
```

Figure 3.6: The instrumented source file `test.fi`

```
        SUBROUTINE MATMUL(A,B,C,N)
        REAL X, A, B, C
        INTEGER I, J, N, K
        DIMENSION A(N, N), B(N, N), C(N, N)
⇒       CALL START_INTERVAL(2)
        DO I = 1, N, 1
           DO J = 1, N, 1
              X = 0.0
              DO K = 1, N, 1
                 X = X+B(I,K)*C(K,J)
              ENDDO
              A(I,J) = X
           ENDDO
        ENDDO
⇒       CALL END_INTERVAL(2)
        RETURN
        END
```

```
        SUBROUTINE IDENT(A,N)
        REAL A
        INTEGER I, J, N
        DIMENSION A(N, N)
⇒       CALL START_INTERVAL(3)
        DO I = 1, N, 1
           DO J = 1, N, 1
              A(I,J) = 0.0
           ENDDO
           A(I,I) = 1.0
        ENDDO
⇒       CALL END_INTERVAL(3)
        RETURN
        END
```

Figure 3.7: The instrumented source file `subs.fi`

```
1       TEST:7
2       MATMUL:3
3       IDENT:14
```

Figure 3.8: Generated interval file, `LOOPS.intvl`

This type of program instrumentation is possible with the use of a wide variety of tools ranging from awk scripts to custom application programs. It is our intent to show that with the assistance of a prototyping environment, such as Delta, it is easy to synthesize a transformation from the primitive functions.

The file `LOOPS.intvl`, shown in Figure 3.8, contains the mapping of interval numbers onto interval names. For this example the first loop is at line 7 of the main program, the second loop is at line 3 of the file containing the routine `MATMUL`, and the third outermost loop is at line 14 of the second file that also contains `IDENT`.

3.5 Delta Function Definitions for Instrumentation

The instrumentation task, as defined in Section 3.4, is to add timing calls to all outermost loops in the program. Approaching this task in a top-down manner, we first define the driver to iterate over the Fortran source files and the compilation units included in each source file.

Delta provides a standard interface to the Unix command line when used as a compiled stand-alone application. The design of the top level driver will accommodate the compiled Delta interface and will also be useful as an interactive function. The compiled Delta interface calls a function named `main()` when executed, passing the Unix command line arguments as a tuple of character strings.

In this function, we wish to supply a tuple of Fortran source file names and have all the compilation units in each of the source files instrumented and written to files with the same root name as the original file name but with the file name extension of `".fi"` instead of `".f"`.

The first statement in the function (shown in Figure 3.9) defines the options passed to the scanner when it is invoked. The options variable contains a database of global configurations for Delta. The `"+line"` directive to the scanner instructs it to add source line numbering information to each program object. This directive gives us a unique correlation between statements in the Delta program object and lines in the original source file. This flag is disabled by default to conserve memory.

The `main()` routine is designed to accept an optional command line switch, `"-r"`. If this switch is present on the command line, then the required parameter to the switch is the root for the name of the interval file; otherwise the root will default to the file name `"LOOPS"`. The file name extension of the interval file will always be `.intvl"`. Thus if the `"-r"` switch is not specified, the interval file name defaults to `"LOOPS.intvl"`.

The Delta function `getopt()` mimics the behavior of the Unix library function of the same name [ATT]. In this example, the `getopt()` function scans the args tuple to find a switch of the form `"-r root"`. The function then removes the switch and its argument from the args list, installs `["root"]` as the value of the database flags for the index `"r"`, and returns the

```
main := func( args );
  local do_loops, f, fd, flags, i, root;

  options("scanner") := " +line ";

  [ flags, args ] := getopt( "r:", args );
  root := (flags("r") ? ["LOOPS"])(1);

  do_loops := [];

  for f in args do
      fd := openw( f + "i" );
      foreach_compilation_unit( f,
          func(pgm_unit);
              local   pgm;
              [pgm, do_loops] :=
                  instrument_program(pgm_unit,root,do_loops);
              write_program_fd( pgm, OM, fd );
          end);
      close( fd );
  end for;

  $  Create the file to map interval numbers to loop names
  fd := openw( root + ".intvl" );
  for f = do_loops(i) do
      writeln i, "    ", f to fd;
  end for;
  close( fd );

  return OM;
end;
```

Figure 3.9: Driver function for instrumentation process

database along with the updated argument list. This database is a set of two-tuples where the first element of the tuple is the switch character and the second element is the value.

Since we are requiring an argument for the "-r" switch, the value in the flags database will be a tuple of values. This behavior is defined because the command line switch may be specified multiple times. The statement `root := (flags("r") ? ["LOOPS"])(1);` finds the tuple for flag "-r" and assigns the first element of the tuple to the variable `root` otherwise if defaults to the name "LOOPS".

Next we initialize the variable `do_loops` to the empty tuple to signify that no loops have been encountered, and iterate over the files that remain in the argument list. In each iteration we first open a file for the instrumented program; then we call the generic routine to handle invoking the scanner on the Fortran source code.

The function `foreach_compilation_unit()` takes two arguments. The first argument is the name of the Fortran source file to load. The second argument is a function that will be applied to each compilation unit in the source file. The value returned by the SETL function `foreach_compilation_unit()` is a tuple of the values resulting from the application of the second functional argument. For this example, the responsibility of the invoked anonymous function is to instrument the compilation unit, accumulate the list of loops, and write the modified program to the open file descriptor.

This programming style takes advantage of the scope rules for variable nesting. Because the anonymous function was created inside the function `main()`, it has access to all of `main()`'s local variables. In this manner, the variable `do_loops` can be updated and communicated to the rest of the function without the use of a global variable.

As the last operation of the loop iteration, the file descriptor for the output source file is closed before processing the next input source file. To conclude the instrumentation the interval file must be written. We open the interval file, write out the index and name associated with all the loops that have been instrumented, and close the file.

3.5.1 Routine Instrumentation

Next, we describe the function (Figure 3.10) responsible for determining where to add the instrumentation calls. The arguments to this function are the compilation unit to instrument, `pgm`, the name of the intervals file, `root`, and a list of the loops already processed.

The first function call to the routine, `setup()`, handles analysis such as annotating the program object with a flow graph, and in-out sets for each statement. After the preliminary preparations are complete, a copy of the name of the compilation unit is stored in the local variable `name` to ease future use. If a name was not declared for the compilation unit, then the default will be the empty string.

Notice the special format of the expression using the `?` operator. As mentioned in Section 3.2, SETL uses `OM` as an object to represent an undefined value. The `?` operator returns its left operand if it is not `OM`; otherwise it returns its right operand. A more verbose and expensive method would be to use the statement `name := (if routine_name(pgm) = OM then "" else routine_name(pgm) end if);`.

The instrumentation we are performing requires that an interval file containing the mapping from interval identifiers to interval names be opened at the beginning of the program's execution and closed at the end of the program's execution. The call to "INIT_INTERVALS" should only be placed as the first executable statement in the main program. To find this location we check

```
$   Instrument a compilation unit to add the interval tracing statements.

instrument_program := func( pgm, root, do_loops );
    local   name, s, stmt;

    pgm  := setup(pgm);
    name := routine_name(pgm) ? "";

    if pgm("routine_type") = "PROGRAM" then
        s := first_ENTRY( pgm ) ? pgm("initial_statement");
        pgm := instrument_add_call( pgm, s, "INIT_INTERVALS",
                                    COMMA([CST("'"+root+".intvl'")]));
    end if;

    for s in list_stmts(pgm) do
        stmt := pgm("statements")(s);
        if stmt("st") = "DO" and stmt("outer") = OM then
            do_loops := do_loops with
                (if stmt("line") /= OM then
                    name+":"+integer_to_string(stmt("line"))
                else
                    name+":$"+integer_to_string(#do_loops+1)
                end if);
            pgm := instrument_add_call( pgm, preceding_stmt(pgm, s),
                        "START_INTERVAL", COMMA([CST(#do_loops)]));
            pgm := instrument_add_call( pgm, matching_enddo(pgm, s),
                        "END_INTERVAL", COMMA([CST(#do_loops)]));
        elseif stmt("st") = "STOP" then
            pgm := instrument_add_call(pgm, preceding_stmt(pgm, s),
                        "EXIT_INTERVALS", COMMA([CST("'"+root+".sum'")]));
        end if;
    end for;

    return [pgm, do_loops];
end;
```

Figure 3.10: Function to instrument a compilation unit

the "routine_type" of the compilation unit to see if we are currently operating on a routine of type "PROGRAM". If we are working on the main program, we insert the initialization call either after the entry point, if the program was written with an explicit PROGRAM statement, or after the initial executable statement if no PROGRAM statement was found.

The arguments to a subroutine or function call are children of a "," expression node. The "," node is created with a call to the Delta function COMMA(). A constant expression node is created with the CST() call, where the type of the node is determined from the type of the SETL object passed as its argument. An integer will be of Fortran type INTEGER, a character string starting with "'" is the Fortran type CHARACTER and otherwise a character string is of Fortran type REAL. The SETL expression COMMA([CST("'"+root+".intvl'")]) creates an imploded argument list with one argument that is a Fortran character string.

Next we want to iterate over the statements in the program and add the instrumentation calls at the appropriate points in the compilation unit. The function list_stmts() returns a tuple of statement tags in lexical order for the compilation unit. For each statement in the program we copy the statement properties into a local variable to simplify the access to the fields.

If the statement is a DO loop header, we may need to add instrumentation calls to the loop. We need to check if this loop is nested inside any other loop. The "outer" field in the statement contains the statement tag of the lexically enclosing loop. If this statement is the outermost loop, then the "outer" field will not be present and a reference to that field will return the value OM.

The interval for this loop will now be defined. If the scanner added a "line" field to each statement (as requested by the "+line" directive to the scanner), then we will concatenate the name of the routine with the line number of the loop to be used as the name of this interval. The position in the do_loops list is the interval number. The SETL expression #do_loops returns the length of the do_loops tuple. If the scanner did not insert a line number, then we will sequential number the loops as they are encountered. Finally we add the "START_INTERVAL" and "END_INTERVAL" calls before and after the loop.

This almost completes the instrumentation of the loops in the compilation unit. We may need to add a call to close the interval file when the program terminates. In Delta, the only way a program may terminate is for it to execute a Fortran STOP statement. Thus, by searching for every STOP statement in each compilation unit and adding a call to "EXIT_INTERVALS" before the STOP statement, we are assured to call this routine wherever the program could possibly terminate.

The instrument_program() routine returns the modified program object and the current list of loops to the calling routine. This function is structured to return these values to eliminate access to global variables and strictly limit the communication between functions to values explicitly passes as arguments and the values returned as results from the functions.

Figure 3.11 defines a support function to add a CALL statement to the program. In Delta a CALL statement is represented as a map that includes an entry for the name of the called routine along with an entry for the arguments at this call site. The function make_call_stmt() creates an entry in the statement table representing a "CALL" statement to the routine specified by the variable name with the argument list referenced by e.

The function explode_args_tree() translates from the “imploded” form of an expression tree and installs it as the “exploded” form in the compilation unit's expression table. The values

```
$ Make a CALL statement after line <s>
$ to routine <name> with <args> in <pgm>.

instrument_add_call := func( pgm, s, name, args );
    local   e, t;

    [pgm, e] := explode_args_tree(pgm, args);
    [pgm, t] := make_call_stmt(pgm, name, e);

    return add_after_stmt( pgm, s, t );
end;
```

Figure 3.11: Support function to create and link a `CALL` statement into a program

returned from this function are the modified compilation unit and the index of the expression in the expression table.

Finally, we complete this support function by linking the new call statement `t`, into the program, lexically after the statement specified as the function's argument `s`. These three SETL functions (Figures 3.9, 3.10, and 3.11) constitute a solution to the problem of adding instrumenting call statements around outermost loops.

3.6 Possible Enhancements

The example in this chapter illustrates well the concepts of a prototyping environment. However, the implementation of this instrumentation strategy is missing a feature that is allowable in Fortran. Assume a loop, with an exit from within the loop. The `"EXIT_INTERVAL"` call may not be made when the loop is actually terminated. Luckily, because the full power of Delta is available, it is trivial to add the enhancements to detect when a branch of some kind leaves the scope of the loop. Other transformations are also possible and may benefit from the ability of Delta to compute dataflow, control dependence or data dependence information along with any other static analyses one might wish to implement.

3.7 Conclusion

Delta is the environment used to implement the experiments in this thesis. Without the use of systems such as Delta, it would have been more difficult to finish the work described in the other chapters.

Through the use of a simple example, we have shown that it is possible to build an open, extensible prototyping environment for restructuring compilers. We have found that because of the concise nature of the SETL language, along with the flexible self documenting data structures and garbage collection of dynamic objects, we can rapidly design and implement many transformations. It is our desire that tools such as Delta can be used to help with the development of new techniques and be used as a tool to educate the next generation of restructuring compiler writers.

Chapter 4

CRITICAL PATH ANALYSIS

4.1 Introduction

Compilers that translate sequential programs into semantically equivalent parallel or vector forms, known as parallelizing compilers [KKP+81, PW86], are an integral part of most supercomputers and have a significant influence on their performance. Despite their importance, there have been few attempts to evaluate systematically the effectiveness of parallelizing compilers, even though such evaluation is necessary to compare the effectiveness of different translation strategies and would be of great help in determining which techniques would benefit from further development [Blu92].

Two main approaches have been followed in the past to evaluate parallelizing compilers. The first measures the execution time of a sequential program or synthetic `DO` loop and that of the parallelized version, and uses the ratio

$$speedup = \frac{execution\ time\ of\ sequential\ program}{execution\ time\ of\ automatically\ parallelized\ program} \tag{4.1}$$

or just the denominator as a measurement of translation effectiveness. The execution times in equation (4.1) are measured either by executing the program on a parallel computer or by static estimation [KBC+74, Lee80, KSC+84]. The second approach uses a collection of synthetic `DO` loops to evaluate the parallelizer by counting how many, and observing which loops it parallelizes [CDL88]. Both of these approaches produce useful but incomplete information. We believe other complementary approaches are necessary to improve our understanding of the performance of parallelizers.

In this chapter we describe a new approach to evaluating parallelizing compilers that is easy to apply and provides important information that cannot be obtained by following the two traditional approaches just discussed. The method we propose, described in sections 4.2 and 4.3, measures the program parallelism missed by the parallelizer by comparing the execution time of the automatically parallelized program with an upper bound of the optimal parallel execution time. The information provided by this analysis should be very useful for improving the parallelizer and for studying the effectiveness of parallelization.

In section 4.4 we use this approach to evaluate the effectiveness of a commercially available parallelizer, KAP/Concurrent [Kuc90], on a few linear algebra kernel routines from *Numerical Recipes* [PFTV88] and a selection of the Perfect Benchmarks® [Per89] programs.

4.2 Overview of the Evaluation Method

In the method described here, the translation quality of a parallelizer is evaluated by comparing the execution time of the parallelized program with an upper bound of the optimal parallel execution time. Both the execution times and the upper bound are computed assuming an ideal machine that has an unlimited number of processors and therefore can exploit an unbounded amount of parallelism. In the work reported here, we assume that in the ideal machine each arithmetic operation, memory write, and intrinsic function invocation consumes one unit of time. All other activities — including forking and synchronization overhead, memory reads, and I/O — are assumed to be free. This choice was made in part for historical reasons, but mainly because we felt that evaluating execution time in this way reflects the characteristics of the programs in a machine-independent manner. For example, the overhead for synchronization and forking is clearly a function of the target machine and not of the program, and therefore we decided to ignore such times. Also, in many machines it is possible to overlap the execution of operations with memory reads and writes. We decided to represent this occurrence by ignoring the time to do memory reads. Clearly, the choices we have made could be the subject of debate, but we believe that substantially the same results as reported below would be obtained with any other sensible choice. The assumption that the ideal machine has an unbounded number of processors also contributes to making the evaluation machine-independent. However, complete machine-independence in this regard is not always possible because some parallelizers may restrict the parallelism they exploit due to some architectural characteristics of the target machine. For example, parallelizers for the Alliant FX/80 [All85] attempt to exploit only one level of loop parallelism.

The upper bound on the optimal parallel execution time used for our evaluation is the maximum length over all *flow*-dependence chains generated by an execution of the program. This value is computed by instrumenting the program to keep track of all memory references as discussed below. This instrumentation strategy was developed by Kumar [Kum88] to measure the implicit parallelism present in a program. This technique was later used by Yew and Chen [Che89, CSY90] to measure other important program characteristics. Several other authors have used similar approaches to measure inherent parallelism [NF84, KBG90, Fu90]. In the work reported here, the effects of the *output*- and *anti*-dependences are disregarded because these dependences can always be eliminated, although sometimes at a high cost, through storage management transformations performed by the parallelizing compiler.

Our proposal is to use the ratio

$$\frac{\textit{execution time of automatically parallelized program}}{\textit{upper bound of optimum parallel execution time}} \tag{4.2}$$

to estimate, on the ideal machine described above, how close the speedup produced by the parallelizer is to the best possible speedup. We believe this is a good indication of the effectiveness of a parallelizer on a particular program because it takes into account the potential parallelism present in the program. This approach is similar to that used to evaluate page replacement strategies by comparing their behavior to an optimal strategy such as OPT [Bel66]. One difference is that it is not possible for us to obtain the optimal execution time because parallelizers can (and often do) apply algorithm substitution. For example, parallelizing compilers may eliminate induction variables and replace recurrence solvers and matrix-multiplication loops with invocations to library subroutine. Finding all the algorithms in a program and replacing them

with the best possible algorithm in existence is, in general, not possible. Therefore instead of computing an optimal execution time, we compute an upper bound that should still be useful to help us in the evaluation process.

Because parallelizing compilers may change some algorithms, the execution time of the automatically parallelized program could be less than the upper bound computed by instrumenting the program. To avoid this problem, the upper bound of the optimal parallel execution time is computed by instrumenting the automatically parallelized program, not the original program. In this way the upper bound is computed only after all the algorithm substitutions have taken place, and therefore the following inequality is guaranteed to hold:

$$\frac{\textit{execution time of automatically parallelized program}}{\textit{upper bound of optimal parallel execution time}} \geq 1 \tag{4.3}$$

The instrumentation approach we use is valid only for sequential programs. Therefore, the automatically parallelized program has to be serialized before it is instrumented. This is easy for the parallelizer we evaluate in this chapter, where serialization can be done just by ignoring the generated annotations.

In our work we compute two upper bounds of the optimal execution time. For the first upper bound, no restrictions are made. This corresponds to the maximum length of the *flow*-dependences computed between operations and therefore assumes that parallelism at the operation-level can be exploited. The second upper bound restricts the program to loop-level parallelism. The loop-level parallelism restriction is implemented by assuming an artificial control dependence from each statement in the program to its sequential successor in the serial execution unless the two statement instances are in different iterations of the same `DO` loop nest. The operation level upper bound gives us an estimation of the total amount of parallelism available. The loop-level parallelism is used because today's parallelizers deal almost exclusively with the extraction of parallelism at the loop-level.

The two types of upper bounds (operation and loop level) were also used by Yew and Chen for the MaxPar project [Che89, CSY90]. However, they allow the concurrent execution of iterations from disjoint loop nests. Our approach is more restricted. At the loop-level, we only allow the statements in different iterations of the same loop nest to execute concurrently. This restriction should allow a better correlation between the automatic parallelization tools and our simulations.

4.3 Implementation

The approach to program instrumentation that is used in this experiment is similar to the method used by M. Kumar [Kum88]. We chose this approach because it is efficient enough to allow us to run several programs with large execution times that are costly to process through MaxPar because of resource limitations.[1] In addition to the instrumentation of sequential programs, we added the ability to instrument explicitly parallel programs as generated by KAP/Concurrent [Kuc90]. Furthermore, the existence of two instrumentation systems, MaxPar and ours, has the additional advantage that the replication can aid in the debugging of both systems because we can compare the results and make corrections when discrepancies arise.

[1] MaxPar is able to compute more parameters than our system. The instrumented code has a subroutine call inserted after each primitive operation to collect the timing information. This makes the system flexible but has a detrimental effect on performance.

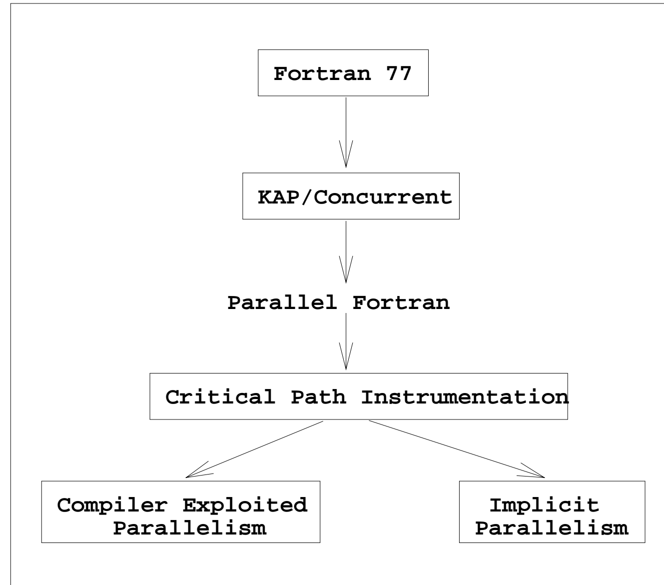

Figure 4.1: General experiment overview

The implementation of the instrumentation tool described below was facilitated by the Delta program manipulation system described in Chapter 3. Figure 4.1 illustrates the major steps we followed in the experiment reported here. First the original source code is transformed through KAP to obtain the reference program. Using this reference program as the base, the program is instrumented to compute the run-time behavior on the target machine. Figure 4.1 shows that two versions of the program are created. One evaluates the execution time of the automatically parallelized program on the ideal machine. The other version computes the implicit parallelism as an upper bound on the optimal parallel execution time. Furthermore, the implicit parallelism is computed in two ways as discussed above, namely loop and operation level.

Throughout this section the reader should remember that, even though we are discussing parallelism in its various forms, all experiments were done on a sequential computer. In the rest of this section we describe the way that programs are instrumented by our system to compute the sequential and parallel execution times on the ideal machines. The instrumentation code accurately computes the execution time of the ideal machine in terms of *operations* that are accumulated during the execution. The instrumented code reveals the *flow*-dependences implicit in the program's execution. The *flow*-dependences are used to derive the upper bounds of the optimal execution time.

In section 4.3.1 we present the instrumentation used to count the number of arithmetic operations in the program that corresponds to the sequential execution time on the ideal machine. In section 4.3.2 we briefly describe KAP/Concurrent and its target parallel language. This is the parallelizer used for the experiments in section 4.4. KAP/Concurrent is a Fortran source-to-source preprocessor that discovers loop-level parallelism. The result of this preprocessor is a program written with Concurrent's parallel programming directives (similar to the PCF Fortran [Par90] parallelism specification) to represent the parallelism. In section 4.3.3 we discuss the instrumentation used to measure the execution time of the automatically parallelized program,

and from section 4.3.4 to 4.3.8 we discuss how the upper-bound to the optimal execution time is computed.

4.3.1 Measuring the Sequential Execution Time of the Parallelized Program

To compute the sequential execution time on the ideal machine, the output of the parallelizer is instrumented and then executed. For KAP/Concurrent, the restructured program is annotated with compiler directives indicating where parallel activity is requested. By disregarding these directives, we have a sequential program to use as the basis for these experiments.

The code is instrumented with a common block in each of the subroutines that allows the operation count, `N$N`, to be accumulated in a single shared variable across the entire program. The value of this operation count at the end of the program is our sequential execution time. To decrease the number of operations in the instrumented program, the operation count is incremented only once in each block of assignment statements.

Consider, for example, the following subroutine.

```
S1:   SUBROUTINE DAXPY(N,DA,DX,DY)
S2:   DIMENSION DX(*), DY(*)
S3:   IF (N .LT. 1) RETURN
S4:   DO I = 1, N
S5:      DY(I) = DY(I) + DA*DX(I)
S6:   ENDDO
S7:   END
```

After the operation count phase of the instrumentation has been completed, the subroutine has the form as shown next.

```
S1:   SUBROUTINE DAXPY(N,DA,DX,DY)
S2:   DIMENSION DX(*), DY(*)
      COMMON /N$NBLOCK/ N$N
I1:   N$N = N$N + 1
S3:   IF (N .LT. 1) RETURN
S4:   DO I = 1, N
S5:      DY(I) = DY(I) + DA*DX(I)
I2:      N$N = N$N + 3
S6:   ENDDO
S7:   END
```

Statement I_1 counts the operation in S_3, and I_2 counts the three operations, two arithmetic operations and one memory write, in S_5. Notice that array offset computations, memory reads, and the bookkeeping needed to implement the `DO` loops do not influence the value of `N$N`. However, if necessary, these values could be added to improve the correlation with real machines.

4.3.2 Parallel Constructs Produced by KAP/Concurrent

As already mentioned, KAP/Concurrent automatically translates Fortran programs into parallel form for the Concurrent Computer Corp. workstations. As an example of the transformations performed, consider the following simple loop.

```
S1:   DO I = 1, N
S2:      DY(I) = DY(I) + DA*DX(I)
S3:   ENDDO
```

This loop can be trivially transformed into the following parallel loop with the addition of the PARALLEL PDO directive.

```
CCUR$ PARALLEL PDO NEW( I)
S1:   DO I = 1, N
S2:      DY(I) = DY(I) + DA*DX(I)
S3:   ENDDO
```

In the above loop you will notice that no data dependences exist. In the next example we will include a reduction operation in the loop body.

```
S1:   DO I = 1, N
S2:      DY(I) = DY(I) + DA*DX(I)
S3:      SUM = SUM + DY(I)
S4:   ENDDO
```

The user has the option of requesting that KAP parallelize loops like this by reordering some of the operations, as shown next.

```
CCUR$ PARALLEL PDO NEW( I,SUM1)
CCUR$ INITIAL SECTION
      SUM1 = 0.
CCUR$ END INITIAL SECTION
S1:   DO I = 1, N
S2:      DY(I) = DY(I) + DA*DX(I)
S3:      SUM1 = SUM1 + DY(I)
S4:   ENDDO
CCUR$ FINAL SECTION LOCK
      SUM = SUM + SUM1
CCUR$ END PDO
```

However, in the experiments reported in section 4.4, we restrict KAP by not permitting the associative reordering of operations. Although this restriction does inhibit KAP's capabilities,

the upper bounds of the optimal execution times used in this chapter also do not consider reordering of associative operations. Therefore, even though we are not allowing KAP to perform to its full potential, we obtain a reliable comparison in the absence of reduction optimization. When reordering is inhibited, KAP generates the following code for the previous loop.

```
CCUR$ PARALLEL PDO NEW( I)
S1:   DO I = 1, N
S2:      DY(I) = DY(I) + DA*DX(I)
CCUR$ ORDERED SECTION( 1)
S3:      SUM = SUM + DY(I)
CCUR$ END ORDERED SECTION( 1)
S4:   ENDDO
```

In this loop, the different iterations of the ordered section execute in the same order as in the sequential program. Through the addition of the `ORDERED SECTION` directives, KAP preserves the original summation order of the array `DY`. Generally, any number of ordered sections is possible in a `PARALLEL PDO`. In this approach, the number of parallel processors that may be used profitably is limited by the number and nature of the statements in the loop body as well as the number of iterations. Without ordered sections, the parallelism is limited only by the number of iterations. The ordering constraint of the ordered section implies that the iterations will be executed in sequential order with statements S_2 and S_3 overlapped in successive iterations.

The standard name used for this type of synchronized parallelism is a `doacross` loop [PKL80, Cyt86]. The `doacross` model of parallelism is more powerful than the simple `ORDERED SECTION` implementation because `doacross` allows any type of synchronization in a parallel loop as long as it goes from lower to higher iterations. In the `ORDERED SECTION` implementation, synchronization is always between consecutive iterations. The `doacross` loop optimization is applicable only when several partially independent statements exist in the loop body. If the summation (S_3) had been the only statement in the previous loop, no parallelism could have been exploited.

4.3.3 Measuring the Parallel Execution Time of the Parallelized Program

The parallel execution time is measured by instrumenting the automatically parallelized program and then executing the resulting program sequentially. The generation of the instrumented code is guided by the Concurrent parallelism directives (`CCUR$`). The measurements are taken on the same program version used to compute the sequential execution time.

The execution time on the ideal parallel machine is computed by running an instrumented version that accumulates the execution time using the tracking variable `N$N`. Notice that this variable has the same name as that used to compute the sequential execution time. This does not cause any problems because the sequential execution time is computed in a separate run. Before entering a parallel loop, two other tracking variables, `N$Sn` and `M$Sn`, are initialized. S_n represents the label of the nth `DO` loop. The variable `N$Sn` contains the execution time before entering the loop. At the beginning of each iteration, the current execution time (contained in the tracking variable `N$N`) is assigned the value of `N$Sn` that simulates a scheduling policy in

which each iteration of the loop is assigned to a separate processor. This is possible because, in the ideal machine, we assume the existence of an unbounded number of processors.

Our model of execution for loop-level parallelism requires a barrier after the execution of a parallel loop. To compute the earliest completion time of the barrier, we need to know the maximal completion time over all iterations of the parallel loop body. The tracking variable `M$Sn` is created to compute this value. At the end of the loop body we compute the maximal execution time of the loop body (`M$Sn`) by comparing it with the maximum so far and recording the larger value. After the loop exits, the accumulated execution time `N$N` is replaced by the maximum completion time for a loop iteration, `M$Sn`.

To illustrate these ideas, consider the following annotated loop.

```
CCUR$ PARALLEL PDO NEW( I)
S1:   DO I = 1, N
S2:      A(I) = B(I) + C(I)
S3:   ENDDO
```

The transformed loop, shown next, is created by the addition of tracking statements to simulate the effects of parallel execution.

```
      M$S1 = N$N
      N$S1 = N$N
S1:   DO I = 1, N
         N$N = N$S1
S2:      A(I) = B(I) + C(I)
         N$N = N$N + 2
         M$S1 = MAX(M$S1,N$N)
S3:   ENDDO
      N$N = M$S1
```

The `ORDERED SECTION` directive is handled similarly to the parallel loop directive. A variable records the completion time of the ordered section, and then the same variable is used to compute the earliest possible time when the ordered critical section on the next iteration may begin. As an example, we will examine the following annotated loop containing an ordered section.

```
CCUR$ PARALLEL PDO NEW( I)
S1:   DO I = 2, N
      ...
CCUR$ BEGIN ORDERED SECTION( 1)
S2:      A(I-1) = A(I) + 1
CCUR$ END ORDERED SECTION( 1)
      ...
      ENDDO
```

Notice that the above core code fragment defines an ordered section surrounding statement S_2. The calculation of the starting and completion times are done through the addition of statements to update the tracking variable M$S2. The variable M$S2 tracks the earliest completion time of the ordered critical section on each iteration. The transformed code is shown next.

```
      N$S1 = N$N
      M$S1 = N$N
      M$S2 = N$N
S1:   DO I = 2, N
         N$N = N$S1
         ...
         N$N = MAX(N$N, M$S2)
S2:         A(I-1) = A(I) + 1
         N$N = N$N + 3
         M$S2 = MAX(M$S2,N$N)
         ...
         M$S1 = MAX(M$S1,N$N)
      ENDDO
      N$N = M$S1
```

4.3.4 Shadow Variables and Operation-Level Implicit Parallelism

To compute the intrinsic operation-level parallelism, we follow an approach similar to that described in [Kum88]; that is, we associate a shadow variable with each of the program's variables and array elements. The purpose of the shadow variable is to make the data dependences explicit during the program's execution. The convention we have used is to concatenate the string "C$" with the variable name to create a new variable called the shadow of the original variable. The shadow variables are initialized to zero.

Starting with the program version as transformed by KAP, we introduce shadow variables and tracking statements. Immediately after an assignment to a variable, its shadow variable is assigned the earliest possible time that this assignment can take place under the assumptions of the ideal machine and ignoring *output-* and *anti-*dependences. Special attention must also be paid to variables that are equivalenced or in common blocks, as well as parameters to functions and subroutines to make sure that all specific declarations are propagated to the shadow variables.

The creation of the tracking code is straightforward. Given an expression A=B+C and the shadow variable for each of the core variables (CA, CB, C$C), tracking is done by an expression that mirrors the run-time behavior of the statement on our ideal machine. For this example the tracking statement would be C$A = MAX(C$B,C$C)+2. We can interpret this assignment to mean that the value assigned to A will be available as soon as both variables B and C become available and two operations, the addition and the memory store into A, are executed.

To illustrate the instrumentation we use to measure operation-level parallelism, consider the following loop:

```
S1:   DO I = N, 1, -1
S2:      SUM = B(I)
S3:      DO J = I+1, N
S4:         SUM = SUM - A(I,J) * B(J)
S5:      ENDDO
S6:      B(I) = SUM / A(I,I)
S7:   ENDDO
```

The operation-level execution time is computed by assignments to the shadow variables, one for each original assignment statement. This is illustrated in the following instrumented version of the previous loop.

```
      C$I = C$N
S1:   DO I = N, 1, -1
S2:      SUM = B(I)
         C$SUM = MAX(C$B(I),C$I)+1
         C$J = MAX(C$I+1,C$N)
S3:      DO J = I+1, N
S4:         SUM = SUM - A(I,J) * B(J)
            C$SUM = MAX(C$SUM,MAX(C$A(I,J),C$B(J),C$I,C$J)+1)+2
S5:      ENDDO
S6:      B(I) = SUM / A(I,I)
         C$B(I) = MAX(C$I,MAX(C$SUM,C$A(I,I),C$I)+1)+1
S7:   ENDDO
```

Notice that here we assume that independent operations within an expression can proceed in parallel, with the `MAX` functions inserted according to this criteria. For example, after statement S_4, two nested `MAX` functions are invoked, each corresponding to a different operation.

The execution time of the program on the ideal machine corresponds to the maximum over the value of all the shadow variables at the end of the program.

`DO` loops are considered a special form for parallel execution. It is assumed that all iterations may begin at the same time. No extra dependence is assumed to exist to update the index variable. This assumption is a point of difference between this work and the work presented by M. Kumar [Kum88].

4.3.5 Shadow Variables and Loop-Level Implicit Parallelism

The computation of loop-level parallelism requires the addition of a control dependence from every statement instance to its successors, except when the successor is in a different loop iteration. Thus, a dependence chain is created using a new set of tracking variables of the form `S$Sn`, that forces the sequential execution of statements S_2, S_3, and S_4. The value of the tracking variable `S$Sn` is the earliest possible completion time of the statement labeled S_n. The loop of the previous section is instrumented as follows:

```
      C$I = MAX(S$S0, C$N) ! S0 is the statement immediately
      S$S1 = C$I           ! before S1
      M$S6 = S$S1
S1:   DO I = N, 1, -1
S2:      SUM = B(I)
         C$SUM = MAX(S$S1,C$B(I),C$I)+1
         S$S2 = C$SUM
         C$J = MAX(S$S2,C$I+1,C$N)
         S$S3 = C$J
         M$S4 = S$S3
S3:      DO J = I+1, N
S4:         SUM = SUM - A(I,J) * B(J)
            C$SUM = MAX(S$S3,C$SUM,C$A(I,J),C$B(J),C$I,C$J)+3
            S$S4 = C$SUM
            M$S4 = MAX(S$S4,M$S4)
S5:      ENDDO
         S$S5 = M$S4
S6:      B(I) = SUM / A(I,I)
         C$B(I) = MAX(S$S5,C$I,C$SUM,C$A(I,I),C$I)+2
         S$S6 = C$B(I)
         M$S6 = MAX(S$S6,M$S6)
S7:   ENDDO
```

In this instrumented loop, the variables `M$S4` and `M$S6` keep track of the completion time of the loop as described in Section 4.3.3. The example shows that for loop-level parallelism, we also require that a statement in an iteration does not start until all the values needed are available from earlier iterations. The examples above show that when we measure loop-level parallelism, we require that all operands for an entire statement be available before the statement can execute. The addition of the control dependences to enforce sequentiality in a block of statements also ensures that only statements in different iterations of the same loop nest can execute concurrently.

4.3.6 Declaration of Shadow Variables

A few points need to be raised about the shadow variables used in the instrumented code. In order for the instrumentation to work, the aliasing between the shadow variables must mirror the aliasing between the core program variables. In particular the static aliasing caused by equivalences must be preserved. This is easy to implement. Assuming we have a type correct program; duplicate the equivalence statements, and change the core variables into shadow variables.

The declarations

```
      REAL X(100), Y(100)
      EQUIVALENCE (X(1), Y(1))
```

are transformed into the following. Notice that the elements of X and Y that are aliased are the same elements of C$X and C$Y that are aliased.

```
REAL X(100), Y(100)
INTEGER C$X(100), C$Y(100)
EQUIVALENCE (X(1), Y(1))
EQUIVALENCE (C$X(1), C$Y(1))
```

This works as long as the program is type correct. If the original program contained equivalenced variables whose base type were different sizes, then the shadow variables could not be aliased in the same manner.

Common blocks are another declaration that must be propagated into the shadow variables. The following declaration

```
REAL X(100), Y(100)
COMMON /BLOCK/ X, Y
```

must be replicated in the shadow variables as:

```
REAL X(100), Y(100)
INTEGER C$X(100), C$Y(100)
COMMON /BLOCK/ X, Y
COMMON /C$BLOCK/ C$X, C$Y
```

This is another form of static aliasing. Here the aliasing is between common blocks in different compilation units. For static aliasing to work, the storage required for the core program variable must be a constant ratio to the storage required for the shadow variable.

One common case arises that violates this assumption. It is a common Fortran programming practice to alias REAL and COMPLEX datatypes. This aliasing is guaranteed to be correct by the Fortran standard. The COMPLEX type is defined to be composed of a contiguous sequence of two REAL storage locations. The instrumentation must accommodate this programming practice. The solution to this problem is to promote the shadow variable of a COMPLEX core variable by adding a dimension to the shadow variable.

A program fragment that declares an aliased COMPLEX array must be treated specially.

```
REAL X(100)
COMPLEX Y(50)
EQUIVALENCE (X(1), Y(1))
```

Instead of defining the dimension of the shadow to be identical to the dimension of the core variable, we add an extra leading dimension to the shadow of the COMPLEX variable. This leading dimension had a length of 2. In effect, we are now shadowing the components of the COMPLEX variable rather than treating it as an indivisible entity.

```
REAL X(100)
COMPLEX Y(50)
INTEGER C$X(100), C$Y(2,50)
EQUIVALENCE (X(1), Y(1))
EQUIVALENCE (C$X(1), C$Y(1,1))
```

In addition to the changes in the declaration, the additional dimension must be added to each reference of the `COMPLEX` variables shadow. A read of the core variable `Y(I)` is shadowed by the code, `MAX(C$Y(1,I),C$Y(2,I))`. A write to the core variable `Y(I)` is shadowed by two writes, the first to `C$Y(1,I)` and the second write to `C$B(2,I)`.

4.3.7 Subroutine and Function Calls

Subroutine and function calls must propagate the availability times of the arguments from the call site to the body of the called routine. This propagation is accomplished by associating a shadow variable with each formal argument, effectively doubling the number of parameters.

An additional formal argument is added. It is named `C$Sn`, where S_n is the name of the current statement. This final parameter is used both to calculate the initial starting time of the subroutine and to propagate the implicit completion time back to the calling routine. For example, the subroutine call:

```
S1:   CALL DAXPY(N, A, X, Y)
```

is transformed into the following code by the addition of the shadow variables expressions for each of the formal arguments and the tracking variable for the control dependences.

```
      C$S1 = ···
S1:   CALL DAXPY(N, A, X, Y, C$N, C$A, C$X, C$Y, C$S1)
```

4.3.8 Control Dependence

In addition to the *flow*-dependences handled by the shadow variables, we also need to model the effects of control dependences in calculating potential speedups at the operation and loop levels. The reason for including control dependence information is to preclude speculative parallelism. Future experiments may relax this constraint to study specifically the effects of speculative execution.

The control dependence algorithm, described in the calculation of the SSA form [CFR+88] and the dominators algorithm from Lengauer and Tarjan [LT79], are used as the basis of the control dependence instrumentation for this experiment. To take control dependences into account, all statements that are control dependent on a given conditional statement, for example S_n, must wait until S_n completes execution. The tracking variable `S$Sn` contains the earliest time at which statements that are control dependent on statement S_n may begin executing.

For example, in the following code fragment there is a control dependence from S_1 to S_2. Thus the availability time of the core variable `A` must wait until statement S_1 is executed.

```
      S$S1 = MAX(···, C$B+1)
S1:   IF (B .GT. 0) THEN
S2:      A = B + C
         C$A = MAX(S$S1, C$B, C$C)+2
      ENDIF
```

4.4 Experimental Evaluation of KAP/Concurrent

The experiments were run using two collections of programs. The first collection is a subset of the Perfect Benchmarks [Per89] (listed in Table 4.1), and the second collection is a set of subroutines from *Numerical Recipes* [PFTV88] (listed in Table 4.4). Because of the minimal overhead of our approach, it was possible to use the complete data-sets as released for the Perfect Benchmarks rather than a subset of the data. Also, the dense linear algebra kernels from *Numerical Recipes* were executed with larger (100×100) matrices rather than the 20×20 matrices used in the driver program distributed by the publisher.

A network of SPARC-based workstations running a cross development version of KAP/Concurrent were used to perform the experiments. Always, as noted in section 4.3.2, reductions were executed sequentially.

4.4.1 Speedup for the Perfect Benchmarks

The Perfect Benchmarks are a collection of programs that are not simple loop kernels. This suite was created to represent typical supercomputer workloads to evaluate the relative performance of large machines. In this experiment, we used these programs to illustrate our approach in evaluating the effectiveness of a parallelizing compiler.

Table 4.2 lists the sequential execution times on the ideal machine of the selected Perfect Benchmarks programs. Also included in the table are the optimal parallel execution times at both the operation-level and the loop-level as well as the execution time on the ideal machine of the program after it is transformed by KAP.

Table 4.3 shows how the parallelism in the programs is affected when the granularity is changed to allow only concurrent execution of loop iterations. In this table we show four columns corresponding to the four ratios in equations (4.4) to (4.8).

$$\mathcal{A}/\mathcal{B} = \frac{\text{sequential execution time}}{\text{upper bound of opt. oper. level parallel execution time}} \tag{4.4}$$

$$\mathcal{A}/\mathcal{C} = \frac{\text{sequential execution time}}{\text{upper bound of opt. loop level parallel execution time}} \tag{4.5}$$

$$\mathcal{D}/\mathcal{C} = \frac{\text{execution time of automatically parallelized program}}{\text{upper bound of opt. loop level parallel execution time}} \tag{4.6}$$

$$\mathcal{A}/\mathcal{D} = \frac{\text{sequential execution time}}{\text{execution time of automatically parallelized program}} \tag{4.7}$$

$$\mathcal{C}/\mathcal{B} = \frac{\text{operation level speedup}}{\text{loop level speedup}} = \frac{(4.4)}{(4.5)} \tag{4.8}$$

Equation (4.4) gives a lower bound of the best possible speedup if the operation-level granularity is assumed. Equation (4.5) gives a lower bound of the best possible speedup if loop-level granularity is assumed. Equation (4.6) estimates how much more parallelism is available at the loop-level beyond what was extracted by KAP. Equation (4.7) is the program speedup on the ideal machine resulting from the automatic parallelization. Finally, equation (4.8) computes the speedup missed by limiting program to loop-level parallelism.

Code	Lines	Institution	Description	Application Area
`adm(AP)`	6101	IBM	Air Pollution	meteorology
`arc2d(SR)`	3965	Cray	Supersonic Reentry	aerodynamics
`bdna(NA)`	3977	IBM	Nucleic Acid simulation	physical chemistry
`dyfesm(SD)`	7608	CSRD	Structural Dynamics	structural mechanics
`flo52q(TF)`	1986	Princeton	Transonic Flow	aerodynamics
`mdg(LW)`	1238	IBM	Water Molecule	physical chemistry
`mg3d(SM)`	2812	Cray	Seismic Migration	geology
`ocean(OC)`	4343	Princeton	Ocean Circulation	fluid mechanics
`qcd2(LG)`	2326	Caltech	Lattice Gauge	particle physics
`spec77(WS)`	3885	CSRD	Weather Simulation	meteorology
`spice(CS)`	18521	CSRD	Circuit Simulation	electronics
`track(MT)`	3784	Caltech	Missile Tracking	signal processing
`trfd(TI)`	484	IBM	Integral Transforms	physical chemistry

Table 4.1: Codes that constitute the Perfect Benchmarks suite

Column $\mathcal{C}/\mathcal{B}$ of Table 4.3 show that the ratio of operation-level speedup to loop-level speedup can be large. With a few exceptions (`mg3d(SM)` and `spec77(WS)`), the ratios show factors of less than 1000 lost by limiting the parallelism to loop nests. The reasons for the exceptional cases may be arithmetic reductions and undetected induction variable computations that serialize entire loop nests.

```
S1:  SUBROUTINE FN(A, B, C, N)
S2:  DIMENSION A(*), B(*), C(*)
S3:  DO I = 1, N
S4:     A(K) = B(K) + C(K)
        ...
S5:     K = K + INC
S6:  ENDDO
S7:  END
```

Figure 4.2: Example of serialization owing to induction

For example, a large number of the loops in `mg3d(SM)` have an inductive sequence used as the indices of an array similar to that shown in Figure 4.2. KAP is unable to replace the induction

	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$
	KAP Processed	Upper Bound of Optimal Parallel		KAP Processed
	Sequential	Oper. Level	Loop Level	Parallel
Program	Execution Time	Execution Time	Execution Time	Execution Time
`adm(AP)`	882543678	370067	19376214	293283144
`arc2d(SR)`	3111818981	484295	9260970	46944930
`bdna(NA)`	1486434605	1924852	10653147	1174503030
`dyfesm(SD)`	493417301	17312687	27541614	125697238
`flo52q(TF)`	1137847449	1314872	5498661	14827201
`mdg(LW)`	3702652187	1164733	695103388	2622955696
`mg3d(SM)`	40208052983	104165	32113276595	34343884638
`ocean(OC)`	3666136327	151746	13459093	2751447397
`qcd2(LG)`	513706328	14450029	217907412	444952424
`spec77(WS)`	105341264921	1008972	7614414132	100263251801
`track(MT)`	119955628	422324	3098391	108472677
`trfd(TI)`	637768455	98509	7256709	61740989

Table 4.2: Simulated and optimal execution times for selected Perfect Benchmarks programs

	$\mathcal{A}/\mathcal{B}$	$\mathcal{A}/\mathcal{C}$	$\mathcal{D}/\mathcal{C}$	$\mathcal{A}/\mathcal{D}$	$\mathcal{C}/\mathcal{B}$
	Lower Bound of Optimal		KAP Parallelized Program		Operation
	Oper. Level	Loop Level	Loop Level Speedup		Level Speedup
Program	Speedup	Speedup	Remaining	Obtained	Remaining
`adm(AP)`	2384.8	45.5	15.1	3.0	52.4
`arc2d(SR)`	6425.5	336.0	5.1	66.3	19.6
`bdna(NA)`	772.2	139.5	110.2	1.3	5.5
`dyfesm(SD)`	28.5	17.9	4.6	3.9	1.6
`flo52q(TF)`	865.4	206.9	2.7	76.7	4.2
`mdg(LW)`	3179.0	5.3	3.8	1.4	596.8
`mg3d(SM)`	386003.5	1.3	1.1	1.2	308292.4
`ocean(OC)`	24159.7	272.4	204.4	1.3	88.7
`qcd2(LG)`	35.6	2.4	2.0	1.2	15.1
`spec77(WS)`	104404.5	13.8	13.2	1.1	7546.7
`track(MT)`	284.0	38.7	35.0	1.1	7.3
`trfd(TI)`	6474.2	87.9	8.5	10.3	73.7

Table 4.3: Speedup ratios for selected Perfect Benchmarks programs

by a direct calculation, and the induction variable computation encloses each iteration in a cycle that includes a *flow*-dependence from S_5 to S_4 and a control dependence from S_4 to S_5. This cycle serializes the entire loop because statement reordering is not considered for loop-level parallelism. The corresponding problem does not appear for operation-level parallelism since we are able to calculate the effects of statement reordering when calculating operation-level parallelism. Clearly, if statement reordering were done, the implicit loop parallelism would lie somewhere between the loop-level reported here and the operation-level parallelism.

The speedups obtained by KAP (except for `arc2d(SR)`, `flo52q(TF)`, and `trfd(TI)`) are small (column $\mathcal{A}/\mathcal{D}$). The actual speedup on a real machine would probably be lower because, as mentioned above, we have not considered the overhead that may be associated with a given parallel loop, or with overheads for architecture-specific transactions such as memory latency.

We find that in general, when interprocedural analysis and reduction transformations are precluded, KAP usually detects little parallelism even though more is available at the loop-level (in two cases, `bdna(NA)` and `ocean(OC)`, up to several 100 times more parallelism, as shown in column $\mathcal{D}/\mathcal{C}$). However, we should point out that the types of inherent parallelism that our method detects in the applications may not be efficiently executable on existing machines. The conventional wisdom that "most of the computation is in the loops" would imply that without further transformations, restricting parallelism to only the `doall` and `doacross` forms will still be missing a substantial amount of the potential concurrency. Optimizations such as high-level spreading or loop distribution may be needed to allow the loop nests to execute concurrently.

4.4.2 Speedup for Numerical Recipe Kernels

The routines that have been selected for inclusion in this chapter (listed in Table 4.4) are dense linear algebra kernels. The published data-set size for the dense matrices is 20×20. However, to increase the amount of work being done, the results reported in Tables 4.5 and 4.6 are based on randomly constructed 100×100 matrices.

Code	Lines	Dominant Algorithm
`GAUSSJ`	135	Gauss-Jordan elimination with full pivoting
`LUDCMP`	136	LU decomposition with partial pivoting
`LUBKSB`	128	LU back substitution
`TRIDAG`	60	Tridiagonal matrix solution
`MPROVE`	171	Iterative improvement of linear equations
`VANDER`	108	Solution of Vandermonde matrices
`TOEPLZ`	116	Solution of Toeplitz matrices
`SVBKSB`	312	Single value back substitution
`SVDCMP`	290	Single value decomposition

Table 4.4: Selected codes from the *Numerical Recipes* programs

As in Section 4.4.1 for the Perfect Benchmarks, we again see a wide range of parallelism present at the operation-level. The smallest operation-level speedup is a factor of 3.4 reported for `TRIDAG` in column $\mathcal{A}/\mathcal{B}$ of Table 4.6. The inner loops of this routine include a tight recurrence that is the reason for the small inherent parallelism.

	$\mathcal{A}$	$\mathcal{B}$	$\mathcal{C}$	$\mathcal{D}$
	KAP Processed	Upper Bound of Optimal Parallel		KAP Processed
	Sequential	Oper. Level	Loop Level	Parallel
Program	Execution Time	Execution Time	Execution Time	Execution Time
GAUSSJ	7610382	3348	26579	1473773
LUDCMP	4172506	11419	52736	1124197
LUBKSB	1258863	22020	68166	1150557
TRIDAG	349979	80001	170011	349979
MRPOVE	21974235	2292085	3905964	21875037
VANDER	586972	22982	76059	468568
TOEPLZ	279077	10799	74442	205223
SVBKSB	27508854	216715	538829	1804296
SVDCMP	31336552	216310	537719	1802278

Table 4.5: Simulated and optimal execution times for selected *Numerical Recipes* programs

	$\mathcal{A}/\mathcal{B}$	$\mathcal{A}/\mathcal{C}$	$\mathcal{D}/\mathcal{C}$	$\mathcal{A}/\mathcal{D}$	$\mathcal{C}/\mathcal{B}$
	Lower Bound of Optimal		KAP Parallelized Program		Operation
	Oper. Level	Loop Level	Loop Level Speedup		Level Speedup
Program	Speedup	Speedup	Remaining	Obtained	Remaining
GAUSSJ	2273.1	286.3	55.4	5.2	7.9
LUDCMP	365.4	79.1	21.3	3.7	4.6
LUBKSB	57.2	18.5	16.9	1.1	3.1
TRIDAG	4.4	2.1	2.1	1.0	2.1
MRPOVE	9.6	5.6	5.6	1.0	1.7
VANDER	25.5	7.7	6.2	1.3	3.3
TOEPLZ	25.8	3.7	2.8	1.4	6.9
SVBKSB	126.9	51.1	3.3	15.2	2.5
SVDCMP	144.9	58.3	3.4	17.4	2.5

Table 4.6: Speedup ratios for selected *Numerical Recipes* programs

As with the Perfect Benchmarks, these results show the vast disparity in inherent parallelism present in the linear algebra kernels. The ratios of potential operation-level parallelism to potential loop-level parallelism (column $\mathcal{C}/\mathcal{B}$) again illustrating the need for further study in the methods for extracting parallelism.

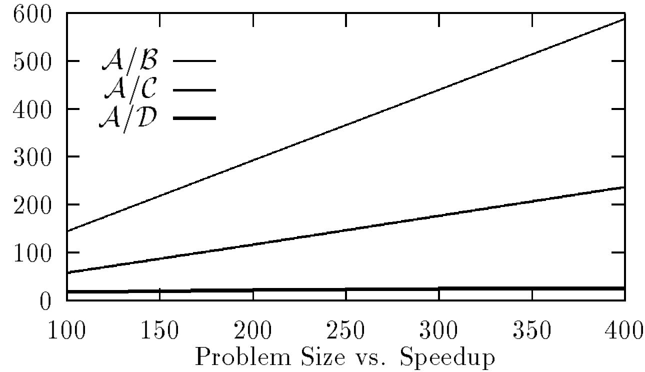

Figure 4.3: Speedup curves for SVDCMP

Finally, by observing the speedups over the automatically restructured code, we see that in several cases KAP has not extracted a significant portion of the available loop parallelism, shown by the potential speedups over KAP (column $\mathcal{D}/\mathcal{C}$ in Table 4.6). Another important factor is that as the data-set size grows, so does the parallelism in the programs. This is illustrated in Figure 4.3 (for the `SVDCMP` program) whose horizontal axis corresponds to the size of N where an $N \times N$ matrix was input data. The corresponding figures for the other programs are similar or show even worse KAP/Concurrent performance on these kernels.

A curious fact shown by Figure 4.3 is the speedup obtain by KAP. This speedup, denoted by the thick black line, is consistently less that the inherent parallelism measured for the routines. The poor performance is directly related to the use of semi-sequential algorithms that KAP/Concurrent was not able to analyze properly. It is important to observe that the speedup of the automatically parallelized program remains practically constant while the potential speedup grows linearly with the data-set size.

4.5 Conclusion

This chapter presents a method of evaluating parallelizing compilers by providing a "yardstick" with which to measure them. The results suggest that intrinsic parallelism is as widely varied as the parallelism extracted by KAP. From examinations of the source code, apparently factors such as variables whose values are unknown at compile time, subscripted subscripts, non-parallelizable statements, and subroutine calls are prime candidates for increased effort. Another large category of missing parallelism is in unrecognized `doacross` loops.

It is important to remember that these experiments were performed without allowing the parallelization of reductions and other recurrences. Therefore, the speedups obtained by the parallelizers are not the best possible, and the evaluation of KAP/Concurrent presented here is not exhaustive. Despite this, the parallelizer is sometimes successful in obtaining a performance close to optimum. In future studies we plan to analyze the influences of the parallelization of recurrences on the overall performance of the program and characterize the types of parallelism leading to the high potential speedups found in the analyzed programs.

Chapter 5

PARALLEL ACTIVITY MEASUREMENTS

5.1 Introduction

The calculation of inherent parallelism presented in Chapter 4 produces summarized information for the results of an optimally executed program on an unlimited number of processors. It may also be useful to create a more detailed picture of the processor activity throughout the program's execution in order to measure processor utilization. M. Kumar [Kum88] collected the processor activity histogram (PAH) for this purpose.

The histogram representation of processor activity may provide some vague indication of the shape of the parallelism, but it is difficult to conclude anything quantitatively meaningful using this representation. It may be obvious that the parallelism present in the program operates in phases, with bursts of activity. Or it may be evident that the parallelism is constant throughout the execution. The shape of the PAH helps to determine the homogeneity of the parallel activity.

In this chapter we discuss the computation of the PAH and use it in two ways. The first is traditional: we will graph the PAH as an intuitive representation of parallel execution (though direct display). The second use will be as data in the computation of bounds on the speedup for restricted numbers of processors.

Section 5.2 describes the background for this experiment and gives an overview of the techniques used. In Section 5.3 we describe a method of collecting the histogram data accurately on an operation-by-operation basis. We provide a heuristic to approximate processor activity in Section 5.4 based on loop iterations. Two methods of recording the loop intervals are discussed, one based on histograms with fixed bucket sizes, and one that creates histogram buckets with variable sizes determined by the temporal overlap of the loop iterations. In Section 5.5 we show how to calculate upper and lower bounds on the inherent parallelism when only a limited number of processors are available. These bounds are derived from the PAH calculated assuming unlimited resources. Finally, we present our conclusions in Section 5.7.

5.2 Overview of the Evaluation Method

As noted in Chapter 4, a sequential program is a sequence of operations related by data and control dependences specifying a minimum partial order of the calculations for correct execu-

tion. Sequential programs are overspecified, including such extraneous notions such as lexical statement ordering and the reuse of storage.

In Chapter 4, we described an instrumentation technique called critical path analysis for propagating timestamps along a dataflow graph to determine the critical path length. This technique provides information about the execution time and thus the overall parallelism (assuming unlimited processors) for the loop, module, or program granularity.

The results of critical path analysis are reported as a coarse summary of the information available during the computation. We would like to collect more precise information about the parallel activity during a program's execution. In particular, we would like to know the number of processors that are active at each time step. One possible presentation of this information is through a processor activity histogram (PAH). The shape of the PAH indicates the processor utilization. As the shape of the PAH approaches rectangular, the processor efficiency approaches one. That is, the percentage of time that a processor is idle approaches zero.

The PAH can also be used as raw data for further calculations. One use is the derivation of bounds on the average parallelism for limited numbers of processing elements. We will expand on this topic further in Section 5.5.

5.3 Histogram Generation

A useful tool in examining parallel execution is the processor activity histogram (PAH). The PAH describes the number of concurrent operations that can be executed in each clock period, assuming an unlimited number of processors. The generation of this histogram requires that we record the time when each operation in the program was executed.

In Chapter 4 we describe a method, with low overhead, for performing the critical path calculation. With this method, the calculations of the critical path length are included inline with the program. Owing to the increased overhead, it may be impractical to add subroutine calls to collect operation-granularity data on the parallelism.

The information needed to update the PAH may only be available as intermediate results in the critical path calculations. The addition of the subroutine calls and the statements needed to generate explicitly the intermediate timestamps necessary to update the PAH would drastically change the style and size of the generated code and also increase the overhead.

As another possibility, Section 5.4 describes an alternative instrumentation method that integrates well within the framework of the critical path calculations.

An alternative not used in this chapter, but available as a more accurate method, is to change the instrumentation method by incorporating operation-granularity measurements. The experiment presented in Chapter 7 describes a method that instruments each operation in the program. The purpose of the instrumentation is to determine the effects of induction variables on the critical path length. Since each operation is individually instrumented, it is trivial to add histogram collection to this method.

5.3.1 Histogram Buckets

From Chapter 4, we observe that the execution time for loop-level parallelism varies over a wide range. In particular, the execution time ranges from approximately 3 million time units to execution times in the billions of time units. Thus, it may be impractical to collect a fine grain

representation of the PAH. For reasons of efficiency and practicality, histogram generation may require that the time-quanta or bucket size (q) be greater than one time unit.

The standard method of defining larger buckets for histogram creation purposes is to use a mapping function f to translate from time units into buckets. The mapping function is $f(t) = \lfloor t/q \rfloor$. This function has the effect of including in bucket $b = f(t)$, of quanta size (q), the time period $[b * q \ldots (b+1) * q - 1]$.

It is easy and natural to increase the PAH bucket size in response to constraints on the instrumented program. A drawback of the larger bucket sizes is a loss of precision on the inherent parallelism bounds. In Section 5.5.3, we illustrate the loosening of the parallelism bounds when less precise information is recorded in the PAH.

We can safely assume that at least one operation must be executed during each internal time-quanta. The construction of the instrumentation for critical path analysis requires that each operation begin execution when all of its operands are available. Because of this construction, the critical path can only be lengthened by the execution of an additional operation that depends on a calculation already in the critical path.

5.4 Histogram Generation from Loop Iterations

One approach to the generation of the PAH, is to observe that we can model the parallel execution of a program, using loop-level parallelism, by observing that each loop iteration is executed sequentially. Figure 5.1 illustrates this type of parallelism with four iterations. Each iteration starts at a temporal offset from the previous iteration. The figure used in this

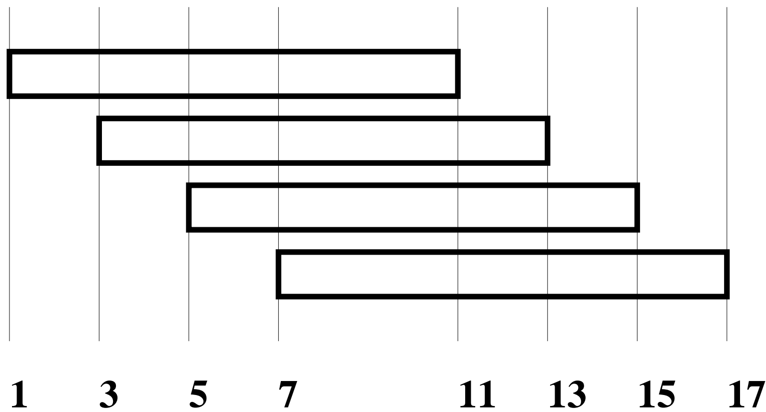

Figure 5.1: Doacross style loop scheduling

example illustrates a `doacross` loop where a constant positive delay (of 2 units) is added to each iteration. If all the iterations started simultaneously, with no delay, we would have an example of the classic `doall` loop. The other extreme is for each iteration to begin executing after the previous iteration has completed. This last class of `DO` loops are examples of sequential loops.

If we know the time when the iteration starts (S) and the time at which it terminates (T), we can be assured that, ignoring any intra-iteration delays or stalls, a processor was allocated to the iteration over the interval $S \ldots T$. Using this heuristic we can eliminate the calls to the support routines that record processor activity for each operation and replace them with a pair

of subroutine calls at the beginning and end of each loop iteration. This reduction in subroutine calls represents a significant overhead reduction.

5.4.1 Natural Interval Storage

We may wish to take advantage of the natural boundaries defined in Figure 5.1. In this figure, regions exist where the loop iterations overlap each other. If the histogram intervals are defined to correspond to these natural intervals, then we may be able to reduce the storage required for the PAH.

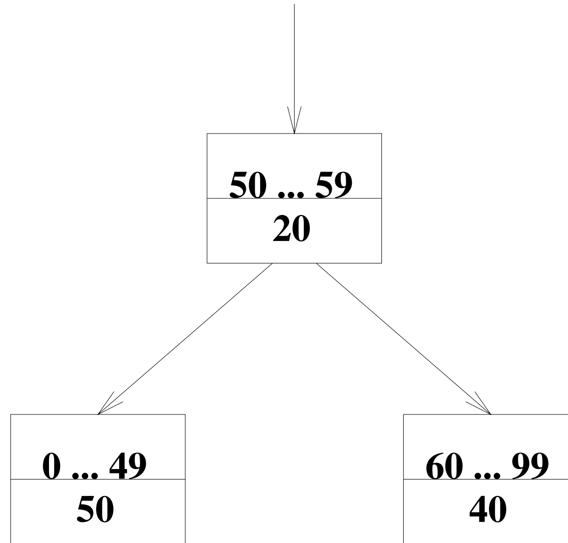

Figure 5.2: Example interval decomposition

Assume we have two intervals, one [0...99] with an operation count of 100, and the other [50...59] with an operation count of 10. When these intervals are merged, we create three new non-overlapping intervals to describe the effects of the two original intervals. For this example shown in Figure 5.2, the two original intervals have been decomposed into three intervals, [0...49] with count 50, [50...59] with count 20, and [60...99] with count 40.

Unfortunately, the storage requirements for each interval in a natural decomposition is considerably larger than for a fixed interval. In the fixed interval decomposition, only one memory word is required for each interval to store the number of operations accumulated over that interval. Since the interval size is constant for all PAH buckets, one global interval size is sufficient to calculate the starting and ending times of the interval covered by the bucket.

In contrast, when using a natural interval decomposition, much more information is required that is specific to each interval. The starting time of the interval and its size are both necessary. Also, it is beneficial to have the PAH represented as a dynamic balanced tree for efficiency in accessing the PAH. Two or three additional memory words are required to maintain the tree structure. The words are used as pointers to the children and possibly the parent of the interval. Thus, the natural interval size must be at least four times larger than the fixed interval size in order for the additional storage requirements of the natural interval decomposition to be amortized. The natural interval decomposition has not been found to generate intervals more than four times larger than the fixed interval decomposition. Therefore, we have chosen to

use the fixed interval decomposition as the storage mechanism implemented for experiments described in the remainder of this chapter.

5.4.2 Heuristics for Interval Measurements from Loop Iterations

The idealized approach, described previously in Section 5.4.1 for recording intervals, has a problem that needs to be addressed. The execution model of parallel loops, which was described in Chapter 4, assumes that all loop iterations begin at the same moment, at time S, immediately after the `DO` loop header has executed. Each loop iteration executes concurrently until it requires information that has not yet been computed. When an iteration needs such a value, it stalls or waits for the information to become available. This process continues until the last statement in the iteration has been executed. For this model, we see that iteration i starts at time S and terminates when all dependences have been enforced and all operations have been executed at time T_i.

When we look at these timestamps, we see that the timestamp T_i represents an accurate termination time for loop iteration i. However, each iteration begins execution at time S. To estimate the real start time (S_i) for each iteration we assume that the operations executed during the iteration are packed as late as possible and the delay slots are all scheduled as early as possible. This heuristic approximates the execution semantics of a `doacross` loop where a delay is inserted at the beginning of each loop iteration to enforce the data dependences present in the loop. If we assume that N_i operations are executed by iteration i, then we can approximate the starting time by $S_i = T_i - N_i$.

Once the true starting (S_i) and terminating (T_i) times for each iteration are known, we can record this interval in the PAH. For each iteration interval, we know that one processor was assigned to execute the operations in that iteration.

5.5 Average Parallelism for Limited Processors

Computing the average inherent parallelism for a program is done through the determination of the critical path during an execution of the program. A limitation of the simple algorithms that determine critical path length is that they do not take into account limited resources such as having a restricted number of processors available.

When taking limited resources into account, you need to assume an algorithm for scheduling the resources. It is not difficult to design an algorithm for any particular scheduling policy. However, it may be impossible to design an optimal scheduling algorithm without knowledge of future events. Even if the information is known, the algorithm to generate the optimal schedule is NP-Complete [BC76].

We may also want to determine the results for different numbers of processors. Even if we could design and implement a scheduling policy, we would be required either to re-execute the program for each resource level or to keep track simultaneously of all resource levels during one execution. The first approach is time consuming, requiring the program to be rerun for each datapoint. The second approach is complex and cumbersome, requiring the support code to consider all resource levels simultaneously.

We have decided to avoid the aforementioned problems by performing post-process analysis of the PAH to determine the average parallelism for a specific number of processors. We do not have a perfect record of the dependences present during the program's execution. The only

information recorded is that each operation in a histogram bucket is dependent on the results of one or more operations in the previous bucket. Thus we are not able to generate the optimal schedule that produces the maximal inherent parallelism for a given number of processors. We can, however, provide bounds to the inherent parallelism.

5.5.1 Upper Bounds of Parallel Execution Time

If we examine the PAH, we see that each bucket of the histogram contains a number of operations. The operations were placed in a specific bucket because they could not be placed in any earlier bucket; doing so would violate the data dependences in the program. Similarly, each operation in every subsequent bucket could not be moved to an earlier bucket because the operation depends on a value produced by at least one operation in an earlier bucket. The critical-operation is depicted by a bold line encircling the operation.

Assuming we have a PAH with granularity of a single clock period, we can easily derive a lower bound for the parallel execute time. For each bucket in the PAH with operation count n, we execute $\lfloor n/p \rfloor$ cycles will all p processors followed by one cycle using $\mathrm{mod}(n, p)$ processors. Each clock period in the original histogram is replaced by $\lceil n/p \rceil$ clock periods. We will refer to this method as cyclic scheduling.

If we assume that the elements in each bucket are ordered from first/bottom to last/top, then cyclic scheduling assumes that each subsequent operation depends on the last operation in the current bucket as shown in Figure 5.3. Examples of cyclic scheduling of the PAH in Figure 5.3 with $p = 2$ and $p = 5$ are shown in Figures 5.4 and 5.5, respectively.

We are assured that selecting last-critical is a correct schedule since we are enforcing all dependences that were present in the original execution. Cyclic scheduling provides us with an upper bound on parallel execution time on p processors, and a lower bound for the average parallelism or speedup.

5.5.2 Lower Bounds of Parallel Execution Time

The simplest upper bound for average parallelism (speedup) is to assume that all operations can be perfectly packed onto the p processors with no regard for the existing data dependences. This approximation always produces an average parallelism linear in the number of processors. Obviously this provides an upper bound on the average parallelism and a lower bound on the parallel execution time, but we would like to have a tighter bound.

To generate the lower bound on the parallel execution, it seems natural to focus on the element in the other extreme position of the bucket from the critical-operation used by cyclic scheduling. If we assume, as shown in Figure 5.6, that all operations in subsequent buckets are dependent only on one operation in the bucket, then we may be able to overlap the operations in subsequent buckets with some of the operations in the current bucket. For the lower bound on parallel execution, define the first operation as the critical-operation shown by the bold lines in Figure 5.6. Figures 5.7 and 5.8 show the results of optimistic scheduling for the PAH shown in Figure 5.6 for $p = 2$ and $p = 5$.

We can show that the first-critical scheduling policy is the tightest lower bound that can be placed on the parallel execution time derived from the PAH. For each histogram bucket i, each operation must be dependent on at least one operation in bucket $i - 1$ or else it would be in bucket $i - 1$. However, we do not know which operation causes the dependence. The best we can do is to assume that the first operation executed for the ith bucket satisfies all

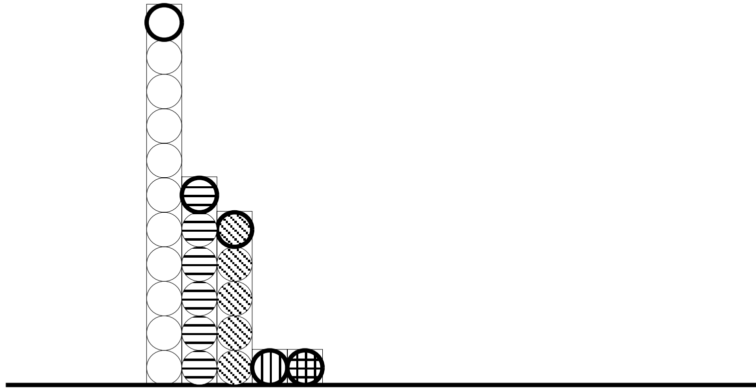

Figure 5.3: Assume the last/top operation is the critical-operation

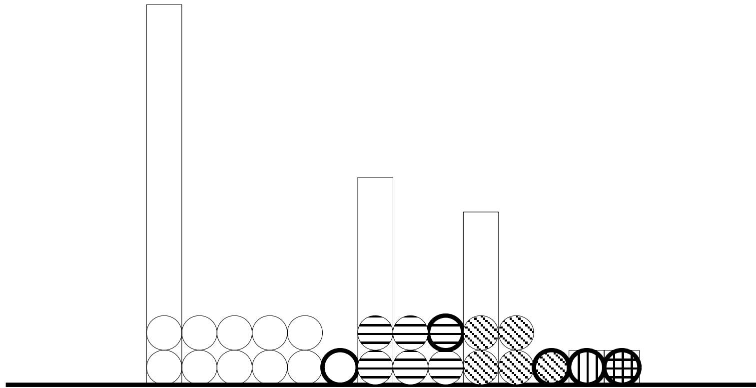

Figure 5.4: Cyclic (*last-critical*) scheduling with $p = 2$

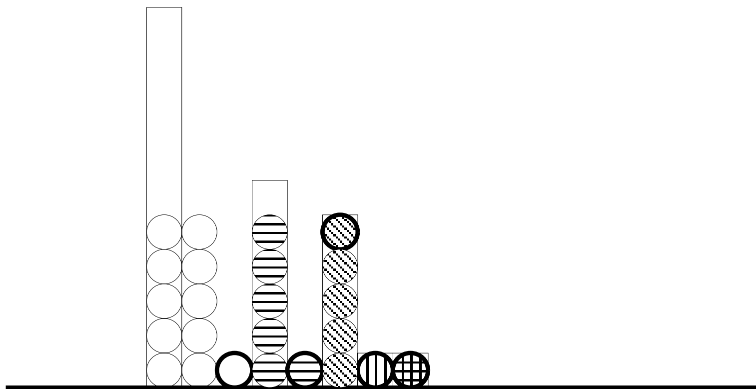

Figure 5.5: Cyclic (*last-critical*) scheduling with $p = 5$

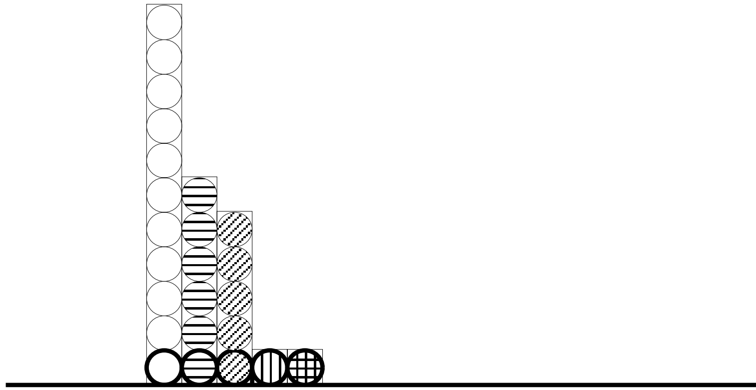

Figure 5.6: Assume the first/bottom operation is the critical-operation

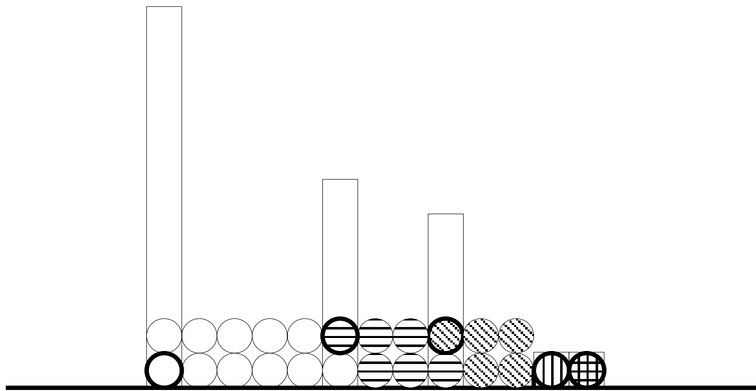

Figure 5.7: Schedule (*first-critical*) assuming most optimistic assumptions, $p = 2$

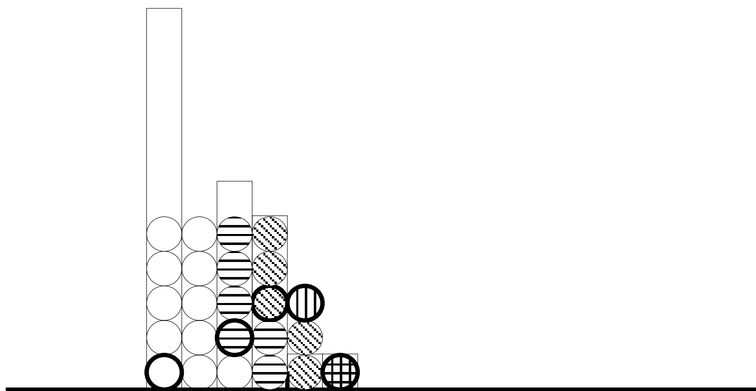

Figure 5.8: Schedule (*first-critical*) assuming most optimistic assumptions, $p = 5$

forward dependences for every operation in the ith bucket. If we assume that it is not the first operation in the bucket that causes the dependence then we might be assuming wrong, (i.e. it was the first operation), and thus we have not computed the lower bound on the parallel execution time. Therefore, the first-critical choice leads to the computation of the lower bound on the parallel execution time.

```
total ← 0
for i ← 1 to number_of_buckets do
    n ← bucket[i].count
    total ← total + ⌈ n/p ⌉
end for
```

Figure 5.9: Calculate upper bound of parallel execution time for $q = 1$

Figure 5.9 lists the code to compute the upper bound of parallel execution time. This algorithm uses a cyclic schedule for a PAH with $q = 1$ to generate a correct schedule.

```
total ← 0
extra ← 0
for i ← 1 to number_of_buckets do
    n ← bucket[i].count + extra
    total ← total + ⌊ n/p ⌋
    extra ← mod(n,p)
end for
total ← total + ⌈ extra/p ⌉
```

Figure 5.10: Calculate lower bound of parallel execution time for $q = 1$

In Figure 5.10 we show how to compute the lower bound of parallel execution time. The second algorithm uses a heuristic that is not conservative to determine the lower bound of the parallel execution time.

5.5.3 Using Larger Time Quanta

The best results can be obtained when each bucket in the PAH corresponds to one unit of time. However, it may be impractical to keep information of this extreme detail. We may be required to reduce the detail of the PAH to make it practical to compute the histogram information. Inaccuracies occur with larger quanta when trying to determine the distribution of the operations inside the bucket. It is possible that almost any distribution of the operations may occur. However, we can distinguish two distributions as the logical extremes for the purposes of this experiment. We assume, for the computation of the lower bound of the parallel execution time, that we have an `L`-shaped distribution of the operations. The `L` distribution is the most optimistic distribution since at least one operation must be executed during each time period. If, during each time period, we only require one operation to be executed, then the remaining operations are free to be placed anywhere in the interval. Placing the remaining

operations as early as possible (i.e., at the left end of the interval) allows them to be scheduled with the most flexibility. In Figure 5.11 you can see that in a histogram bucket containing n operations with size q corresponding to a time-quanta of q clock units, $n-q+1$ of the operations are assumed to execute during the first time unit of the bucket, while one operation is assumed to execute for each of the subsequent time units in the bucket.

If we execute the optimistic scheduling algorithm with $p = 2$ and $p = 5$, we observe that the resulting operation schedule is shown in Figures 5.12 and 5.13.

Similarly, if we assume the opposite extreme that schedules any extra operations as late as possible, we generate a distribution looking like a reverse `L`, as shown in Figure 5.14. The bold circles represent the operations that are the cause of the serializing dependences.

Using cyclic scheduling on the PAH with the number of processors set to 2 and 5 gives the results shown in Figures 5.15 and 5.16.

5.5.4 Algorithms for Larger Time Quanta

It is possible to use the algorithms presented in Figures 5.9 and 5.10 to handle a PAH with larger quanta sizes.

For the upper bound on parallel execution time, distribute the operations in each bucket as illustrated in Figure 5.14. Each bucket contains q time steps. Partition each bucket into q sub-intervals of size one. In each sub-interval, select the last/top operation as the critical-operation. Since we have repartitioned the bucket into sub-intervals of one time unit, we can apply the algorithm from Figure 5.9 to the sub-intervals.

A corresponding change can be made for the lower bound on the parallel execution time. To calculate the lower bound on the parallel execution time, distribute the operations as shown in Figure 5.11. Partition each bucket or time-quanta into individual time units. Select the bottom operation in each sub-interval as the critical-operation. Now since each bucket has time quanta $q = 1$, we can directly apply the previous algorithm shown in Figure 5.10.

It may be more efficient to calculate the bounds directly when using larger bucket sizes rather than to reduce the problem to the previous case. The algorithm, shown in Figure 5.17, computes the upper bound of parallel execution time assuming a time-quanta (q) or bucket size greater than one.

Corresponding to Figure 5.10 for $q = 1$, Figure 5.18 describes the algorithm to compute the lower bound of parallel execution time assuming a time-quanta (q) or bucket size greater than one.

5.5.5 Limiting the Resources Used by the Execution

If a program executes several million or billion operations during its run, we may require an enormous amount of storage to maintain the processor activity histogram (PAH). One approach to alleviate this problem is to summarize the information as it is being collected.

The constrained execution times for limited resources can be calculated over the interval $[0 \ldots T]$, if we have recorded all operations that occur in this interval. However in the simulation methods used, each iteration of a loop begins at the same time. For any time T after the entry into a loop, the summary information cannot be computed until the loop is terminated. It has been informally observed that many of the Perfect Benchmarks® programs spent the majority of their execution inside at least one loop nest. Since we would only be able to summarize the histogram when no loop nests are active, this approach offers little advantage.

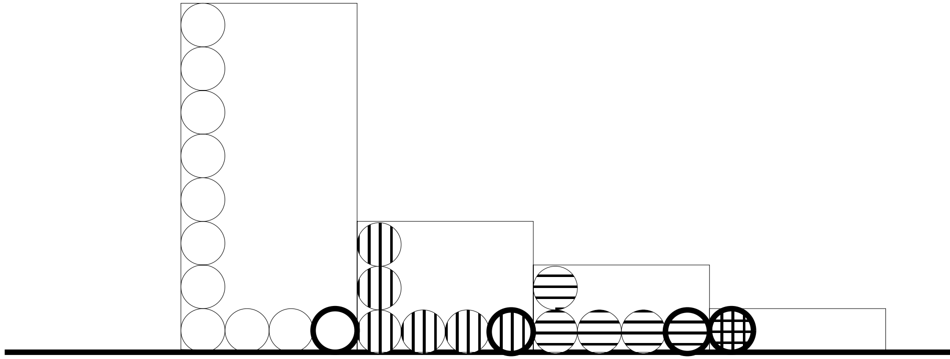

Figure 5.11: Assuming leftmost distribution

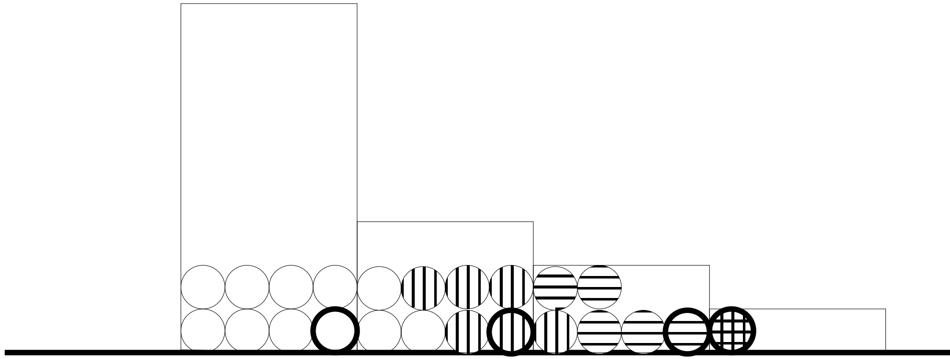

Figure 5.12: Assuming leftmost scheduling distribution $p = 2$

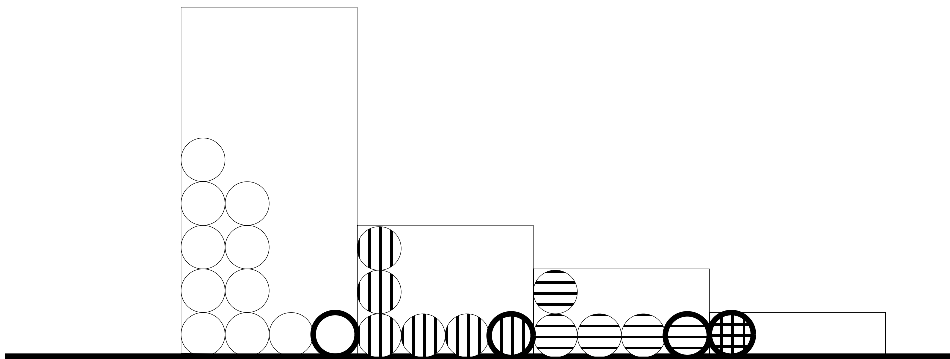

Figure 5.13: Assuming leftmost scheduling distribution $p = 5$

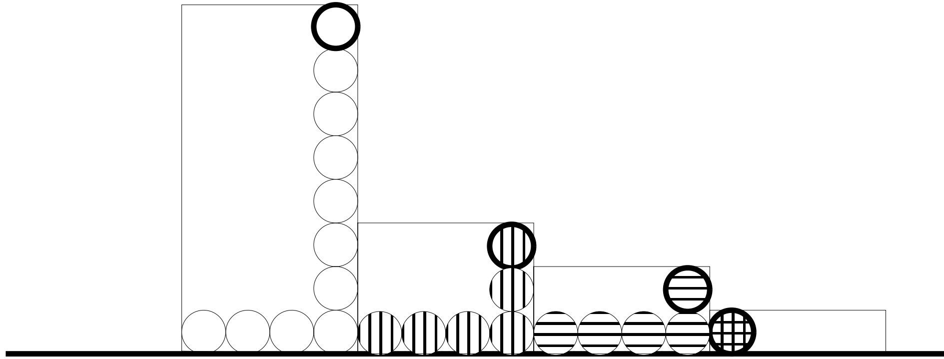

Figure 5.14: Assuming rightmost distribution

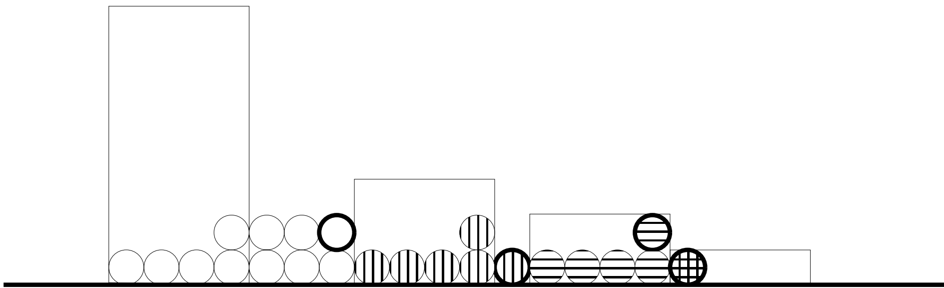

Figure 5.15: Assuming rightmost scheduling distribution $p = 2$

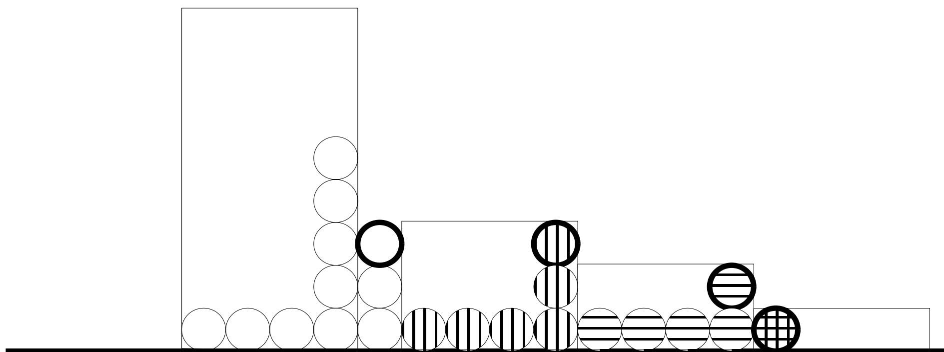

Figure 5.16: Assuming rightmost scheduling distribution $p = 5$

```
total ← 0
for i ← 1 to number_of_buckets do
    count ← bucket[i].count
    if (count < q) then
        total ← total + count
    else
        total ← total + (q - 1)
        count ← count - (q - 1)
        if (count > 0) then
            total ← total + ⌈ count/p ⌉
        end if
    end if
end for
```

Figure 5.17: Calculate upper bound of parallel execution time for $q > 1$

```
total ← 0
extra ← 0
for i ← 1 to number_of_buckets do
    count ← bucket[i].count + extra
    if (count > 0) then
        total ← total + q
        count ← count - (q * p)
        if (count ≥ p) then
            total ← total + ⌊ count/p ⌋
            count ← mod(count,p)
        end if
        if (count ≤ 0) then
            extra ← 0
        else
            extra ← count
        end if
    end if
end for
if (extra > 0) then
    total ← total + ⌈ extra/p ⌉
end if
```

Figure 5.18: Calculate lower bound of parallel execution time for $q > 1$

The ability to specify a time-quanta larger than one allows the PAH to be reduced in size, at the expense of accuracy, to fit into the available memory constraints. A direct correlation exists between the size of the time-quanta and the tightness of the bounds. Thus, we would like to use the smallest time-quanta possible without overflowing the available storage. The minimum time-quanta may be unknown at the start of the program's execution since it is based on the parallel execution time for an unlimited number of processors. If the minimum time-quanta is known then we can customize each run to use the minimum possible quanta. However, if the parallel execution time is unknown, then the minimum quanta time can be determined dynamically.

The strategy for dynamically determining the minimum quanta size is first to assume that a quanta of one is sufficient. When the PAH is about to overflow, double the size of the quanta and combine adjacent entries in the PAH to create a new histogram table based on the new quanta. Since we have combined adjacent histogram buckets, the latter half of the table will be free to record future operations. This strategy results in $O(\log_2(\mathit{final\ time\text{-}quanta}))$ adjustments to the PAH.

5.6 Average Parallelism Results

The methods described in the previous sections were applied to several programs. First a simple benchmark, the 10x10 matrix multiply, was used. This program exhibits almost linear speedup.

Figures 5.19 to 5.21 illustrate the effects of different time-quanta or bucket sizes on the calculated speedup values. Figure 5.19 gives the truest picture of the real speedup obtainable with limited numbers of processors. This figure shows that for a 10x10 matrix multiply, *as it was written*, we can only use less than 100 processors. The upper bound for this example is linear in the number of processors, the lower bound is a stairstep, where dramatic improvements are noticed whenever the number of processors is a divisor of 100.

Figures 5.20 and 5.21 show the effects that larger time-quanta or PAH bucket sizes have on the inherent parallelism bounds. In both cases the number of processors required to achieve maximum parallelism has increased owing to the operation distribution effects and uncertainty described in Section 5.5.3. It is also apparent from these graphs that the lower bound, calculated by cyclic scheduling, changes from the stair-step appearance in Figure 5.19 to a much smoother appearance in the other two figures.

An important property of the average parallelism calculation for limited resources is shown by these figures. The bounds on the average parallelism are the tightest when the quanta size of the PAH buckets is the smallest. When the quanta size is increased, the bounds for the larger quanta are looser than the bounds for smaller quanta. This property permits the techniques described in this chapter to be used for larger quanta sizes at the expense of precision while maintaining the correctness of the bounds.

5.6.1 Perfect Benchmarks Results

The remaining examples are from selected Perfect Benchmarks. The time-quanta chosen for each program was computed by the dynamic processes described in Section 5.5.5. Each example shows the results portrayed as accurately as possible within the bounds of the given memory constraints. A 16-megabyte buffer was used to store the PAH for these experiments.

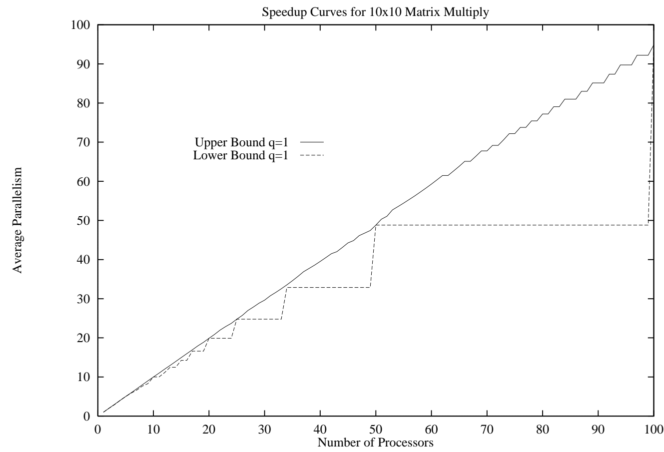

Figure 5.19: Average parallelism for 10x10 matrix multiply, $quanta = 1$

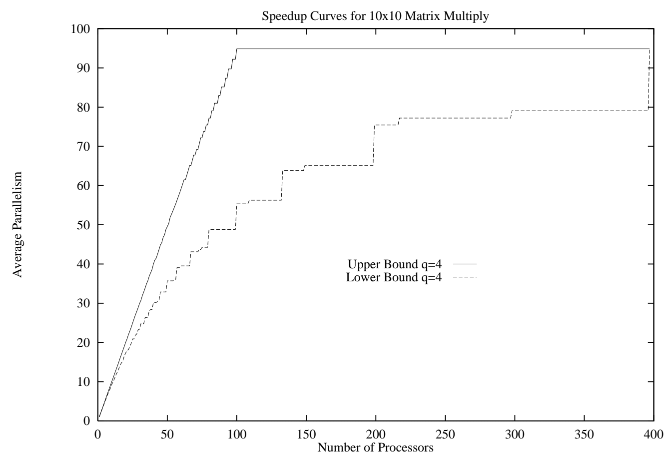

Figure 5.20: Average parallelism for 10x10 matrix multiply, $quanta = 4$

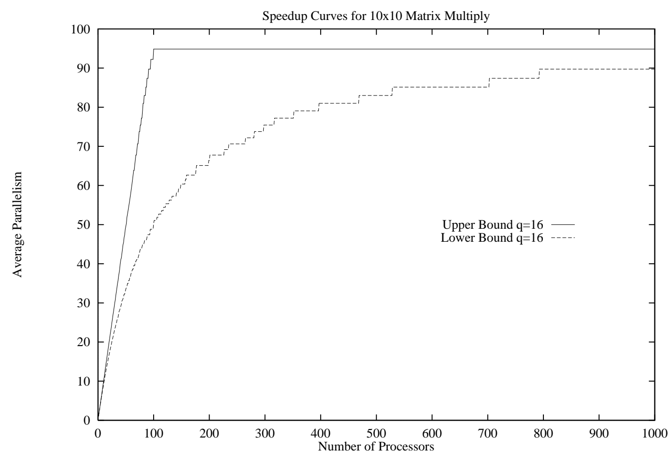

Figure 5.21: Average parallelism for 10x10 matrix multiply, $quanta = 16$

The parallel activity histogram for each program lists three curves that, because of limited resolution, may not all be visible. The PAH is divided into 512 display periods and the minimum, maximum, and average values are computed for each interval in the display period. The y-axis displays the number of concurrent operations at each time interval. The tops of some curves are not displayed since they exceed the maximum value of the y-axis.

Each of the average parallelism plots contains two curves. The curve drawn in a dashed line is the lower bound on the average parallelism, and the solid line is the upper bound. All graphs of average parallelism show the same asymptotic behavior. This behavior is expected since there is no penalty for the saturation of a resource such as memory bandwidth. The bounds asymptotically converge on the average parallelism for the unlimited processor case.

Two observations stand out when examining the graphs of the average parallelism. The first observation is that the upper and lower bounds are tight. Only a small amount of uncertainty exists about the true parallelism at each point on the graphs. The second observation is that, on all the graphs, a knee in the curve exists where increasing the number of processors does not have a significant impact of the average parallelism.

The programs presented in these graphs fall into three categories based on the knees in the average parallelism graphs. The first category is held by `qcd2(LG)`. This program is unique for having the lowest knee. For `qcd2(LG)` it would be impractical to use more than 20 processors. The second category are those programs where up to 500 processors would be useful. Most of the programs fit in this category. The exceptions are the two programs that could use up to 1000 processors efficiently. This last category consists of `ocean(OC)` and `flo52q(TF)`.

Other measures can be taken of the graphs depending on the characteristics of interest. Examples might be the number of processors required to get 90% of asymptotic parallelism or the number of processors needed to get 50% of the asymptotic parallelism.

5.7 Conclusion

Inherent parallelism is a concept that has been used to show the potential for parallel execution. We have shown in this chapter that the processor activity histogram (PAH) can be easily collected as a by product of critical path analysis. We presented two methods for collecting the PAH of parallel activity for an unbounded number of processing elements.

The first method computes the PAH exactly, at operation-level granularity, by issuing a subroutine call for each operation. This method is not used by the experiments in this chapter. The method we used summarizes the activity inside a loop iteration by estimating processor activity from the number of operations performed and completion time of the iteration. The first method collects the parallel activity for each operation. The second method uses a heuristic to summarize the parallel activity for an entire loop iteration.

Given the PAH for an unlimited number of processors, we have shown that it is possible to compute bounds for the case of limited processing resources. The lower bound is computed using cyclic scheduling of the operations in each histogram bucket. The upper bound is calculated by allowing some of the operations in adjacent buckets to execute concurrently. We have shown in this chapter that for large Fortran programs selected from the Perfect Benchmarks, the upper and lower bounds are tight and provide a good estimate on the real speedup curve. We have also shown that histogram bucket sizes greater than one cause degradation in the accuracy of the bounds.

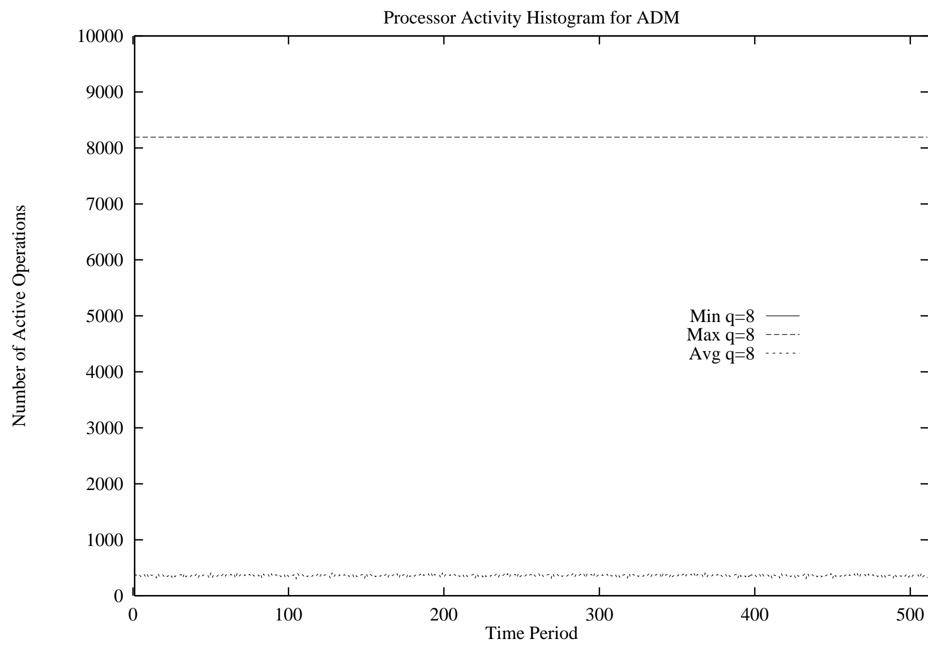

Figure 5.22: Parallel activity histogram for `adm(AP)`

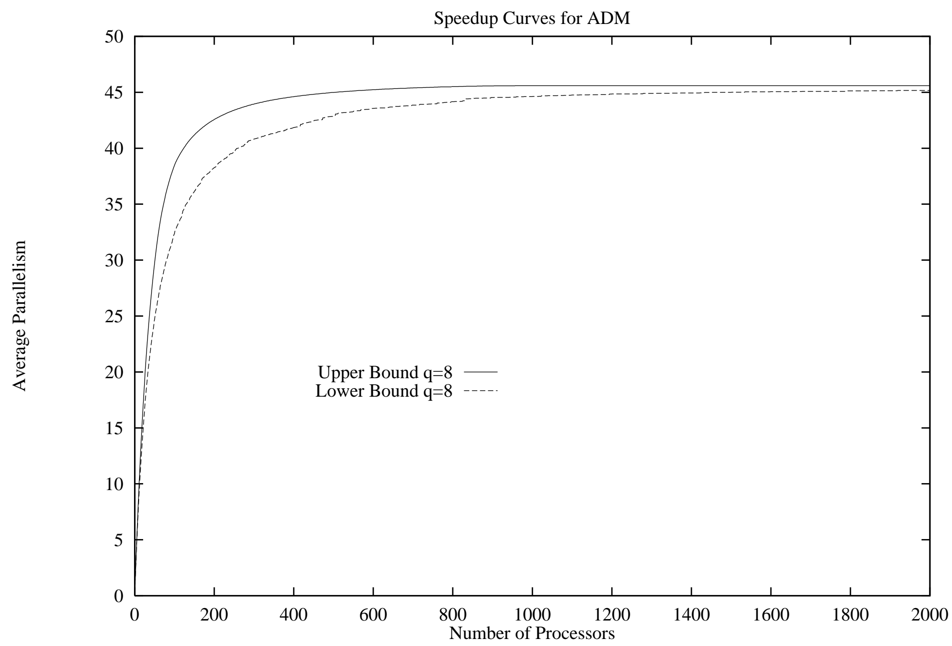

Figure 5.23: Average parallelism for `adm(AP)`

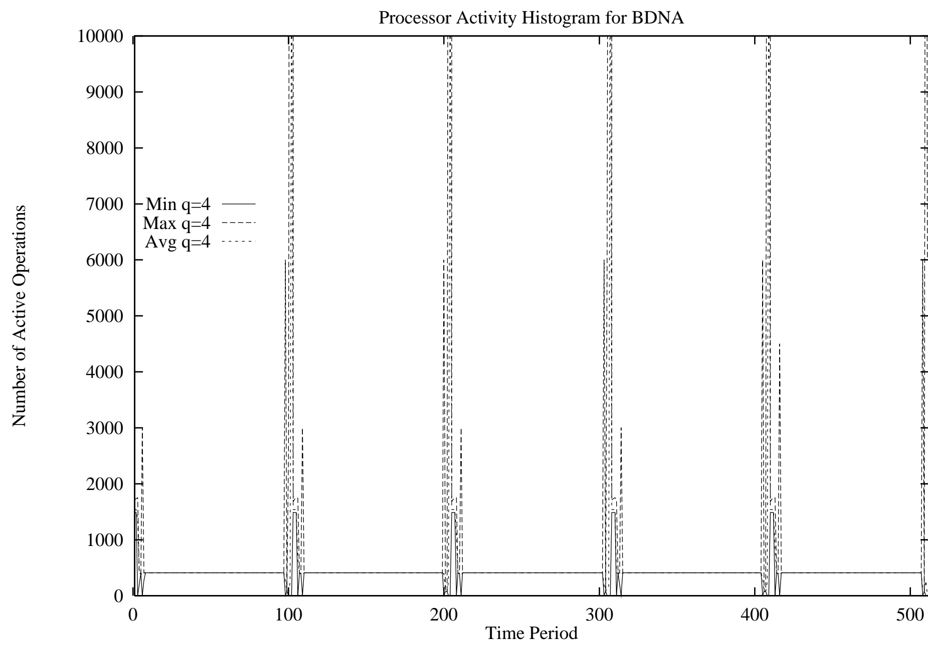

Figure 5.24: Parallel activity histogram for `bdna(NA)`

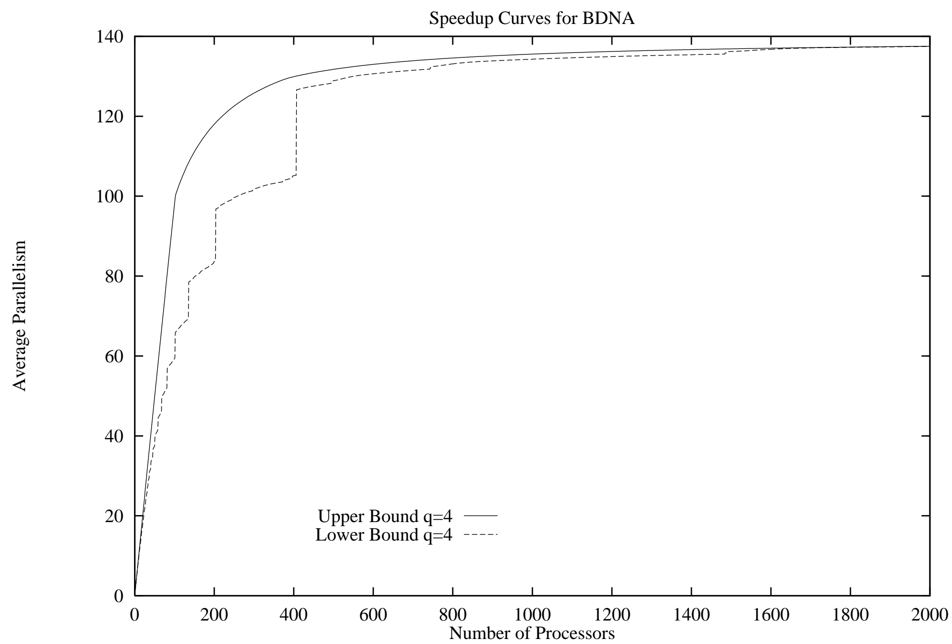

Figure 5.25: Average parallelism for `bdna(NA)`

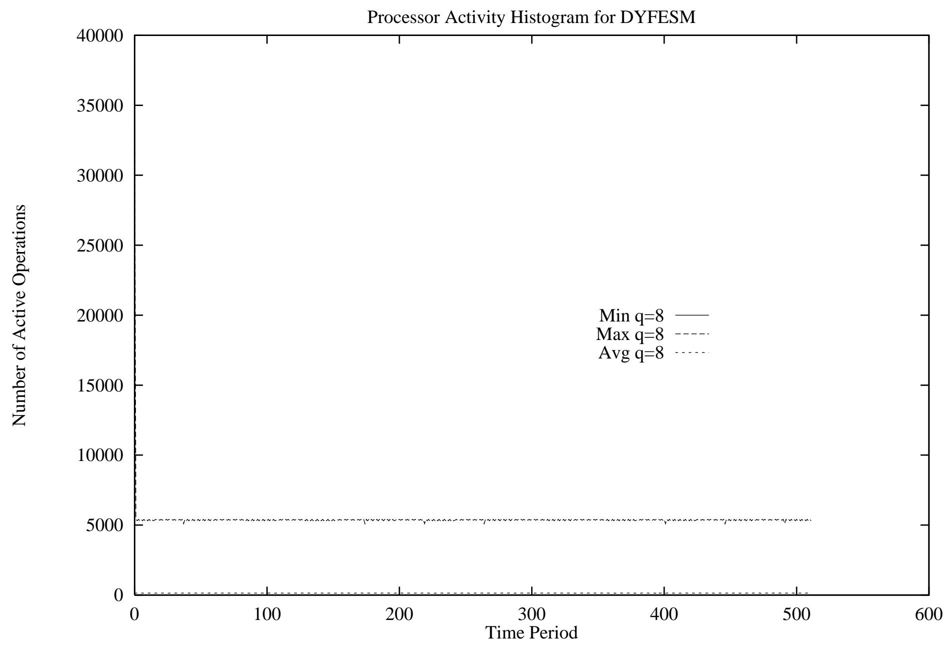

Figure 5.26: Parallel activity histogram for `dyfesm(SD)`

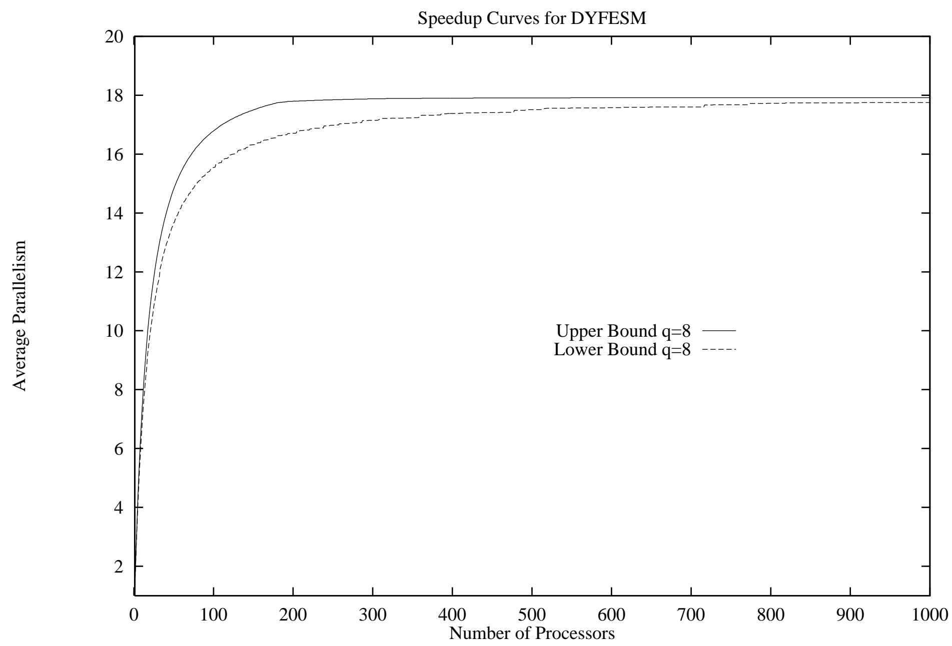

Figure 5.27: Average parallelism for `dyfesm(SD)`

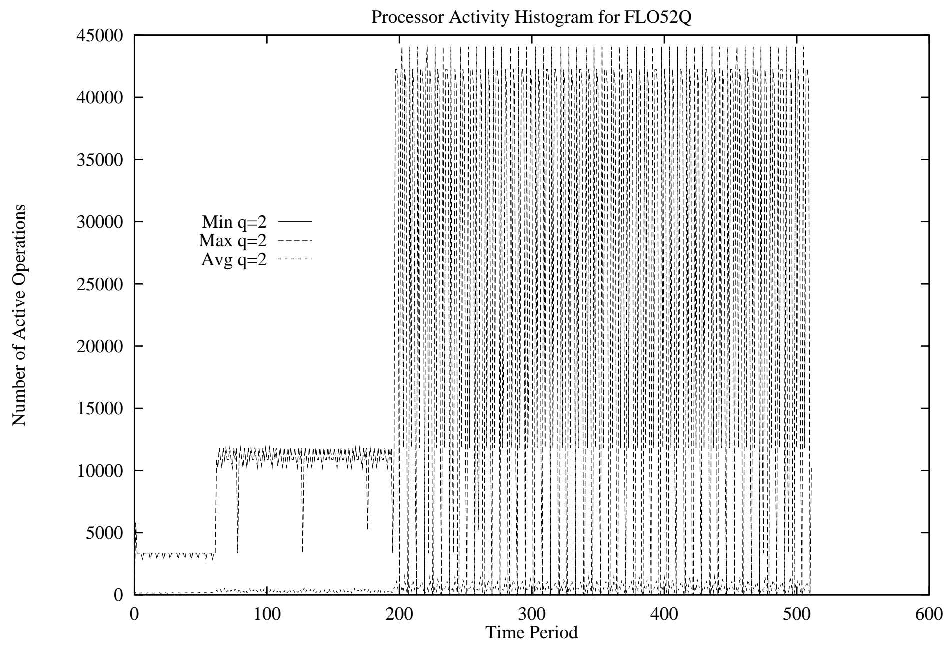

Figure 5.28: Parallel activity histogram for `flo52q(TF)`

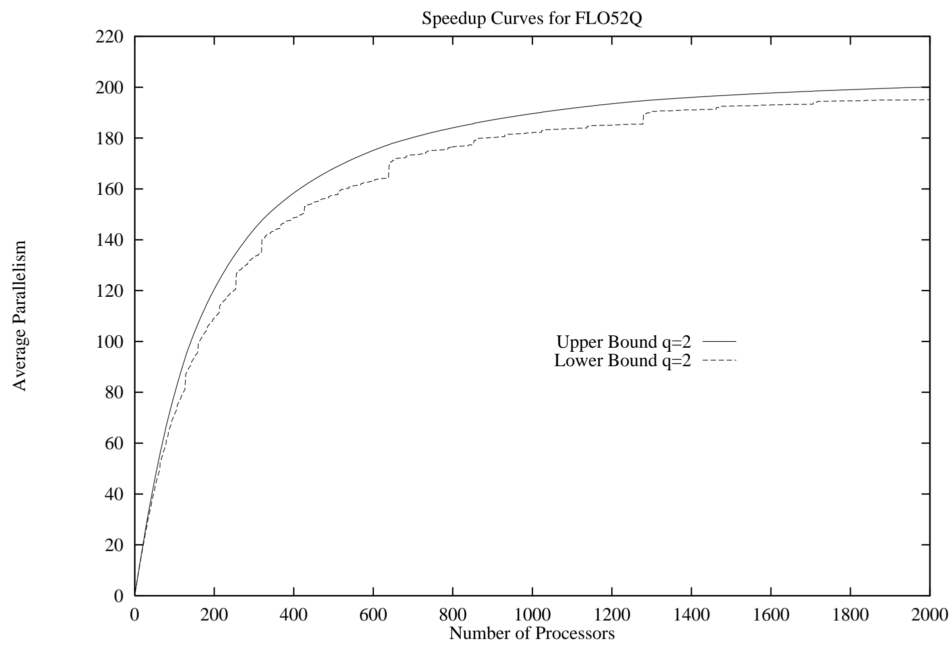

Figure 5.29: Average parallelism for `flo52q(TF)`

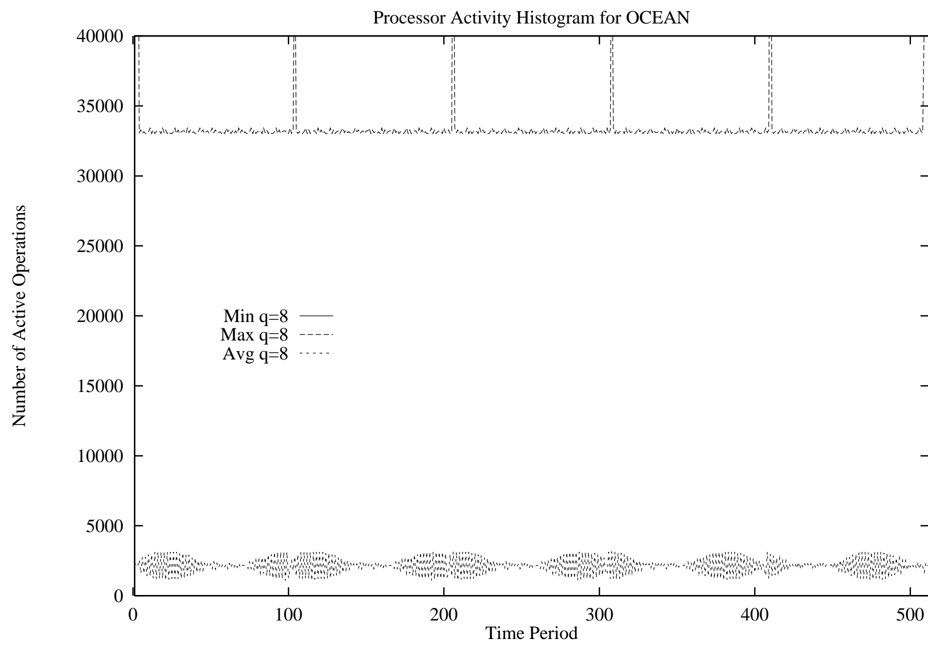

Figure 5.30: Parallel activity histogram for `ocean(OC)`

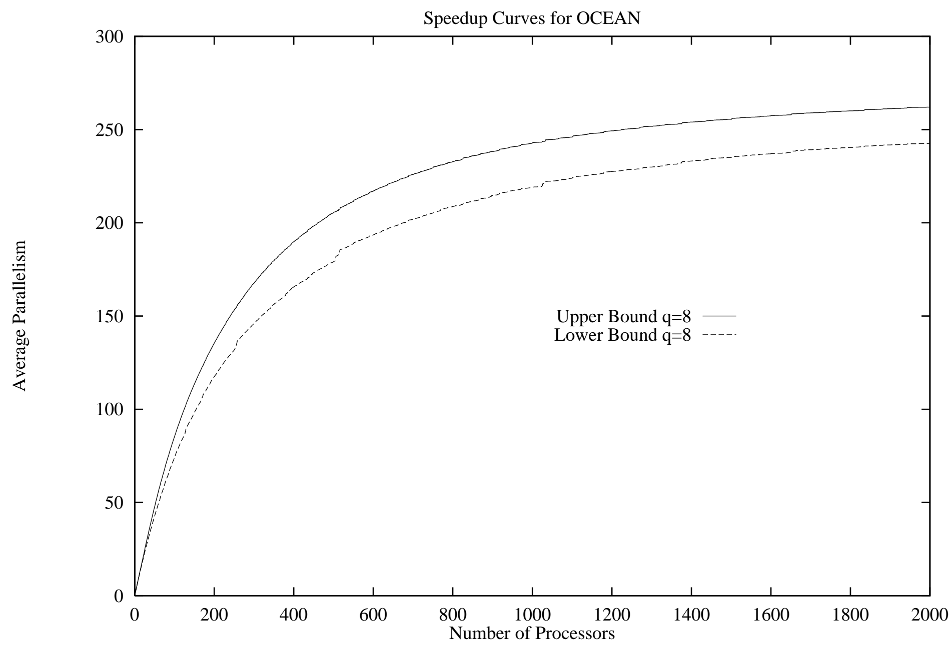

Figure 5.31: Average parallelism for `ocean(OC)`

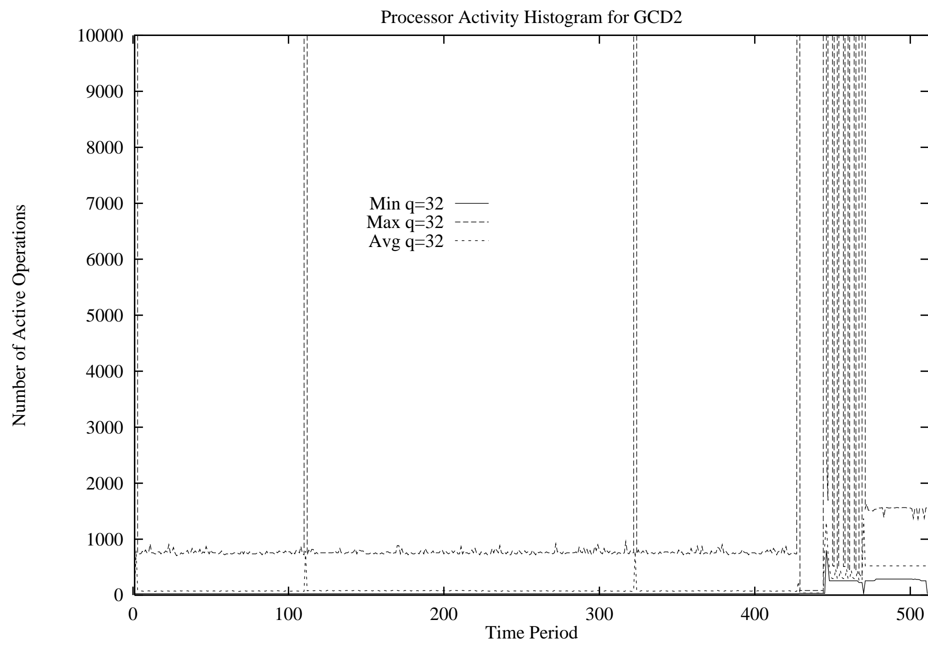

Figure 5.32: Parallel activity histogram for `qcd2(LG)`

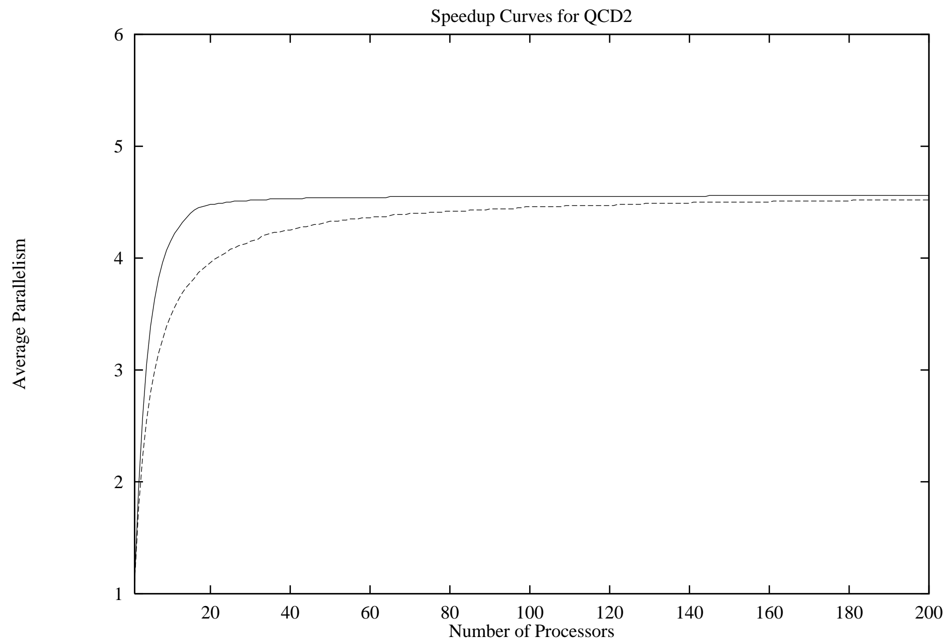

Figure 5.33: Average parallelism for `qcd2(LG)`

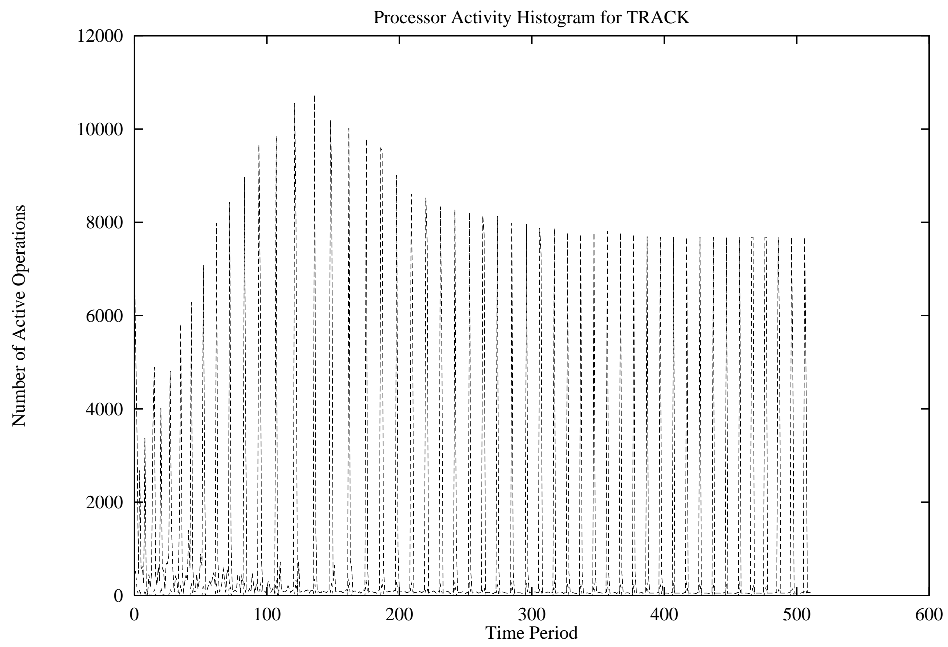

Figure 5.34: Parallel activity histogram for `track(MT)`

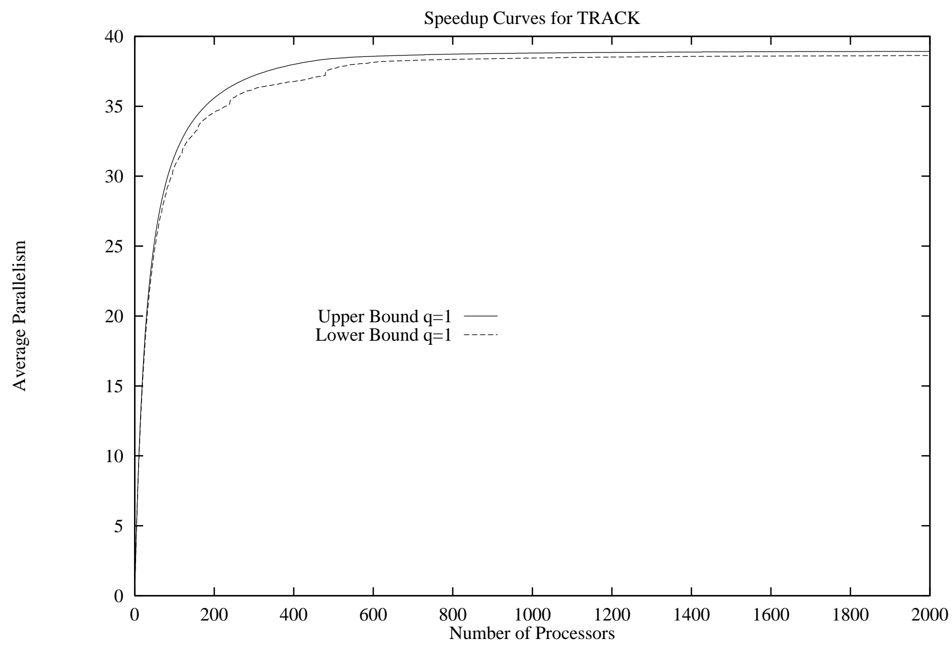

Figure 5.35: Average parallelism for `track(MT)`

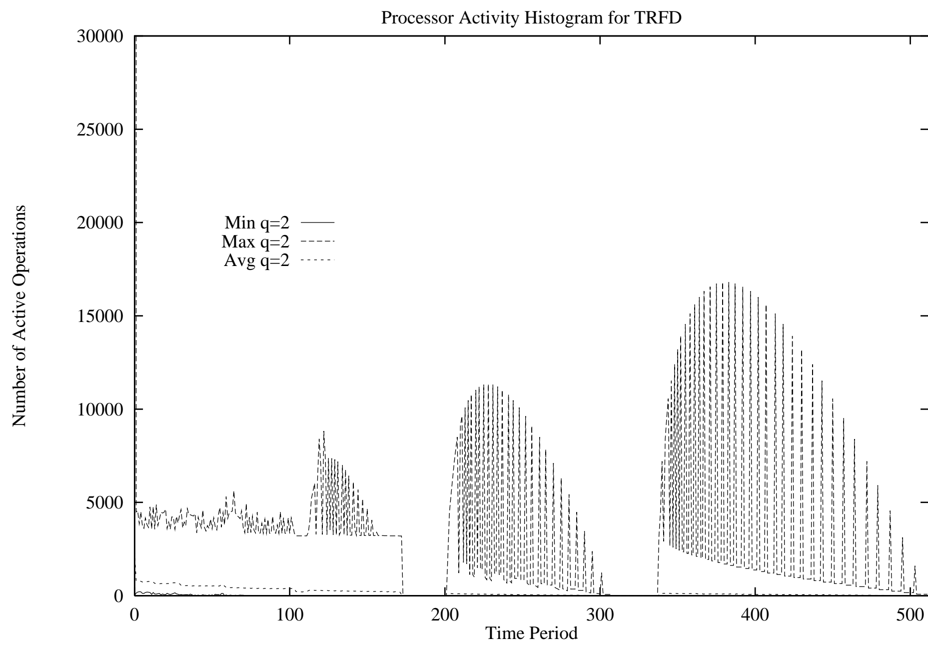

Figure 5.36: Parallel activity histogram for `trfd(TI)`

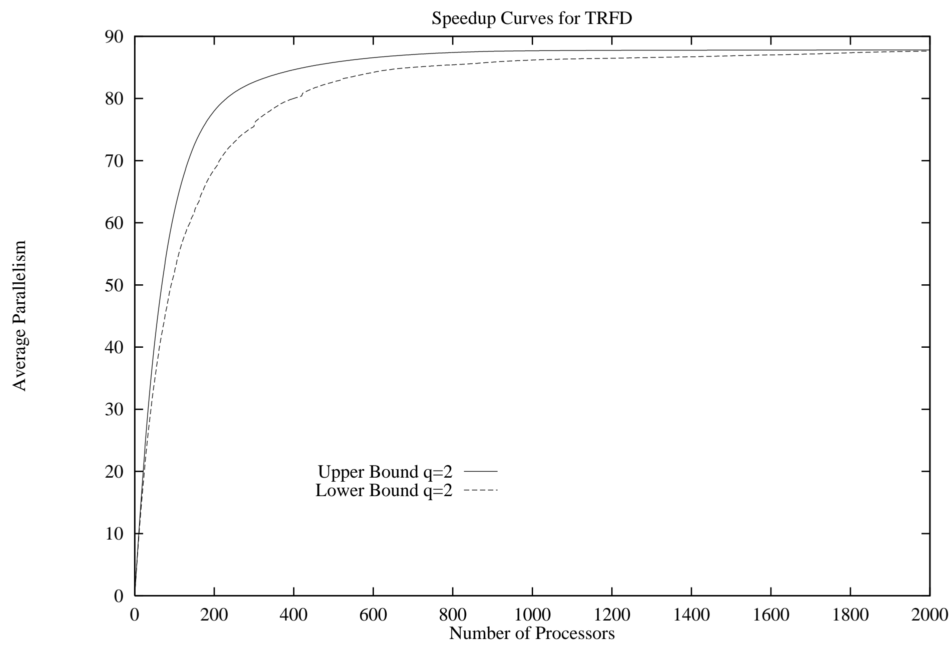

Figure 5.37: Average parallelism for `trfd(TI)`

Chapter 6

STRESS ANALYSIS

6.1 Introduction

An analogy can be formed between the dependence arcs present in a program and the load-bearing members in a physical structure. The stresses present in the load-bearing members of a structure can be calculated from the forces imposed on the structure. The forces present at each point in a structure and by analogy in a program need to be balanced in order to have a static equilibrium.

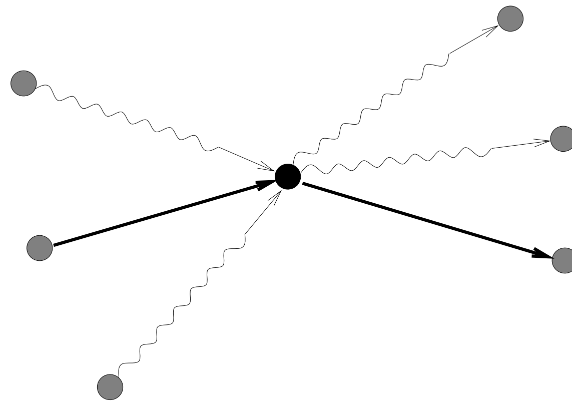

Figure 6.1: Critical path representation via node stress

The dependence arcs in the program can be divided into load-bearing and non-load-bearing components. The load-bearing components are those that give rise to the critical path. All other components are considered redundant because they are enforced by the load-bearing dependence arcs. Critical path analysis attempts to schedule all operations as early as possible in the parallel execution. The dependence arcs apply an opposing force to counteract the tendencies of operations to move earlier in time.

At every point in the program, each incoming dependence arc contributes a time when the operations may begin. The arc with the maximum timestamp is the critical dependence arc. In Figure 6.1, the critical path is illustrated by the bold arrows. The curved arrows represent the data dependences that are redundant in determining the execution time of the operation.

We describe in this chapter a technique for determining the impact of a particular statement on the critical path length of a program. This idea is the basis for the experiments in Chapters 7 and 9. In addition to defining an interesting idea, we present a tool to display directly the locations in the program that induce the highest stress on the operations contributing to the length of the critical path.

6.2 Overview of the Evaluation Method

The ability to calculate internal stress is a direct consequence of the methods used to calculate the critical path length. The method shown in Chapter 4 to instrument Fortran programs propagates timestamps through the program's dynamic data-flow graph. Stress analysis measures the differentials between the dependences that act on each node in the data-flow graph. The results of the stress analysis are locations that locally influence the critical path. Since the differentials are a local phenomenon and the critical path is a global phenomenon, some discrepancies do exist. It is possible for some of the statements reported by stress analysis to be redundant because the local forces induce stress that is not globally propagated.

Inherent parallelism, as calculated by critical path analysis, reflects dependences that are intrinsic to a program. In addition to these dependences, the program can be augmented with artificial control dependence arcs, as described in Chapter 4, to simulate the restrictions placed on execution to simulate loop-level parallelism.

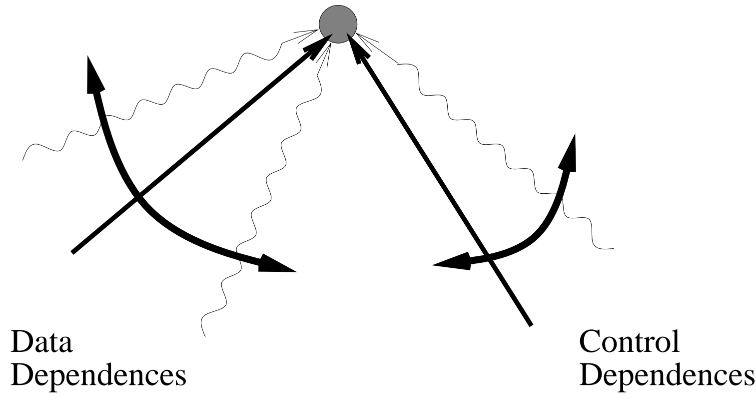

Figure 6.2: Grouping of dependence arcs for stress calculations

The ability to calculate a program's internal stress arises when the dependences or forces acting on a statement can be divided into several classes. Each class represents a programming idiom. For the purposes of this experiment we will define two classes, as shown in Figure 6.2. We will group all *flow*-dependences into the first class. The main idiom recognized in the first class is the use of recurrences to specify inductions and reductions. The second class will be control dependences. The control dependences enforce the flow of control required for conditionals and loops as well as the artificial dependences introduced to sequentialize the statements in loop iterations. The idioms recognized by the control dependences include speculative calculations and statement reordering.

In each class of dependences, we define the critical dependence as that which imposes the largest force on the operation. Alternatively, the force can be interpreted as the value of the

timestamp propagated by the dependence arc. The arc that has the largest timestamp is then the critical dependence arc.

In Figure 6.2, let X be the timestamp propagated by the critical dependence arc in the group of data dependences (indicated by the bold arc). From the same figure we can let Y be the timestamp propagated by the critical control dependence arc. The dependence differential is the absolute value of $X - Y$.

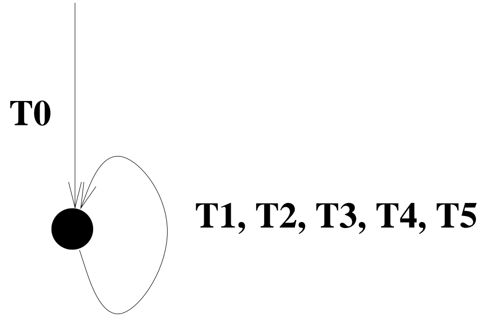

Figure 6.3: Recurrences in stress analysis.

To see how each of these classes is used, consider the effects of the dependences present during the execution of a program. Figure 6.3 illustrates a node with one control dependence arc that has a value of $T0$ and one data recurrence arc with a series of timestamps $T1, \ldots, T5$. In the series of timestamps generated and consumed by the recurrence, we have the property that $T0 < T1 < T2 < \cdots < T5$ and thus $T_i > T0$ for all $i > 0$. We can capture the information that the recurrence is degrading system performance by recording the sum of the dependence differentials between the timestamps propagated by the critical arc in each iteration, that is $\sum_i (Ti - T0)$.

Accumulating the dependence differentials may not be the best method of determining the importance of the stress placed on an operation. It may be more enlightening to calculate the maximum of the dependence differentials instead of the sum. The sum of the differentials has the intuitive advantage that it represents the entire contribution of the node in the data-flow graph to the critical path length. Since each instance contributes a small part to the total, summing the individual differentials reflects its average contribution.

However, two operations with the same stress value may not be comparable. If the first is a summation of a large number of tiny dependence differentials, it may not be advantageous to remove any particular one of the arcs that cause the stress. But, if the entire stress value reported is the result of one instance, a minor change to the program may have a large impact on the critical path length.

It may be important to use both methods of calculating the final weight of stress on an operation. For the remainder of this chapter we illustrate the examples using the method of adding the individual dependence differentials.

6.3 Results

In Sections 6.1 and 6.2 we saw that, at each point in the program, stress analysis calculates the forces (i.e., dependences) that act on a statement or operation. It is a simple matter to

compare the forces/dependences at each point to determine if they are uniform. The dependence differential as defined in Section 6.2 is used to determine uniformity. When the forces are not uniform, or one force is dominant, we can use this information to infer where the program needs to be changed to eliminate this force and potentially reduce the length of the critical path.

```
Line#
1       PROGRAM EX1
2       REAL A(5)
3       K = 0
4       DO I = 1, 5
5          K = K + 1
6          A(K) = I
7       ENDDO
8       END
```

```
     ----- SUMMARY: Stress Relation Recognition -----

EX1                 MODULE
 <S> EX1:5           K:20
```

Figure 6.4: Example one

Figure 6.4 gives a simple example of a dependence cycle that serializes a loop. Here we are dealing with an induction variable, but in the general case it could be any data dependence that forces future iteration to wait for the current iteration. Figure 6.5 shows graphically how the execution of the program proceeds. The thin arrows represent control dependences and the thick arrows represent data dependences. We can redraw this graph as shown in Figure 6.6 to make the critical path more evident. It can be seen in this figure that the cross-iteration *flow*-dependences from statement 5 to itself determines the critical path.

The format of the resulting output is as follows. For each routine in the program we create a list of statements. Each statement starts with the label `<S>` to show a stress relationship. Following the label is a statement tag of the form *routine*:*line*. For example, `EX1:5` shows a reference to line 5 in the file that contains the routine `EX1`. A list of pairs of the form *cause*:*count* follow the statement tag. If the cause is a data dependence induced by a variable in the statement, the name of the variable is shown. If the cause is a control dependence, the line number of the source of the control dependence is shown. In Figure 6.4, the stress report shows that the statement on line 5 of routine `EX1` was differentially delayed owing to the use the of variable `K`. In no other statement of this example was a differential delay or unbalanced force detected.

The count field of the stress report gives a relative weight to the importance of this particular stress. The higher the count, the more importance this statement has, at least locally. The result of `K:20` shows that the variable `K` caused a cumulative delay of `20` time units. This is easy to observe from the program. Each loop iteration starts at the same time. Statement 5 requires

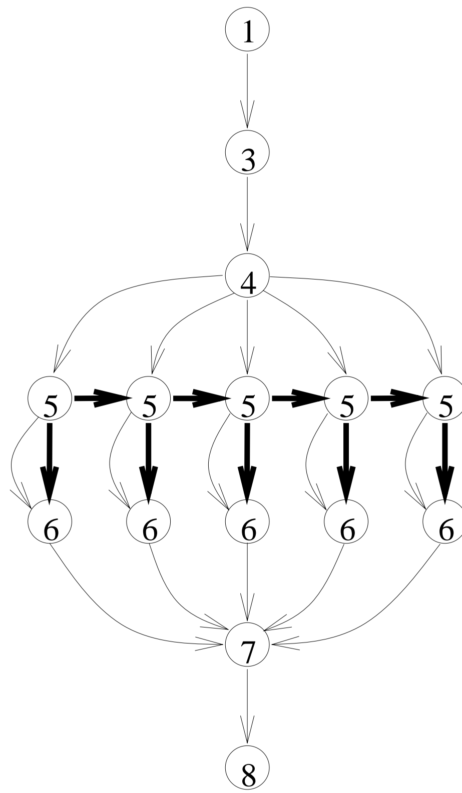

Figure 6.5: Parallel execution graph of statements in figure 5.4

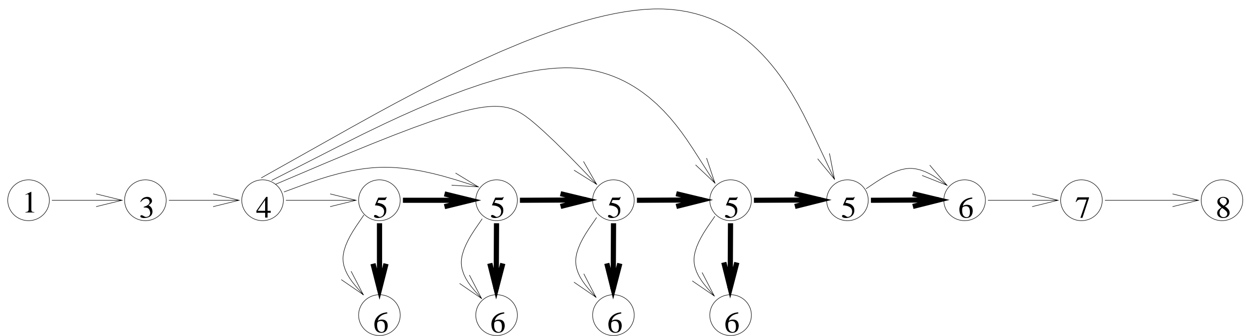

Figure 6.6: Parallel execution graph resulting from critical path analysis of the statements in figure 5.4

2 time units to execute, one for the addition and one for the store. At the first iteration we have a delay of 0. The second iteration must wait for the value to be computed during the first iteration, so it has a delay of 2. Similarly we add 2 to the delay for each subsequent iteration. The total $0 + 2 + 4 + 6 + 8$ is 20.

```
PROGRAM EX1
REAL A
INTEGER S$S1, S$S2, S$S3, S$S4, S$S5, S$S6
INTEGER C$EX1, K, C$K, N$Z, N$N, I, C$I, C$A
DIMENSION A(5), C$A(5)
COMMON /N$NBLOCK/ N$N, N$Z
CALL MSTRESS$ENTRY('EX1',C$EX1)
S$S1 = C$EX1
S$S3 = C$EX1
K = 0
CALL STRESS$STMT('EX1:3',C$K,1,0,1,S$S1,'K!1')
S$S2 = C$K
CALL STRESS$STMT('EX1:4',C$I,0,0,1,S$S2,'I!3')
S$S6 = C$I
DO I = 1, 5, 1
   S$S3 = C$I
   K = K+1
   CALL STRESS$STMT('EX1:5',C$K,2,2,C$K,S$S3,1,S$S3,'K!K!4!4')
   S$S4 = C$K
   A(K) = I
   CALL STRESS$STMT('EX1:6',C$A(K),1,3,C$I,C$K,S$S3,2,S$S4,S$S3,'A(K)!I!K!4!5!4')
   S$S5 = C$A(K)
   S$S6 = MAX(S$S5,S$S3,S$S6)
ENDDO
C$I = MAX(C$I,S$S6)
S$S3 = MAX(S$S3,S$S6)
C$EX1 = MAX(S$S3,C$I,C$EX1,C$K)
CALL MSTRESS$EXIT('EX1',C$EX1)
STOP
END
```

Figure 6.7: Example one instrumented code

Figure 6.7 shows a program listing after the original code has been instrumented. In addition to the support routines at the beginning and end of the routine (that maintain the dynamic call tree and collect parallelism statistics), we call the routine `STRESS$STMT()` to record the differences in the dependences arcs. The support routine `STRESS$STMT()` has a variable number of arguments. The first argument is the statement tag of the form *routine*:*line*. The tag is

printed by the stress report as a means of correlating the results with the original source code. The second argument is the shadow variable for the lefthand side of the assignment statement.

The next argument is an integer describing the number of time units required to execute this statement once all the operands are available. For example, if the shadow expression is `$X=MAX($A,$B)+2`, then the third argument to the `STRESS$STMT` call would be `2`. Two lists of shadow variables follow the operation count. Since these lists vary in length, each is preceded by the number of elements it contains. The first list of shadow variables includes the data dependences; the second list contains the control dependences. Finally, as the last argument we have a character string using the "!" character as a separator of the names for each of the shadow variables. If we are describing a control dependence, then the line number of the dependence is recorded as its name.

The routine `STRESS$STMT` determines the maximum timestamp in each list of shadow variables. It then finds the difference between the largest timestamp in each list and adds the dependence differential to an array kept for each statement. This array has one element for each of the dependences passed into the routine. A pointer to the final argument is cached in the internal structures so that the results can be displayed in terms of the program's original variables, instead of being displayed as a position in the call to the support routine.

6.3.1 Stress Analysis Examples

Another property of stress analysis is pointed out in Figure 6.8. Here we have two unrelated inductions, one in the variable `A` and one in the variable `X`. The first line of the stress report shows the data dependence on the variable `A` at line 5. The second line of the stress report, however, does not tell us about the variable `X`. It does correctly identify line 7 as a problem and reports that a control dependence to line 6 degraded performance. The counts on the two lines of the report show that the induction variable in line 5 is of much greater impact than the control dependence at line 7. What this is really indicating is that we need a `cobegin/coend` pair inside the loop body to compute the two sequences concurrently. We note that the small *count* value for `EX2:7` is a consequence of the effects of loop-level parallelism. The bulk of the delay required to calculate the induction w.r.t. `X` is already forced to occur owing to the calculation of `A`. Thus, the delay penalty for the second induction calculation is hidden by the first.

Finally, in Figure 6.9, we see that a loop can purely have control dependences as the only points of stress in the program. In this example we calculate three values into local variables and then use these three values to compute a result. The stress results show that statements 6 and 7 are delayed because of the immediately previous statement. Statement 6 must arbitrarily wait for the calculation of `A(I)`, and statement 7 must wait for the calculation of both `A(I)` and `B(I)`. Here the code could be written more efficiently by either distributing the computation into separate loops, or introducing a `cobegin/coend` pair to calculate concurrently the values of `A(I)`, `B(I)`, and `C(I)`.

6.4 Conclusion

Analogous to the load-bearing members in a physical structure, dependences give form and support to parallel programs. Interpreting the dependence arcs in a constrained program in a

Line#
```
1      PROGRAM EX2
2      A = 0
3      X = 0
4      DO I = 1, 10
5         A = A + 1
6         B = A + 1
7         X = X + 1
8         Y = X + 1
9      ENDDO
10     END
```

```
    ----- SUMMARY: Stress Relation Recognition -----

EX2                 MODULE
 <S> EX2:5           A:90
 <S> EX2:7           6:4
```

Figure 6.8: Example two

Line#
```
1      PROGRAM EX3
2      REAL A(10), B(10), C(10)
3
4      DO I = 1, 10
5         A(I) = I*2+1
6         B(I) = I*3+2
7         C(I) = I*4+3
8         D(I) = A(I) + B(I) + C(I)
9      END DO
10     END
```

```
    ----- SUMMARY: Stress Relation Recognition -----

EX3                 MODULE
 <S> EX3:6           5:30
 <S> EX3:7           6:60
```

Figure 6.9: Example three

similar manner to the components of a physical structure allows one to grasp intuitively the concept of a critical path and to calculate the "load-bearing" dependences in a program.

If we separate the dependences into data and control sets, we can compare the relevance of the arcs of each type on an operation in the program. Summing the difference in tension (i.e., timestamps) along the maximum arcs in each set indicates the general cause of the stress at that operation. If the data arcs have a higher sum of the differences then it is a recurrence that has serialized this section of code. On the other hand, if the control arcs have a higher sum, then it may be the artificial order of the statements under loop-level parallelism that is constraining the parallelism.

In the first case of degradations owing to recurrences, it may be possible to reduce the stress with a parallel recurrence solver, or through induction variable elimination. The second case of the control arcs being predominant may be eliminated by reordering the statements, or concurrently executing the lexically sequentialized statements.

Chapter 7

INDUCTION VARIABLES

7.1 Introduction

Parallelizing compilers are used to recognize explicitly the parallelism that is implicit in a program. Through the use of program transformations, the parallelizing compiler may be able to substitute an alternative algorithm that eliminates the data dependences contained in the original code. The process of induction variable elimination is an example of this class of transformation.

```
S1:   K = 0
S2:   DO I = 1, N
S3:      K = K + 3
S4:      A(K) = A(K) + 1
S5:   ENDDO
```

Figure 7.1: Example of a loop with an induction variable

A variable K is called an induction variable of a loop L if the value of the variable at every point in L is strictly a function of the iteration number and loop invariant values. The values of the induction variable form an arithmetic sequence. A basic induction variable is defined to be an induction variable K such that every time the variable K changes value, it is incremented or decremented by a constant value [ASU86]. In the experiments reported here, we are interested in generalized induction variables [EHLP91], that is, we allow multiplication as well as addition for the basic induction variables and do not restrict the values to be in an arithmetic sequence.

For the loop shown in Figure 7.1, the variable K in statement S3 is incremented by a constant (3) in each iteration of the loop. It is important to note that the code, *as it is written*, has a cross-iteration *flow*-dependence from S3 to itself. The effect of this *flow*-dependence is to serialize the loop since iteration I+1 must wait until the value of K, in iteration I, has been computed.

If we replace the induction variable K with an expression involving the loop's index variable (shown in Figure 7.2), then we break the *flow*-dependence and allow the loop to execute in parallel. The parallel execution is only possible if we also eliminate the *output*-dependence on

```
S1:   K = 0
S2:   DO I = 1, N
S3:      K = 3*I
S4:      A(K) = A(K) + 1
S5:   ENDDO
```

Figure 7.2: After induction variable elimination

`K` possibly via localization of the scalar variable. We are assuming throughout these examples that the computation of the index variable for a loop does not introduce a serializing bottleneck. We can make this assumption because the iteration space of a loop can be pre-computed from the information in the `DO` statement.

```
S2:   DO I = 1, N
S4:      A(3*I) = A(3*I) + 1
S5:   ENDDO
```

Figure 7.3: After forward substitution and dead code elimination

The final result is shown in Figure 7.3. If the value of `K` is not live on exit from the loop, then we can do forward substitution and deadcode elimination to obtain the final form.

The sequence of transformations presented here are traditionally used by parallelizing compilers to expose parallelism that is not immediately obvious. We have transformed a loop that must be executed sequentially into the final form that can be executed as a parallel `doall` loop. These transformations are not always beneficial. You will notice that we replaced a single addition from the first example with 2 multiplications in the last. This is the opposite of strength reduction. If the multiplications are more expensive than the addition and the iteration space of the loop is small, we may have introduced more overhead into the program than can be eliminated through the use of parallelism. Also, if the elimination of induction variables does not eliminate cross-iteration *flow*-dependences caused by other statements in the loopnest we may produce an equivalent serial loop that has higher overhead than the original. Because of this effect, commercial parallelizing compilers such as KAP/Concurrent do not remove the induction variable unless it can create a parallel loop.

It is natural that parallelizing compilers should attempt to recognize all induction variables in a program, and to remove those that are inhibiting the parallel performance of the program. One would like to be able to measure the success of the parallelizing compiler at this task. However, at the present time, we are unaware of any technique that can measure the performance degradation of induction variables that either were not recognized as such or were not removed from the program.

Through the technique of critical path analysis (defined in Chapter 4), we present a method by which the effects of generalized induction variables can be measured. In Section 7.2, we

describe the general overview of our method. Section 7.3 describes the implementation used to compute the results. Section 7.4 displays the results from the experiments. Section 7.5 presents our conclusions.

7.2 Overview of the Evaluation Method

Induction variable elimination is a mature area where many useful techniques exist. However, no technique has been proved optimal in the presence of incomplete information resulting from imprecise interprocedural analysis or variables whose values are known only at run-time. In the absence of optimal solutions, we are reduced to making comparative measurements. To make the comparison, you need a "better" method with which to compare.

In Chapter 4 we described a technique that can be used to calculate the inherent parallelism present in an application; the method used for the experiments described in this chapter is a variation of that technique. For the experiments presented in this chapter, we use the speedup measured by critical path analysis as the basis for our comparisons.

The calculation of inherent parallelism involves the instrumentation and execution of the source program. The instrumentation relies on a shadow variable associated with each of the original variables, containing a "timestamp" that indicates the earliest time that any computation using that value may begin.

In general terms, the critical path length computation requires that each statement compute the expression `MAX(`*forall operand timestamps*`)+`*time for operators*; that is, each statement must wait until each of its operands is available before it can begin, and then the results from the statement will be available a constant amount of time later.

This general strategy is adequate if we are interested only in tracking one parameter per variable (i.e., the timestamp). However, to calculate the effects of induction variables, we need to be able to classify all the variables in the program.

To accomplish the bookkeeping required for this additional information we define the shadow variable as a pointer to an auxiliary structure instead of a timestamp. Most Fortran 77 compilers, including ours, are capable of allocating all variables statically at compile-time. We can take advantage of this behavior by noting that uninitialized shadow variables contain the value 0. We can interpret this value as a NULL pointer in the support libraries, and use this value to signify that a shadow structure has not been allocated for this variable. Whenever a shadow variable is passed into the support library we first check the shadow's value; if it is NULL we create a shadow structure and place the address of the shadow structure into the shadow variable of the Fortran program.

In addition to the modifications in the instrumentation needed to collect shadow structures, we also need to calculate the mode of each variable definition. In Section 7.2.1 we define a lattice of states that the mode of a variable definition may take. These states describe what is known about the type of the variable. It classifies the values as unknown, constant, loop invariant, induction variable, or anything. The instrumentation support library handles the transitions among these states owing to operations such as addition or multiplication being applied to the values.

7.2.1 State Space for Variable Identification

Table 7.1 lists the states that the mode of a statement make take. The statement's mode is the classification of the value that it produces. Each statement that produces a value is classified according to this table. If a statement does not produce a value, then by default it is placed in the `BOT` state. Every statement and every value produced by the program is classified as being in one of these states.

State	Definition
`TOP`	nothing of interest
`MAx`	a multiple of an additive induction variable
`AMx`	an addition of a multiplicative induction variable
`Mx`	a multiplicative induction variable
`M1`	a partial multiplicative induction variable
`Ax`	an additive induction variable
`A1`	a partial additive induction variable
`LIV`	a loop invariant variable
`CST`	a constant
`BOT`	could be anything

Table 7.1: Definition of lattice points

Each variable has a field in its shadow structure pointing back to the statement that produced the value. By following this pointer back to the statement, we can classify the value contained in a variable to determine if it is part of an inductive sequence. If the variable is an induction variable, then we can eliminate its effects on the length of the critical path to calculate a bound on the benefit derived from eliminating the induction calculation.

Transitions between the states in Table 7.1 are shown by Figures 7.4 and 7.5. Details are shown in Tables 7.2 and 7.3. When an operation is performed during the program's execution, the states of the values are used as indices into the tables to find the resulting state.

In Figures 7.4, 7.5, and 7.8 we have omitted the arcs from `BOT` to all other nodes, and from all nodes to `TOP` for clarity in the diagrams. Self-transition arcs are also omitted from the diagrams. In each of the figures the arcs may be traversed on meeting with the appropriate variable state. Figures 7.4 and 7.5 describe symmetric transition diagrams. The initial state in the diagram may be that of either the left or the right operand of the operation.

Table 7.2 and Figure 7.4 show the transition diagram when two values meet under an addition/subtraction. We treat subtraction in an identical manner to addition. This treatment allows for the possibility of recognizing more potential induction sequences, but it also opens the possibility to erroneously classify a sequence of operations as an induction. No distinction is made between the forms $X - Y$ and $Y - X$. As can be seen from the diagram, an addition/subtraction can only move into the states `A1`, `Ax`, `AMx`, or `TOP`.

Table 7.3 and Figure 7.5 show a similar situation to Figure 7.4 except that the transitions assume we are processing a multiplication/division operation. As in the case for transitions on addition and subtraction, division is treated identically to multiplication. Again this opens the method to inaccuracies owing to the ordering of the operands. The expressions X/Y and Y/X

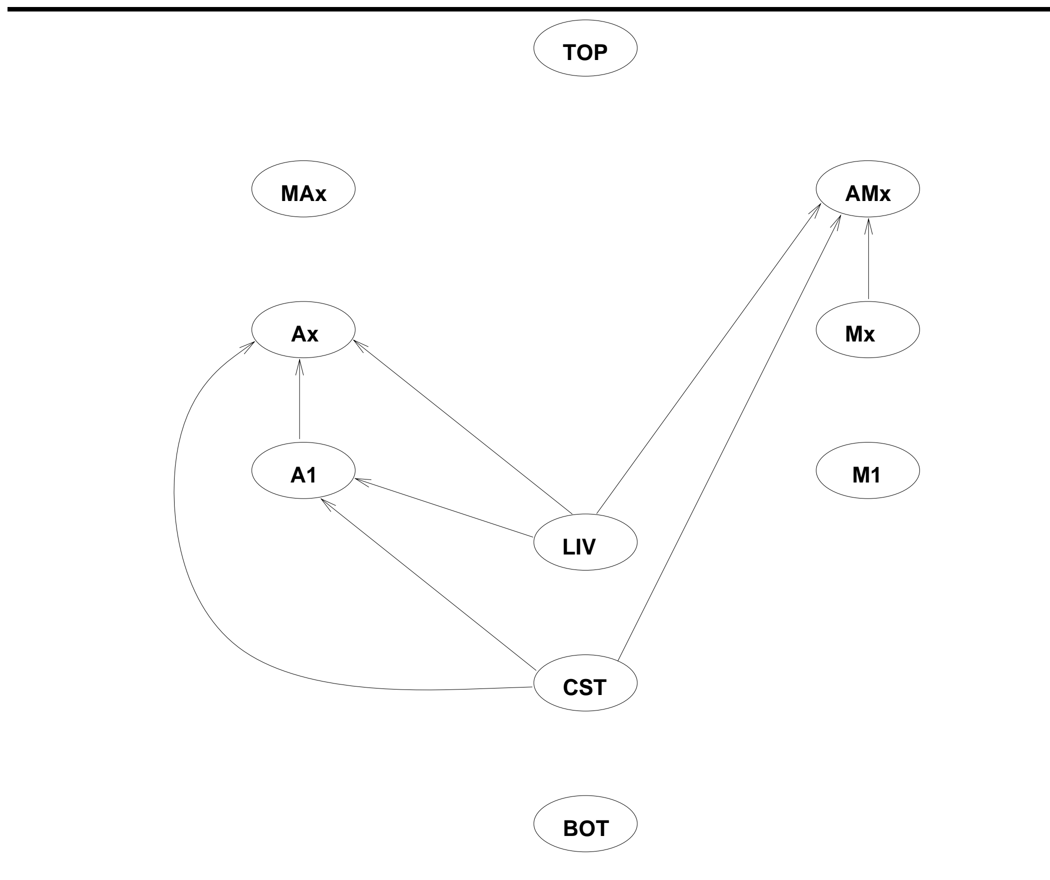

Figure 7.4: Lattice for ADD merge

BOT	CST	LIV	A1	Ax	M1	Mx	MAx	AMx	TOP
CST	CST	A1	Ax	Ax	TOP	AMx	TOP	AMx	TOP
LIV	A1	A1	Ax	Ax	TOP	AMx	TOP	AMx	TOP
A1	Ax	Ax	A1	Ax	TOP	TOP	TOP	TOP	TOP
Ax	Ax	Ax	Ax	Ax	TOP	TOP	TOP	TOP	TOP
M1	TOP	TOP	TOP	TOP	TOP	TOP	TOP	TOP	TOP
Mx	AMx	AMx	TOP	TOP	TOP	TOP	TOP	TOP	TOP
MAx	TOP	TOP	TOP	TOP	TOP	TOP	TOP	TOP	TOP
AMx	AMx	AMx	TOP	TOP	TOP	TOP	TOP	TOP	TOP
TOP	TOP	TOP	TOP	TOP	TOP	TOP	TOP	TOP	TOP

Table 7.2: Transition table for addition and subtraction merge

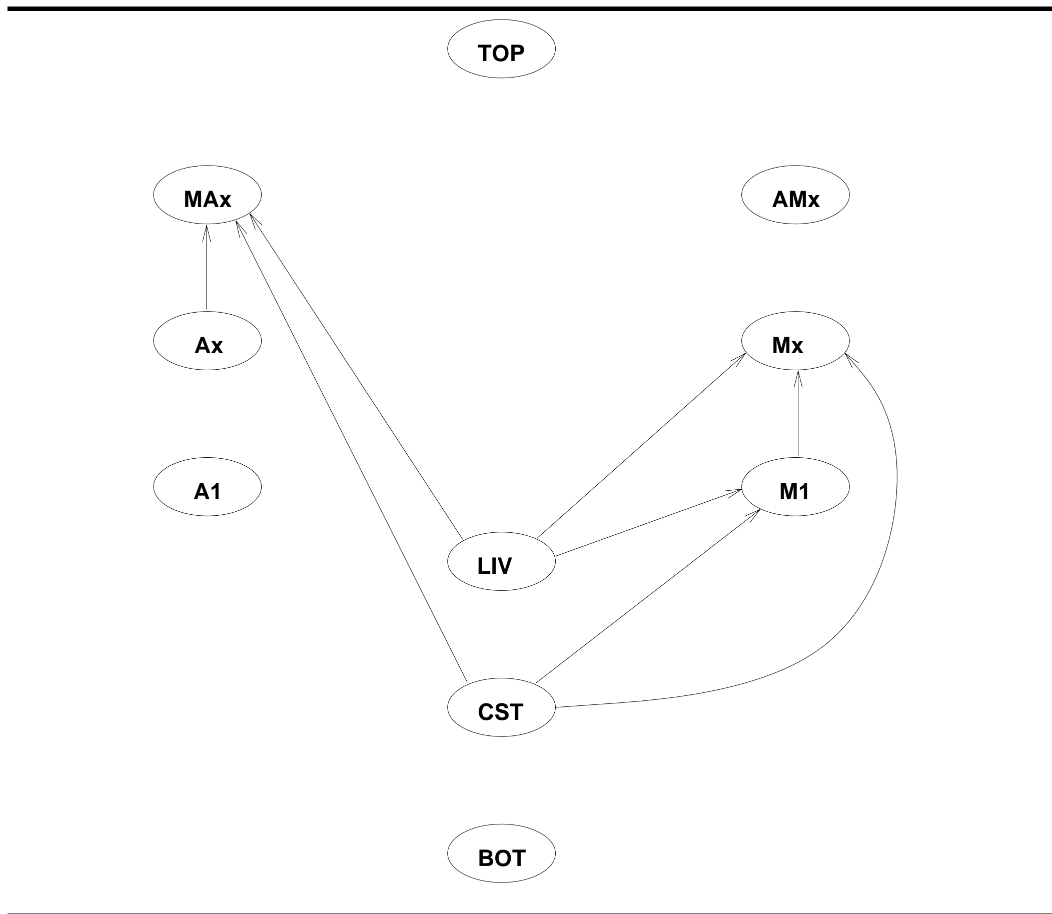

Figure 7.5: Lattice for MUL merge

BOT	CST	LIV	A1	Ax	M1	Mx	MAx	AMx	TOP
CST	CST	M1	TOP	MAx	Mx	Mx	MAx	TOP	TOP
LIV	M1	M1	TOP	MAx	Mx	Mx	MAx	TOP	TOP
A1	TOP	TOP	TOP	TOP	TOP	TOP	TOP	TOP	TOP
Ax	MAx	MAx	TOP	TOP	TOP	TOP	TOP	TOP	TOP
M1	Mx	Mx	TOP	TOP	M1	Mx	TOP	TOP	TOP
Mx	Mx	Mx	TOP	TOP	Mx	Mx	TOP	TOP	TOP
MAx	MAx	MAx	TOP	TOP	TOP	TOP	TOP	TOP	TOP
AMx	TOP	TOP	TOP	TOP	TOP	TOP	TOP	TOP	TOP
TOP	TOP	TOP	TOP	TOP	TOP	TOP	TOP	TOP	TOP

Table 7.3: Transition table for multiplication and division merge

are treated in the same manner. Using these transitions, we can only move into the states M1, Mx, MAx, or TOP.

7.2.2 Use of Transition Tables

To illustrate the use of the transition diagrams, consider the simple loop shown in Figure 7.6.

```
S1:   K = 0
S2:   DO I = 1, 10
S3:      K = K + 1
S4:      A(I) = K
S5:   ENDDO
```

Figure 7.6: Example for induction variable recognition

Assume that the shadow of a variable is declared to be the name of the original variable preceded by a '$'. Initially all shadow variables are 0 (i.e., NULL). On the first reference to each shadow variable by the support library, the shadow is initialized to a newly created shadow structure. Initially this structure refers with its .from field to a statement whose mode is in the BOT state.

After the execution of statement S1 we have $K.from=S1 and S1.mode=CST. The DO statement initializes $I.from=S2 and S2.mode=Ax. We assume that all index variables used in DO loops are classified as additive induction variables.

The righthand side of statement S3 is the expression K+1. The first time statement S3 is executed we set $K.from=S1 which is outside the loop, so instead of using S1.mode, we substitute the mode of LIV to indicate that at this point in the loop, K's value is loop invariant. The constant 1 obviously has the mode CST, and the transition table for addition says that the meet of the state LIV with the state CST produces the state A1. The state A1 indicates that this variable might be an induction variable, but it is too early to be sure.

After executing statement S3 we have $K.from=S3 and S3.mode=A1. Executing statement S4 copies the information in $K into $A(I). However, since $I.from=S2 and S2.mode=Ax, which is not constant or loop invariant, we must assign S4.mode=TOP. This additional step allows our method to distinguish induction on array elements from reductions over the array elements.

The second iteration of the loop produces slightly different results. Statement S3 again has the expression K+1. This time $K.from=S3, which is inside the loop; therefore we do not modify the mode of $K. The mode of $K=S3.mode=A1. The meet of A1 and CST from Figure 7.4 produces Ax, which is the final state, describing the variable as an induction variable.

Again S4 copies $K into $A(I) with the alteration that the use of a subscript on the lefthand or righthand side that is not loop invariant caused the mode of the statement to be TOP state. For any subsequent iteration we have the same pattern repeating. The mode of the shadow $K has stabilized to the state Ax.

7.2.3 Enhancement to the Model

A problem exist in the use of the lattices that have been proposed. In particular, the prior mode of the statement is not considered when computing the new mode of the statement. Consider the simple loop in Figure 7.7.

```
S1    DO I = 1, N
S2       K = 0
S3       K = K + 1
S4       K = K + 1
S6    ENDDO
```

Figure 7.7: Problem example for induction variable recognition

It is obvious that the variable K in this loop has a constant value at each statement in the loop. However, if we use the lattice defined for addition, we have S2.mode=CST, S3.mode=A1, and S4.mode=Ax. From this example we see that increasing the height of the lattice (i.e., adding A2, A3, A4) will just defer the problem since we can always construct an example that will incorrectly classify the statements.

We have settled on a two-stage definition (A1, Ax) for this experiment to eliminate the obviously incorrect classifications. The problem with dynamic execution is that statements S3 and S4 look like an unrolled loop. It would seem possible to use the dynamic loop nesting structure of the program to eliminate these erroneous classifications. However, this would entail the creation of a database of variables assigned inside each loop, along with the overhead of maintaining the database. The overhead for this approach makes it infeasible.

The solution we have adopted for this problem of incorrectly classifying some statements is to create a lattice transformation based on the assignment or store operator.

Figure 7.8 is the final important transition diagram. Unlike Figures 7.4 and 7.5, Figure 7.8 describes a nonsymmetric transition diagram. The current state of the lefthand side of the assignment is always used as the initial state. In Table 7.4 the new state of the statement is computed by using the previous state as the index into the rows, and using the new state as the index into the columns. The basic premise of this transition is that a mode can never propagate down in the lattice. The mode must remain the same or move up. For example, if the previous mode was Ax and we want to define the statement to have mode CST, then we would assign the mode of TOP to the statement.

The basic idea is that the classification state of a statement can never go down in the lattice. If the lefthand side of the assignment has a previous mode, then we use Figure 7.8 to determine the new state by meeting the old state with the state of the righthand side. Thus for our example we merge the previous definition of the variable K with the current definition of K+1.

For the first iteration of statement S2, we have the previous definition S2.mode=BOT, and the current definition to be CST. The meet of these states assigns the definition of S2.mode=CST. Likewise for statements S3 and S4 we assign the modes S3.mode=A1 and S4.mode=Ax.

During the second iteration of the loop, we now have the current definition of K coming from statement S4.mode=Ax. The meet of Ax with CST attempts to move down the lattice. This is not allowed; thus the result is TOP. Therefore, we define S2.mode to be TOP. The state of

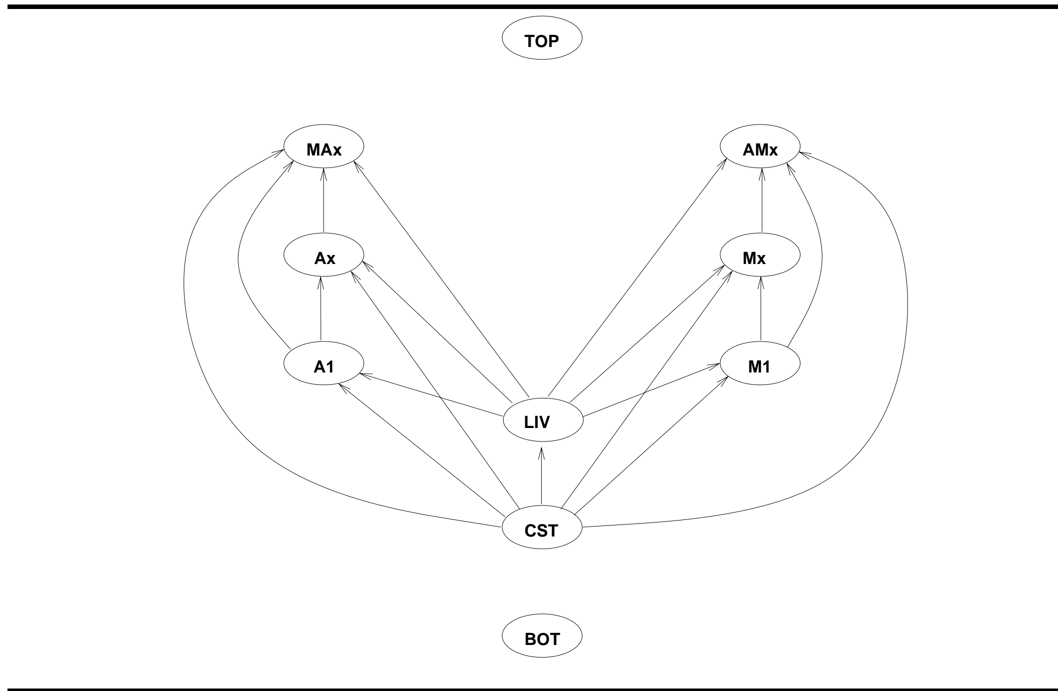

Figure 7.8: Lattice for STORE merge

BOT	CST	LIV	A1	Ax	M1	Mx	MAx	AMx	TOP
CST	CST	LIV	A1	Ax	M1	Mx	MAx	AMx	TOP
LIV	TOP	LIV	A1	Ax	M1	Mx	MAx	AMx	TOP
A1	TOP	TOP	A1	Ax	TOP	TOP	MAx	TOP	TOP
Ax	TOP	TOP	TOP	Ax	TOP	TOP	MAx	TOP	TOP
M1	TOP	TOP	TOP	TOP	M1	Mx	TOP	AMx	TOP
Mx	TOP	TOP	TOP	TOP	TOP	Mx	TOP	AMx	TOP
MAx	TOP	TOP	TOP	TOP	TOP	TOP	MAx	TOP	TOP
AMx	TOP	TOP	TOP	TOP	TOP	TOP	TOP	AMx	TOP
TOP	TOP	TOP	TOP	TOP	TOP	TOP	TOP	TOP	TOP

Table 7.4: Transition table for store merge

K, S2.mode, propagates through the statements causing both S3.mode and S4.mode to be TOP. This is a stable configuration, so no further changes will be made.

7.2.4 Deficiencies of this Approach

Our method is based on a simple state machine for the value in each variable. This approach has limitations to its accuracy. In the example shown in Figure 7.9, we have two inner loops.

```
S0    K = 0
S1    DO I = 1, N
S2       K = K + 1
S3       DO J = 1, N
S4          K = K ** 2
S5       ENDDO
S6       DO J = 1, N
S7          K = K + 1
S8       ENDDO
S9    ENDDO
```

Figure 7.9: Inaccuracies for the induction variable recognition

We can see from a hand analysis of this fragment that the only induction variable is the variable K in loop S6 at statement S7. However, because of the way that loop invariant variables are computed, we declare K at statement S2 to also be an induction variable.

To see why our method arrives as this conclusion we will simulate the algorithm. In the S1 loopnest, first $K.from=S0, which produces $K.from=S2 and S2.mode=A1, after statement S2 is executed. Then in loop S3 we determine that $K.from=S4 and S4.mode=TOP.

On entry to loop S6, we see that $K.from=S4, which is outside the S6 loop; thus instead of using the mode TOP, we substitute the mode LIV during the first iteration producing $K.from=S7 and S7.mode=A1. The second iteration has $K.from=S7 (i.e., inside the loop body) converging to S7.mode=Ax.

On the second iteration of the outer loop, at statement S2 we have the variable $K.from=S7, and S7.mode=Ax. Meeting the previous state of S2.mode (i.e., A1) with S7.mode (i.e., Ax), we climb the lattice and store Ax. Thus the stable configuration of this loopnest is: S2.mode=Ax, S4.mode=TOP, and S7.mode=Ax.

As with Section 7.2.3, this is another example where a stack-based database of variable classifications would be needed to save and restore the mode of each variable on entry and exit from a loop.

Since the purpose of our experiment is to compute an upper bound of the effects of induction variables on the critical path length, we can ignore these inaccuracies as long as they err on the side of recognizing more induction variables than are present in the program.

The technique presented here will overestimate the number of induction variables in the program and thus can only provide an upper bound to the parallelism that is present in the programs after removing all induction variables.

7.3 Implementation

The need to maintain multiple data items in the shadow structure changes the style of the instrumentation code. In Chapter 4 we were able to use the built-in Fortran intrinsics to propagate the shadow timestamps since they were basic Fortran datatypes. Now that a structure is required to hold the shadow information, we are forced to use auxiliary support routines for most of the work, instead of coding the bulk of the instrumentation inline. The reliance on support routines has the advantage that we can arbitrarily change the behavior of the instrumented program without changing the instrumentation code, but it has severe performance problems owing to the subroutine call overhead when compared with inline code.

```
typedef struct shadow {
    DATATYPE        timestamp;
    DATATYPE        industamp;
    DATATYPE        last_access;
    struct node    *from;
} SHADOW;
```

Figure 7.10: Shadow structure datatype

The structure shown in Figure 7.10 lists the fields of the structure that are maintained by the support routines. The two timestamps, `timestamp` and `industamp`, are the current length of the critical path with and without the influence of induction variables respectively. The `last_access` field gives the global iteration number for the current loop. Finally, the `.from` pointer refers to the statement that defined this value. The mode of this value can be obtained by following this pointer.

The support routines all follow a similar pattern. The first argument to the routines is a character string that describes the source statement that is being executed. The string starts with the name of the routine being executed. If there exists `ENTRY` statements in the body of the routine, then the routine name will be the name in the `SUBROUTINE` or `FUNCTION` declaration. If we are referring to a specific line in the source code, a colon and the line number are appended to the string. For example, if we are at line 13 of the subroutine `MATMUL`, then the first argument to the support routines would be `'MATMUL:13'`.

We have chosen a varargs format for the argument list of the auxiliary routines. Many of the support routines are required to evaluate the effects of an arithmetic expression on the critical path length, and on the classification of the variables. The last two sections of the argument list to the support routines are the operator string and the operand list. The number of operands present in the subroutine call is calculated from the number of `'_'` operators in the operator string. We will describe the operator string in more detail in Section 7.3.2.

The instrumented code relies on many support routines. To delineate when a module boundary has been crossed, the routines `AB$PENTRY` and `AB$PEXIT` are used. The entry routine assigns a value to the root of the control dependence graph. The exit routine copies the final node of the control dependence graph into the variable used to summarize the execution time for the routine.

```
AB$PENTRY( 'MATMUL', S$S1, 1, S$MATMUL )
AB$PEXIT( 'MATMUL', S$MATMUL, 1, S$S999 )
```

Similar routine are used for the entry and exit from loops.

```
AB$LENTRY( 'MATMUL:36', S$S36, 1, S$S35, 'C_CAA$', C$N )
AB$LEXIT( 'MATMUL:36', S$S36, 1, S$S45 )
```

Each iteration of a loop in the routine is assigned a unique global iteration number. The only purpose of the `AB$IENTRY` and `AB$IEXIT` routines are to assign the correct GIN to the `current` field of the active loop. The use of the loop tag as the argument allows error checking for consistency.

```
AB$IENTRY( 'MATMUL:36' )
AB$IEXIT( 'MATMUL:36' )
```

The main instrumentation support routines are called `AB$STMT` and `AB$CONTROL`. All possible induction values must be defined via an assignment statement in the program. The `AB$STMT` routine interprets the effects of the operations applied in the statement on the mode of the variable and the timestamp of the results. The routine `AB$CONTROL` is similar, except that the control dependence calculations can never define an induction variable.

```
AB$STMT( 'MATMUL:13', S$S3, 2, S$S1, S$S2, C$K, '_C+$', C$K )
AB$CONTROL( 'MATMUL:13', S$S3, 2, S$S1, S$S2, C$K, '_C+$', C$K )
```

I/O is an important consideration for the timing model and for determining induction variables. Any value read by an input statement is defined to have the mode `TOP`. The execution time of the program is also altered by the I/O statements. We enforce the rule that all I/O must occur in the same order as in the original program.

```
AB$INPUT('MATMUL:44', 0, S$S5, 1, S$S4, '$')
AB$OUTPUT('MATMUL:44', 0, S$S5, 1, S$S4, '$')
```

7.3.1 Invariant Variable Recognition

The purpose of each call to the support routines is to update the shadow structures. One important aspect of the shadow variable is the iteration count (`last_access`) associated with each variable. To determine an optimistic set of induction variables correctly we must first be able to determine if a variable is loop-invariant.

We will define a variable to be loop-invariant if the last update to the variable occurred before the current loop was entered. If the variable has been modified since the current loop was entered, then we classify the variable as loop-variant.

The method used to determine when a variable was last modified is based on a global iteration number (GIN). The GIN is initialized to zero at the start of the program and incremented

by one each time a loop iteration begins. We record two values associated with the DO statement. The first value is stored in the field named start. The GIN is assigned to the start field just before the DO statement is executed. The second value is stored in the field named current. We increment the GIN and assign it to the current field at the beginning of each iteration of the loop.

The GIN provides us with a frame of reference against which to determine if a variable contains a value that is invariant in the current loop. Each loop in the program is tagged with a pair (S, C) that are the global iteration numbers for the start (i.e., first iteration) and the current (i.e., the current iteration).

If we have a variable whose last_access field has the value A, and $A < S$, we are sure that the variable has not been modified since the invocation of the loop. We can prove this statement by noting that if $A < S$, we had not even arrived at the DO loop in question when this variable was modified. Conversely, if we have a variable with last_access field A where $A \geq S$, then we are assured that the variable has been modified since the entry into the loop because the access stamp A could not be created until the loop had entered.

The mode of LIV is reserved for the case of a loop-invariant variable. When executing the '_' (push) instruction of our stack-based interpreter, we check the last_access field of the shadow variable, and if the last definition of the variable is outside the current loop, then we assume that the variable is invariant with respect to the current loop. It should be noted that we are being slightly lax in the traditional definition of loop-invariant for induction variable recognition.

```
S1    DO I = 1, N
S2       M = ···
S3       DO J = 1, N
S4          ...
S5          K = K + M
S6          ...
S7       ENDDO
S8    ENDDO
```

Here we recognize K as being an induction variable (which is correct in the innermost loop) and M as being loop-invariant; the only relaxation we have done is not to be specific about what loops we are referring to for M and K. In our model, every reference of K dynamically after the assignment in S5 will be assumed to be an induction variable.

7.3.2 RPN Statement Form

The recognition of induction variables requires that we trace the operations that are performed on the values.

We translate the expression into a stack-based form and assign a one-character token to each operation. The expression D=(A+B)*(B+C) is translated into the call:

```
AB$STMT('SUB:2', S$S2, 1, ···, C$D, '__+__+*$' C$A, C$B, C$B, C$C)
```

We interpret the statement to mean that, `S$S2` and `C$D` are both assigned values from the expression `MAX(S$S1, C$A, C$B, C$B, C$C)+3`.

To interpret this statement, '_' represented the push operation, '+' is an addition, '*' is a multiplication, and '$' is the end-of-string or result token. The following list of tokens describes the operator set used by the stack machine. We are only interested in sequences of operations involving addition/subtraction or multiplication/division. The '.' operator is used for all other binary operations, causing the mode to be defined as `TOP`.

The operators "+-*/.XA" are all binary operators. For all these cases we Pop A, Pop B, calculate C, and Push C onto the operand stack. For all the binary operators, C.time = MAX(A.time, B.time).

'+' The addition operator is treated identically as the '-' operator. This operation takes one time unit. The mode of C is calculated with the add-merge table in Figure 7.4 as C.mode = add-merge(A.mode, B.mode).

'*' The multiplication operator is treated identically as the '/' operator. This operation also takes one time unit. As with addition, C.mode = mul-merge(A.mode, B.mode).

'.' The point signifies an operation that forces the result of the merge to be `TOP`. This operation takes one time unit. We force C.mode = `TOP`.

'X' Assume the top two items on the evaluation stack are the two halves of a complex variable. Create a new entry with C.mode = `TOP` (i.e., assume complex variables can not be induction variables).

'A' This is the anything or abstract operation. It takes no time and removes one entry from the evaluation stack. The mode of C is the more general of the modes of A and B.

We also have a collection of unary operators. These operators all remove a value from the stack and optionally push a result onto the stack.

'N' The negation operation takes one time unit. It forces the mode of the operand at the top of the stack to be `TOP`.

'@' Pop an operand off the evaluation stack and assume it is being used for array subscripting. If this value is not loop-invariant, then set the flag `global=TOP`. Set the global time, timestamp = MAX(timestamp, A.time).

',' Pop an operand from the stack and assume it is an operand to a function call or an intrinsic function call. Set the global time, timestamp = MAX(timestamp, A.time).

Finally, we have operators that do not remove any operands from the stack and optionally push a result onto the operand stack.

'C' Push an operand of mode `CST` onto the evaluation stack.

'_' This operation corresponds to a variable usage. It pushes the next variable in the argument list of the call to the support routine onto the evaluation stack. If the variable was defined outside the current loop, then it is said to have mode `LIV` when placed on the evaluation stack. If the mode of the variable is one of `Ax`, `Mx`, `MAx`, or `AMx`, then erase the `industamp` field of the shadow to avoid propagating the time constraint.

'F' A function call, force the result to be TOP.

'I' An intrinsic function call, force the result to be TOP.

'$' The end of string or result token.

7.3.3 Example Instrumentation

To see how this fits together, we will start with the example shown in Figure 7.11. This example shows a doubly nested loop that defines a triangular iteration space. In this simple example we are copying a portion of the matrix A into the array B. Each iteration of the loopnest increments a counter and copies an element of the triangular portion of the matrix A into a linear array B. This is a good example for induction variable usage because the form of the program with the variable K removed is almost unreadable, while it is easy to determine the purpose of the original version of the program.

```
1      PROGRAM TCOPY
2      REAL A, B
3      INTEGER I, J, K, N
4      PARAMETER (N=100)
5      DIMENSION A(N, N), B(N*N)
6      K = 0
7      DO I = 1, N, 1
8         DO J = 1, I, 1
9            K = K+1
10           B(K) = A(I,J)
11        ENDDO
12     ENDDO
13     PRINT *, B(K)
14     STOP
15     END
```

Figure 7.11: Example of a triangular loop with an induction variable

The instrumented version of the program follows in Figure 7.12. Notice the proliferation of the calls to the support routines to update the shadow structures. Each operation in the original program must be individually simulated by the support routines to catch any sequence of operations that might define an induction variable.

After we have instrumented the program and created an executable by compiling and linking with the support library, we are ready to execute the program. Listed in Table 7.5 are the results produced by the support routines.

In this example we have identified line 9 as the definition of an additive induction variable (Ax). The output also reports the obvious fact that line 6 defines a constant value. Lines 10 and 13 are of mode TOP because we are indexing into an array with a variant expression. The use of the loop variant expression check removes reductions from consideration while allowing

```
PROGRAM TCOPY
REAL A, B
INTEGER I, J, K, N
INTEGER C$I, C$J, C$K, C$N, C$A, C$B, N$N, N$Z, C$TCOPY
INTEGER S$S1, S$S2, S$S3, S$S4, S$S5, S$S6, S$S7
INTEGER S$S8, S$S9, S$S12
PARAMETER (N=100)
DIMENSION A(N, N), C$A(N, N), B(N*N), C$B(N*N)
COMMON /N$NBLOCK/ N$N, N$Z
CALL AB$PENTRY('TCOPY',S$S1,C$TCOPY)
K = 0
CALL AB$STMT('TCOPY:7',S$S2,1,S$S1,C$K,'C$')
CALL AB$LENTRY('TCOPY:8',S$S3,1,S$S2,C$I,'CCCAA$')
CALL AB$CONTROL('TCOPY:8',S$S8,1,S$S3,'$')
DO I = 1, N, 1
   CALL AB$IENTRY('TCOPY:8')
   CALL AB$LENTRY('TCOPY:9',S$S4,1,S$S3,C$J,'C_CAA$',C$I)
   CALL AB$CONTROL('TCOPY:9',S$S7,1,S$S4,'$')
   DO J = 1, I, 1
      CALL AB$IENTRY('TCOPY:9')
      K = K+1
      CALL AB$STMT('TCOPY:10',S$S5,1,S$S4,C$K,'_C+$',C$K)
      B(K) = A(I,J)
      CALL AB$STMT('TCOPY:11',S$S6,2,S$S4,S$S5,C$B(K),'_@__@@_$',C$K
         C$I,C$J,C$A(I,J))
      CALL AB$IEXIT('TCOPY:9',S$S7,3,S$S7,S$S4,S$S6)
   ENDDO
   CALL AB$LEXIT('TCOPY:9',S$S4,2,S$S7,S$S4)
   CALL AB$IEXIT('TCOPY:8',S$S8,3,S$S8,S$S4,S$S3)
ENDDO
CALL AB$LEXIT('TCOPY:8',S$S3,2,S$S8,S$S3)
PRINT *, B(K)
CALL AB$OUTPUT('TCOPY:14',0,S$S9,1,S$S3,'_@_$',C$K,C$B(K))
CALL AB$PEXIT('TCOPY',C$TCOPY,1,S$S9)
STOP
END
```

Figure 7.12: Instrumented code of triangular loopnest

Stmt Tag	Record Type	Variable Type	Original Speedup	Serial Time	Parallel Time	New Speedup	New Parallel Time
TCOPY		<none>	1.5	15151	10102	1893.9	8
TCOPY:6	STMT	CST					
TCOPY:7	LOOP	<none>	1.5	15150	10101	2164.3	7
TCOPY:8	LOOP	<none>	1.0	15150	15150	30.4	499
TCOPY:9	STMT	Ax					
TCOPY:10	STMT	TOP					
TCOPY:13	STMT	TOP					

Table 7.5: Results from induct variable evaluation of TCOPY

inductions on array elements to be considered. As expected from this example, the inherent parallelism of the algorithm changes from 1.5 in the presence of the induction variable to 1893.9 after the affects of the induction variable are removed.

7.3.4 Inductions on Array Elements

The dynamic induction evaluation routines can find additional opportunities that other static methods may not find. The first example uses an array element as an induction variable. Given the loop fragment shown in Figure 7.13, we have a simple induction calculation using an array element as the induction variable.

```
DO I = 1, N
   A(K) = A(K) + 3
   ...
ENDDO
```

Figure 7.13: Induction on an array element

Since the subscript to the array is a loop invariant value, we can treat the array element in the same way we treat a scalar variable for the purposes of induction variable recognition. We can transform this by eliminating the induction of the array element `A(K)` as shown in the code fragment in Figure 7.14.

7.3.5 Interprocedural Induction Variables

Since the technique tracks the dynamic call graph of the program's execution, we are not limited to intraprocedural induction variable recognition. In the example shown in Figure 7.15, we have a calculation defining an induction on the variable `K` in the routine `SUB`. This induction is not statically visible from an intraprocedural analysis of the main routine.

If we recognize that the prior pair of routines define an interprocedural induction variable, then we can eliminate the induction by re-writing the code as shown in Figure 7.16. This is

```
AKO = A(K)
DO I = 1, N
   A(K) = AKO + I*3
   ...
ENDDO
```

Figure 7.14: Removal of an induction on an array element

```
PROGRAM TEST              SUBROUTINE SUB(I)
COMMON K                  COMMON K
DO I = 1, N               K = K + 2
   CALL SUB(I)            ...
   ...                    ENDDO
ENDDO                     END
END
```

Figure 7.15: Interprocedural example of an induction variable

only one possible way of eliminating the induction. In addition to removing the induction it may also be required to remove *output*-dependences by localization of variables and forward substitution before the parallelism in the outer loop can be exploited.

```
PROGRAM TEST              SUBROUTINE SUB(I,KO)
COMMON K                  COMMON K
KO = K                    K = KO + 2*I
DO I = 1, N               ...
   CALL SUB(I,KO)         ENDDO
   ...                    RETURN
ENDDO                     END
END
```

Figure 7.16: Example removal of an interprocedural induction variable

7.3.6 Correctness of the Upper Bound

The results generated by the method presented in this chapter would be more concrete if we could show that they represent a guaranteed upper bound on the improvement due to the recognition of induction variables. If we assume that only variables in the states `Ax`, `Mx`, `MAx`, and

AMx are induction variables, then we do not have a guaranteed upper bound on the improvement in the parallelism for removing the induction variables.

Fortunately, by choosing an alternative set of states, we can show that the technique always simulates the effects of removing all induction variables. The assumption made to assure correctness of the bounds is that at least one statement defining the induction variable must be executed on every iteration. Additionally it is assumed that the only operations allowed in the induction calculation are addition, subtraction, multiplication, and division. Other operations could be considered by augmenting the transition lattices with the requisite states.

The states A1, Ax, M1, Mx, MAx, AMx will be classified as induction variables. This classification means that whenever we compute the shadow expressions involving variables in these states we ignore the timestamp for these variables. If we ignore a timestamp, we can substitute the value 0 for the timestamp since for any timestamp t, $\max(0, t) = t$.

The first observation we make is that the classification of the variables does not matter during the first iteration of any loop. We enforce lexicographical ordering of the statements, and the structured control dependences from DO loops and IF statements. Every statement in a loop iteration is dependent on the previous statement in the iteration and on the loop header. Thus, the timestamp of either the loop header or the previously executed statement will dominate the timestamp of any induction variables used by a statement in the first iteration of a loop.

For a variable to be classified as an additive induction variable it must be incremented by the same value in every iteration of the loop. Since it must be incremented in every iteration, it must also be incremented in the first iteration. If we assume the first increment looks like K=K+1, then we know that during the first iteration the state of K on the righthand side must be LIV since this is the first definition of K in the loop body. For K to be an induction variable, we must add a loop invariant value. This loop invariant value will either be in state CST or LIV. From Table 7.2 we see that the meet of these states generates the state A1 which we have classified as an induction variable. A similar argument holds for multiplicative induction variables.

At the end of the first iteration we must have executed the statement that defines the induction variable. From the previous paragraph we have shown that this is sufficient to classify the variable as an induction. Since we have defined it to be illegal for a variable to move down in the lattice, the variable must remain an induction variable for every subsequent iteration. If the state of the variable ever reaches TOP, then we have erroneously represented the variable for a few iterations, but we have erred on the side that guarantees an upper bound.

7.4 Experimental Results from the Perfect Benchmarks

The techniques described in Sections 7.2 and 7.3 are only useful when applied to a program and the results produced are observed.

Table 7.6 shows the results of applying these techniques to a selection of the Perfect Benchmarks programs. In Table 7.6 the first column lists the name and abbreviation for the program. The second column shows the base loop-level speedup for the program. The third and fourth columns show the serial execution time and the parallel execution time when all operations are considered part of the critical path. The fifth column is the speedup obtained by ignoring dependences caused by the presence of induction variables. The last column shows the execution time when the effects of induction variables are ignored.

	Original Program			Induction Variables Removed	
PROGRAM	Loop-Level Speedup	Serial Execution Time	Parallel	Loop-Level Speedup	New Parallel Time
adm(AP)	45.5	881953965	19376214	46.0	19174249
arc2d(SR)	336.0	3111626215	9260772	336.0	9260772
bdna(NA)	139.5	1486171820	10653112	139.5	10653104
dyfesm(SD)	17.8	490791996	27545569	17.8	27544641
flo52q(TF)	206.9	1137847045	5500562	206.9	5500562
mdg(LW)	5.3	3702617624	695104451	5.8	635430792
ocean(OC)	272.1	3662121314	13459093	387.8	9442602
qcd2(LG)	1.8	511357356	277166352	2.2	237825002
track(MT)	40.6	119833481	2949194	43.3	2765112
trfd(TI)	87.9	637768445	7256699	808.1	789253

Table 7.6: Average parallelism: with and without induction constraints

Several observations can be made from this table. Of all the programs considered by this experiment, only two program (`ocean(OC)` and `trfd(TI)`) have a significant change in the speedup when ignoring the effects of induction variables. In [EHLP91], it was noted that for the same two programs, induction variable elimination was required to parallelize these codes. The program `flo52q(TF)` has no change between columns two and five, which is surprising since it is able to recognize several additive induction variables.

7.5 Conclusion

The advent of techniques capable of measuring the inherent parallelism in sequential programs has provoked the need for more advanced analysis. In particular, the existing techniques can only disregard the effects of *output*- and *anti*- dependences when calculating the critical path length. We propose a method of identifying and accounting for the impact of induction variables during the calculation of the critical path.

We have shown in this chapter that it is possible to measure the effect of induction variables on critical path parallelism. Generally, it is not as significant as previously believed. Only two of the programs exhibited a noticeable improvement when the effects of induction variables were removed.

Chapter 8

STATIC DEPENDENCE ANALYSIS

8.1 Introduction

Data dependence analysis is the most important step in the automatic detection and exploitation of implicit parallelism. Currently, parallelizing compilers are used on all commercial parallel computers and many sequential computers (for increasing cache performance). The data dependence information allows the compiler to restructure the code to take the most advantage of the machine.

As discussed in detail in Chapter 2, the goal of data dependence analysis is to determine whether or not a system of equations has an integer solution inside a given region of $\mathbf{Z}^n$. One of the first techniques used, answered the dependence question accurately [Tow76]. However, this method was too expensive to use in any practical compiler. For this reason, faster but approximated techniques have been developed that will sometimes assume the existence of a solution to the system of equations when no solution actually exists. The use of approximate techniques that sometimes have conservatively to assume dependence will not lead to an incorrect parallel program, but could preclude some optimizations.

Approximate dependence techniques, especially those developed by Utpal Banerjee in his Ph.D. thesis [Ban79] have been widely adopted. In the last few years, a renewed interest in the subject of dependence analysis has arisen, and techniques that are in some cases more accurate than Banerjee's have been developed [LYZ89, KKP90].

Despite the importance of dependence analysis, there is little experimental analysis of the accuracy of these approximate techniques. Such analysis could be useful to guide research and also help compiler writers decide what approach to take and the risks involved. In this chapter, we present an experimental analysis of some approximation techniques including the GCD method and three variants of Banerjee's test [Ban88]. To evaluate the accuracy of these methods for linear subscripts, we compare them against two exact methods, based on integer-programming algorithms.

Initially we had chosen to use only one integer-programming based dependence algorithm, the Branch-and-Bound method based on the simplex algorithm. However, the Omega Test [Pug91] has become available and, in order to use the most powerful dependence test that is widely available, we have chosen to include this test.

In addition to the published variations of Banerjee's inequalities, for rectangular and trapezoidal iteration spaces, we have extended the implementation to handle unknown upper and lower bounds correctly. The Banerjee infinity test was developed after some preliminary results from the experiments in this chapter. We later found that J. R. Allen [All83] describes a method that works in a similar manner. We believe it is important to include this variation as a separate dependence analysis test to determine empirically the necessity of knowing the loops bounds.

We feel that a major contribution of our study is to show the effectiveness of a naive version of the standard implementation of Banerjee's inequalities. It was found that the existing versions of Banerjee's inequalities could not be applied in over 70% of the loops because the upper bound was unknown. Another guiding piece of information was that the lower bound and stride were unknown in less than 12% of the loops tested. Given this high disparity between the occurrences of unknown information in the loop bounds, it is natural to study a dependence test that can use but does not require the loop's bounds.

8.2 Description of the Experiments

To evaluate the dependence tests described in Chapter 2, we used the Perfect Benchmarks [CKPK90], which are a collection of thirteen Fortran programs that represent the applications most frequently run on parallel and vector computers. To be consistent with the other experiments in this thesis, we will omit the program `spice(CS)` from this experiment. Thus, we are using a total of twelve programs. All the programs in this experiment were processed via KAP/Concurrent (which was also used by all the other experiments in this thesis) before the static dependence analysis is performed. The use of KAP/Concurrent removes induction variables and performs some source level scalar optimization. The number of potential dependences is larger than in the original code owing to two of the transformations performed by KAP. The KAP preprocessor unrolls and distributes loops to increase performance.

As mentioned above, we will be omitting the program `spice(CS)` from this experiment. One obvious change with this omission will be a marked decrease in the number of array references found in subscripts. The program `spice(CS)` is almost statically unanalyzable today owing to the presence of indirect addressing.

Two experiments were run corresponding to two different sequences of dependence tests. In both cases, the dependence tests were applied to each potential dependence in the order specified by the sequence. The results of these experiments are shown in Tables 8.3 and 8.4 where we also list the dependence test sequence used in each case in the order they were applied.

All the data dependence tests used in the experiments are defined in detail in Chapter 2. For the purposes of this chapter, we can assume that Banerjee's tests (rectangular iteration space, trapezoidal iteration space, and infinite or unbound iteration space) and the integer-programming tests were applied once for each direction vector of the potential dependences. The constant and the GCD tests were also applied once for each potential dependence.

To be able to compare the different measures, the units of all counts kept in this experiment are the "number of feasible direction vectors"; this statement is clarified below. An accumulator is associated with each dependence test. It is incremented each time the corresponding test is the first to detect independence. The accumulators associated with Banerjee's tests and the integer-programming tests are incremented by one each time the corresponding test breaks a potential dependence for a given direction vector. The constant and GCD test accumulators

are incremented by the number of feasible direction vectors for the potential dependence since these two tests apply to all direction vectors. Notice that, counting in this way, the weight of each potential dependence grows with its level of nesting.

In each experiment and before applying the sequence of tests, each potential dependence is analyzed, the subscript expressions are simplified, and any common loop invariant variables are canceled from the subscripts. The potential dependence is passed to the sequence of tests only if the coefficients and constant terms of the subscript function are known at compile-time. Otherwise, an accumulator for the unknown dependences is incremented by the number of feasible directions of the potential dependence. In this way we keep a count of the potential dependences that cannot be analyzed due to lack of compile-time information. We also keep a count of all the potential dependences (one per direction vector) that are conclusively proven dependent by either the constant test or the integer-programming tests.

The unanalyzable subscripts are divided into two categories based on the form of the subscripts. The first category includes as unanalyzable, those subscripts that are linear in the index variables of the loopnest (with constant coefficients) but also may include symbolic additive terms that do not involve the index variables. We will refer to this category as "quasi-linear dependent." All other unanalyzable subscripts are categorized as "unknown dependent."

Each of the two experiments consists of two parts. The first part (shown in the *Original Loop Bounds* column of Tables 8.3 and 8.4) uses the loop limits as they appear in the source program. In this case, the Banerjee rectangular and trapezoidal tests and the simplex-based integer-programming test cannot be applied to all loops since they require that the loop limits be known at compile-time.

For the second part of each experiment (shown in the *Loop Bounds Assumed Constant* column of Tables 8.3 and 8.4) we chose to force all loops to have limits computable at compile-time. This variation is intended to show the maximum effect of unknown loop limits in dependence analysis. Any lower bound of a loop that was not a linear function of the indices in the enclosing loop nest was defined to be 1, and any upper limit that was also not a linear function of the indices in the loop nest was set to the constant 40. The stride or step of the loop was defined to be 1 if it was not an integer constant. For example, in the following loop, the upper limit of the iteration space, `IP(K)`, will be replaced by the constant 40.

```
DO I = 1, IP(K)
   A(I) = A(I) + 1
ENDDO
```

The total number of potential dependences will decrease when we force the loop limits to be compile time constants. As in this example, there exist a few cases in the Perfect Benchmarks where the loop limits are an expression involving array elements. The dependences that are being ignored in the second part of each experiment are those that arise from dependences because of the loop limits.

The choice of 40 as the upper bound is arbitrary and was chosen for historical reasons to maintain consistency with earlier experiments such as [SLY90]. A condition for the chosen upper bound is that it should not significantly change the set of dependences present in the program. Therefore a reasonable criterion is that the upper limit should be larger than the maximal dependence distance present in a loop. We feel that the value 40 satisfies this criteria for the programs being examined. The experiment in Chapter 10 indicates that the most

Benchmark	Lower Bound	Upper Bound	Step
adm(AP)	5.0%	97.7%	0.0%
arc2d(SR)	77.6%	95.1%	0.0%
bdna(NA)	4.2%	62.7%	0.0%
dyfesm(SD)	0.9%	73.6%	0.0%
flo52q(TF)	2.2%	89.2%	1.6%
mdg(LW)	3.8%	66.0%	3.8%
mg3d(SM)	14.2%	100.0%	36.1%
ocean(OC)	3.7%	93.4%	6.6%
qcd2(LG)	7.0%	66.9%	0.0%
spec77(WS)	2.6%	22.1%	0.0%
track(MT)	7.7%	42.9%	0.0%
trfd(TI)	4.1%	64.9%	0.0%
TOTAL	11.9%	71.0%	3.2%

Table 8.1: Percentage of statically unknown loop bounds

common dependence distance is 1. Thus almost any choice for the upper bounds should give comparable results.

The loops in the Perfect Benchmarks range widely in the information available to a restructuring compiler. Table 8.1 shows the distribution of symbolic loops bounds after all the standard optimizations have been completed (constant propagation, induction variable elimination, and dead-code elimination). We observe from this table that the information about the upper and lower bounds of the iteration space as well as the stride vary from program to program in the Perfect Benchmarks. Also, when averaged over the entire collection, it is significant to note that at compile-time the stride is almost always known, the upper bound is almost never known, and the lower bound is known most of the time.

8.2.1 Classification of Unanalyzable Potential Dependences

One way to improve data dependence information is by being able to apply the dependence tests to a larger percentage of the potential dependences. Classifying the reason a potential dependence is unanalyzable by the techniques discussed in this chapter is useful in determining where further effort may prove beneficial.

There are two reasons for a potential dependence to be unanalyzable. One reason is that a subscript is nonlinear. It is rare in these programs to find a subscript that is truly nonlinear. FFT algorithms are the main source of these subscripts. The FFT algorithm uses subscripts that are based on powers of 2.

Unknown information is the second reason that a subscript is unanalyzable. The unknown value may be a coefficient of the loop indices, a additive constant, or an array element used for indirect addressing.

Table 8.2, which records the totals for the types of unanalyzable potential dependences, created by examining all the unknown dependence arcs and partitioning them by the type of the coefficient sets. In each of these sets, the classification precedence was the sequence {Array, Loop Variant Scalar, Loop Invariant Scalar, Constant}. If two or more different classification

	Coefficient Type			
Constant Type	Numeric Constant	Loop Invariant Scalar	Loop Variant Scalar	Array
Numeric Constant	---	1709	103	477
Loop Invariant Scalar	367	3599	141	---
Loop Variant Scalar	3204	2517	105	---
Array	3736	91	---	---

Table 8.2: Classification of unanalyzable subscripts

types were present in the same part of the subscript pair, the one with the higher precedence was chosen.

Each of the groups created by first examining the coefficients was also divided into four categories based on the type of components in the constant term. The same precedence sequence was used in this second partitioning of the unknown dependences. The constant term was defined to be any additive term not containing an index variable from the current loop nest.

Examining Table 8.2, we see that the most common reason a potential dependence is unanalyzable is the presence of an array reference in the subscript. The use of subscripted subscripts is a difficult problem in dependence analysis; it is comparable to using pointers to reference an element. The second most common source are unknown coefficients of the loop indices that are loop invariant. In this case, without knowing the sign or the magnitude of the coefficient, even though it is constant we are unable to use any of the dependence tests. Following closely, the third most common source of unknown coefficients is loop variant additive values. Here all the coefficients of the loop indices are compile-time constants. But a value, possibly an undetected induction variable, is added into the subscript.

All the categories can be reduced by having more information about the subscripts. Interprocedural analysis is one method of collecting more information about the calculations that go into the variables involved in a subscript expression.

The problem of subscripted subscripts is the most challenging but may be resolved through user assertions such as declaring that the subscripting array is an injective map. The second most prevalent category, invariant variables, may be solved by doing more complex analysis to propagate the relationships between invariant variables. Techniques such as improved induction variable recognition may help to reduce the number of variant variables and thus the size of the third category.

8.3 Experimental Results

The results in this chapter can only represent the programs chosen for these experiments. The generality of these results is tied directly to the extent to which these programs are representative of the programs one wants to analyze.

Several important conclusions can be derived from Tables 8.3 and 8.4. The first conclusion is the unexpected effectiveness of the infinity test. Consider the *Original Loop Bounds* column in Table 8.3 where the loop limits are processed as they appear in the source program. In this column we observe that when the infinity test is applied after the rectangular and trapezoidal

tests, it breaks potential dependences in over 6% of the cases, more than the two other tests combined. In Table 8.4, it is shown that when the infinity test is applied before the other two tests, it breaks dependences in over 9% of the cases, and the application of the rectangular and trapezoidal tests contribute nothing extra. This dependence sequence illustrates that none of the dependences requires information about the upper loop bound, and that none of the potential dependences removed by Banerjee's inequalities truly require the use of a trapezoidal dependence test.

A second important conclusion is that after all the traditional tests have been applied, less than 1.0% of the analyzable cases are detected as independent by the integer-programming methods. The integer-programming based methods are the most accurate dependence tests known that can be used when the loop limits are affine functions of the loop indices.

In cases where the loop limits are not known, we can determine an upper bound of what can be achieved by any other conceivable method by looking at the *Loop Bounds Assumed Constant* column. Clearly, all the potential dependences that are broken in the *Original Loop Bounds* column should also be broken in the *Loop Bounds Assumed Constant* column. Therefore, the additional number of dependences broken in the assumed bounds column is an upper bound of what can be achieved by any other new dependence analysis method, which has a subset of the capabilities of the integer-programming method and requires information about the loop limits. The amount is 2471 potential dependences, which is 0.6% of the total potential dependences. We conclude, for this selection of programs, that after the first five tests are applied, any new method could only detect independence in approximately 6453 or 1.5% of the analyzable cases.

The distribution of the potential dependences among the categories of proved dependent, assumed dependent, and proved independent shows no difference when we assume values for the loop limits. The change between the columns of Tables 8.3 show that for almost all dependences proven dependent when we assume loop limits, the same number of dependences were proven dependent by the Omega Test when no assumptions were made. This observation is significant in showing that the symbolic properties of the Omega Test were sufficient to handle unknown loop limits.

At the end of Section 8.2.1 we listed some possible improvements to a parallelizing compiler that might decrease the number of dependences. We can use Tables 8.2 and 8.3 to estimate the upper bound on these improvements. Adding interprocedural analysis would potentially eliminate the loop variant and invariant variables along with the unknown loop bounds for a maximum improvement of 11745 potential dependences or 2.7%. Improved induction variable support might eliminate all loop variant variables for a maximum of 6070 potential dependences or 1.4%. The use of user declarations to eliminate problems resulting from subscripted subscripts would lead to a maximum of 4304 potential dependences or 1.0%.

8.4 Conclusion

The statistical summary of dependence information may not relate to the speed-up that is obtained for a program. A single dependence may be responsible for prohibiting the parallelization of a loop. But we believe that statistical information is useful for determining areas in which to concentrate further research. In Chapter 9 we will extend the evaluation of data dependence analysis into the dynamic domain by considering the effects of the dependence analysis on the parallelism exploited in a program.

Type	*Original Loop Bounds* Count	Percent	*Loop Bounds Assumed Constant* Count	Percent	Difference
Proved Dependent					
Constant Test	116616	(26.7%)	116616	(26.7%)	(0.0%)
Integer Programming	1553	(0.4%)	30697	(7.0%)	(6.6%)
Omega Test	70770	(16.2%)	39684	(9.1%)	(-7.1%)
Total Proved Dependent	188939	(43.3%)	186997	(42.9%)	(-0.4%)
Assumed Dependent					
Unknown Dependent	8796	(2.0%)	8232	(1.9%)	(-0.1%)
Quasi-Linear Dependent	7253	(1.7%)	7283	(1.7%)	(0.0%)
Proved Independent					
Constant Test	150716	(34.5%)	150716	(34.5%)	(0.1%)
Greatest Common Divisor	34645	(7.9%)	34675	(7.9%)	(0.0%)
Banerjee Rectangular	13320	(3.1%)	42778	(9.8%)	(6.7%)
Banerjee Trapezoidal	15	(0.1%)	3406	(0.8%)	(0.7%)
Banerjee Infinity	28567	(6.5%)	15	(0.1%)	(-6.4%)
Integer Programming	92	(0.1%)	1491	(0.3%)	(0.2%)
Omega Test	3982	(0.9%)	727	(0.2%)	(-0.7%)
Total Proved Independent	231337	(53.0%)	233808	(53.6%)	(0.6%)
Total	436325		436320		

Table 8.3: Dependence results for the Perfect Benchmarks

	Original Loop Bounds		*Loop Bounds Assumed Constant*		
Type	Count	Percent	Count	Percent	Difference
Proved Dependent					
Constant Test	116616	(26.7%)	116616	(26.7%)	(0.0%)
Omega Test	72323	(16.6%)	70381	(16.1%)	(-0.5%)
Integer Programming	0	(0.0%)	0	(0.0%)	(0.0%)
Total Proved Dependent	188939	(43.4%)	186997	(42.9%)	(-0.4%)
Assumed Dependent					
Unknown Dependent	8796	(2.0%)	8232	(1.9%)	(-0.1%)
Quasi-Linear Dependent	7253	(1.7%)	7283	(1.7%)	(0.0%)
Proved Independent					
Constant Test	150716	(34.5%)	157016	(34.5%)	(0.0%)
Greatest Common Divisor	34645	(7.9%)	34675	(7.9%)	(0.0%)
Banerjee Infinity	41902	(9.6%)	44901	(10.3%)	(0.7%)
Banerjee Rectangular	0	(0.0%)	0	(0.0%)	(0.0%)
Banerjee Trapezoidal	0	(0.0%)	1234	(0.3%)	(0.3%)
Omega Test	4074	(0.9%)	2282	(0.5%)	(-0.4%)
Integer Programming	0	(0.0%)	0	(0.0%)	(0.0%)
Total Proved Independent	231337	(53.0%)	233808	(53.6%)	(0.6%)
Total	436325		436320		

Table 8.4: Switch the Banerjee Infinity and Banerjee Rectangular Tests, and the Omega Test and Integer Programming Tests.

In [SLY90] a statistical study using a different set of Fortran routines collected by the original Parafrase effort was reported. However, many of the results obtained different conclusions. We have shown the estimation of the importance of coupled subscripts in previous work is not present for nonsymbolic subscripts in the Perfect Benchmarks. It is possible that the coupled subscripts do exist; however, more advanced analysis, such as symbolic dependence tests, must be done to uncover those terms.

It has also been shown that the infinity test seems to be a practical variation that complements the Banerjee rectangular and trapezoidal tests. The main advantage of the infinity test is that it has semi-symbolic properties. These properties allow it to be applied in cases where a full symbolic implementation of the standard Banerjee inequalities might be too expensive.

Finally, the performance of the Omega Test indicates that only a small number of potential dependences require the power of the Omega Test. Most of the applicability of this test is in proving the dependence of potential arcs that would otherwise be assumed dependent.

Chapter 9

DYNAMIC DEPENDENCE EVALUATION

9.1 Introduction

A frequent topic of discussion concerning parallelizing compilers is the effectiveness of data dependence tests. Several recent studies have been published on this topic, all of which count the number of times that a test determines independence as a measure of the success of the test. It is commonly accepted that static evaluation is not sufficient, and additional measurements should be taken to assess accurately the performance of data dependence calculations.

To quote Maydan et al. [MHL91]:

> "In a loop with a thousand independent pairs, being inexact in just one test could have a devastating effect on the amount of parallelism discovered. Ideally, one would like a standard model to measure the parallelism found. Then one could say how much faster a program ran due to exact data dependence. Unfortunately no such system yet exists."

We propose a system whereby it is possible to measure comparatively the differences in effectiveness of a set of dependence tests. The basis for comparison is the amount of implicit parallelism extracted through the application of the dependence tests. Also, we endeavor to define quantitatively or absolutely the effectiveness of the dependence test.

9.2 Overview of the Evaluation Method

The purpose of parallelizing compilers is to transform a program's parallelism, whether implicit or explicit, into a form that is directly executable on a parallel machine. Due to the wide variety of parallel machines available and the varying resource constraints, it is impractical to study the entire parallelizing compiler in depth without limiting our results to a particular parallel machine. Therefore, we have decided that these variations among the different parallel architectures must be factored out to get a better idea of how well the compiler *understands* the program.

On a single machine and using a single program, it is easy to determine which of two compilers is more effective; by creating an executable program with each compiler, running the

programs, and recording the execution times. The compiler that produces the fastest program is obviously the better compiler. However, using a real parallel machine introduces several problems. It is difficult to isolate the effects of the architecture from the understanding and transformation of the program. It may also be that resource constraints mask any differences in parallelism.

9.2.1 The Ideal Parallel Machine

Defining an ideal parallel machine is a natural way to eliminate the variations in performance caused by the differences among existing parallel machines. If we make the assumption that the dependence analysis required of a parallelizing compiler is identical for our ideal parallel machine and for existing parallel machines then by our assumption we are examining the same dependence analysis problem in both cases. Therefore, all the characteristics of the dependence tests are evident when the target is an ideal parallel machine.

Our ideal parallel machine consists of an unlimited number of processing elements. Each processing element has unit time access to a common shared memory. We assume no conflicts among the processors for memory access. For the purposes of this experiment we restrict parallel activity by only allowing statements in different iterations of the same loop nest to execute concurrently. Except for the resource constraints we ignore on memory, memory access, and parallelism, our ideal parallel machine serves as a generic model of available parallel architectures.

The experiments described in this chapter were conducted using the Delta program manipulation system [Pad89] which is described in Chapter 3. Delta is a collection of SETL functions that implement a toolkit of the functions required to perform analysis, instrumentation, and parallelization of Fortran programs. We used the Perfect Benchmarks [Per89] as the source for the Fortran code used in this experiment. As with the experiments in the other chapters of this thesis, all Fortran codes were preprocessed by KAP/Concurrent before being used for this experiment. The experiment determines the number of times each data dependence test is used statically, as well as the impact of each data dependence test on the execution time of the program on the ideal parallel machine.

We measure the effectiveness of our experiments in several ways. First we use the standard method of static analysis. The static results give the first approximation of the effectiveness of the data dependence tests. Examining the number of dependences resolved by each data dependence test shows how often that test was required. The number of potential dependences that were statically unanalyzable indicates the work that is still left to produce a *ideal* dependence test. The second measure of effectiveness is the comparison of the dynamic results for each type of dependence test to the optimal dynamic measurement. The comparison of the dynamic results shows how much parallelism is lost as the result of an imperfect dependence test.

9.3 Data Dependence

The purpose of the experiments described in this chapter is to use the difference between conservative static dependence analysis and run-time reality to evaluate the effectiveness of the former. Given two statements, S_1 and S_2, we say that S_2 is data dependent on S_1 if control can flow from S_1 to S_2, both statements access a common memory location, and at least one

statement writes to the shared location. Chapter 2 provides the background and details of static dependence analysis. For the purposes of this experiment we will be concerned only with *flow*-dependences, which means we examine only the case where S_1 must write a memory location that S_2 reads. The restriction to *flow*-dependences allows us to focus our attention on the transportation of data rather than the effects of memory management, which can be attenuated by compiler transformations such as renaming or expansion.

For scalar variables in Fortran 77, even taking equivalences into account, these is no ambiguity or uncertainty at compile time about whether two variable references refer to the same memory location. For array variables, it can be difficult to determine when two subscripted array references refer to the same element of an array.

To define dependence, consider the loopnest in Figure 9.1.

```
              DO i1 = L1, U1
                 DO i2 = L2(i1), U2(i1)
                    ...
                    DO in = Ln(i1,...,in-1), Un(i1,...,in-1)
S1 :                   X(f1(i1,...,in),...,fm(i1,...,in)) = ...
S1 :                   ...= X(g1(i1,...,in),...,gm(i1,...,in))
                    ENDDO
                 ...
                 ENDDO
              ENDDO
```

Figure 9.1: Loop nest

To determine if a flow dependence exists from statement S_1 to S_2 we need to solve the following problem. Let $\vec{\imath} = (i_1, \ldots, i_n)$ and $\vec{\jmath} = (j_1, \ldots, j_n)$ be two integer n-vectors where $L_k(\ldots) \leq i_k, j_k \leq U_k(\ldots)$ for all k in $[1 \ldots n]$. We refer to the vectors $\vec{\imath}$ and $\vec{\jmath}$, which are composed of the index variables, as elements of the iteration space of the loopnest. A *flow*-dependence exists from S_1 to S_2 if $\vec{\imath}$ is lexicographically less than or equal to $\vec{\jmath}$ and the system of equations derived from the array subscripts, $f_k(\vec{\imath}) = g_k(\vec{\jmath})$ for all k in $[1 \ldots m]$ is satisfied. All the data dependence tests in this section require that, after invariant variable cancellation, both $f_k(\vec{\imath})$ and $g_k(\vec{\jmath})$ be linear functions of the index variables. If we cannot apply the dependence tests to the subscripts, then conservatively we assume dependence.

Suppose we are determining if a dependence exists from statement S_1 with $\vec{\imath} = (i_1, \ldots, i_n)$ to S_2 with $\vec{\jmath} = (j_1, \ldots, j_n)$. The direction vector for this potential data dependence is $\vec{d} = (d_1, \ldots, d_n)$ defined by the equation:

$$d_k = \begin{cases} < & \text{if } i_k < j_k \\ = & \text{if } i_k = j_k \\ > & \text{if } i_k > j_k \end{cases} \tag{9.1}$$

Given a loopnest of depth n containing statements S_1 and S_2, we define the set of potential direction vectors to be all direction vectors where $\vec{i}$ is lexicographically less than $\vec{j}$. Since we

have three choices for each component, the set of potential direction vectors has at most 3^n elements.

The universe of potential data dependence arcs is defined to be the cross-product of all subscript pairs referring to the same array with all potential direction vectors. This way of counting potential dependences is slightly different from the approaches taken by other researchers [GKT91, MHL91], but we feel that it is important to be able to define the universe of potential dependences in such a way that it is possible to partition the universe into non-overlapping subsets. The main difference is the ability to classify a subscript pair to belong in multiple subsets depending on the effects of each direction vector associated with the subscript pair. The importance of the direction vector, $\vec{d}$, will be explained in Section 9.4.4 as a way of determining which loop in the loopnest is affected by a dependence.

Many tests have been proposed to solve the dependence analysis problem. We have chosen four data dependence tests as representative of the spectrum of tests that are currently described in the literature. The two simplest of these tests are the constant test and the generalized GCD test. Neither of these two tests uses the loop bounds or discriminates between direction vectors. We use two complex tests that include the effects of loop bounds and direction vectors: a variation on Banerjee's inequalities described in Chapter 2, that has been extended to handle unknown lower and upper bounds, and the Omega test [Pug91], which represents the integer-programming based dependence tests that are currently the most accurate data dependence tests available.

9.3.1 Examples of Data Dependence from Programs

Before one can understand the evaluation of data dependence techniques, it may be beneficial to examine dependence structures and subscripts that occur in real programs. Studying situations that existing tests can handle gives insight into the boundary between the analyzable and the unanalyzable cases.

The Perfect Benchmarks used in this experiment have provided several examples where the more powerful dependence tests proved useful. The program `dyfesm(SD)` contains several prominent examples of the need for symbolic analysis, analysis of triangular loops, localizable variables, and coupled subscripts. The subroutine `CHOFAC` contains the loop nest shown in Figure 9.2. After the application of the techniques listed above, it should be obvious that loop `35` can be classified as `doall`.

This example is included to illustrate that not all subscripting patterns are really that difficult. The existing techniques used by parallelizing compilers can detect independence for the loop in Figure 9.2.

Another example occurs in the program `dyfesm(SD)` from the Perfect Benchmarks suite. In subroutine `CORRECT`, the loop nest of Figure 9.3 is a classic example showing the need for symbolic relations in the data dependence tests. As is apparent from this loop, constant propagation and symbolic dependence analysis are required to establish the relationship between the subscripts. These techniques are not new, but we wish to emphasize the importance of including them in compilers.

The array `T` is assigned in four statements and used in four other statements. After forward propagation, we have subscripts of the form:

```
      DO 40 I=2,N
         SUM=0.0
         DO 20 L=1,I-1
            SUM=SUM+A(L,I)*A(L,I)*A(L,L)
20       CONTINUE
         A(I,I)=A(I,I)-SUM
         IF (A(I,I).EQ.0.)  RETURN
         DI=1./A(I,I)
         DO 35 J2=I+1,N
            SUM2=0.0
            DO 30 L1=1,I-1
               SUM2=SUM2+A(L1,I)*A(L1,J2)*A(L1,L1)
30          CONTINUE
            A(I,J2)=DI*(A(I,J2)-SUM2)
35       CONTINUE
40    CONTINUE
```

Figure 9.2: Example loop from the routine `CHOFAC` in `dyfesm(SD)`

- `IQ1=IQN+I`
- `IQ2=IQN+I+NSP`
- `IQ3=IQN+I+2*NSP`
- `IQ4=IQN+I+3*NSP`

Canceling the loop invariant term (`IQN`) gives subscripts of the form: `IQ`k`=I+`$(k-1)$`*NSP` for k in $[1 \ldots 4]$. Since $1 \leq$ `I` $\leq$ `NSP`, it is impossible for any instance of the subscripts from different iterations to overlap.

9.4 Critical Path Parallelism

Any program can be viewed as a directed graph where the nodes represent operations and the arcs represent execution constraints required to execute the program correctly. Typical arcs are control-flow arcs, which describe conditional execution, and data-flow arcs, which reflect the requirement that no operation can proceed until all operands are available. By convention, the digraph is augmented with a **start** node joining all entry points of the program, and a **stop** node joining all exit points.

The critical path computation, detailed in Chapter 4, dynamically determines the length of the longest path joining the **start** and **stop** nodes during a program's execution. The ratio of the number of operations executed to the length of the critical path provides a notion of average parallelism or speedup with respect to sequential execution.

The experiments presented in this chapter consider only the effects of *flow*-dependences on the parallelism in a program. We consider *output*- and *anti*- dependences to be primarily an

```
      DO 350 I=1,NSP
         IQ1=IQN+I
         Q1=T(IQ1)
         IQ2=IQ1+NSP
         Q2=T(IQ2)
         IQ3=IQ2+NSP
         Q3=T(IQ3)
         IQ4=IQ3+NSP
         Q4=T(IQ4)
         QNORM=SQRT(Q1*Q1+Q2*Q2+Q3*Q3+Q4*Q4)
         IF(QNORM.NE.0.D0) QNORM=1.D0/QNORM
         T(IQ1)=QNORM*Q1
         T(IQ2)=QNORM*Q2
         T(IQ3)=QNORM*Q3
         T(IQ4)=QNORM*Q4
350   CONTINUE
```

Figure 9.3: Loopnest extracted from `CORRECT`

unintended consequence when a sequential language specifies a potentially parallel computation. The effect of ignoring *anti-* and *output-* dependences is similar to the effect obtained by localizing variables, renaming variables, or performing either scalar or array expansion.

The exceptions are the cases where the only activity of an algorithm is to compute an *output*-dependence. It is possible to construct such examples, but we believe they have little practical value. For example: Initialize an array by overwriting previous values as shown in Figure 9.4.

```
DO I = 1, N
   DO J = 1, N
      A(I*J) = I
   ENDDO
ENDDO
```

Figure 9.4: Loop that computes an output dependence

This example illustrates that it is possible to create a computation that has no explicit *flow*-dependences. A question inspired by this example is: Does there exist an algorithm whose critical path is constrained by *anti-* or *output-* dependences that does not have a corresponding algorithm where the critical path is constrained by *flow*-dependences? For the example presented here, we would require that the state of the array `A` after the loopnest be identical for the original and the new algorithm.

9.4.1 Types of Parallelism

The basic definition of critical path parallelism assumes a granularity corresponding to the smallest atomic operation [Kum88, Che89]. As we increase the size of the granularity of our computations, we decrease the available parallelism. Increasing the granularity to the statement level simulates a form of macro dataflow computation where the operations are n-valued expressions instead of primitive operators, as in the traditional dataflow model.

Existing general purpose parallel machines do not provide the low-cost synchronization hardware needed to execute either the dataflow or macro dataflow paradigms. The parallelism granularity needs to be increased to match the efficiency of the available synchronization facilities.

A logical reason to choose loop-level granularity is that sequential languages use looping constructs as a multiplicative factor in computational complexity whereas, textual replication (i.e., macro-dataflow) is additive to the complexity. Thus one would expect the major computations contained in a program to be concentrated inside a looping construct. This observation is well known and is reflected by available parallelizing compilers that concentrate on loop transformation and extracting loop-level parallelism.

Loop-level parallelism can be modeled in the framework of critical path parallelism. The change required is to place additional constraints into the dataflow graph. In Chapter 4, we introduced the analog of a *program counter* (PC) into the dataflow graph which allows code to be selectively serialized. Traditionally the PC has enforced sequential execution by introducing a *flow*-dependence (PC = PC + 1) between operations. Introducing *artificial* dependence arcs that connect adjacent statements into the dataflow graph is a step toward enforcement of the loop-level parallelism paradigm.

The two predominant forms of loop level parallelism are the `doall` and `doacross` loops. The introduction of *artificial* dependence arcs, simulating the program counter, and the removal of backward control dependence arcs in loops, allow simulation of both the `doall` and `doacross` parallelism forms by preserving any implicit data dependences that may exist between loop iterations.

9.4.2 Dynamic Dependence Evaluation

The program instrumentation described in this chapter is done intraprocedurally by an extension of the method in Chapter 4. The execution of the instrumented code propagates the data and control dependence information across routine boundaries at call sites to compute the interprocedural results. Thus we are able to calculate the effects of interprocedural analysis without actually doing the static analysis and discovering methods of implementing the interprocedural parallelism.

In the ideal parallel machine used to evaluate dependence tests dynamically, we only enforce *flow*-dependence edges present in the original program. The rationale for this choice is that *output*- and *anti*- dependence relate to artificial storage dependences and can be removed through program restructuring. Including the effects of *output*- and *anti*- dependences artificially reduces the available parallelism owing to the semantics of Fortran.

We will define a program transformation $\mathcal{T}$ such that when $\mathcal{T}$ is applied to a program p we obtain a program p' (e.g., $p' = \mathcal{T}(p)$). The program p' has the property that executing program p' produces the same results as program p, and additionally produces the length of the critical path (T_∞) through the *flow*-dependence graph of program p and the number of operations (T_1)

executed by program p. The *flow*-dependence graph is computed at run-time and is therefore completely accurate for the given input data.

The *flow*-dependence graph is calculated by following the data as they pass from one statement to another. We associate a shadow variable with each original program variable; the shadow variable holds the availability time of the value contained in the program variable. For each statement in the program the availability times of the output variables are a function of the availability times for the input variables. Usually the function is the maximum of the times for the input variables plus a constant to account for the time to execute the statement.

If we assume an unlimited number of processing elements with no memory conflicts and ignore memory-related dependences (our ideal machine), the time to execute the program in parallel (T_∞) is equal to the length of the critical path. The number of operations executed is equal to the sequential execution time (T_1) assuming a sequential machine would take one time step to execute each operation. The ratio ($S_\infty = T_1/T_\infty$) gives the speedup from program p executing on the ideal parallel machine.

9.4.3 Instrumentation for Dependence Evaluation

The method chosen for this experiment is to do a source-to-source transformation of the input program. This transformation introduces calls to an instrumentation library which does the recordkeeping necessary to determine if a statically indeterminant dependence is degrading the performance of the program.

If we have the subroutine in Figure 9.5 to instrument, we need to calculate when statements can execute based on the availability of the operands (*flow*-dependences) and the control dependences. We also need to accumulate the number of operations being executed to estimate the sequential execution time.

Line#		
1	S_1 :	`PROGRAM TEST`
2		`REAL A`
3		`INTEGER I, N, C`
4		`PARAMETER (N=100)`
5		`DIMENSION A(N)`
6		`DATA C /1/`
7	S_2 :	`DO I = 1, N, 1`
8	S_3 :	`   A(C*I) = A(C*I)+1`
9	S_4 :	`ENDDO`
10	S_5 :	`END`

Figure 9.5: Sample program with unanalyzed subscript

The output from the instrumented code is listed in Figure 9.6. The important line to notice is the line that is tagged with the symbol `<D>`. This line indicates that at line number 8 in the file which contains the routine `TEST`, the difference between the static and the dynamic dependence analysis is significant.

```
   ---- SUMMARY: Critical Path Analysis Speedup ----

TEST                  (     1.0     1.0)      400      400      400
TEST:7           1.0x (     1.0     1.0)      400      400      400
    <D>  TEST:8          19800
```

Figure 9.6: Output from sample program with unanalyzed subscript

In the instrumented code (shown in Figure 9.7) we have the common block `N$NBLOCK` containing the variable `N$N` to communicate the accumulated operation count between calls to the subroutines. The control dependences and the serializing assumptions of loop-level parallelism are simulated using variables of the form `S$n`. Finally, the *flow*-dependences are calculated using shadow variables of the form `C$X` to hold the availability time of the value in variable `X`.

In summary, we have described a method where one is able to simulate the parallel execution of a sequential program on our ideal parallel machine. Our ideal machine exploits the following features.

- Scalar expansion, renaming, and array expansion are simulated by the removal of *output-* and *anti-* dependences.
- The `doall` and `doacross` parallel scheduling methods are used across procedure boundaries (unless instructed otherwise).
- Perfect data dependence calculation (effectively comparing the machine addresses at run-time to determine dependence).

As with any heuristic/experimental approach, a few limitations must be considered. Our method does not consider the effects of statement reordering. If a dependence cycle exists that covers the entire loop body, the loop will appear to be serialized even though statement reordering would be able to minimize the dependence cycle and expose additional parallelism.

Certain programming paradigms negatively influence the recognition of inherent parallelism. The recurrences introduced, by inductively calculating a sequence, serialize otherwise parallel code. Similarly, naive reductions such as maximization or summation, artificially limit the parallelism that might be extracted from an application. A sufficiently powerful restructuring compiler may be able to remove induction variables and replace reduction calculations with more efficient algorithms. Our current approach to these limitations is to use an existing parallelizing compiler to statically analyze the source code and remove as many of these problems as possible. This approach does not totally eliminate the effects of improper algorithm choice, but does help to minimize the effect. We are planning to investigate the effects of inductions and reductions with techniques similar to those used to compute the critical path length.

9.4.4 Constrained Execution

In addition to the instrumentation required for the critical path parallelism calculations, the program is instrumented by the addition of statements that enforce the set of potential *flow*-dependence arcs that the compiler was unable to prove independent during static analysis. The

```
S1 :  PROGRAM TEST
      REAL A
      INTEGER C, N, I
      INTEGER S$S1, S$S2, S$S3, S$S4, T$0, N$N, N$Z
      INTEGER C$A, C$C, C$I, C$TEST, TA1$S3, TB1$S3
      PARAMETER (N=100)
      DIMENSION A(N), C$A(N)
      COMMON /N$NBLOCK/ N$N, N$Z
      DATA C /1/
      CALL MODULE$ENTRY('TEST',N$N,C$TEST)
      S$S1 = C$TEST
      S$S2 = C$TEST
      C$C = C$TEST
      TB1$S3 = N$Z
      C$I = S$S1
      S$S4 = C$I
      CALL CP$ENTRY('TEST:7',N$N,C$I)
S2 :  DO I = 1, N, 1
         S$S2 = C$I
         TA1$S3 = TB1$S3
S3 :     A(C*I) = A(C*I)+1
         N$N = N$N+4
         T$0 = MAX(S$S2,C$A(I*C),C$C,C$I)
         CALL CP$DEPEND('TEST:8',MAX(TA1$S3-T$0,N$Z))
         C$A(I*C) = MAX(TA1$S3,T$0)+4
         C$TEST = MAX(C$TEST,C$A(C*I))
         S$S3 = C$A(C*I)
         TB1$S3 = MAX(TB1$S3,S$S3)
         S$S4 = MAX(S$S2,S$S3,S$S4)
S4 :  ENDDO
      C$I = MAX(C$I,S$S4)
      CALL CP$EXIT('TEST:7',N$N,C$I)
      S$S2 = MAX(S$S2,S$S4)
      C$TEST = MAX(S$S2,C$TEST,C$I,C$C)
      CALL MODULE$EXIT('TEST',N$N,C$TEST)
S5 :  END
```

Figure 9.7: Instrumented sample program with unanalyzed subscript

additional statements introduce *artificial* constraints into the program's dataflow graph. The effect of these *artificial* arcs is to serialize the loop with respect to the statements involved in the dependence.

As an example, consider a simpler yet incorrect method. We start with the simple loop, in Figure 9.8, with arbitrary functions f_1 and f_2.

```
       DO i1 = L1,U1
S1 :       X(f1(i1)) = X(f2(i1))+1
       ENDDO
```

Figure 9.8: Initial loop

We want to introduce a variable (X') and a function (g_1) into the program such that the combined static *flow*-dependence arcs for statements S_1 and S_1' are identical. The difference occurs during the execution of the program. Statement S_1' forces the loop to be serialized, even though it may not have been dynamically serialized because of statement S_1. In Figure 9.9, we are assuming statements S_1 and S_1' are executed as one combined statement. With this assumption, the total execution time for the loop is the same as the conservative execution time for the previous loop. The difference is that more resources are needed to deal with the shadow computations.

```
       DO i1 = L1,U1
S1 :       X(f1(i1)) = X(f2(i1))
S1' :      X' = g1(X')+1
       ENDDO
```

Figure 9.9: First instrumented loop

As should be obvious to the reader, this simple technique using scalar variables does not generalize to loopnests of depths greater than one.

An interesting problem can be posed. Given a program, p, and its dependence graph, $\mathcal{D}(p)$, create a new program p' on the same statement and control-flow structure, such that the combined *flow*-dependence graph $\mathcal{D}(p) \cup \mathcal{D}(p')$ is equal to $\mathcal{D}(p)$. The program p' has the additional property that all *flow*-dependence arcs in $\mathcal{D}(p')$ are provably dependent during static analysis and therefore are guaranteed to be enforced during the execution of the program.

We did not find a practical solution to this problem, so we found a suitable heuristic. Given a direction vector $\vec{d}$ for a dependence arc, the level l of the dependence is the first non-equal component of $\vec{d}$. We reasoned that the effect of a dependence at level l in a loopnest is to serialize the source of the dependence arc with respect to the sink across level l of the loopnest. The dependence indicates a potential ordering conflict when two iterations from level l of the loopnest are concurrently executed. By serializing the loop at this level, we eliminate this dependence as a potential problem. We recognized that an arbitrary loop in a loopnest can

be serialized by breaking the artificial dependence arc into two components and forcing the execution of the loopnest to connect the components at the proper time.

The loopnest in Figure 9.10 will be used to illustrate how an arbitrary loop at level k in the loopnest may be serialized.

```
S1 :   DO i1 = L1, U1
          ...
S2 :      DO in = Ln, Un
S3 :         X(f1(i1, ..., in)) = X(f2(i1, ..., in))+1
S4 :      ENDDO
          ...
S5 :   ENDDO
```

Figure 9.10: Second loop nest

In this example let $k = 1$ be the level of the loop in the loopnest to serialize. First create two new shadow variables $\$XA_k$ (source) and $\$XB_k$ (sink). Before the DO statement at level k, clear the source of the artificial arc. After the DO statement, copy from the source component of the artificial arc to the sink component. When the sink is referenced, add the shadow variable (sink) to the dependences list. When the source is computed, assign the source of the artificial dependence arc to be equal to the shadow variable of the associated program variable. Figure 9.11 shows the final form of the instrumented code that allows serialization at any level.

```
       $XA1 = 0
S1 :   DO i1 = L1, U1
          $XB1 = $XA1
          ...
S2 :      DO in = Ln, Un
S3 :         X(f1(i1, ..., in)) = X(f2(i1, ..., in))+1
             $X(f1(i1, ..., in)) = MAX($XB1, ...)
             $XA1 = $X(f(i1, ..., in))
S4 :      ENDDO
          ...
S5 :   ENDDO
```

Figure 9.11: Instrumented version of the second loop

The important features of this instrumentation method are:

- The sink of the artificial *flow*-dependence arc is updated only when the iteration boundary of the loop being serialized is crossed.
- The source of the artificial *flow*-dependence arc is assigned whenever the source of the dependence is executed.

- On entry to the loop being serialized, the artificial *flow*-dependence arc is cleared to eliminate any serializing effects from previous invocations of the loop.

9.4.5 Serialization Caused by Variable Dependence Distance

It should be noted that the implicit parallelism recognized by the critical path analysis may come from unusual sources. The results from the program `dyfesm(SD)` pointed to the interesting loop in Figure 9.12.

```
      DO 53 I=2,N
         SUM = 0.0
         DO 51 L=1,I-1
            SUM = SUM + A(L,I)*B(L)
51       CONTINUE
         B(I) = B(I) - SUM
53    CONTINUE
```

Figure 9.12: Example loop from `dyfesm(SD)`

In this loopnest from the subroutine `CHOSOL`, the *flow*-dependence is statically determined correctly. The dependence in question is a *flow*-dependence from `B(I)` in loop `53` to `B(L)` in the next occurrence of loop `51`.

The results indicated that this loop was being artificially slowed down by the conservative assumption of a dependence. Here however, the conservative assumption is not the existence of the dependence arc, it is the dependence distance. Our instrumentation method for serializing loops assumes that all dependence arcs are unit length. Here we have a dependence distance that is variable and increasing on each iteration. By pipelining the computation of successive calculations of the inner dot product, we are able to achieve at least partial parallelism.

9.5 Static Results

Practically all dependence analysis evaluations have been static, with potential dependences classified as dependent, independent, or statically indeterminant [GKT91, MHL91]. Chapter 8 also describes such an experiment. The criterion for the success of a dependence test has been the number of potential dependences it has been able to identify as dependent or independent.

As we discussed in Section 9.4, we are interested only in the *flow*-dependences for this experiment. Our own counts, presented in Table 9.1, consider each potential direction vector of the *flow*-dependences separately. Even though other researchers count dependences in a different way, we believe our way of counting is more likely to reflect the numbers from the dynamic analysis than just counting each potential dependence, since we are effectively weighting each dependence arc by the loopnest depth.

The information presented in Tables 9.1 and 9.2 use the following column identifiers.

`CST`:	Constant Test.
`GCD`:	Generalized GCD Test.
`BAN`:	Banerjee's Inequalities.
`OMT`:	Omega Test.
`UNK`:	Dependence was not statically determined.

Table 9.1 is divided into three sections. The first section lists the number of potential *flow*-dependence arcs that were proved independent and subdivides the total by the test that proved independence. The second section shows the number of potential *flow*-dependences that were either proved dependent by the Constant or Omega test, or were assumed dependent because they could not be analyzed (`UNK` column). The last section shows the total number of potential *flow*-dependences in each program. Table 9.2 presents the information in Table 9.1 in percentage form.

Static dependence counts are only able to show the relative merits among a class of similar dependence tests. Also, such counts are not directly related to the effectiveness of the dependence tests in enabling the compiler to express the parallelism inherent in the program.

Many static dependence experiments reported in the literature present the number of potential dependences that they have been able to remove. The experiments have failed to report the number of dependences they are unable to handle owing either to subscript complexity or unknown information. We report the count (`UNK`) for subscripts that our tool was unable to statically analyze. But even this additional information is not sufficient to understand the program fully. As an example of the effect of imperfect data dependence analysis that will be detected at run-time but not reflected by our static measurements, consider the effect of run-time constraints on the dependence analysis as shown in Figure 9.13.

```
DO I = 1, N
   IF (I .NE. 5) THEN
      A(I) = A(5) + 1
   ENDIF
ENDDO
```

Figure 9.13: Example loop that requires additional information

If the dependence test does not consider the additional constraints to the problem, it will incorrectly (conservatively) determine dependence. This inaccuracy may have a large impact on the parallelism exploited by the program, but will not be reflected by our static measurements. It is possible to statically handle these cases in a wide variety of circumstances. But what about more complex situations? How do you determine when such advanced static analysis is worthwhile? The solution is to do a dynamic evaluation of the static data dependence tests to determine how much parallelism is lost by the conservative data dependence tests.

An important limitation should be noted for the static dependence results presented in this section. Our simulation results reflect only the *flow*-dependence arcs from the original program for the given input dataset. In addition to this limitation, we have not enabled the induction

	Independent					Dependent				All
PROGRAM	CST	GCD	BAN	OMT	Total	CST	OMT	UNK	Total	Total
adm(AP)	0	82	490	6	578	2	373	60	435	1013
arc2d(SR)	1526	5069	1204	57	7856	153	726	18	897	8753
bdna(NA)	0	0	241	12	253	0	276	150	426	679
dyfesm(SD)	296	26	698	26	1046	89	584	60	733	1779
flo52q(TF)	114	958	1009	54	2135	0	199	11	210	2345
mdg(LW)	60	0	866	4	930	92	1867	726	2685	3615
mg3d(SM)	2	165	4409	32	4608	1827	61	3016	4904	9512
ocean(OC)	0	0	473	191	664	46	284	557	887	1551
qcd2(LG)	53536	294	2022	0	55852	37200	13125	0	50325	106117
spec77(WS)	35	0	1565	30	1630	16	1168	263	1447	3077
track(MT)	238	3	133	0	374	56	184	12	252	626
trfd(TI)	0	0	1011	627	1638	8	689	74	771	2409
Total	55807	6597	14121	1039	77564	39489	19536	4947	63972	141536

Table 9.1: Breakdown of dependence results for each program

	Independent (%)					Dependent (%)			
PROGRAM	CST	GCD	BAN	OMT	Total	CST	OMT	UNK	Total
adm(AP)	0.0	8.1	48.4	0.6	57.1	0.2	36.8	5.9	42.9
arc2d(SR)	17.4	57.9	13.8	0.7	89.8	1.7	8.3	0.2	10.2
bdna(NA)	0.0	0.0	35.5	1.8	37.3	0.0	40.6	22.1	62.7
dyfesm(SD)	16.6	1.5	39.2	1.5	58.8	5.0	32.8	3.4	41.2
flo52q(TF)	4.9	40.9	43.0	2.3	91.0	0.0	8.5	0.5	9.0
mdg(LW)	1.7	0.0	24.0	0.1	25.7	2.5	51.6	20.1	74.3
mg3d(SM)	0.0	1.7	46.4	0.3	48.4	19.2	0.6	31.7	51.6
ocean(OC)	0.0	0.0	30.5	12.3	42.8	3.0	18.3	35.9	57.2
qcd2(LG)	50.4	0.3	1.9	0.0	52.6	35.1	12.4	0.0	47.4
spec77(WS)	1.1	0.0	50.9	1.0	53.0	0.5	38.0	8.5	47.0
track(MT)	38.0	0.5	21.2	0.0	59.7	8.9	29.4	1.9	40.3
trfd(TI)	0.0	0.0	42.0	26.0	68.0	0.3	28.6	3.1	32.0
Total	39.4	4.7	10.0	0.7	54.8	27.9	13.8	3.5	45.2

Table 9.2: Breakdown of dependence result percentages for each program

variable recognition phases of our analysis (during our dependence analysis). On average, only 3.5% of the potential *flow*-dependences are unanalyzable. Thus, even though induction variable recognition is important to understanding a program, it would have a minor impact on these static results.

9.6 Effectiveness of Data Dependence Tests

One must ask the question: Are the existing dependence tests sufficient, and if they are not, where must further effort be expended? A number of papers such as [MHL91, GKT91, Pug91] show that each of the many different alternative tests are capable of solving a number of data dependence problems. For evaluation purposes we want to use a *ideal* dependence test. This *ideal* test should be able to determine dependence or independence for all possible subscripts. It must be able to say exactly on each reference to an array, what set of definitions could have contributed to the value placed in this location.

Unfortunately this *ideal* dependence test does not exist. The closest we have been able to come is through dynamic analysis of the program, as discussed in the previous section. Dynamic analysis has its own limitations since it is only able to record what occurred during that particular execution of the program with the dataset used, even though it is possible to increase the coverage by supplying datasets that exercise the range of possibilities that the program may take.

Tables 9.3 and 9.4 list the results of the dynamic analysis. The first two columns in each table are obtained from Chapter 4. The first column is the optimal parallelism present in each program. Optimal parallelism is defined in terms of the critical path length when assuming loop-level parallelism, as discussed in Section 9.4. The second column is the measured performance of the KAP/Concurrent restructuring compiler on our ideal machine. The performance of KAP is expected to be slightly lower than any of the other comparable measurements since it does both static dependence analysis and static mapping of the parallelism.

The last three columns in Table 9.3 list the average interprocedural parallelism, which is computed in the same way as the optimal parallelism except that certain restrictions are applied. The restrictions are based on the potential *flow*-dependences that are not proven independent by the data dependence tests listed in the heading for the column. Table 9.4 reports the effects of intraprocedural parallelism for each of the three types of dependence tests.

9.7 Observations from the Perfect Benchmarks

The purpose of running an experiment is to learn something new about the subject being studied. The major observations of running our analyses on the Perfect Benchmarks are presented next.

The average parallelism, as measured against the Omega test and Banerjee's inequalities for Tables 9.3 and 9.4, show no meaningful difference. The conclusion is that in this experiment, the additional capabilities of the Omega test were not beneficial in extracting additional parallelism from the programs.

The static results from Table 9.2 help to illustrate the reason for this conclusion. In the absence of the Omega test, any potential *flow*-dependences that were not broken by a less complex test would be counted as statically unanalyzable (`UNK`) and assumed to be dependent.

PROGRAM	Optimal Loop-Level	KAP-Obtained Loop-Level	Omega Test CST & GCD	Banerjee CST & GCD	CST & GCD
adm(AP)	45.5	3.0	8.38	8.38	2.77
arc2d(SR)	336.0	66.3	330.81	330.81	3.10
bdna(NA)	139.5	1.3	73.36	72.55	2.38
dyfesm(SD)	17.9	3.9	10.51	10.49	3.61
flo52q(TF)	206.9	76.7	206.36	206.36	2.66
mdg(LW)	5.3	1.4	5.22	5.22	5.12
mg3d(SM)	1.3	1.2	1.25	1.25	1.14
ocean(OC)	272.4	1.3	2.40	2.40	2.03
qcd2(LG)	2.4	1.2	2.27	2.27	2.15
spec77(WS)	13.8	1.1	13.52	13.52	13.20
track(MT)	38.7	1.1	32.90	32.90	1.74
trfd(TI)	87.9	10.3	58.02	58.02	2.89

Table 9.3: Average interprocedural parallelism: constrained by static dependence analysis

PROGRAM	Optimal Loop-Level	KAP-Obtained Loop-Level	Omega Test CST & GCD	Banerjee CST & GCD	CST & GCD
adm(AP)	45.5	3.0	3.00	3.00	1.89
arc2d(SR)	336.0	66.3	252.52	252.52	2.18
bdna(NA)	139.5	1.3	73.35	72.55	2.38
dyfesm(SD)	17.9	3.9	5.31	5.30	1.13
flo52q(TF)	206.9	76.7	206.16	206.15	2.66
mdg(LW)	5.3	1.4	4.61	4.61	3.50
mg3d(SM)	1.3	1.2	1.18	1.18	1.08
ocean(OC)	272.4	1.3	2.40	2.40	2.03
qcd2(LG)	2.4	1.2	1.49	1.49	1.42
spec77(WS)	13.8	1.1	2.25	2.25	2.20
track(MT)	38.7	1.1	1.73	1.73	1.59
trfd(TI)	87.9	10.3	26.60	26.60	1.52

Table 9.4: Average intraprocedural parallelism: constrained by static dependence analysis

We can see from Table 9.2 that the effect of the Omega test was to move only 0.7% of the potential dependences from the unknown column to the independent column, but to move 13.8% from the unknown column to the dependent column. A move from unknown to dependent accomplished nothing in this experiment since all unknown potential *flow*-dependence arcs are assumed dependent. In this experiment, none of the dependences in the 0.7% that moved to the independent column had an important effect on the average parallelism.

One dependence cycle is sufficient to serialize an important loopnest. Thus, as long as the number of dependences in the unknown column of Table 9.1 is non-zero, the potential exists for improving parallelism through better dependence analysis.

9.7.1 Evaluation and Classification

If the count of potential dependence arcs were sufficient to evaluate the effectiveness of data dependence tests, we would expect the results of the static analysis in Tables 9.1 and 9.2 to predict the results in Tables 9.3 or 9.4. However, this is not the case.

An examination of the count of unknown potential dependences shows where the compiler must make conservative assumptions to ensure the correct execution of a program. The programs `mg3d(SM)` and `mdg(LW)` have high percentages of unknown dependences. But in Table 9.3, we see a small loss of parallelism when we compare the optimal parallelism to the parallelism exposed by either the Omega test or Banerjee's inequalities.

The program `qcd2(LG)` has no unknown *flow*-dependences, but we see from Table 9.3 that it too has a small loss of parallelism when the optimal parallelism is compared with the parallelism extracted by static analysis. Here are two different static results but the same dynamic behavior.

The simplest data dependence tests do not predict the dynamic behavior either. In fact, the program `spec77(WS)` has the largest average parallelism when restricted by only the results of the constant and GCD tests. However, from Table 9.2, only 1.1% of the potential dependence arcs were removed by these tests. For the same program, we see that Banerjee's inequalities removed 50.9% of all dependences but only improved the average parallelism in Table 9.3 from 13.20 to 13.52.

9.7.2 Intraprocedural and Interprocedural Parallelism

We have shown that in some cases interprocedural parallelism is required for good performance and in some cases it is not. What is important to note is not that these particular programs have a certain behavior, but that the method used to determine whether a program may benefit from interprocedural parallelism is presented without actually needing to do the interprocedural analysis.

The difference between Tables 9.3 and 9.4 shows the effects of interprocedural parallelism on program execution. Normally the results calculated by the critical path analysis assume that all subroutines are effectively inlined. Table 9.3 gives the results for this assumption.

However, if instead of inlining the subroutine at each call site we add *artificial* dependence arcs connecting all `CALL` statements, we require that no two subroutine calls execute concurrently. Table 9.4 gives the results for this second assumption. This table shows what effect intraprocedural parallelism alone has on the average parallelism present in a program. In both tables we compute the dependence analysis intraprocedurally, but the interprocedural parallelism that is reported in Table 9.3 is removed in Table 9.4.

The programs can be divided into three categories, based on the influence of interprocedural parallelism. The heavily influenced programs are defined to have a factor of approximately two or more loss of parallelism. The programs in this category are `adm(AP)`, `dyfesm(SD)`, `spec77(WS)`, `track(MT)`, and `trfd(TI)`. We can also define a category based on the programs that have no significant parallelism degradation. The programs in this second category are `bdna(NA)`, `flo52q(TF)`, `mdg(LW)`, `md3g(SM)`, and `ocean(OC)`. The third category contains the rest of the programs. These remaining programs have a slight but not significant degradation from the lack of interprocedural parallelism. The programs in the last category are `arc2d(SR)` and `qcd2(LG)`.

9.7.3 Correlation with Interprocedural Parallelism

Sometimes the correlation between the static and dynamic evaluations is as expected. The three best programs for average KAP parallelism are: `flo52q(TF)`, `arc2d(SR)`, and `trfd(TI)`. These three programs also have the largest percentages of independent arcs; all three programs have greater than two-thirds of the potential *flow*-dependence arcs proven independent as shown in Table 9.1.

A comparison of Tables 9.3 and 9.4 for the same three programs shows that they all have a minimal loss of parallelism when serializing subroutine calls and eliminating interprocedural parallelism. As one would expect, we can conclude that using a parallelizing compiler that does not do interprocedural analysis only works effectively when the programs are easy to analyze and do not require interprocedural parallelism.

Sometimes the correlation is not as encouraging. Several of the programs, `track(MT)`, `spec77(WS)`, `dyfesm(SD)`, and `adm(AP)`, show a large loss of average parallelism when Tables 9.3 and 9.4 are compared. These four programs also correspond to instances where KAP was unable to exploit automatically any meaningful parallelism.

9.7.4 Future Enhancements

One obvious enhancement is to increase the descriptive power of the ideal parallel machine to be able to model existing parallel machines more closely. This would extend the applicability of this method to cover not only the semantic analysis portion of a parallelizing compiler but also the ability to predict the behavior of the mapping portion.

Another extension would be to include the effects of *anti-* and *output-* dependences on the critical path. This extension has the potential to fully serialize the program, but also would be able to pinpoint the exact location of the storage conflicts that need to be eliminated.

Inductions and reductions need to be considered. It is possible to recognize these constructs using the same type of critical path analysis. The removal of an induction variable may cause another dependence to be discovered. Thus, removing the induction variables will increase the effectiveness of the static dependence tests, potentially increase the available baseline parallelism, and, last but not least, may allow this instrumentation to discover the true nature of the computation, independent of the means (such as an inductive sequence) used during the calculation.

The current instrumentation assumes that every potential dependence arc that is not proven independent is of distance one. It may be possible to alter the instrumentation to simulate the effects of larger distances if they are known at compile-time.

Since we serialize the statements in each iteration of a parallel loop, the order in which these statements are executed impacts the performance of the loop. Statement reordering is one very important technique for minimizing the size of the dependence cycles contained in a loop. It may be possible to use the artificial dependence arcs that are added to the program to determine where the impact of statement reordering would be most beneficial. Another technique would be to precondition the source code to minimize the size of the dependence cycles in all loops, regardless of whether they are marked as serial or parallel.

Finally, other methods for serializing loop nests may be more effective. The imposition of a two-part serialization arc is purely for the convenience of this study.

9.8 Conclusion

The data presented in this chapter describe a method where existing dependence analysis techniques can be quantitatively measured. This method is similar to performing the actual parallel compilation and execution on a parallel machine. However, our method attempts to isolate the important features of a parallel architecture without being constrained by artificial limitations such as cache/memory bandwidth or a limited number of processors.

We have shown that it is sometimes possible to relate the static dependence counts to the speedup of an application on an ideal parallel machine. We have also shown that it is an imprecise measure of the ability of the dependence analysis portion of the parallelizing compiler to measure only static dependence counts.

The major result of this experiment shows that dynamically, the Omega test does not improve the average parallelism over the parallelism exposed by Banerjee's inequalities. This conclusion is contradictory to the accepted wisdom that more powerful data dependence tests would help to extract more parallelism.

Another benefit of this evaluation technique is the capability to locate exactly the places in the source code that are causing difficulty for the compiler. These locations might be beneficial as targets for hand transformations or as test cases to target further development of the compiler.

Chapter 10

DYNAMIC DEPENDENCE ANALYSIS

10.1 Introduction

Data dependence analysis is used by parallelizing compilers to understand the structure of the computation in a program. Static data dependence analysis tests are used to determine the dependences that might be present in the source code. However, most static dependence analysis techniques are limited to computing intraprocedural information and are also limited to considering simple subscript expressions, usually affine functions with constant coefficients.

Expanding the scope of the dependence analysis to consider interprocedural data dependence arcs makes it possible to determine if loops that call other routines can be parallelized. The technique presented in this chapter works interprocedurally and considers all subscripts regardless of their complexity. As with any dynamic technique, the results are dependent on the input dataset, which means that we will only discover dependences that are executed by one run of the program. More dependences may potentially be present in the source code, but cannot be discovered by this technique.

Parallelizing compilers require information about independent computations in order to schedule the operations on multiple processors. The simplest method is to use a collection of patterns or templates that describe independent computation. For example, if all the subscripts of array elements have the form A(I) where I is the loop's index variable, then we are assured that no cross-iteration data dependences can exist.

A more general method of determining data dependence is by means of data dependence tests. Generically, the object is to solve a system of equations in integer values to determine if two memory references may refer to the same memory location. Chapter 2 goes into the details of the dependence analysis problem and static algorithms for calculating data dependence.

In static data dependence analysis, the problem is: given two array references

$$\begin{array}{lll} A(f_1(i_1,\ldots,i_n), & \ldots, & f_m(i_1,\ldots,i_n)) \\ A(g_1(j_1,\ldots,j_n), & \ldots, & g_m(j_1,\ldots,j_n)) \end{array}$$

generate a set of equations whose solution will determine if the two array references ever reference the same memory location.

The solution to the set of equations

$$\begin{aligned} f_1(i_1, \ldots, i_n) &= g_1(j_1, \ldots, j_n) \\ &\cdots \\ f_m(i_1, \ldots, i_n) &= g_m(j_1, \ldots, j_n)) \end{aligned}$$

corresponds to determining the equality of the subscript in each dimension of the array references. The equations must be satisfied simultaneously for legal values of $i_1 \ldots i_n$ and $j_1 \ldots j_n$.

The static dependence tests evaluated in Chapter 8 are designed either to exactly solve the equations, or to give a conservative approximation. We wish in this chapter to explore a dynamic solution to the dependence analysis problem. The dynamic solution may not use the same formulation as the solution to the static problem. In particular, dynamically we can use the definition of data dependence directly; that is, we can test the memory locations directly to determine dependence.

The technique presented in this chapter has two uses. The first is to generate an interprocedural list of cross-iteration data dependences. The second is to record the dependence distances that occur during the execution. Both uses have many applications. The *output*-dependences show which variables need to be considered for possible localization or expansion to enable the parallel execution of a loop. The *flow*-dependences pinpoint the locations that serialize loops that must be removed to execute the loop concurrently. The absence of all dependences is an indication that the loop may be a candidate for parallel execution. The record of dependence distances can be used to evaluate the suitability of transformation techniques which depend on large dependence distances for their effectiveness. If such distances exist, then these techniques may have applicability to the program being studied. Also, the effectiveness of the `doacross` loop depends on the amount of work in the loop and the length of the dependence distances.

10.2 Overview of the Evaluation Method

Dynamic analysis has advantages over the traditionally used static methods. Since we have the machine state available at the point of the variable reference, we can use this information to resolve easily problems that might arise in the static analysis. For example, during static analysis a difficult problem is to determine if two array references refer to the same memory location. This problem is trivial to determine at run-time. We have the virtual addresses of the array elements, and a simple arithmetic comparison of the addresses determines the dependence or independence of the two references.

Other difficulties that arise from routine boundaries, in a static analysis, are not problems for dynamic analysis. The aliasing that can potentially occur during a subroutine invocation are transparent to run-time observations.

10.2.1 Strategy for Dynamic Analysis

One problem that arises when we look at the dependence problem dynamically is that even though it is trivial to determine if we have a dependence, it is nontrivial to determine if this dependence will be troublesome.

If we have the following code fragment:

```
S1 :   DO I = 1, N
S2 :      A(I) = I
S3 :      B(I) = A(I) + I
S4 :   ENDDO
```

We correctly observe that there exists a *flow*-dependence from S_2 to S_3, but this is not a cross-iteration dependence and does not inhibit parallel execution.

However, in this slightly altered code fragment:

```
S1 :   DO I = 1, N
S2 :      A(I) = I
S3 :      B(I) = A(I-1) + I
S4 :   ENDDO
```

a cross-iteration data dependence does exist. Here the previous iteration of the loop sets the value of A that is read in statement S_3. Without any information other than the virtual addresses of the array elements, we are unable to distinguish between these two cases.

In Chapter 4 we described a method of associating shadow variables with the programs original variables. We place timestamps in these shadow variables that indicate, relative to the start of the program's execution, when a particular value of a variable would be available. The aliasing between the shadow variables allows the instrumentation to trivially determine when two variables share the same timestamp. This method of storing timestamps is adequate for parallelism estimation, but it is inadequate for determining cross-iteration dependences.

If the timestamps are not adequate, what would be sufficient? The critical information required to determine when a cross-iteration dependence occurs is to determine when accesses to a variable occur in different loop iterations. A simple method could store the loop index in the shadow of the array element. For the previous example, this is sufficient to determine if two accesses are in different iterations. Normalizing the loops index values into points in the iteration space eliminates some minor problems with respect to non-unit strides.

However, using index values or a simple iteration space has limitations. If we have a multiply nested loop, then we need to keep a vector of loop indices to determine our location in the iteration space with respect to other loop instances. This additional information is linearly proportional to the maximal loop nesting depth we wish to consider.

We can recast the vector of loop indices, or our position in the iteration space, by using an alternative basis for recording the point in the iteration space. Linearizing the iteration space allows one to pack a multi-dimensional iteration space into a linear representation. The process of index space linearization can be extended to a global iteration numbering (GIN) scheme. In this scheme each time a new loop iteration is started, we record a GIN for the iteration and record the GIN for the first and current iterations of every active loop. To determine if two references to a variable are from different iterations, compare the GINs of the two references and use this information to tell if the references occurred in the same iteration or that the references form a cross-iteration dependence.

10.2.2 Dependence Distances

The dependence distance of an arc, in a singly nested loop, can be calculated by subtracting the GINs for the endpoints of the arc. For multiply nested loops this method no longer is correct. The GIN of successive iterations of outer loops are not consecutive. Thus, we need to record the GIN if we wish to determine dependence distances. To be fully accurate we need one record for each loop iteration. As an approximation to recording the GINs of every iteration, we will only record the GINs for the last k iterations. Searching through this FIFO buffer still allows one to determine precisely distances less than k, and to also determine if a dependence distance greater than k occurred.

Let's examine the following loop nest:

```
S1 :  DO I = 1, 3
S2 :     DO J = 1, 3
S3 :        A(f1(I,J)) = ··· A(f2(I,J)) ···
S4 :     ENDDO
S5 :  ENDDO
```

(1,–) 1	(1,1) 2	(1,2) 3	(1,3) 4
(2,–) 5	(2,1) 6	(2,2) 7	(2,3) 8
(3,–) 9	(3,1) 10	(3,2) 11	(3,3) 12

Table 10.1: Global iteration numbering

In Table 10.1, the effects of GIN are shown. Notice the column that has entries of the form (n, –). The "–" signifies that the J or inner loop, is not active at this time. To illustrate the usage of the GIN scheme consider three situations. First, assume we have the array A written in iteration (2,2) and read in iteration (2,3). The first iteration corresponds to a GIN of 7 and the second to a GIN of 8.

At the time of the second access we have a loop state of:

Loop	Start	Current	FIFO
I	1	5	1
J	6	8	7,6

From this loop state information we can determine that the first access at GIN=7 occurred while the current instance of loop J was executing but in a previous iteration. Examining the FIFO buffer of previous iterations, we see that the last iteration has the same GIN and thus we can conclude that the dependence distance is 1.

For the second example, consider the iterations (1,1) and (3,3). From Table 10.1 we have GINs of 2 and 12.

Loop	Start	Current	FIFO
I	1	9	5,1
J	10	12	11,10

From the loop state table we see that the first reference did not occur in the current loop ($2 < 10$), but it did occur in the outer loop ($1 \leq 2 \leq 9$). Since we know that the I loop is the first common loop that is still active, we check the FIFO field of the table and see that we need to look back two entries in the FIFO buffer to find a GIN that is smaller or equal to the GIN of the first reference. Thus, we have determined that a cross-iteration dependence occurred with respect to the outer loop with a dependence distance of 2.

Finally, modify the example to have two statements in the loop body, and consider the case where the GIN for both references were identical. Here we are assured that a cross-iteration dependence did not occur, or in other words, the dependence distance is zero.

10.3 Uses of the Dynamic Dependence Analysis Method

Experimental techniques can potentially generate vast amounts of data. It may be preferable to show the detailed results by a collection of short examples that display the properties of the technique instead of applying the technique to a large dataset.

In all the examples, only *flow*- and *output*-dependences are reported. The method used during the dynamic analysis records information only about writes to a variable, not about reads. The global iteration number (GIN) of the last write is sufficient to discover an *output*-dependence at the next write or a *flow*-dependence at the next read. To calculate the same information about *input*- or *anti*-dependences, the GIN of the last read would need to be recorded. This is possible but outside the scope of this experiment.

Figures 10.1 to 10.3 show an example program and its results. This example was designed to show the recognition of *flow*-dependences with various dependence distances. Figure 10.2 shows the report generated by running the instrumented program in Figure 10.3. We can interpret the results as follows:

- Each loop in the program is listed in alphabetical order via a tag that is composed of the routine name and the line number in the file. For example `'EX1:4'` refers to line 4 in the file that contains the routine `EX1`.
- After the loop tag is a bracket enclosed list of information. The item `S=1` means that during one execution of the loop at least one cross-iteration dependence was discovered. The item `P=1` means that during one execution of the loop no cross-iteration dependences were found.
- Additional tags are: `ONE` to denote a one trip loop, and `ZERO` to denote a zero trip loop. These tags are important since by definition, zero and one trip loops cannot have any cross-iteration dependences and will therefore be classified as parallel.
- Following the loop tag are a collection of lines starting either with a `<F>` or `<O>`. These markers signify *flow*- and *output*-dependences respectively. The next item on the line is the tag of the sink for the dependence. Again, the form of the tag is *routine*:*line*.

```
Line#
1     PROGRAM EX1
2     PARAMETER (N = 10)
3     DIMENSION A(N)
4     DO I = 1, N
5        A(I) = I
6     ENDDO
7     DO I = 3, N
8        A(I) = A(I-2)
9     ENDDO
10    DO I = 4, N
11       A(I) = A(I-3)
12    ENDDO
13    END
```

Figure 10.1: Example one

```
    ----- SUMMARY: Dynamic Dependence Relation Recognition -----

EX1:4       [ P=1 ]

EX1:7       [ S=1 ]
 <F> EX1:8      A(I-2)  #2:6

EX1:10      [ S=1 ]
 <F> EX1:11     A(I-3)  #3:4
```

Figure 10.2: Example one results

```
PROGRAM EX1
REAL A
INTEGER C$A, C$EX1, I, C$I, N$Z, N, N$N
PARAMETER (N=10)
DIMENSION A(N), C$A(N)
COMMON /N$NBLOCK/ N$N, N$Z
CALL PROC$ENTRY('EX1')
CALL DO$ENTRY('EX1:5',C$I,0)
DO I = 1, N, 1
   CALL DO$NEXT('EX1:5')
   A(I) = I
   CALL DO$STMT('EX1:6',C$A(I),1,C$I,'A(I)!I')
ENDDO
CALL DO$EXIT('EX1:5')
CALL DO$ENTRY('EX1:9',C$I,0)
DO I = 3, N, 1
   CALL DO$NEXT('EX1:9')
   A(I) = A(I-2)
   CALL DO$STMT('EX1:10',C$A(I),2,C$A(I-2),C$I,'A(I)!A(I-2)!I')
ENDDO
CALL DO$EXIT('EX1:9')
CALL DO$ENTRY('EX1:13',C$I,0)
DO I = 4, N, 1
   CALL DO$NEXT('EX1:13')
   A(I) = A(I-3)
   CALL DO$STMT('EX1:14',C$A(I),2,C$A(I-3),C$I,'A(I)!A(I-3)!I')
ENDDO
CALL DO$EXIT('EX1:13')
CALL PROC$EXIT('EX1')
STOP
END
```

Figure 10.3: Example one instrumented

- Following the tag for the dependence are a list of variables and array references that are involved in a dependence. The last section of the line displays the dependence distances summarized for all instances of the statement. The form of the summary is #*distance*:*count*. For example #2:6 means that the leading term of the distance vector was equal to two a total of six times. Since the distances are recorded using a sliding window FIFO buffer, any distance greater than the maximum buffer size will be reported as >*max*:*count*.

From Figure 10.2 we can see that the first loop was executed once with no dependences. The second loop reports cross-iteration dependences of distance 2 and the third loop reports cross-iteration dependences of distance 3.

Figure 10.4 shows a simple example that reports not only the *flow*-dependences, but the *output*-dependences as well. It is interesting to see that by interchanging to two loop (or renaming the subscripts), we can show that either the inner or the outer loop is parallel. It is important to note that the dependence counts of 90 illustrate that the first iteration of the inner loop does not depend on other computation in the loopnest.

Figures 10.6 and 10.7 show that subroutine boundaries pose no additional problem in determining the dependences that occur during a program's execution. We have the same results as in Figure 10.5 with the alteration that the specific location of the dependence sink has changed to line 17, and the local name of the variable that causes the dependence is now `X`.

We can also show in Figure 10.8 that aliasing of variables does not cause a problem. We again have the same results as in Figures 10.5 and 10.7 with the alteration that the *flow*- and *output*- dependences are not attributed to different local variables. Currently we have no means of reporting that `X, Y` were dynamically aliased to `A(I)` and `A(J)`.

We can combine the previous examples into a composite program. In Figure 10.10, we have a *flow*- dependence of distance two in the outer loop of the first loopnest. In the same loopnest we have an *output*-dependence of distance one in the inner loop. By switching the index used in the subscripts, we interchange the types of the dependences reported for the two loops in the second loopnest.

```
Line#
1     PROGRAM EX2
2     PARAMETER (N = 10)
3     DIMENSION A(N)
4     DO I = 1, N
5        DO J = 1, N
6           A(I) = A(I) + 1
7        ENDDO
8     ENDDO
9     DO I = 1, N
10       DO J = 1, N
11          A(J) = A(J) + 1
12       ENDDO
13    ENDDO
14    END
```

Figure 10.4: Example two

```
    ----- SUMMARY: Dynamic Dependence Relation Recognition -----

EX2:4      [ P=1 ]

EX2:5      [ S=10 ]
 <F> EX2:6     A(I)  #1:90
 <O> EX2:6     A(I)  #1:90

EX2:9      [ S=1 ]
 <F> EX2:11    A(J)  #1:90
 <O> EX2:11    A(J)  #1:90

EX2:10     [ P=10 ]
```

Figure 10.5: Example two results

```
Line#
1       PROGRAM EX3
2       PARAMETER (N = 10)
3       DIMENSION A(N)
4       DO I = 1, N
5          DO J = 1, N
6             CALL F( A(I) )
7          ENDDO
8       ENDDO
9       DO I = 1, N
10         DO J = 1, N
11            CALL F( A(J) )
12         ENDDO
13      ENDDO
14      END
15
16      SUBROUTINE F( X )
17      X = X + 1
18      RETURN
19      END
```

Figure 10.6: Example three

```
    ----- SUMMARY: Dynamic Dependence Relation Recognition -----

EX3:4       [ P=1 ]

EX3:5       [ S=10 ]
 <F> F:17           X  #1:90
 <O> F:17           X  #1:90

EX3:9       [ S=1 ]
 <F> F:17           X  #1:90
 <O> F:17           X  #1:90

EX3:10      [ P=10 ]
```

Figure 10.7: Example three results

```
Line#
1     PROGRAM EX4
2     PARAMETER (N = 10)
3     DIMENSION A(N)
4     DO I = 1, N
5        DO J = 1, N
6           CALL F( A(I), A(I) )
7        ENDDO
8     ENDDO
9     DO I = 1, N
10       DO J = 1, N
11          CALL F( A(J), A(J) )
12       ENDDO
13    ENDDO
14    END
15
16    SUBROUTINE F( X, Y )
17    X = Y + 1
18    RETURN
19    END
```

Figure 10.8: Example four

```
    ----- SUMMARY: Dynamic Dependence Relation Recognition -----

EX4:4       [ P=1 ]

EX4:5       [ S=10 ]
 <F> F:17          Y  #1:90
 <O> F:17          X  #1:90

EX4:9       [ S=1 ]
 <F> F:17          Y  #1:90
 <O> F:17          X  #1:90

EX4:10      [ P=10 ]
```

Figure 10.9: Example four results

```
Line#
1      PROGRAM EX5
2      PARAMETER (N = 10)
3      DIMENSION A(N)
4      DO I = 3, N
5         DO J = 1, N
6            CALL F( A(I), A(I-2) )
7         ENDDO
8      ENDDO
9      DO I = 1, N
10        DO J = 3, N
11           CALL F( A(J), A(J-2) )
12        ENDDO
13     ENDDO
14     END
15
16     SUBROUTINE F( X, Y )
17     X = Y + 1
18     RETURN
19     END
```

Figure 10.10: Example five

```
    ----- SUMMARY: Dynamic Dependence Relation Recognition -----

EX5:4       [ S=1 ]
 <F> F:17            Y  #2:60

EX5:5       [ S=8 ]
 <O> F:17            X  #1:72

EX5:9       [ S=1 ]
 <O> F:17            X  #1:72

EX5:10      [ S=10 ]
 <F> F:17            Y  #2:60
```

Figure 10.11: Example five results

10.4 Results from the Perfect Benchmarks

In addition to illustrating the results of this technique on small programs, the dynamic dependence analysis tool was applied to a collection of programs from the Perfect Benchmarks suite. The results are presented in Tables 10.2 to 10.13.

Tables 10.2 to 10.5 list the number of times dynamically that a *flow-* or *output-* dependence that involved an array element respectively occurred during the program's execution

Tables 10.8 to 10.11 include scalars in the counts of dependences. This set of tables shows an even larger bias toward distance one dependences. The reason should be obvious; if we include scalars, then any cross-iteration dependence will have distance one unless it is covered by an conditional statement.

From these tables we see that almost all dependence distances are either one or two. When larger dependence distances appear, they usually do so because all larger distances also occur. Dependence distances that are input data dependent are common. For example, a loop that finds the maximum value in an array may execute an arbitrarily large number of iterations before it assigns an element to the maximum value.

10.5 Conclusion

Traditionally, data dependence analysis has been the realm of static analyses. We show that it is possible to collect *flow-* and *output-*data dependences dynamically. We isolate the loop carried dependences for each loop and also record the iteration distance spanned by the dependence. This approach to dependence collection has several advantages: it works interprocedurally, has no difficultly with aliasing, and has no artificial limits with respect to loop nesting depth.

We have shown that with global iteration numbers (GIN) we can determine *flow-* and *output-*dependences that occur during a program's execution. Additionally with the inclusion of a FIFO buffer of the GINs for the last k iterations of each loop we can determine the dependence distance (up to a maximum of k) of the loop that carries the dependence.

The dynamic dependence recognition procedure provides an indication to the user of whether any cross-iteration dependences existed during the program's execution. This information is calculated interproceduraly and summarized for each `DO` loop in the program. The information gained though dynamic dependence recognition can also be used to automatically guide the parallelizing compiler by providing locations in the program to create multiversion loops. Since the dynamic analysis can recognize parallel loops, the compiler can insert run-time checks into the code to see if the particular relationship between the variables that allowed a loop to be executed in parallel for the test run are generally true.

Finally, the data summarized for the Perfect Benchmarks indicates that almost all dependences that occur dynamically during the program's execution refer to data that was written in the previous iteration of the loop that carries the dependence. A consequence of this observation is that restructuring transformations that rely on large dependence distances may not be applicable to this collection of programs.

Code	#1	#2	#3	#4
adm(AP)	49494854	0	0	0
arc2d(SR)	207300417	67070800	0	0
bdna(NA)	59255818	219315	173655	100575
dyfesm(SD)	109286250	1799958	1490807	1422337
flo52q(TF)	33563778	0	0	0
mdg(LW)	95667545	5011164	2860092	2145618
ocean(OC)	6822672	0	0	0
qcd2(LG)	14516959	2441216	195840	64512
track(MT)	15847034	49462	46769	45194
trfd(TI)	189224350	0	0	0

Table 10.2: Number of occurrences of each *flow*-dependence distance for array references, distances #1 to #4

Code	#5	#6	#7	#8	>8
adm(AP)	0	0	0	0	0
arc2d(SR)	0	0	0	0	0
bdna(NA)	74643	93471	71739	53679	888666
dyfesm(SD)	1369313	1285460	1217050	1148659	12267143
flo52q(TF)	0	0	0	0	0
mdg(LW)	1253718	705582	238815	120330	696042
ocean(OC)	0	0	0	0	0
qcd2(LG)	0	0	16128	0	755712
track(MT)	43868	43003	41980	40566	807669
trfd(TI)	0	0	0	0	0

Table 10.3: Number of occurrences of each *flow*-dependence distance for array references, distances $\geq$ #5

Code	#1	#2	#3	#4	#5	#6	#7	#8	>8
adm(AP)	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
arc2d(SR)	75.6%	24.4%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
bdna(NA)	97.2%	0.4%	0.3%	0.2%	0.1%	0.2%	0.1%	0.1%	1.5%
dyfesm(SD)	83.2%	1.4%	1.1%	1.1%	1.0%	1.0%	0.9%	0.9%	9.3%
flo52q(TF)	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
mdg(LW)	88.0%	4.6%	2.6%	2.0%	1.2%	0.6%	0.2%	0.1%	0.6%
ocean(OC)	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
qcd2(LG)	80.7%	13.6%	1.1%	0.4%	0.0%	0.0%	0.1%	0.0%	4.2%
track(MT)	93.4%	0.3%	0.3%	0.3%	0.3%	0.3%	0.2%	0.2%	4.8%
trfd(TI)	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Table 10.4: Percentage of occurrences of each *flow*-dependence distance for array references

Code	#1	#2	#3	#4
adm(AP)	37721502	11197760	0	0
arc2d(SR)	178509573	0	0	0
bdna(NA)	83750657	402648	248700	140796
dyfesm(SD)	115543052	240762	76	300
flo52q(TF)	1668343	0	1	1
mdg(LW)	502402967	16714241	9824396	7000879
ocean(OC)	74526468	0	0	0
qcd2(LG)	53892821	704512	0	0
track(MT)	8599911	72730	23515	10100
trfd(TI)	195543420	0	0	0

Table 10.5: Number of occurrences of each *output*-dependence distance for array references, distances #1 to #4

Code	#5	#6	#7	#8	>8
adm(AP)	0	0	0	0	0
arc2d(SR)	0	0	0	0	0
bdna(NA)	100260	112524	79260	61572	920580
dyfesm(SD)	15448	16	8	0	0
flo52q(TF)	3	0	5	1	69
mdg(LW)	4564260	2745239	1054609	518887	2857284
ocean(OC)	0	0	0	0	0
qcd2(LG)	0	0	0	0	0
track(MT)	6894	6144	4665	3046	23497
trfd(TI)	0	0	0	0	0

Table 10.6: Number of occurrences of each *output*-dependence distance for array references, distances $\geq$ #5

Code	#1	#2	#3	#4	#5	#6	#7	#8	>8
adm(AP)	77.1%	22.9%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
arc2d(SR)	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
bdna(NA)	97.6%	0.5%	0.3%	0.2%	0.1%	0.1%	0.1%	0.1%	1.1%
dyfesm(SD)	99.8%	0.2%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	0.0%	0.0%
flo52q(TF)	100.0%	0.0%	<0.1%	<0.1%	<0.1%	0.0%	<0.1%	<0.1%	<0.1%
mdg(LW)	91.7%	3.1%	1.8%	1.3%	0.8%	0.5%	0.2%	0.1%	0.5%
ocean(OC)	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
qcd2(LG)	98.7%	1.3%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
track(MT)	98.3%	0.8%	0.3%	0.1%	0.1%	0.1%	0.1%	<0.1%	0.3%
trfd(TI)	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Table 10.7: Percentage of occurrences of each *output*-dependence distance for array references

Code	#1	#2	#3	#4
adm(AP)	62149689	0	0	0
arc2d(SR)	209855158	67070800	0	0
bdna(NA)	67625525	236562	185323	108018
dyfesm(SD)	135233341	1800010	1490851	1422364
flo52q(TF)	34631739	1163	1160	1158
mdg(LW)	180147729	7033746	3402968	2421227
ocean(OC)	70286802	0	0	0
qcd2(LG)	16294790	2444288	195840	64512
track(MT)	16589223	53515	48219	45996
trfd(TI)	193140087	0	0	0

Table 10.8: Number of occurrences of each *flow*-dependence distance for all references, distances #1 to #4

Code	#5	#6	#7	#8	>8
adm(AP)	0	0	0	0	0
arc2d(SR)	0	0	0	0	0
bdna(NA)	80734	100248	76902	58078	966374
dyfesm(SD)	1369341	1285481	1217063	1148663	12271107
flo52q(TF)	1154	1152	1151	990	17233
mdg(LW)	1344933	786307	256327	128708	742255
ocean(OC)	0	0	0	0	0
qcd2(LG)	0	0	16128	0	755712
track(MT)	44492	46544	52294	61635	1688710
trfd(TI)	0	0	0	0	0

Table 10.9: Number of occurrences of each *flow*-dependence distance for all references, distances $\geq$ #5

Code	#1	#2	#3	#4	#5	#6	#7	#8	>8
adm(AP)	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
arc2d(SR)	75.8%	24.2%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
bdna(NA)	97.4%	0.3%	0.3%	0.2%	0.1%	0.1%	0.1%	0.1%	1.4%
dyfesm(SD)	86.0%	1.1%	0.9%	0.9%	0.9%	0.8%	0.8%	0.7%	7.8%
flo52q(TF)	99.9%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%
mdg(LW)	91.8%	3.6%	1.7%	1.2%	0.7%	0.4%	0.1%	0.1%	0.4%
ocean(OC)	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
qcd2(LG)	82.4%	12.4%	1.0%	0.3%	0.0%	0.0%	0.1%	0.0%	3.8%
track(MT)	89.0%	0.3%	0.3%	0.2%	0.2%	0.2%	0.3%	0.3%	9.1%
trfd(TI)	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Table 10.10: Percentage of occurrences of each *flow*-dependence distance for all references

Code	#1	#2	#3	#4
adm(AP)	157819812	11198011	9	21
arc2d(SR)	495979193	0	0	0
bdna(NA)	250432130	421275	260598	148239
dyfesm(SD)	150564026	240779	82	303
flo52q(TF)	44282543	9	7	13
mdg(LW)	613647362	20387302	11333672	7955313
ocean(OC)	479772792	68971	0	24340
qcd2(LG)	60823687	977920	0	0
track(MT)	19031958	214196	80653	59518
trfd(TI)	206570081	0	0	0

Table 10.11: Number of occurrences of each *output*-dependence distance for all references, distances #1 to #4

Code	#5	#6	#7	#8	>8
adm(AP)	12	36	21	0	2970
arc2d(SR)	0	0	0	0	0
bdna(NA)	106351	119301	84423	65971	998288
dyfesm(SD)	15451	16	8	0	0
flo52q(TF)	9	3	8	1	117
mdg(LW)	5096774	3085091	1171742	575780	3175448
ocean(OC)	0	0	0	0	1
qcd2(LG)	0	0	0	0	0
track(MT)	35336	40640	32810	16309	133057
trfd(TI)	0	0	0	0	0

Table 10.12: Number of occurrences of each *output*-dependence distance for all references, distances $\geq$ #5

Code	#1	#2	#3	#4	#5	#6	#7	#8	>8
adm(AP)	93.4%	6.6%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%
arc2d(SR)	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
bdna(NA)	99.1%	0.2%	0.1%	0.1%	<0.1%	<0.1%	<0.1%	<0.1%	0.4%
dyfesm(SD)	99.8%	0.2%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	0.0%	0.0%
flo52q(TF)	100.0%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%	<0.1%
mdg(LW)	92.1%	3.1%	1.7%	1.2%	0.8%	0.5%	0.2%	0.1%	0.5%
ocean(OC)	100.0%	<0.1%	0.0%	<0.1%	0.0%	0.0%	0.0%	0.0%	<0.1%
qcd2(LG)	98.4%	1.6%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
track(MT)	96.9%	1.1%	0.4%	0.3%	0.2%	0.2%	0.2%	0.1%	0.7%
trfd(TI)	100.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%

Table 10.13: Percentage of occurrences of each *output*-dependence distance for all references

Chapter 11

CONCLUSIONS AND FUTURE DIRECTIONS

11.1 General Observations

In the beginning there was a dream of unlimited performance through parallel processing. The potential linear scalability of performance through the addition of processors is appealing. However, frustration sets in as the difficulty of determining the parallel structure of the computation and of discovering what information is shared becomes apparent. Gradually, these problems turn the dream into a nightmare.

To wake up from the nightmare, one needs the illumination that the tools and techniques presented in this thesis provide. In the proper light, the information shared by components of the program can be identified and dealt with to make the parallelism explicit. It is hoped that the techniques presented in this thesis allow a greater understanding of a program's parallel structure to become available to the user and possibly to the compiler.

The experiments described in this thesis are all based on the idea that a program's parallel structure is embedded in the computation performed by the program. By observing the computation, we can observe how information is shared by the statements. A snapshot taken during one run of the program can shed light on at least one instance of the program's behavior. From this snapshot, the information gained can expose the threads of execution to be independent that might at first appear to be entangled.

The information generated by the tools described in the chapters of this thesis provide methods for generating additional information unavailable by other means. The information may be about the maximal inherent parallelism for various parallel models such as loop, statement, or operation granularity. It may be about inherent parallelism when only a limited number of processors are available. The additional information may be about the effects of unrecognized induction variables on the parallel execution of the program. The information may also describe the effectiveness of data dependence analysis both in static terms and as measured by a program's parallel performance. It is possible the information may be just a report of the presence of cross-iteration data dependences classified by the number of iterations they span. Or, the information provided about the program could be a prioritized list of locations where the way computation is performed potentially degrades the performance of the program.

11.1.1 Dynamic Evaluation and Analysis

Static analysis determines what a program is capable of doing. Dynamic analysis determines what a program actually did. It is the subtle distinction between static and dynamic analysis that makes it possible to use the latter to measure the effectiveness of the static analysis methods.

The focus of this research has been on discovering ways in which dynamic analysis can be applied to the tasks of increasing ones understanding of the behavior of programs and the effects of parallelizing compilers. These two aspects are highly interrelated, and knowing the answer to one of them can lead to the answer of the other.

As described in the introduction and shown in Figure 1.1, all tools fit into a framework encompassing the different forms of dynamic analysis. Each of the tools presented in this thesis measure one part of the overall picture; it is hoped that when the results are integrated, there will be a conclusion that will more closely represent the properties of the program.

In each chapter, we examined a solution to some portion of the problem. A summary of the major results of each chapter will put the individual techniques into perspective.

The machine model used throughout this research was an ideal parallel machine assuming an unlimited number of processors. Since no resource constraints are considered, we have isolated the program from its environment. The aim of this isolation is to evaluate the program, independent of any artificial constraints.

The first real component in this thesis is the Delta program manipulation system, described in Chapter 3. The flexibility of this environment allowed the rapid prototyping and exploration of the concepts described in the experimental chapters. Without the availability of such a tool it is unlikely that as comprehensive a collection of techniques could have been developed.

The experimental foundation of the dynamic analysis techniques and of the instrumentation methods is described in Chapter 4. Here the critical path length is the dominant measure of a program. Comparing the critical path length with the number of operations executed by the program indicates the parallelism inherent in the program. Critical path analysis uses the dynamic structure of the program to determine how well it might execute on a parallel machine.

The next innovation described is the notion of stress analysis. In Chapter 6, we analogize the dependence structure of a program to the load bearing members in a building. Stress analysis points out hotspots or points of high stress along the critical path, that is the points where the dependences introduce a stress that degrades the program's performance.

After laying the foundation of the analysis, we move on to some calculations derived from critical path analysis. We again look at the results from critical path analysis in the light of limited processor resources. We describe how to collect the processor activity histogram and to use this information to derive a speedup curve for varying numbers of processors. The curve is defined by upper and lower bounds on the parallelism for a fixed number of processors. Chapter 5 shows that the bounds are tight and generate an accurate picture of the change in parallelism when additional processors are used for a computation.

One of the program aspects not dealt with by the standard critical path analysis is the removal of induction variables that is performed by parallelizing compilers. We introduce, in Chapter 7, a method to recognize the existence of induction variables dynamically during a program's execution and to simulate the effect of removing these variables. The results in Chapter 7 indicate that, for the Perfect Benchmarks®, only two of the codes will benefit substantially from the removal of the induction calculations. Through the manual optimization

listed in [EHLP91], the technique of induction variable removal was cited as beneficial to the same two programs as found by our technique.

Next we digress again to provide some background on the static effectiveness of data dependence tests. Another view of a program is through the eyes of a parallelizing compiler. In Chapter 8, we give a profile of the static dependence results for the Perfect Benchmarks. The conclusion from this experiment is that for the Perfect Benchmarks, the addition of dependence tests that are more powerful than Banerjee's inequalities had minimal impact. Most of the contribution was in proving the dependence of potential dependence arcs that had been assumed dependence without the advanced tests.

Static evaluation is not sufficient. The percentage of potential dependence arcs discovered by static dependence analysis may not correlate with the parallelism. In Chapter 9, we use a dynamic evaluation of the effectiveness of the static dependence tests to show that, for the Perfect Benchmarks, the more complex dependence tests do not show any advantage when we look at achieved parallelism. This result is not surprising in light of the static evaluation results in Chapter 8. As a byproduct of determining the effectiveness of the data dependence analysis, locations of statically unanalyzable dependences and an indication of their importance to the parallel execution is generated. This list of dependences can be used by the users as locations to rewrite the code, or as a list of test cases to be used by compiler writers to create the next generation of static dependence analysis techniques.

Finally, we totally remove the static analysis of data dependence from the picture. In chapter 10, we present a method where the dynamic cross-iteration dependences that occur during the program's execution are recorded and displayed at the end of the run. With each dependence we also record the dependence distance of the loop that carried the dependence. The major experimental conclusion from this experiment is that large constant dependence distances do not occur with any frequency in the Perfect Benchmarks. Most dependence distances are of distance one, with many also of distance two.

11.2 Observations on the Perfect Benchmarks

The Perfect Benchmarks can be divided into three categories based on the ability of a commercially available parallelizer, KAP/Concurrent, to recognize and exploit the inherent parallelism. This division of the programs may reflect particular deficiencies in the compiler or particular characteristics of the programs.

11.2.1 Poor Performance by KAP/Concurrent

The collection of programs where KAP obtained the smallest amount of the inherent loop-level parallelism is the largest category. Most of the programs in this category exhibit characteristics, measured by the experiments in this thesis, that point out reasons why a compiler based on intraprocedural static analysis is unable to exploit any inherent parallelism. A summary of the results from each experiment will now be discussed with regard to each Perfect Benchmarks program.

- `adm(AP)`: From chapter 9 we observe that this program's performance degrades when subroutine calls are serialized. Thus, the interprocedural analysis necessary to parallelism loops that contain `CALL` statements should benefit this program. It is also possible the number of unanalyzable dependences contribute to its poor showing.

- `bdna(NA)`: The only characteristic that stands out for this program is a drop in performance owing to static dependence analysis. Interprocedural analysis to detect independent subroutine calls would not benefit this program as no performance was lost when the subroutine calls were serialized.

- `ocean(OC)`: Almost all of the lost performance can be explained by the inability of static dependence analysis to understand the subscripts in the program. Also, induction variables play a minor part in degrading the performance.

- `spec77(WS)`: All parallelism seems to be interprocedural. With out this capability, KAP does not even have a chance of performing well on this program. The dependence analysis is relatively easy. A striking observation is that this program exhibits a large difference between the operation-level and the loop-level parallelism measurements. More parallelism might be extracted by finding the statements that are degrading the performance of the loop-level model.

- `track(MT)`: A relatively easy data dependence problem. The drop in performance is entirely owing to the inability of the compiler to recognize interprocedural opportunities for parallelism.

11.2.2 Mediocre Performance by KAP/Concurrent

The next largest category in the Perfect Benchmarks are those for which KAP did a reasonable job at finding the parallelism that was present.

- `dyfesm(SD)`: Two experiments displayed reasons why the performance of this program were not as expected. The dependence analysis was not able to detect independence in a few important cases. Also, the parallelism in loops with subroutine calls was not recognized.

- `mdg(LW)`: The experiments do not provide clear reasons why this program does not perform well. A clue is the high number of dependences that are statically unanalyzable. Another clue is the high discrepancy between the operation-level and the loop-level parallelism.

- `qcd2(LG)`: This is another program where the experiments were not sufficient to determine why the performance is poor. The static dependence analysis was able to classify all dependences. The loss when only allowing intraprocedural parallelism is significant but small. One characteristic of this program may be contributing to its low inherent parallelism. The use of a user-coded random number generator introduces a dependence cycle that must be honored.

- `trfd(TI)`: The reason for the performance of this program is obvious. The lack of induction variable elimination serialized the most important loops in the program. Also, because of the unknown induction variables, the static dependence analysis assumed dependence when it might have shown otherwise in the presence of better information.

11.2.3 Good Performance by KAP/Concurrent

Unfortunately, as a measure of the field of parallelizing compilers, the group of programs where KAP was able to exhibit good performance is the smallest of the categories. The main characteristic binding these programs together is the lack of difficulties in the analyzing or the implementation of the parallelism. The world would be a much nicer place if all programs were similar to these two:

- `arc2d(SR)`: All the experiments in this thesis show that no difficulties exist for this program. It does not have any inductions that degrade performance. It does not have difficult dependences to compute. And finally, it does not require interprocedural parallelism to show good performance.

- `flo52q(TF)`: This program is similar to `arc2d(SR)`. All of the experiments indicate that none of analysis that is considered difficult must be performed.

11.3 Future Directions

As with most scientific endeavors, the experiments in this thesis raise as many questions and challenges as they solve. The future directions of research that are indicated may prove just as fruitful as the experiments so far performed.

All the experiments were performed assuming an ideal parallel machine. This machine has unlimited memory, bandwidth, and processors. Changing the simulated machine to have characteristics closer to a real machine would allow the numbers generated by experiments such as these to have a more immediate relationship with reality.

Extending the simulation model to cope with *anti*-dependences, while increasing the overhead, would allow for a truer representation of the program's structure. Ignoring these dependences may be adequate for performance measurements. However, eventually the location of these dependences must be discovered to eliminate them.

Methods of summarizing the parallelism on objects smaller than a program should be developed. Here, the average parallelism of a loop or routine should indicate the likelihood that transformations in that routine would create explicit parallelism. The challenge is not only to accurately determine the time interval where a routine is active, but also to correctly merge multiple instantiations whether they overlap or are temporally disjoint.

It would be beneficial to continue this line of research to discover more classes of static analysis that can also be performed dynamically. In this way the effectiveness of the static techniques can be empirically measured against the inherent parallelism.

BIBLIOGRAPHY

[All83] John Randal Allen. *Dependence Analysis for Subscripted Variables and Its Application to Program Transformations.* PhD thesis, Rice University, April 1983.

[All85] Alliant Computer Systems Corp., Acton, Massachusetts. *FX/Series architecture manual*, 1985.

[Amd67] Gene M. Amdahl. Validity of the single processor approach to achieving large scale computing capabilities. In *AFIPS Proc. of the SJCC*, volume 31, pages 483–485, 1967.

[ASU86] Alfred. V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: Principles, Techniques and Tools.* Addison–Wesley Publishing Company, 1986.

[ATT] AT&T. *System V Interface Definition*, third edition.

[Ban79] Utpal Banerjee. *Speedup of Ordinary Programs.* PhD thesis, University of Illinois at Urbana-Champaign, October 1979.

[Ban88] Utpal Banerjee. *Dependence Analysis for Supercomputing.* Kluwer Academic Publishers, 1988.

[BC76] John L. Bruno and Edward Grady Coffman. *Computer and Job Shop Scheduling-Theory.* Wiley, 1976.

[Bel66] L. A. Belady. A Study of Replacement Algorithms for a Virtual-Storage Computer. *IBM Systems Journal*, 5(2):78–101, 1966.

[Blu92] William Joseph Blume. Success and Limitations in Automatic Parallelization of the Perfect Benchmarks Programs. Master's thesis, Univ. of Illinois at Urbana-Champaign, Center for Supercomputing Res. & Dev., July 1992.

[CDL88] David Callahan, Jack Dongarra, and David Levine. Vectorizing Compilers: A Test Suite and Results. In *Proceedings of Supercomputing '88*, pages 98–105, November 1988.

[CFR+88] Ron Cytron, Jeanne Ferrante, Barry K. Rosen, Mark N. Wegman, and F. Kenneth Zadeck. An Efficient Method of Computing Static Single Assignment Form. Technical Report CS-88-16, IBM T.J. Watson Research Center, 1988.

[Che89] Ding-Kai Chen. MAXPAR: An Execution Driven Simulator for Studying Parallel Systems. CSRD Report no. 917, University of Illinois at Urbana-Champaign, Center for Supercomp. Res.&Dev., October 1989.

[CKPK90] George Cybenko, Lyle Kipp, Lynn Pointer, and David Kuck. Supercomputer Performance Evaluation and the Perfect Benchmarks. Technical Report 965, University of Illinois at Urbana-Champaign, Center for Supercomp. Res.&Dev., March 1990.

[CSY90] Ding-Kai Chen, Hong-Men Su, and Pen-Chung Yew. The Impact of Syncronization and Granularity on Parallel Systems. CSRD Report no. 942, University of Illinois at Urbana-Champaign, Center for Supercomp. Res.&Dev., May 1990.

[Cyt86] Ron Cytron. DOACROSS: Beyond Vectorization for Multiprocessors. In *Proc. 1986 International Conf. on Parallel Processing*, pages 836–844, August 1986.

[EHLP91] Rudolf Eigenmann, Jay Hoeflinger, Zhiyuan Li, and David Padua. Experience in the Automatic Parallelization of Four Perfect-Benchmark Programs. *Proceedings of the Fourth Workshop on Languages and Compilers for Parallel Computing, Santa Clara, CA*, pages 65–83, August 1991.

[Fu90] Chuigang Fu. Evaluating the Effectiveness of Fortran Vectorizers by Measuring Total Parallelism. CSRD Report no. 1033, University of Illinois at Urbana-Champaign, Center for Supercomp. Res.&Dev., August 1990.

[GKT91] Gina Goff, Ken Kennedy, and Chau-Wen Tseng. Practical dependence testing. *SIGPLAN Notices*, 26(6):15–29, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[GMB88] John L. Gustafson, Gary R. Montry, and Robert E. Benner. Development of Parallel Methods for a 1024-Processor Hypercube. *SIAM Journal on Scientific and Statistical Computing*, 9(4), July 1988.

[Hag90] Mohammad Reza Haghighat. Symbolic Dependence Analysis For High Performance Parallelizing Compilers. Master's thesis, Univ. of Illinois at Urbana-Champaign Center for Supercomputing Res. & Dev., May 1990.

[KBC+74] D. Kuck, P. Budnik, S-C. Chen, Jr. E. Davis, J. Han, P. Kraska, D. Lawrie, Y. Muraoka, R. Strebendt, and R. Towle. Measurements of Parallelism in Ordinary FORTRAN Programs. *Computer*, 7(1):37–46, Jan., 1974.

[KBG90] Andrew W. Kwan, Lubomir Bic, and Daniel D. Gajski. Improving Parallel Program Performance Using Critical Path Analysis. In D. Gelernter, A. Nicolau, and D. Padua, editors, *Languages and Compilers for Parallel Computing*, chapter 18, pages 358–373. The MIT Press, 1990.

[KKP+81] D. J. Kuck, R. H. Kuhn, D. A. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. *Proceedings of the 8th ACM Symp. on Principles of Programming Languages (POPL)*, pages 207–218, Jan., 1981.

[KKP90] Xiangyun Kong, David Klappholz, and Kleanthis Psarris. The I Test: A New Test for Subscript Data Dependence. In *Proc. 1990 International Conf. on Parallel Processing*, August 1990.

[KSC+84] David J. Kuck, Ahmed H. Sameh, Ron Cytron, Alexander V. Veidenbaum, Constantine D. Polychronopoulos, Gyungho Lee, Tim McDaniel, Bruce R. Leasure, Carol Beckman, James R. B. Davies, and Clyde P. Kruskal. The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance. *Proceedings of 1984 International Conference on Parallel Processing*, pages 129–138, Aug.21-24, 1984.

[Kuc90] Kuck & Associates, Inc. *KAP/Concurrent User's Guide.* KAI, Champaign, IL 61820, 1 edition, May 1990. #9005010.

[Kum88] Manoj Kumar. Measuring Parallelism in Computation-Intensive Science / Engineering Applications. *IEEE Transactions on Computers*, 37(9):5–40, 1988.

[Lee80] Ruby Bei-Loh Lee. Emperical Results on the Speed, Efficiency, Redundancy and Quality of Parallel Computations. In *Proc. 1980 International Conf. on Parallel Processing*, pages 91–100, August 1980.

[Lev89] Gary Mark Levin. *An Introduction to ISETL - Version 2.0.* Clarkson University, October 1989.

[LT79] Thomas Lengauer and Robert Endre Tarjan. A Fast Algorithm for Finding Dominators in a Flowgraph. *ACM Transactions on Programming Languages and Systems*, 1(1):121–141, July 1979.

[LYZ89] Zhiyuan Li, Pen-Chung Yew, and Chuan-Qi Zhu. Data Dependence Analysis on Multi-dimensional Array References. In *Proc. 3rd International Conf. on Supercomputing*, pages 215–224, June 1989.

[MHL91] Dror E. Maydan, John L. Hennessy, and Monica S. Lam. Efficient and exact data dependence analysis. *SIGPLAN Notices*, 26(6):1–14, June 1991. *Proceedings of the ACM SIGPLAN '91 Conference on Programming Language Design and Implementation.*

[NF84] Alexandru Nicolau and Joseph A. Fisher. Measuring the Parallelism Available for Very Long Instruction Word Architectures. *IEEE Transactions on Computers*, C-33(11), November 1984.

[Pad89] David A. Padua. The Delta Program Manipulation System — Preliminary Design. CSRD Report no. 808, University of Illinois at Urbana-Champaign, Center for Supercomp. Res.&Dev., June 1989.

[Par90] Parallel Computing Forum. *PCF Fortran*, April 1990.

[Per89] Perfect Club, et al. The Perfect Club Benchmarks: Effective Performance Evaluation of Supercomputers. *International Journal of Supercomputer Applications*, pages 5–40, Fall 1989.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes.* Cambridge University Press, 1988.

[PKL80] David A. Padua, David J. Kuck, and Duncan H. Lawrie. High-Speed Multiprocessors and Compilation Techniques. *IEEE Transactions on Computers*, C-29(9):763–776, September 1980.

[Pug91] William Pugh. The Omega Test: A Fast and Practical Integer Programming Algorithm for Dependence Analysis. *Supercomputing'91*, 1991.

[PW86] David A. Padua and Michael J. Wolfe. Advanced Compiler Optimizations for Supercomputers. *Communications of the ACM*, 29(12):1184–101, December 1986.

[SDDS86] J.T. Schwartz, R.B.K. Dewar, E. Dubinsky, and E. Schonberg. *Programming with Sets: An Introduction to Setl.* Springer-Verlag, 1986.

[SLY90] Zhiyu Shen, Zhiyuan Li, and Pen-Chung Yew. An Empirical Study of FORTRAN programs for Parallelizing Compilers. CSRD Report no. 983, University of Illinois at Urbana-Champaign, Center for Supercomp. Res.&Dev., April 1990.

[SM89] Harvey M. Salkin and Kamlesh Mathur. *Foundations of Integer Programming.* North-Holland, 1989.

[Sny90] W. Kirk Snyder. *The SETL2 Programming Language.* Courant Institute of Mathematical Sciences, May 1990.

[Tow76] R. Towle. *Control and Data Dependence for Program Transformations.* PhD thesis, University of Illinois at Urbana-Champaign, March 1976.

[WB87] Michael Wolfe and Utpal Banerjee. Data Dependence and Its Application to Parallel Processing. *International Journal of Parallel Processing*, October 1987.

[WT92] Michael Wolfe and Chau-Wen Tseng. The Power Test for Data Dependence. *IEEE Transactions on Parallel and Distributed Systems*, 3(5):591–601, September 1992.

Vita

Paul Marx Petersen was born on June 5, 1963 in Grand Island, Nebraska. He entered the University of Nebraska–Lincoln, in 1981 and received a B.S. in Computer Science in 1986. After that, he started his graduate studies at the University of Illinois at Urbana–Champaign. He received his M.S. degree in Computer Science in 1989. Continuing his appointment as a graduate research assistant at the Center for Supercomputing Research and Development (CSRD), he switched his research efforts to the evaluation of compilers. He finished his Ph.D. in Computer Science in December 1992.