Computer Science Department Technical Report
University of California
Los Angeles, CA 90024-1596

TOOLS AND TECHNIQUES FOR PERFORMANCE MEASUREMENT AND PERFORMANCE IMPROVEMENT IN PARALLEL PROGRAMS

Carl Kesselman

June 1991
CSD-910015

UNIVERSITY OF CALIFORNIA
Los Angeles

Tools and Techniques for Performance Measurement and Performance Improvement in Parallel Programs

A dissertation
submitted in partial satisfaction
of the requirements for the degree
Doctor of Philosophy in Computer Science

by

Carl Kesselman

1991

Table of Contents

1 Introduction **1**

1.1 Contributions 3

1.2 Organization of the Dissertation 4

2 Performance, Measurement and Parallel Programs **7**

2.1 The Performance of Systems 7

2.1.1 A definition of performance 7

2.1.2 Methods of performance evaluation 9

2.2 Performance and Parallel Programs 11

2.2.1 Characterizing parallel program execution 11

2.2.2 Performance indexes for parallel programs 13

2.2.3 Measurement models 16

2.2.4 Deriving performance indexes from performance measurements 19

2.3 Issues in Performance Measurement of Parallel Systems 20

2.3.1 Instrumentation and the implementation of sensors . . 21

2.3.2 The placement of instrumentation in a parallel program 23

2.3.3 The probe effect 23

2.3.4 Volume of performance data 25

2.4 Current Performance Measurement Systems for Parallel Programs 25

2.4.1 Generating event traces on the BBN TC2000 25

2.4.2 Collecting performance data on the **NCUBE** multiprocessor 26

2.4.3 Performance measurement in IPS-2 27

2.5 Summary 27

3 A New Approach to Performance Measurement **29**
3.1 Limitations of Current Performance Measurement Systems . . 30
3.2 Our Approach to Performance Measurement 31
3.3 Profiling Parallel Programs 33
3.4 Summary 37

4 Programming Parallel Computers and Performance Measurement **39**
4.1 Program Composition Notation 39
4.1.1 Basic concepts of *PCN* 42
4.1.2 *PCN* data types 43
4.1.3 The *PCN/* programming model: syntax and semantics 44
4.1.4 Executing *PCN* on multiple processors 47
4.1.5 A *PCN* example 48
4.2 Implementation of *PCN* 50
4.2.1 The Program Composition Machine 51
4.2.2 Core *PCN*and the *PCM* execution model 53
4.2.3 The reduction component 55
4.2.4 The communications component 57
4.3 Summary 59

5 Profiling Techniques for Parallel Programs **61**
5.1 Measurement and Execution Profiling 61
5.1.1 Principles of design 62
5.1.2 Determining how to measure 63
5.1.3 Direct measurement 64
5.1.4 Statistical methods (sampling) 65
5.1.5 Indirect methods 66
5.2 Accuracy in Performance Measurement 68
5.3 Implementation and Measurement 70
5.3.1 Design goals 70
5.4 Test Programs 71
5.5 Measuring Event Frequencies 73
5.5.1 Frequency measurement of parallel composition 74
5.5.2 Frequency measurement of choice composition 76
5.5.3 Implementation of guard evaluation 77
5.5.4 Profiling guard evaluation 83

5.5.5 A comparison of the storage overhead for linear search and decision tree implementations . . . 85
5.5.6 Comparing the runtime overhead of linear search and decision tree implementations . . . 87
5.5.7 Reducing the measurement overhead . . . 89
5.5.8 Statistical models of guard execution . . . 91
5.5.9 Accuracy of the simplified performance model . . . 94
5.5.10 Reducing the measurement error in the simplified flow graph . . . 96
5.5.11 Assignments and definitions . . . 100
5.5.12 Foreign programs . . . 101
5.6 Measuring Execution Time . . . 102
5.6.1 *PCN* procedures . . . 102
5.6.2 Validation of the *PCN* execution time model . . . 103
5.6.3 Execution time of foreign programs . . . 105
5.7 Profiling Multiple Processor Execution . . . 106
5.7.1 Profiling idle time . . . 106
5.7.2 Measuring idle time . . . 108
5.7.3 Profiling communication time . . . 109
5.8 Summary . . . 113

6 Implementation of a Parallel Execution Profiler 115
6.1 Profiler Overview . . . 115
6.2 Runtime Support for Profiling . . . 116
6.2.1 Storage for counters and timers . . . 117
6.2.2 Measurement in the *PCM* . . . 119
6.2.3 Measuring program activity . . . 119
6.2.4 Measuring *PCM* activity . . . 121
6.3 Compiler Support for Profiling . . . 122
6.4 Collecting the Profile Data . . . 127
6.5 Summary . . . 132

7 Interactive Performance Visualization for Parallel Execution Profiles 133
7.1 *Gauge* . . . 133
7.1.1 Obtaining an execution profile from measurement data 134
7.2 Presentation of Parallel Execution Profile . . . 137

7.3 Visualizing Parallel Profile Data . . . 138
7.3.1 Related work in performance visualization . . . 138
7.3.2 Performance displays in *Gauge* . . . 139
7.4 Interactive Data Analysis . . . 141
7.4.1 Related work in interactive data analysis . . . 142
7.4.2 Interactive data analysis in *Gauge* . . . 143
7.4.3 Creating alternative views of performance data . . . 144
7.4.4 Querying values in a histogram . . . 146
7.5 Summary . . . 147

8 Profiling to Improve Program Performance: A Case Study 149
8.1 The Application . . . 149
8.1.1 Parallel implementation . . . 150
8.1.2 Execution environment . . . 151
8.2 Performance Improvement of FLOW . . . 152
8.2.1 Preliminary analysis . . . 153
8.2.2 Improving the load balance in FLOW . . . 155
8.3 Decreasing the Cost of Global Synchronization . . . 163
8.4 Decreasing the Cost of Relaxation Step . . . 170
8.5 Summary . . . 173

9 Summary and Conclusions 175
9.1 Future Work . . . 176
9.1.1 Alternative uses of performance data . . . 176
9.1.2 Snapshots and profiling . . . 176
9.1.3 *Gauge* and performance visualization . . . 177
9.1.4 Application level performance measurement . . . 177
9.2 Final Thoughts . . . 178

A A Summary of Notation 179

B A Summary of the Program Composition Machine 181
B.1 Machine Registers . . . 181
B.2 Instruction Summary . . . 182

C Failure Probabilities of Guard Test Instructions 185

Bibliography 187

List of Figures

2.1 An abstraction hierarchy for parallel program execution 16

3.1 Execution graph of a program to be profiled 34
3.2 A more complete parallel profile 35

4.1 A multilingual program 41
4.2 Syntax of *PCN* composition operators 46
4.3 A *PCN* program to sum the nodes in a tree 49
4.4 A *PCN* program to perform the DAXPY operation 50
4.5 The abstract execution algorithm for core *PCN* 54
4.6 Basic structure of *PCM* code representation of a core *PCN* program 56
4.7 The reduction process within the *PCM* 58
4.8 The communications component with the *PCM* 59

5.1 The measurement hierarchy for a *PCN* computation 64
5.2 The form of a performance model 67
5.3 Basic structure of the *PCM* code for a parallel composition . . 75
5.4 Flow graph of linear search implementation of guard evaluation 78
5.5 A *PCN* program in which linear search executes tests that are redundant or tests that cannot succeed 79
5.6 Linear search implementation of sample program 80
5.7 Flow graph of decision tree implementation of guard execution 81
5.8 Decision tree implementation of sample program 82
5.9 Storage overhead required to profile guard evaluation on the basis of execution path. 86
5.10 Runtime overhead for measurement of execution paths 88
5.11 Storage overhead for measuring implication local execution path 91

5.12 A simplified flow graph for linear search implementation of guard evaluation 92
5.13 Space overhead with measurements based on a simplified flow graph 94
5.14 Time overhead with measurements based on a simplified flow graph 95
5.15 Maximum possible error in guard execution 96
5.16 Measurement error using conditional failure probabilities to estimate the guard test that fails 100
5.17 Variance versus the number of instruction classes 104
5.18 Distribution of errors in execution time 105
5.19 Example of how assignment of idle time to programs can be fooled 108
5.20 Code sample showing how communication is unrelated to a program 110
5.21 Runtime overhead for communications measurements 112
5.22 Distribution of errors in times including communications measurements 113

6.1 An overview of a performance analysis system 116
6.2 The layout of sections and fields in a *PCM* module 118
6.3 The *PCN* compilation process 123
6.4 The structure of the *PCM* code generated by the encoder . . . 125
6.5 The format of the data output for each processor by the profile server 129
6.6 The master profiler algorithm 130
6.7 The node profiler algorithm 131

7.1 A flat 3D display of execution time 134
7.2 The representation of profile data in *Gauge* 135
7.3 A stacked histogram showing the idle time and execution time 141
7.4 A direct query of profile data from a histogram 146

8.1 The communications pattern between lines in FLOW 151
8.2 The execution time profile of FLOW 154
8.3 Breakdown of time spent in FLOW 156
8.4 Load balance for FLOW 157

8.5 Per processor execution frequency for **gauss1** in FLOW 158
8.6 Execution frequency of **gauss1** in FLOW1 160
8.7 Execution time for **gauss1** in FLOW1 161
8.8 Breakdown of execution times in FLOW1 162
8.9 The per node idle times for FLOW1 163
8.10 The per node, per program breakdown of idle times in FLOW1 164
8.11 *PCN* code for line1 166
8.12 *PCN* code for wait_int 167
8.13 Breakdown of idle times in FLOW2 169
8.14 The idle time in FLOW3 171
8.15 Breakdown of idle times in FLOW3 172
8.16 Breakdown of execution times in FLOW3 174

List of Tables

5.1 Test programs used to study measurement overhead 71

6.1 Model components for each *PCN* program 126

6.2 Model components for each *PCN* implication 127

6.3 Fraction of *PCN* devoted to supporting execution profiling . . 132

8.1 The measurement overhead for FLOW 152

8.2 Breakdown of program activity during the execution of FLOW156

8.3 Breakdown of execution time for FLOW1 162

8.4 The breakdown of execution time for FLOW3 172

C.1 Failure probabilities of guard tests 186

ACKNOWLEDGMENTS

This dissertation is the culmination of many years work. I would not have been able to reach this point without the help and support of many people.

My advisor, Professor Miloš Ercegovac has been a great help throughout this process, doing everything possible to ensure that I completed this dissertation. I would also like to thank the other members of my committee: Professors Kirby Baker, Rajive Bagrodia, David Jefferson and Stephen Jacobsen for their time and effort. Other professors at UCLA have been helpful as well. In particular I want to thank Professor Gerald Estrin for getting me through the first couple of years and Professor Stott Parker for being generally supportive.

Much of the work in this dissertation was done while I was an Aerospace PhD fellow. Aerospace has provided a fertile environment, where I was given the opportunity to refine my ideas and develop as a scientist. Dr. Steve Crocker is responsible for hiring me at Aerospace and suggesting that I pursue a PhD at UCLA. Dr. Mel Cutler has looked over my career from the very first day. I am indebted to him for his patience and understanding. My coworkers at Aerospace have kept me reasonably sane through this process. In particular, Dr. Craig Lee and Dr. Joe Bannister were always ready for a beer.

Early recognition of my work came from Argonne National Laboratory. I would like to thank Dr. Ross Overbeek and Dr. Rusty Lusk for their enthusiastic support early on. Rick Stevens, also at Argonne, merits special mention. I want to thank him for his collaboration, for putting up with me while I was a visiting scientist at Argonne and for taking me to all of the best blues bars in Chicago.

I owe a great deal to Dr. Ian Foster and Dr. Steve Taylor. Their belief in my work has been unfailing and a great source of encouragement to me when there seemed to be no end in sight.

The research presented in this dissertation builds upon the work of others. The *PCN* programming system is central to my results. I want to thank the *PCN* development team, especially Sharon Brunett and Steve Tuecke for their help in integrating my work into the *PCN* system and answering my questions. Jim Hugunin is responsible for the graphics routines used

by *Gauge*, described in Chapter 7. The FLOW program used for the case study of Chapter 8 was written by Dong Lin of the California Institute of Technology. I thank him for his help in understanding the program and assistance in conducting the study itself.

Most importantly, I want to thank Phyllis, my wife. Without her I could not have gotten through this. Since we have met, she has not known me without a UCLA reg card or a beard; I suppose it would have been easier to shave.

ABSTRACT OF THE DISSERTATION

Tools and Techniques for Performance Measurement and Performance Improvement in Parallel Programs

by

Carl Kesselman
Doctor of Philosophy in Computer Science
University of California, Los Angeles, 1991
Professor Miloš D. Ercegovac, *Chair*

Programming a parallel computer is inherently more complex than programming a sequential computer. This complexity is due to: 1) additional design parameters, such as program partitioning and task to processor mapping, and 2) nonintuitive performance tradeoffs. Because of this complexity, it is inevitable that the initial design of a program will not utilize the available processing resources as effectively as possible. In this dissertation, we investigate methods for understanding and improving the performance of parallel programs. There are two aspects to this work: measurement and presentation. Our approach is to base performance measurement on extending execution profiling to parallel programs. Although profiling has long been recognized as a valuable tool in sequential programming, its value to parallel programs has not been extensively investigated. In this dissertation we show that there are compelling advantages to profiling over other forms of measurement in parallel programs. We develop a set of low overhead techniques for measuring an execution profile and integrate these techniques into a parallel programming system called *PCN*.

To present a parallel execution profile to a programmer, we have developed a performance visualization tool called *Gauge*. *Gauge* is unique in its

simple and concise method of data presentation and its use of interactive data analysis techniques to aid in the comprehension of multidimensional performance data. To demonstrate the effectiveness of our approach, we examine the performance of a large parallel application executing on 160 processors. Using the tools and techniques developed in this thesis, the execution time of the application is reduced by almost 20%.

Chapter 1

Introduction

Within the past few years, parallel computing has moved from the research laboratory into computing centers. As parallel hardware becomes more prevalent, the community of people who write parallel programs expands from specialists to a broad range of applications programmers. The task facing these programmers is to obtain the best use possible from a parallel computer, in a word: *performance.*

Performance plays an important role in parallel computing. Parallel processing is justified only when the solution to a problem is more cost effective on a parallel computer than on a uniprocessor. Since the main motivation for writing a parallel program is performance, a "working" program must not only be functionally correct, but it must also meet performance and/or efficiency requirements.

Programming a parallel computer is inherently more complex than programming a sequential computer. In addition to the normal algorithmic concerns, a parallel program must be split into tasks that can execute concurrently. These tasks must be mapped onto the available processors and the order of task execution on each processor determined. In addition, the costs associated with communicating data between processors should be minimized and the interaction of program actions that occur in parallel must be controlled.

Because of this complexity, it is inevitable that the initial design of a program will not utilize the available processing resources as effectively as possible. In light of this, we ask:

How can we understand and improve the performance of parallel programs?

Our answer to this question is to measure the performance of a parallel program and to use these measurements to guide the process of improving the program. Measurement is the key to understanding a parallel computation for without measurement, a programmer must guess which elements of a program are limiting performance. Even in sequential programs, a programmer's intuition about what the program is doing is often wrong [Ben82]. Moreover, in parallel programs, the issues of program partitioning, mapping, communications and synchronization make the situation even more difficult. With performance measurement, a systematic and directed approach to improving a parallel program's performance is possible. That is: 1) understand a program's behavior, 2) identify performance bottlenecks, 3) formulate corrective actions and 4) evaluate the improvement.

Measuring and improving the performance of parallel programs is hard for several reasons:

- The process of measuring parallel execution can change a program's behavior. Because of this, ensuring the validity of the performance data is difficult.
- Parallel programs can generate large amounts of performance data. The sheer volume of this data can make performance measurement impractical in real parallel programs.
- To be of practical value, we must be able to measure the behavior of programs on the available parallel hardware, but there is minimal hardware support for measurement on current parallel computers.
- Performance data from parallel programs is complex. This makes interpretation of the data difficult.

In light of these difficulties, we identify five specific issues as important to the use of measurement as a practical tool for performance improvement. These are:

- What are the appropriate measurements to make?

- How accurate do performance measurements need to be?
- What is the impact of measurement on program performance?
- What is the minimum hardware support needed for practical performance measurement tools?
- How can performance data be presented to a user in a manner that facilitates performance improvement?

We address these problems by proposing a new method for making performance measurements. Our focus is on developing techniques and tools that aid in the production of large scale parallel applications running on hundreds or thousands of processors.

1.1 Contributions

In this dissertation, we advocate the use of execution profiling as the fundamental means of performance measurement for parallel programs. Although the value of execution profiling has been downplayed in other works, we demonstrate that it is a valuable and useful tool with advantages over current methods of performance measurement.

The contributions of this dissertation are:

- Novel, low overhead, measurement techniques to obtain execution profiles from parallel programs. No hardware support beyond a microsecond timer is required.
- A concrete demonstration of these measurement techniques integrated into the implementation of a parallel programming system.
- A new approach for visualizing performance data obtained from parallel programs.
- A methodology for performance understanding and improvement based on parallel execution profiling.

The techniques developed in this dissertation have been used to construct a working system for profiling the execution of parallel programs. In Chapter 8, we show how our profiler is used to improve the performance of a computational fluid dynamics program. Additionally, our measurement tools have been used to improve other applications in domains such as climate modeling and genetics [Fos90]. In a real sense, our most important results are the improved parallel programs that have resulted from this work.

1.2 Organization of the Dissertation

In Chapter 2, we introduce the concept of performance and show how it applies to parallel programs. The problems associated with measuring the performance of parallel programs are discussed in detail and current approaches to performance measurement of parallel programs are presented.

Limitations of current methods of measurement are discussed in Chapter 3. We then introduce execution profiling as a means to overcomming these limitations.

Chapter 4 introduces *PCN*, a parallel programming system which we use to demonstrate how the techniques developed in this dissertation are applied in practice.

In Chapter 5, we present a framework for designing measurement techniques for execution profiling. Using this framework, we develop a measurement process specifically for profiling the execution of parallel *PCN* programs.

The implementation of a profiling system for *PCN* is given in Chapter 6. In this chapter, we describe how the measurement techniques developed in Chapter 5 are integrated into the implementation of *PCN*.

Chapter 7 focuses on the presentation of performance data to the user. We describe *Gauge*, the interactive performance visualization tool we have developed to present the data collected by our profiler.

A major claim of this dissertation is that execution profiles are effective in improving the performance of a parallel program. To support this claim, Chapter 8 shows how the tools and techniques developed in this dissertation are used to improve the performance of a nontrivial parallel application program: a simulation of wavy Taylor vortices using the three dimensional Navier-Stokes equations.

Chapter 9 summarizes the main results of this dissertation and proposes future research.

Chapter 2

Performance, Measurement and Parallel Programs

The term "performance" is often used without much thought as to what it really means. In this chapter, we will clarify what is meant by performance, how the performance of a system is determined and what performance means in the context of a parallel program. We conclude with a survey of the methods currently used to measure the performance of parallel programs.

2.1 The Performance of Systems

Performance is an abstract idea that can be applied to any system. We first introduce performance in this general context and then discuss how these concepts apply to parallel programs.

2.1.1 A definition of performance

Performance can be defined as the property of a system that makes that system valuable to its user [Fer79]. In other words, performance indicates how well a system does the task it was designed for. From this definition, performance provides a somewhat subjective means of system evaluation. What one user considers to be important might be of no value to another. If performance is to be used to evaluate the effectiveness of a system, or to compare the relative worth of two systems, a more quantitative assessment

is required.

This is achieved by representing the performance of a system by a set of one or more *performance indexes*. Each index has a value which is a measurable aspect of the system or its behavior. When more than one performance index is used, the performance of the system is described by:

$$P = \sum p_i w_i \tag{2.1}$$

where P is the performance of the system, p_i is the value of the i^{th} performance index and w_i indicates the relative importance of each performance index with respect to overall system performance. The w_i are defined such that $\sum w_i = 1$.

For the purposes of this dissertation, all performance indexes will be assumed to take on numeric values. This eliminates subjective indexes such as "programmability" from consideration unless they can be assigned a numeric value. In examining the characteristics of program execution, one is primarily interested in indexes that reflect how a system uses resources; for example, how long it takes to complete an operation or how many processors are being used. Clearly, indexes such as these can be assigned numeric values.

Selecting an appropriate set of performance indexes can be difficult. For example, the value of a computer system may be determined by how powerful it is. One measure of computing power is the speed at which the computer can perform basic operations. A number of different indexes can be used to represent this notion of performance, including:

- The number of floating point operations completed in second (FLOPS)
- The number of instructions executed per second (MIPS)
- The amount of time required to complete the execution of a benchmark programs (Whetstones, Dhrystones, etc)

This set of performance indexes could be further extended by considering whether the value of an index is determined by the system's best case behavior, its peak performance, or the behavior of the system over a range of activities, its sustained performance.

The users of a system must ultimately decide which indexes best represent their requirements. If multiple indexes are used, then the relative weighting between them must be determined as well. It is important to realize that no combination of performance indexes might fully capture the true value of the system. However, by selecting a specific set of performance indexes, we have a basis from which an objective system evaluation can be made. In this dissertation, the word *performance* is used to mean *the value of the weighted sum of performance indexes being used to represent a system's value*, as in Equation 2.1.

2.1.2 Methods of performance evaluation

The performance of a system is determined by a *performance evaluation study*. Such a study consists of two parts. First, we identify the performance indexes to be used to represent system performance and second, we determine the value of these indexes. There are two types of performance evaluation studies — performance analysis and performance measurement.

In *performance analysis*, the values of performance indexes are derived from system parameters using mathematical models. Many different tools for performance analysis exist. Of particular importance to parallel programs and parallel computers are queuing models and models based on Petri nets [MBC86]. Analytic methods are most useful when the system has not yet been built, or when the cost of performing experiments on the actual system would be prohibitive.

Alternatively, if the system in question has already been constructed, the values of performance indexes can be obtained directly by taking measurements as the system accomplishes its task. Obtaining the value of performance indexes in this manner is called *performance measurement*.

There is an important distinction between the values obtained through performance analysis and the values obtained though performance measurement. Performance analysis methods generally determine the value for a performance index via statistical abstractions. For example, the index values obtained might be representative of a class of system inputs whose size varies according to a Poisson distribution with a specific mean.

The values obtained through performance measurement differ in that they are based on one specific sequence of actions taken by the system in response to one specific set of inputs. Generalizations beyond this sequence of system

actions require that the input to the system used for performance measurement be representative of the range of inputs the system will have during actual use.

When performance evaluation is conducted via measurement, the system is instrumented and measurements are made while the system operates. At some point during the development of the system, a decision is made to enter a performance evaluation phase. An alternative is to make measurement a permanent part of the system. In this approach, called *continuous monitoring*, performance measurements are made during the entire lifetime of the system — from the time it is deployed until it is no longer used. Continuous monitoring allows the user to constantly evaluate the performance of a system.

Continuous monitoring has a number of advantages over "one shot" performance evaluation studies. An important aspect of continuous monitoring is that, by definition, the performance of the system is obtained from actual use and not specially constructed test cases. Performance data obtained from a continuously monitored system is more representative than performance data obtained during a performance evaluation study. When making performance measurements, one must be concerned with interfering with the operation of the system. Continuously monitored systems have the property of including the effect of measurement as part of the behavior of the system; so by definition, there is no interference caused by the measurement.

There are disadvantages to continuous monitoring, however. Continuous monitoring is not without cost; the performance of a continuously monitored system will always be less than optimal. To justify its use, the loss of performance incurred by continuous monitoring must be kept small. Consequently the performance data collected by continuous monitoring will be limited. It can not be as detailed or as comprehensive as data obtained though a carefully designed performance evaluation study. In spite of this limitation, we will show in Section 3.3 that continuous monitoring can be an important tool.

This dissertation studies techniques for performance measurement of parallel programs. Given a parallel program and a parallel computer, we wish to determine how well the program executes. Since we assume that the parallel program and parallel computer already exist, we limit our attention to performance measurement and continuous monitoring. Analytical performance evaluation techniques are not discussed further.

2.2 Performance and Parallel Programs

Performance is an important aspect of many programs, those intended for sequential execution as well as those intended for parallel execution. However, when one sits down to write a parallel program, it is with the expectation that the program will execute faster in parallel than not. Furthermore, there is an expectation that increasing the number of processors will result in a commensurate decrease in execution time. Even if it is not explicitly specified, performance is part of the design specification of a parallel program; a failure to perform well means the program is wrong. Thus *understanding* performance is an essential aspect to developing parallel programs.

To the programmer, performance data provides a means of identifying which elements of a program are degrading its performance. After performance bottlenecks are identified, alternative algorithms or data structures can be tried in order to alleviate the performance problem. When used to improve program performance, the presentation of performance data to the user is very important. In Chapter 7, we will introduce a tool designed to present performance data for a parallel program graphically, allowing the user to interactively explore the measurements in order to gain an understanding of the data.

Performance improvement by the programmer is not the only use for performance data. Other elements of the computing environment, such as the compiler or operating system, can also make use of performance data. For example, in [GT88], the execution time of a procedure is used to determine which procedures in a program should execute in parallel. Other resource allocation problems, such as partitioning and scheduling, can be aided by performance data as well. While the performance data obtained with the techniques developed in this dissertation can be used for this purpose, this application of performance data is not pursued further.

2.2.1 Characterizing parallel program execution

Many factors contribute to the runtime behavior of a parallel program. The particular characteristics of a program's execution are the result of the interaction of the program with components of the parallel computer on which it executes. In addition to the program itself, the compilers, operating system and underlying parallel hardware all have an impact on a program's

performance. The details of these interactions determine the values of the performance indexes for program execution.

The value of a performance index is determined by conducting a performance experiment during which measurement are collected. Before proceeding further, we define the parameters of a performance experiment. An experiment is defined by the tuple:

$$(\mathcal{A}, \mathcal{N}, \mathcal{P}, \mathcal{I})$$

where:

$\mathcal{A}$ is a parallel architecture. We direct our attention solely to multiple instruction stream, multiple data stream (MIMD) parallel computers. The two basic types of MIMD computers are shared memory multiprocessors and message passing multicomputers, both of which are considered in this dissertation. $\mathcal{A}$ parameterizes all aspects of the computer system as visible from the program, accounting for the effects on program behavior caused by compilers, the operating system, the processor node architecture and interprocessor communications mechanisms. For example, $\mathcal{A}$ encompasses the time required to complete basic operations such as floating point arithmetic as well as the time required to communicate data from one processor to another.

$\mathcal{N}$ is the number of processors allocated to the computation. In practice, it is common to execute a program with less than the total number of processors available on a parallel computer. If the total number of processors available on the computer is $\mathcal{N}_p$, then $1 \leq \mathcal{N} \leq \mathcal{N}_p$.

$\mathcal{P}$ is the program to be executed. $\mathcal{P}$ is assumed to be in source code form so as not to become involved in issues of the mapping of a high level language program onto an instruction set. The impact of translation of a program to an architecture specific form is included in the parameter $\mathcal{A}$.

An example will clarify the relationship between $\mathcal{A}$ and $\mathcal{P}$. In Section 5.6.1, the execution time of a program is determined by expressing it in terms of a sequence of abstract operations and counting the number of times each operation occurs in the program. The abstract operations that are needed are a characteristic of the program. The parameter $\mathcal{P}$

is the number of times each abstract operation is used in the program. The parameter $\mathcal{A}$ specifies the amount of time required to perform each type of abstract operation, accounting for factors such as the mapping of the operation onto the computer's instruction set architecture, the operating system overhead (if paging or I/O are required), and the speed at which the computer executes instructions.

$\mathcal{I}$ is the input data to the parallel program. Measured values of performance indexes are meaningful only with respect to the input data used for the experiment. The input data used for a performance experiment should represent the range of inputs the program will process in actual use.

Our performance experiments are developed from the perspective of the performance of the program. That is, given $\mathcal{N}$ processors on computer $\mathcal{A}$ and input data $\mathcal{I}$, what is the performance of program $\mathcal{P}$?

2.2.2 Performance indexes for parallel programs

As discussed earlier, the performance of a program is represented by a set of performance indexes. We now examine performance indexes that are of particular interest to the execution of parallel programs. Two basic performance indexes for program execution [Fer79] are:

Execution Time: The length of time, t, required for the program to complete its task.

Cost: The resources required for the program to complete its task. These could be a number of cost metrics, such as the actual execution cost in dollars, the number of processors used or the amount of memory required.

Execution time is an index that can be applied to any program — parallel or sequential. However, having spent time and money developing a program capable of being executed in parallel, we would like to have an index that reflects scalability: how much faster a program runs as $\mathcal{N}$ increases. This requires a performance index that combines the execution time with the cost of the computation in terms of the number of processors used, $\mathcal{N}$. There are a number of performance indexes with these characteristics [KBC+74].

One such index is *speedup*. We find that there are two commonly used definitions for speedup. In one, the speedup of a program $\mathcal{P}$ executing on $\mathcal{N}$ processors, $S_{\mathcal{P}}(\mathcal{N})$, is defined as:

$$S_{\mathcal{P}}(\mathcal{N}) = \frac{t_1}{t_{\mathcal{N}}} \tag{2.2}$$

the ratio of the execution time for the program on one processor to the execution time of the program when executed on $\mathcal{N}$ processors. To simplify the notation, we drop dependence on $\mathcal{P}$ in the sequel. The problem with this definition is that it only indicates how well the program executes in parallel rather than how well the program executes. A program with large speedup can have an execution time greater than a better program with lower speedup. A more honest evaluation is obtained by defining speedup as:

$$S(\mathcal{N}) = \frac{t_s}{t_{\mathcal{N}}}$$

where t_s is the execution time for the best sequential program that performs the same task.

Recently there has been some argument as to whether speedup is a good performance index. An alternative metric called *scaled speedup* has been proposed in [GMB88]. In calculating scaled speedup, the size of the problem being solved is scaled by the number of processors being used. For example, in a grid problem, the number of grid points on each processor is kept constant regardless of how many processors are being used. Scaled speedup is then defined by:

$$S_s(n) = 1 + (1 - \frac{t_n}{t_s})\mathcal{N}$$

Scaled speedup is motivated by the fact that for many types of problems, a parallel computer is not used to solve a specific problem faster, but in order to solve a larger problem. Many scientific applications, such as computational fluid dynamics, fall into this category.

An index related to speedup is *efficiency*. Efficiency, defined as:

$$E = \frac{t_1}{\mathcal{N} t_{\mathcal{N}}}$$

indicates how effectively a parallel program used a parallel computer. Note that efficiency can be related to Equation 2.2 by:

$$E(\mathcal{N}) = \frac{S(\mathcal{N})}{\mathcal{N}}$$

.

An index similar to efficiency is *processor utilization.* Utilization is the fraction of a program's execution time spent working. Utilization differs from efficiency in that utilization does not discount additional work introduced into a computation in order to parallelize it.

Average parallelism [EZL89] provides another characterization of the performance of a parallel program. Average parallelism is defined by:

$$P = S(\infty)$$

This index is a good representation of program performance in that a good lower bound for $S(\mathcal{N})$ for any $\mathcal{N}$ can be derived from P.

The performance indexes of execution time, speedup and efficiency view program execution at a high level of abstraction, the program as a whole. Not surprisingly, these indexes can only determine the existence of a performance problem in the program as a whole. At this level, it is not possible to identify where in a program performance bottlenecks occur. To accomplish this, it is necessary to use a performance index that represents program execution at the level of abstraction where the bottlenecks exist. For example, if a specific procedure is responsible for a performance bottleneck, a performance index that abstracts program execution as a set of procedures is appropriate.

For the purpose of measurement, program execution can be viewed as a hierarchy of abstractions. At the top level is the execution of the program as a whole. At the bottom are the activities of the hardware that make up the computer. In traversing the hierarchy, program execution is decomposed into smaller and smaller components. Figure 2.1 shows an example of one set of abstractions; alternative hierarchies are given in [BGK89, MCH+90, McK88]. Corresponding to the execution hierarchy is a hierarchy of performance indexes which, in turn, structures a hierarchy of possible measurements.

It is important to select the appropriate level at which to represent program performance. As the level of abstraction decreases, the amount of detail revealed by the performance index increases. However, the number

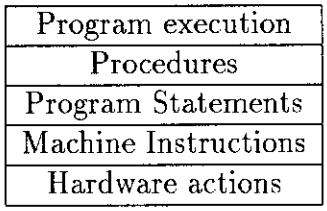

Figure 2.1: An abstraction hierarchy for parallel program execution

of measurements that must be made to evaluate the performance index also increases. McKerrow estimates that for each level of the measurement hierarchy descended, one order of magnitude more data is generated [McK88].

While speedup is often the performance index of choice when describing a parallel program, we do not consider it further. To calculate speedup or efficiency, one must conduct at least two performance experiments. Instead, we prefer to focus on execution time and processor utilization, both of which can be determined in single performance experiment. In Chapter 5, we develop techniques for determining the execution time of a program at the program statement level of the execution hierarchy. In Chapter 8 we show how this index combined with processor utilization is sufficient to improve the performance of a parallel program.

2.2.3 Measurement models

We now turn our attention to the actual process of measurement and how measurements are related to a performance index. The relationship between measurement and performance indexes is established by defining a *measurement model.* A measurement model defines:

- The aspects of program execution that are measurable.
- The data associated with a measurement.
- How measurements are related to the value of performance indexes.

Several different measurement models have been proposed for program execution. In [Fer79], Ferrari associates a set of states, Σ, with a program.

Each state corresponds to the program performing a specific activity. For example, a state might be associated with the execution of a procedure. Another state could be defined to represent a program waiting for the availability of some system resource. Program execution is modeled by 1) the sequence of states that a program is in and 2) the amount of time spent in each state. An *event* is defined as the program being in a particular state for an amount of time. A measurement can be made whenever an event occurs. The value of a performance index is obtained from the measurement sequence:

$$M_d = (\sigma_1, t_1), (\sigma_2, t_2) \cdots (\sigma_m, t_m)$$

where $\sigma_i \in \Sigma$ is the i^{th} state of the program and t_i is the amount of time the program spends in σ_i. Measurements of this type are said to be generated by a *state/duration* measurement model.

By decomposing program execution into alternative sets of states, different performance indexes can be obtained. However, a performance index does not determine a unique decomposition; a performance index can be determined from several different decompositions. Conversely, it is also possible to compute the value of several performance indexes from a single decomposition.

The state duration measurement model can be directly applied to a parallel program in the following way. Let $\sigma_i^{\mathcal{N}} \in \Sigma^{\mathcal{N}}$ where $\Sigma^{\mathcal{N}}$ is the $\mathcal{N}$ dimensional cartesian product of Σ with itself. Each $\sigma_I^{\mathcal{N}}$ is the instantaneous state of the parallel computation. The measurement sequence becomes:

$$M_d^{\mathcal{N}} = (\sigma_1^{\mathcal{N}}, t_1^{\mathcal{N}}), (\sigma_2^{\mathcal{N}}, t_2^{\mathcal{N}}) \cdots (\sigma_m^{\mathcal{N}}, t_m^{\mathcal{N}})$$

In this model, a computation enters a new state whenever the state on any processor changes.

This model has a practical limitation in that detecting an event requires determining the state of the entire parallel computation. Since the state of a computation on a parallel computer depends on the status of each processor within the computer system, it is distributed throughout the computer. Although it is possible to obtain the global state of a program [CL85], in general it is difficult.

A more practical measurement model can be defined by restricting the types of events that can be detected within the model. We extend the sequential model to parallel programs by recording the events from each processor

as a separate sequence. Each event in this model records the local state on a processor and the amount of time spent in that state. The measurement is now a set of sequences:

$$M_d^{\mathcal{N}} = \begin{cases} (\sigma_1^1, t_1^1), (\sigma_2^1, t_2^1) \cdots (\sigma_{m_1}^1, t_{m_1}^1) \\ \vdots \\ (\sigma_1^{\mathcal{N}}, t_1^{\mathcal{N}}), (\sigma_2^{\mathcal{N}}, t_2^{\mathcal{N}}) \cdots (\sigma_{m_{\mathcal{N}}}^{\mathcal{N}}, t_{m_{\mathcal{N}}}^{\mathcal{N}}) \end{cases}$$

The primary drawback of this model is that the time relationship between states on different processors is implicit rather than explicit; states are not placed absolutely in time. If the measurement sequence does not account for every action that the program may take, relating states between processors will not be possible.

Rather than observing program activity as a sequence of state/time duration pairs, Svobodova proposed modeling program activity by observing the times at which a state change took place [Svo76]. We will call this a *state/timestamp* model. In this model a program is decomposed as above. However, events are associated with changes in state rather than with the states themselves. The data associated with an event is the state transition and the time at which the transition takes place.

Measurements consist of the sequence:

$$M_t = \sigma_{0,1}(\tau_1), \sigma_{1,2}(\tau_2) \cdots \sigma_{m-1,m}(\tau_m)$$

where $\sigma_{i,i+1}$ is the state transition from state σ_i to state σ_{i+1} and τ_i is the time at which the transition took place. With the appropriate set of states, measurements from a state/timestamp model can be converted into a state/duration form. In practice, events are associated only with entering or leaving a state and not the transition between two states.

This model can be easily generalized to parallel execution by augmenting the event data with the identity of the processor on which the state change took place. This requires that a timestamp be available globally on the parallel computer. Fortunately, this problem is easier to solve than determining global state. Some computers such as the Sequent Symmetry [Seq99] maintain a single clock accessible by every processor. On computers without this facility, algorithms for synchronizing local clocks exist [Fre89]. Most current performance measurement tools for parallel programs are based on the state transition/timestamp.

2.2.4 Deriving performance indexes from performance measurements

When a performance index is computed from a set of measurements, that index is called a *derived index.* Svobodova has identified four different ways in which the value of a performance index can be derived from a measurement model [Svo76]. These are:

- An event trace: The sequential record of each event that occurred during execution. An important state decomposition is to assign a different state to each statement within a program. If every event is measured, the resulting data is called a *program trace.*
- Relative activity: The ratio of resource utilization for a specific activity to the total resource utilization of the program.
- Event frequency: The total number of times an event occurred during the experiment.
- Distribution of activity intervals: The distribution of resources used by a specific activity.

In addition to a program trace, we identify one other type of data collection: an *execution profile.* An execution profile is obtained by defining a different program state for each subprogram (or program statement) and the data collected is limited to relative activity and event frequency. Knuth [Knu71] originally defined a execution profile to consist solely of frequency data. However, current usage of the term generally includes time data as well [Ben82]. In Section 3.3, we will show how execution profiles can be applied to parallel program execution.

As an example of how different decompositions determine performance indexes, assume that we wish to determine processor utilization. This index is given by:

$$\text{Utilization} = \frac{\text{Time the processor is busy}}{\text{Time the processor is busy} + \text{Time the process is waiting}}$$

One possible decomposition splits program execution into two states, one for processor busy and one for processor idle. Utilization is determined by the

relative activity of the processor busy state. An alternative decomposition is to use a program trace. Typically, specific statements within a program, such as lock requests or message reads, cause a processor to become idle. The execution for the program statements can be summed, separating the statements that cause processor idle time from all the others. Processor utilization can then be calculated as in the first decomposition. However, the many additional performance indexes can be calculated from a program trace.

2.3 Issues in Performance Measurement of Parallel Systems

In performance measurement of parallel program execution there are three concerns: 1) deciding the measurements to make, 2) dealing with the interaction between the measurement process and the program being measured and 3) managing the volume of measurement data. In this section, we examine how measurements of parallel programs are made and the aspects of parallel execution that make the measurement process difficult.

The process of performance measurement can be broken into three steps:

- Instrumentation. Measurements are made by instrumenting a program (or underlying system) with *sensors*. The type and placement of a sensor is determined from the measurement model. Generally, a different sensor is associated with every event in the measurement model. When an event occurs, its sensor is given the opportunity to observe the data associated with the measurement.

- Data collection and storage. During program execution, the values of sensor observations must be collected and stored. The collected data can be simply stored unchanged, stored along with additional information such as a timestamp or some data reduction can take place and the result stored.

- Data analysis and formatting. Collected data is analyzed and processed. The value of derived performance indexes are determined. The data is formatted and presented to the programmer in a comprehensible form.

In this section, we only address the first two steps of the measurement process: instrumentation and data collection. We present a more detailed discussion of the problems associated with instrumentation and data collection. The problems associated with data analysis and formating are discussed in Chapter 7.

2.3.1 Instrumentation and the implementation of sensors

During a performance experiment, data is collected from an executing program by instrumenting the system with *sensors*, objects responsible for actually obtaining the measurement data. The functionality of a sensor can be broken into three parts [GS85]:

1. **Detection.** A sensor must determine when an event in the measurement model has occurred.

2. **Isolation and Filtering.** Not all events in the measurement model need to be measured. The sensor can apply various criteria to determine if measurement data will be collected for an event.

3. **Notification.** An identifier for the event with its associated data is transferred to the performance measurement system for collection and storage.

When a parallel program is instrumented with sensors, a tradeoff is made between the type and accuracy of the data obtained, the impact the sensor has on program execution and the ease of use and flexibility of the sensor. Three implementation options are possible: sensors can be implemented in hardware, in software, or as some combination of the two.

The advantage of hardware sensors is that very accurate measurements can be made with little or no impact on the program being executed. In addition, if the data required pertains to the operation of subsystems in the parallel computer itself, a hardware sensor may be the only way to capture the information. For example, a hardware sensor is the only practical approach to accurately measure the impact of contention for memory modules in a shared memory parallel computers. However, it can be very difficult to capture events defined by actions within the program, such as the execution

of a specific program statement, with a hardware sensor. More significantly, designing and installing hardware sensors can be both costly and difficult. Finally, by its very nature, hardware solutions to sensor implementation limit the flexibility of the measurement system.

An example of hardware measurement facilities can be found on the Cray X-MP. The Cray hardware is instrumented with 32 counters that collect performance data [CRA]. The counters record the number of floating point operations, the number of memory conflicts and instruction use histograms. Access to these values is provided to the user program via a system call. Because of the overhead associated with a system call, frequent examination of these counters slows down program execution. Details of the overhead associated with examining the counters can be found in [MLR90].

The obvious alternative to hardware sensors is to implement sensors completely in software. This approach is attractive because of its flexibility and the relative ease of construction and installation. The primary drawback of software sensors is that their execution can interfere with the execution of the program that is being measured. As discussed in the next section, this can have dire consequences in a parallel program.

The approach to sensor implementation in the Parasite system [AG88] is an interesting example of a software based measurement system. Parasite was designed to execute on the Encore Multimax, a shared memory multiprocessor. When a parallel program executes, Parasite creates a monitor process which attaches itself to the program by mapping a shared memory segment from the program into the monitor. The monitor can instrument the parallel program by setting a breakpoint, much like a debugger. When the breakpoint is reached, the necessary data is passed to the monitor through shared memory. The program then continues execution while the monitor finishes processing and storing the collected data. Because the monitor runs on a "spare" processor using shared memory, the disturbance to program behavior is minimized.

The final alternative in sensor design is to implement part of the sensor in hardware and the remainder in software. Although there are many ways in which the measurement task can be divided, flexibility is maximized and overhead minimized by performing event detection in software and the remaining sensor functionality in hardware.

An example of a hybrid approach is found in HPERMON [MAA+89], a performance measurement system for the Intel iPSC/2 (a multicomputer based

on the Intel 80386 microprocessor). The HPERMON hardware is responsible for capturing events, event processing and analysis and storage of events onto a disk storage device. Each HPERMON unit contains two microprocessors and 512 Kbytes of memory and can service up to 16 processing nodes. Events are generated in software by issuing an I/O write request from a processing node. The I/O request with four bits of data is passed over a special global bus to the HPERMON event capture hardware. In the hardware, a timestamp is generated for the event and the resulting data is stored. There is a cost associated with event detection; the I/O request required to signal an event requires at least 12 clock cycles on the node processor to complete.

2.3.2 The placement of instrumentation in a parallel program

The location of instrumentation in a program is determined by the measurement model from which the desired performance indexes are to be obtained. With the measurement model defined, instrumentation can be inserted into a program manually or automatically. With manual placement, the programmer is responsible for identifying the appropriate location in the program to place the instrumentation. Automatic placement is typically performed in the compiler or thorough the use of instrumented library routines. If the compiler is responsible for placement, the location of the instrumentation is determined by an analysis of the program code.

In sequential profiling, instrumentation is typically placed on routine entry and exit and basic block boundaries. However, the activities of interest in a parallel program, such as sending a message or obtaining a memory lock, can occur at levels below the basic block. Consequently, instrumenting parallel execution can require more detail than instrumenting sequential execution. This increases the measurement overhead and the amount of performance data generated during performance measurement.

2.3.3 The probe effect

A fundamental issue in measuring a parallel program is to design a sensor that can observe a parallel program without altering the program's behavior. The execution of a parallel program depends not only on the actions

being performed, but also on the spatial and temporal relationships between concurrently executing program components. Unlike sequential programs, changes in the time needed to perform an operation can have drastic effects in the behavior of the program. To make matters worse, the impact of a disturbance is not necessarily local. Interference with program behavior at one location can impact the behavior of other components of the program [Col79].

The changes in the behavior of a program when it is instrumented are called the *probe effect* [Gai86]. The probe effect was first discussed in the context of debugging parallel programs. It was shown that as instrumentation overhead in a program changed, different errors present in the program would manifest themselves. In some cases, the errors would not appear at all. As with debugging, the probe effect is of concern with performance measurement. If the behavior of a parallel program is altered by the process of measurement, the measurements are of questionable value.

Three approaches to managing the impact of the probe effect have been identified [MH89]. These are:

- **Fast sensors:** Minimize the impact of the probe effect by making sensor operations as fast as possible. However, there is no guarantee that the program's behavior has not been altered in some way. Unfortunately, this seems to be the technique most commonly used in performance measurement.

- **Leave the sensors in:** Sensors are not added and removed as needed. Rather, a set of sensors is inserted into the program when it is created and never removed. This approach results in a continuously monitored program. Since the behavior of the system is defined with the sensors in place there is no probe effect. Obviously, this approach will result in some degradation in program performance. Therefore, the overhead of the sensors must be kept low so as not to outweigh the benefits of avoiding the probe effect.

- **Logical time:** Sometimes the impact of sensor overhead can be compensated for by viewing program activity on a virtual timeline rather than the actual timeline. For example, in PIE [LSV+89], measurement overhead is compensated for by adjusting the values of all of the event timestamps.

2.3.4 Volume of performance data

Measurements of a parallel program can generate large amounts of performance data. Each processor in a parallel process is a source of performance data. Thus, a one thousand processor computer has the potential to generate a thousand times more performance data than a uniprocessor. In addition, applications that warrant parallel execution tend to be large problems requiring hours or days of computer time. The longer a program runs, the more performance data it can generate.

As the amount of data grows, it becomes increasingly difficult to extract the data from the computer and store it. Once a measurement is made, the resulting performance data must be stored somewhere. This can be in a processor's memory, off loaded onto a mass storage system or displayed immediately. Storing hundreds of megabytes of performance data in processor memory is not feasible. If the data is transferred from the node onto a disk or displayed, valuable I/O and communications bandwidth is consumed. Both of these factors complicate the measurement process.

2.4 Current Performance Measurement Systems for Parallel Programs

In this section, we review several different performance measurement systems for parallel computers. All the systems discussed in this section rely on event traces as the primary means of data collection. Although this survey is not exhaustive, it is representative of the current state of the art in performance measurement of parallel programs.

2.4.1 Generating event traces on the BBN TC2000

The TC2000 is a non-uniform access shared memory multicomputer developed by BBN Advanced Computers. It can support up to 528 processors interconnected by an eight-by-eight butterfly switch. BBN provides two software based facilities for generating event traces, or as BBN refers to them, event logs [BBN90]. Performance data on the TC2000 is collected by instrumenting a program with calls to the ELOG event logging library. Each log entry is 16 bytes long and contains a timestamp, an event type, a processor

number and an event value. As the program executes, logged events are stored in a per node buffer located in the program's virtual address space. The amount of execution time overhead caused by generating event traces depends on the program and the number of calls to the logging library. However, overheads in the range of 10% or higher are typical.

The second facility is a low level event logging mechanism in the operation system kernel. An system call is used to store event data into a buffer the kernel maintains. Access to the buffer is provided through a separate utility program. In addition to user-defined events, the kernel itself generates a set of events to monitor such things as locking, page faults and I/O activity.

2.4.2 Collecting performance data on the NCUBE multiprocessor

The **NCUBE** is a distributed memory computer with up to 1024 nodes connected in a hypercube topology. Each node has 512 Kbytes of memory; there is no virtual memory. Simplex [KCHC89] is a nodal operating system for the **NCUBE** that supports performance measurement. Two types of performance data are collected: summary statistics are collected directly by the nodal operating system and event traces can be generated by placing calls to event logging library.

Event traces are generated by placing calls to an event generation routine in an application program. The event collection implementation capabilities are much like those on the TC2000 [Cou88]. However, because of the limited memory on an **NCUBE** node, much less buffer space is available. To overcome this problem measurements are stored into a circular buffer; thus new entries overwrite the oldest entries. Simplex has a mechanism to transfer a range of buffer entries to the system host.

Simplex automatically collects a large number of summary statistics such as the number of messages read and sent, the current number of processes on a node, etc. The statistics provide information on a per node basis, a per process basis and a per communication channel basis. The value of individual statistics can be collected by monitor program executing on a workstation with network access to the NCUBE. To minimize the impact of collecting the statistics, a program on the NCUBE can be run in *burst mode*. In burst mode, each node runs for a time slice, then program execution on every node

is suspended and statistics collected.

2.4.3 Performance measurement in IPS-2

IPS-2 is a performance measurement system developed for performance measurement of parallel and distributed programs [MCH+90]. It currently runs only on the Sequent Symmetry, a shared memory parallel computer with from 2 to 28 processors. In IPS-2, performance data is collected in the form of event traces. Sensors to generate the trace are automatically placed into a program at the entry and exit points of a procedure during program compilation.

An interesting aspect of event generation in IPS-2 is that a state/duration measurement model is used to reduce the number of events that must be collected. For example, measuring the acquisition of a lock on a shared memory computer would require two timestamped events: one to capture when the request for the lock was initiated and one when the lock was granted. If the exact time of the lock request is not required, then the two timestamped events can be replaced by a single state/duration event.

2.5 Summary

In this chapter, we discussed the concept of performance and how it is represented by performance indexes. We then extended these concepts to parallel computers, presenting a framework for conducting performance measurement experiments on parallel programs.

In reviewing the current approaches to performance measurement, we find that all the tools generate an execution trace. In the next chapter, we will discuss the problems with this form of measurement and propose an alternative approach to performance measurement.

Chapter 3

A New Approach to Performance Measurement

The performance measurement systems surveyed in the previous chapter all suffer from a common set of problems: their runtime overhead can be 7% or higher and they can generate arbitrarily large amounts of data. In this chapter, we present an approach to performance measurement that overcomes these limitations. Based on execution profiling, our approach has the following advantages:

- The runtime overhead is less than 3%.
- The space overhead is less than 10%.
- There is no probe effect.
- A constant amount of data is collected regardless of how long a program executes.
- The measurement technique scales to large numbers of processors.
- It can be used on any program with any input data without programmer intervention.

We start with this chapter with a discussion of the problems with current approaches to performance measurement. We then introduce our approach and show how it overcomes these problems.

3.1 Limitations of Current Performance Measurement Systems

The measurement systems discussed in Section 2.4 rely on event tracing as their primary means of data collection with the events defined on a per processor basis using the state/timestamp measurement model. The major advantage to this approach is the level of detail that can be obtained. Assuming a time source with adequate precision, a complete accounting of program activity is possible. However, trace based performance measurement has some significant limitations. These are: 1) nontrivial execution time overhead of the sensors, 2) controlling the volume of data generated and 3) the level of user interaction required for use.

The runtime overhead of generating an event trace is nonnegligible. The source of the overhead is in the work that a sensor must do to detect an event, collect the event data, allocate a storage spot for the data and then actually copy the data. At a minimum, the data copied includes an event type and a timestamp. Additional data may also be required. Each element of data associated with an event requires at least one memory operation to collect and store. In addition, a slot in the log buffer must be allocated, causing additional overhead. Although the total runtime overhead will greatly depend on the program being instrumented and the amount and placement of instrumentation, experience indicates that an overhead of over 7% is not exceptional [Cou].

More significantly, the collection of event traces can require a huge amount of storage space. The longer a program runs, the greater the amount of data that must be collected. Typically, the designer of the performance experiment is responsible for keeping the amount of data collected reasonable. There are three options available to the programmer: measuring at a higher level of abstraction, selective tracing, and limiting the execution time of the program. Each has disadvantages.

Reducing the number of states in the measurement model reduces the amount of performance data generated. One way to accomplish this is to use a performance index higher in the measurement hierarchy described in Section 2.2.2. However, this defeats one of the great advantages of tracing — detail.

Alternatively, one could collect performance data from a subset of the

events in the measurement model. This requires the programmer to guess which parts of the program are causing the performance problem and then limit the measurement to those parts. The question then arises: Where does the programmer get *a priori* knowledge about the performance of the program? Experience has shown that programmers are bad guessers when it comes to making these types of assessments [Ben82].

The final option is to limit the execution time of the program by executing it on smaller input data set or test case. It is then the responsibility of the programmer to design a test input that is small enough to collect data from, while still accurately reflecting the program's behavior when it executes using actual input data.

Recognizing these problems, most trace based performance measurement systems require the programmer to manually place sensors into the program. Some measurement systems, IPS-2 for example, places a limited number of sensors automatically. Usually these sensors capture procedure entry and exit. However, in programs in which parallelism is present within a procedure, such as parallel loop execution, this level of measurement not detailed enough and programmer intervention is still required.

3.2 Our Approach to Performance Measurement

We have developed a performance measurement technique specifically to overcome the limitations of measuring parallel programs with event traces. Our approach is based on a parallel generalization of an execution profile. The profile data is collected by simple software sensors that generate a fixed amount of data, regardless of how long the program runs. By leveraging compile time information, much of the data in a profile is collected by simple counters; the other data collection relies on lapse time computed from a microsecond clock.

Recall that an execution profile records the number of times an action within a program is performed and the amount of time spent performing that action. Because a profile collects totals rather than individual events, the amount of storage needed for the measurement data is fixed and determined by only by the text of the program. Regardless of how long a program runs,

the size of its execution profile remains constant. This has two ramifications:

- Performance data can be collected from the entire program. We do not need to guess time what parts of a program are important.
- The input data normally processed by the program can be used as input for the performance experiment as well.

In measuring the overhead of a range of parallel applications, we have found that the storage requirements for profiling do not exceed 10% of a program's code size.

The sensors are placed into a program automatically based on an analysis of the program text. Because the storage overhead of an execution profile is low, the placement of the sensors can detail operations within a program that are relevant to parallel execution, not just procedure calls. The fixed storage requirements and automatic placement of instrumentation in the program eliminates any need for the programmer to direct the process of performance measurement. Because profiling can be applied to any program with any input, it is a more general approach to performance measurement than execution profiling.

The sensors required for execution profiling are very simple. As a result, they are implemented in software and have little runtime overhead. Our experiments show that using our techniques described in Chapter 5, a parallel program can be profiled with an execution time at of at most 3%. Often it is an order of magnitude less. An advantage of software sensors is portability. Our measurement techniques are currently in use on four completely different parallel architectures including both multicomputers and multiprocessors.

Because the runtime and storage overhead of our instrumentation is so low, we make the sensors an integral part of the parallel programming environment. *Profiling is always performed.* By defining the behavior of a parallel program with sensors in place, we are freed from consideration of the probe effect.

Of course, profiling is not without its disadvantages. The primary drawback is that there is less detail available in an execution profile than provided in an event trace. Information about specific instances of procedures or statements, temporal information, and most information about sequences of events is not available. However, our experience has shown that a great

deal of insight into parallel program execution can be obtained without temporal information. Furthermore, by exploiting causal information implicit in the structure of the program, some causality of actions can be inferred from a profile. Still more temporal data can be recovered by generating a sequence of profiles through the use of snapshots. Finally, if an execution profile does not provide enough information to identify and correct a performance bottleneck, it can be used as a tool in designing a more detailed performance experiment utilizing execution tracing.

A performance measurement system based on low overhead measurements, execution profiling and continuous measurement provides a basic set of performance measurements that should be provided in *any* parallel programming system. With its global view of program execution, execution profiling answers the first set of questions that must be answered before more detailed analysis can proceed. It is a general technique in that it can be applied to any program without concern to program size or execution time. The program can be instrumented for profiling automatically, requiring no interaction by the programmer. Consequently, no *a priori* knowledge about problem areas in the program is required to generate a profile.

3.3 Profiling Parallel Programs

Of all the current performance measurement systems for parallel programs, none directly support execution profiling. While it is true that program profiles can be extracted from procedure or statement level program traces, all the inherent limitations of event tracing remain. To fully exploit its advantages, execution profiling should be directly supported as part of a parallel programming environment.

Profiling is considered to be essential for sequential programs [Ben82, Knu71] and tools for profiling sequential programs have been available for some time. However, these tools are lacking for parallel programs. To our knowledge, profiling techniques specifically designed for parallel program have not been developed. In fact, some authors have gone so far as to claim that profiling is not important for parallel programs [Mal89]. It is our belief that nothing could be further from the truth.

The argument made against profiling asserts that sequencing information is required to properly identify performance bottlenecks. For example,

consider the call graph fragment in Figure 3.1. Each block represents a subprogram with its execution time indicated along side. Assume that the left and right branches of the graph are executing on different processors.

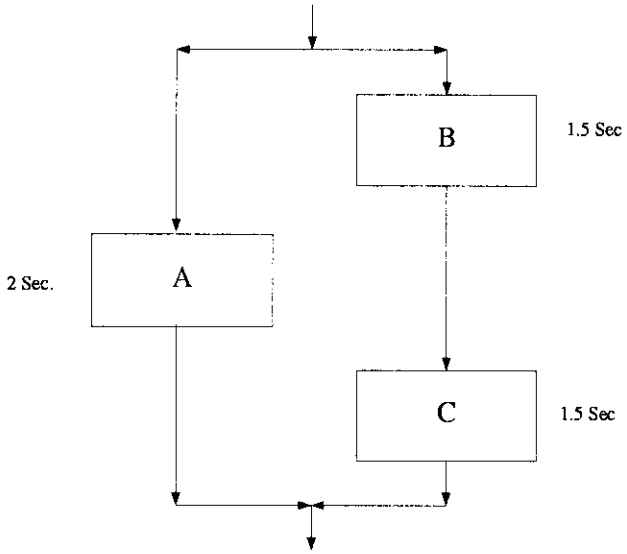

Figure 3.1: Execution graph of a program to be profiled

The existence of a performance bottleneck can be identified because the execution time of three seconds on two processors results in a speedup of 1.66 when the best case is a speedup of 2. Consulting the sequential program profile would lead one to focus on the execution of procedure A with 2 seconds of execution time. The profile does not indicate that in fact the performance bottleneck is the result of the combined execution of B and C.

The preceding analysis is limited by its attempt to apply a sequential execution profile to a parallel program. In order to understand parallel program execution, sequential execution profiles must be generalized. We define a parallel execution profile by:

Definition 1 *A parallel execution profile of a program is obtained by defining a state/duration measurement model extended for parallel programs as defined in Section 2.2.3. The program is decomposed into components such as procedures or statements. For each component c, three states are defined:*

σ_c^e *Component is executing user code.*

σ_c^i *Component is waiting (idle).*

σ_c^c *Component is communicating with another processor.*

If measurements are limited to the total number of times a program is in each state and the total time spent in each state, the resulting data is a parallel execution profile.

Let us return to Figure 3.1 and investigate what happens when we collect a parallel profile. First, we observe that the call graph of Figure 3.1 is incomplete. There must be some form of synchronization to coordinate the termination of the left branch executing A and the right branch executing B and C. We augment the original graph by placing a *barrier* procedure to perform this synchronization in processor one. Figure 3.2 shows the graph with the barrier added.

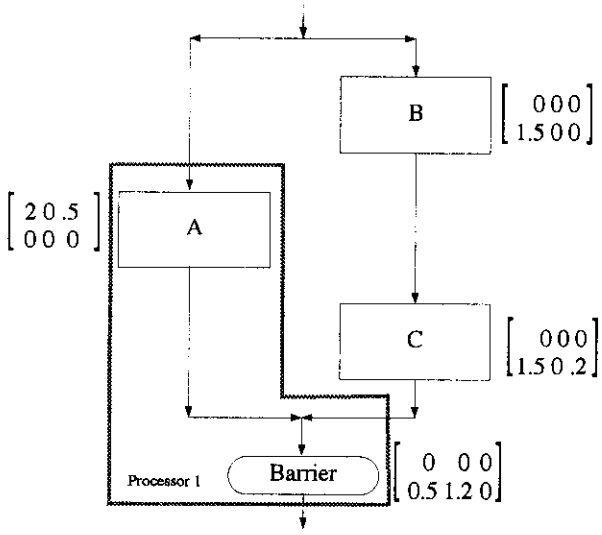

Figure 3.2: A more complete parallel profile

We have annotated each procedure with parallel profile data. A matrix formulation is used where for a procedure c, the kth row of the matrix is:

$$(\sigma_c^e, \sigma_c^i, \sigma_c^c)$$

where the elements are the total time spent in the states of Definition 1 on processor k.

The total time spend in this graph is 3.7 seconds. From the profile data, we can compute the processor utilization. On processor one, we spend 2.0 seconds in A and 1.0 seconds idle and 0.5 seconds executing the barrier. Comparing the computation time to the total execution time gives us a utilization of 68%. On processor two, we spend a total of 3.0 seconds executing B and C. In addition, C spends 0.2 seconds communicating with the barrier. This results in a utilization of 81% in processor 2. The difference in utilization between the two processors indicates a performance bottleneck on processor two.

To find the cause of the bottleneck, we consult the graph of Figure 3.2 and the execution times of the programs. The bottleneck manifests itself as idle time in the barrier. Execution of the barrier requires that A and C complete and C requires B. From this information we conclude that the difference in execution times between procedure A and procedures B and C is responsible for the idle time at the barrier and consequently degrades the performance. The performance of the program fragment can be improved by decreasing the execution time of B or C or splitting B or C into smallers tasks and allocating one second's worth of computation to processor one.

The purpose of this simple example is to demonstrate that it is possible to understand and improve the performance of a parallel program using an execution profile. A more convincing argument is presented in Chapter 8 where using a method similar to that in the preceding paragraph, we obtain a 19% performance improvement on a 5000 line program executing on 160 processors.

The values in the matrixes in Figure 3.2 constitute a parallel execution profile. While the profile in this example is based on a decomposition of program execution into procedures, other measurement models are allowed by Definition 1. In Chapter 5, we discuss the issues that arise in selecting a measurement model and develop a set of measurement techniques to obtain the data needed for a profile.

3.4 Summary

In this chapter, we have introduced a new approach to measuring the performance of a parallel program: parallel execution profiles. This approach eliminates the issue of the probe effect while limiting the amount of data to be collected. Almost all current performance measurement systems use some form of event tracing. While there will always be a place for performance measurement that is detailed and accurate, this should not be the first line of defense when developing parallel programs. Rather, execution profiling, which is generally applicable without programmer intervention, should form the basis of performance measurement.

We now turn our attention to ways in which parallel computers are programmed and how these interact with the process of performance measurement.

Chapter 4

Programming Parallel Computers and Performance Measurement

Our approach to measuring and improving the performance of parallel programs is not dependent on a specific programming language. However, a demonstration of our techniques requires that a specific language be chosen. For this purpose, we use a parallel programming system called the Program Composition Notation or *PCN*. *PCN* has the dual advantage of being an effective method for expressing parallel computation and having a structure that facilitates performance measurement. In this chapter, the syntax and semantics of *PCN* are described and an overview of its implementation is presented.

4.1 Program Composition Notation

The Program Composition Notation (*PCN*) is a high level parallel programming language. Designed by Taylor and Chandy [CT89], *PCN* combines the lessons learned from the *Strand*[TF89] programming language with the formal foundations of the Unity [CM88] parallel programming model. In this section we present a brief overview of *PCN*. A more complete discussion can be found in [CT90].

PCN is based on one simple idea:

A complex program can be built by composing simpler programs.

The concept of composition is essential to building complex parallel programs from existing parts [Bac78]. The more powerful the composition operators, the easier is to construct complex programs within a simple language framework.

The focus of *PCN* is on providing a means to combine programs without placing restrictions on the programs themselves. *PCN* is a complete programming language and the programs being composed can be written totally in *PCN*. However, *PCN* is just as capable of combining programs written in other languages, such as Fortran or **C**. This gives rise to a programming style called *multilingual programming.*

From the early days of high level languages, multilingual programming has been a means to improve the performance of a program. Most high level language compilers provide a "trap door" from which in-line assembly code can be embedded into a program or assembly language subroutines called. Multilingual programming is a means of exploiting the efficiencies that can only be obtained from a lower level of abstraction.

Applied to parallel programs, multilingual programming combines operations written in a sequential programming language and a high level parallel programming language in a single program. Parallel multilingual programming is motivated by the following observations:

- Most commercially successful MIMD computers are built from standard high performance microprocessors which execute an instruction set optimized for sequential high level language programs.

- Compilers for sequential programming languages do a good job of generating code that exploits the capabilities of high performance microprocessors.

- Even a highly parallel application contains some tasks which are essentially sequential.

By utilizing existing sequential compilers to encode sequential operations, a multilingual program can offer performance superior to a program written in only a parallel programming language. In addition, since representing

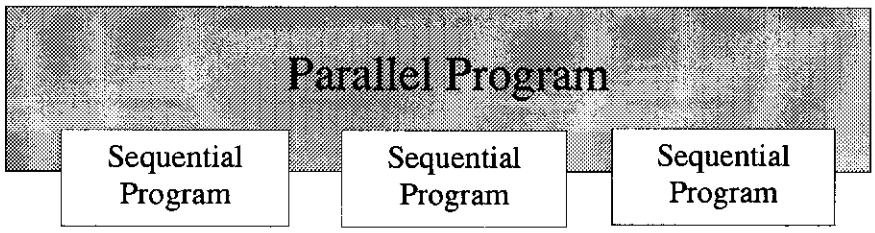

Figure 4.1: A multilingual program

sequential computation in a parallel language can be cumbersome, the resulting sequential code is easier to write and understand. Likewise, using a high level parallel language to represent the coordination and interaction of the sequential components yields a direct, clear and concise representation of the parallel aspects of the program.

An important benefit of multilingual programming is that existing sequential procedures can be embedded into a parallel program, preserving the time and money investmented in sequential algorithm development. Furthermore, multilingual programming provides a migration path from an existing sequential application into a parallel application.

Experience to date has shown that multilingual programming is an effective means for developing high performance parallel applications [FO90]. The portability inherent in a high level language approach to parallel execution results in programs that are easily ported to different parallel computers. By coding a program's kernel operations in an efficient sequential programming language, performance degradation of less than 1% is seen in single processor execution[FO91].

The structure of a multilingual *PCN* program is shown in Figure 4.1. The top level structure of the program is written in *PCN*, forming a collection of concurrently executing tasks. At some lower level in the program, the basic mode of computation switches from parallel to sequential. A sequential programming language such as C or Fortran is used from that level on down.

Multilingual parallel programming has advantages from the perspective of performance measurement. Programming for observability is the process of developing a parallel program in a manner that facilitates the observation of performance data [GS85]. Multilingual *PCN* programs have several characteristics that aid in this process. First, *PCN* is a simple language in

which parallel operations are represented in a straightforward manner. Consequently, it is easy to instrument a *PCN* program. Second, the boundary between parallel and sequential code provides a built-in abstraction mechanism. Sequential operations that are not relevant to the parallel performance of a program are easily identified by the boundary.

This dissertation focuses on execution profiling for parallel applications written as multilingual *PCN* programs. This choice is motivated both by the advantages of *PCN* as a means of expressing complex parallel computations in an efficient and portable manner and by the degree to which *PCN* enables observability. *PCN* currently executes on a number of commercially available parallel computers. One of the contributions of this dissertation is the design and implementation of the elements in the *PCN* system that support performance measurement [CTKF90].

4.1.1 Basic concepts of *PCN*

The building blocks of *PCN* are *programs*, which correspond to procedures in other languages, and *composition operators*. The mechanism used to create a program from simpler programs is a composition operator. If $P_1, \ldots P_n$ are programs and O is a composition operator, then $\{OP_1, \ldots P_n\}$ is a program. Note that this is really no different from the way most traditional programming languages work; however, in *PCN*, rather than only being able to compose programs sequentially, a richer set of compositions is provided.

The programs that are glued together by a composition operator can in fact be compositions themselves. Thus a new *PCN* program can be created from existing *PCN* programs by composition. A composition operator can also be used to create a new *PCN* program from programs written in "foreign" languages such as Fortran and **C**. In the context of *PCN*, all foreign code components are referred to as programs, regardless of whether they are actually complete programs, or just subroutines or procedures.

Programs interact with each other by means of shared variables. There are two classes of variables in *PCN*, *mutable variables* and *definitional variables*. *PCN* variables can contain integer and floating point values and structured data such as arrays. Mutable variables are like the variables found in a sequential programming language such as **C**. The value of a mutable variable is set by an assignment, a program that is a primitive in the *PCN* language.

A definitional variable is equated to its value by a definition, which is also

a primitive program. Definitional variables obey the single assignment rule; a variable can appear on the left hand side of an assignment at most once. The following rule applies to all uses of definitional variables:

> If a program needs the value of a definitional variable and that variable has not yet been defined, then execution of the program is delayed until such a definition has been made.

This single rule forms the basis of all synchronization and communication within a *PCN* program.

4.1.2 *PCN* data types

PCN has three scalar data types: integers, floating point numbers and characters. These data types can be stored either in a mutable or a definitional variable. In addition, *PCN* has two complex data types: arrays and tuples. An array is a linear sequence of any of the scalar data types. As with the scalar data types, arrays can be both definitional and mutable. A string is an array of characters and can be represented within a program by enclosing its value within double quotes.

Tuples differ from arrays in that: 1) they are only definitional and 2) the elements of a tuple can be any nonmutable *PCN* data type or a definitional variable. A tuple is similar to a **C** structure or a Pascal record except that elements are accessed by integer index and not by a field name. A tuple is written in a program using the syntax:

$$\{e_1, \ldots e_n\}$$

where the e_i are the elements of the tuple. A tuple with n elements is said to have *arity* n. With tuples, recursive data structures such as trees and lists can be built.

Lists, which are nonmutable, are often used in *PCN* programs. In *PCN* lists are written as:

$$[e_1, \ldots e_n]$$

The empty list is represented by a tuple of arity zero and is written as []. Finally, a vertical bar is used to designate the tail of a list, i.e., ["a" | T] is a list whose head is the string a and whose tail is the value of the variable T.

Tuples and lists can be explicitly written in a *PCN* program. When such a program executes, data structures of the specified type and contents are automatically created. Arrays are also dynamically created. However, there is no syntax to specify an array in a program. Rather, arrays are created via a declaration, much like automatics variables in **C**.

4.1.3 The *PCN* programming model: syntax and semantics

The basic syntactic structure from which a *PCN* program is built is a *block*. A block is one of:

- program call
- assignment
- definition
- implication
- composition

The simplest type of block is a program call. The syntax for a program call module:program(a_1, ... a_n). Every *PCN* program belongs to a module. When called from outside its defining module, the module name must be specified. If a program is called from within its defining module, the module name and colon can be eliminated from the calling syntax. Foreign program names are global and a module name is not required. The a_i are the actual parameters to the call and can be constants, *PCN* variables or expressions.

Foreign programs are written in accordance with the syntax and semantics of the programming language. No special interface between *PCN* and the foreign language program is required, however, there are some restrictions as to what data types can be passed from *PCN* to a foreign program.

PCN programs are defined in a manner similar to **C** procedures: the program name is followed by an optional declarations section which is followed by the body of the program. The exact format of a *PCN* program is:

program(p_1, ... p_n)
  $type_1$ $var_{1,1}$, var_{1,m_1}
  ⋮ ⋮
  $type_n$ $var_{1,n}$, ... var_{n,m_n};
B

where $type_i$ are *PCN* data types and B is a block. Only mutable variables are declared in a *PCN* program. If a variable is not declared, it is assumed to be a definitional variable and can contain any *PCN* data type.

An assignment is specified by:

X := *expression*

where X is a mutable variable. If the expression contains any definitional variables that have not yet been defined, then the assignment is delayed until the value of the expression has been completely defined.

The value of a definitional variable is defined by the block:

X = *expression*

Before a definition can take place, the right hand side must be completely defined. When this is true, *expression* is evaluated, and the resulting value equated into X. If any mutable variables appear in the defining expression, then a *snapshot* of the mutable values is made; the value of X is defined with the current values of the mutable variable.

Conditional execution is performed by an implication block. An implication contains two parts: a *guard* and a *body*. The guard is a sequence of comma-separated tests which are evaluated from left to right in the order that they appear within the guard. The guard tests are predicates that evaluate to true or false. The tests include relational tests and type checking. The syntax for an implication is:
T_1, ... T_n -> B
where T_i are guard tests and B is a block.

If all the guard tests in a guard evaluate to true, the guard succeeds and the body of the implication is executed. If the ith guard test of a guard fails and all other tests T_j, $j < i$ evaluate to true, the guard fails and no further action is taken. If the i^{th} test in a guard requires the value of a definitional variable and the variable is not yet defined and all tests T_j, $j < i$, evaluate

to true, the guard suspends. In this case, the guard evaluation is repeated until it either succeeds or fails.

To simplify guard tests on *PCN* tuples, a pattern matching guard is provided. For example, the guard test: X ?= [{"foo",x} | Xs] is true if the variable X is defined as a list whose head is a pair whose first element is the string "foo". The variable x is bound to the second element of the tuple and the variable Xs is bound to the tail of the list. No variable in a pattern matching test may appear more than once on the right hand side of a pattern.

Complex programs are built from simpler ones with compositions. The syntax of a composition is:

$$\{op\ \mathsf{B}_1, \ldots \mathsf{B}_2\ \}$$

where *op* is one of the *PCN* composition operators and the B_i are the blocks that are being composed.

There are three composition operators defined in *PCN*: parallel composition, sequential composition and choice composition. The syntax for each of these composition operators is shown in Figure 4.2.

Composition Type	Syntax
Parallel	$\{\|\|\ \mathsf{B}_1, \ldots \mathsf{B}_n\ \}$
Sequential	$\{;\ \mathsf{B}_1, \ldots \mathsf{B}_n\ \}$
Choice[a]	$\{?\ \mathsf{I}_1, \ldots \mathsf{I}_n\ \}$

[a]The arguments of a choice composition must be implications

Figure 4.2: Syntax of *PCN* composition operators

Sequential composition is used to sequence operations on mutable variables. If B_i and B_j manipulate a mutable variable in any way and $i < j$, then sequential composition ensures that the execution of B_i is complete before B_j is allowed to start execution.

The blocks within the scope of a parallel composition operator execute concurrently. The execution of blocks in a parallel composition is fair: every block in the composition that has not terminated will eventually execute. Mutable variables are not allowed within the scope of a parallel composition

operator. A parallel composition terminates when all blocks in the composition terminate. However, termination is neither required nor detected by *PCN*.

Both synchronization and conditionals are expressed in *PCN* by a choice composition. Only implications can be composed by a choice. When executed, the choice composition arbitrarily selects an implication and attempts to evaluate its guard. This process repeats until either one guard succeeds or all guards fail. If a guard succeeds, then that guard's body is executed and the choice terminates. If all guards fail then the choice simply terminates. A special implication with the single guard test **default** is allowed within a choice composition. The **default** guard succeeds only if all other guards within the implication fail. It is important to recognize that the order in which implications in a choice are selected for guard evaluation is completely arbitrary. If more than one guard is true, any of them can be selected for body evaluation.

4.1.4 Executing *PCN* on multiple processors

The parallel composition operator defines semantics for parallel execution. Regardless of how many processors a program executes on, these semantics must be followed, even if the composition is executed totally on a single processing node. However, because of these semantics, blocks composed by a parallel composition operator may be executed simultaneously.

Parallel composition fulfills an important function in addition to enabling simultaneous execution. Parallel execution semantics within a processor can be used to mask the latency of interprocessor communications or the unavailability of required values. Specifically, if the value of a definitional variable is needed and it is not currently available, either because it has not yet been defined or it resides on another processor, execution can be suspended and another program in a parallel composition executed. The ability to tolerate latency is an important property of a parallel programming language [AI83].

To execute a *PCN* program in parallel, one must map specific parallel compositions onto different processors. Partitioning and mapping a parallel program onto a processor graph is a difficult problem and has been studied extensively [Bok81, KT86, Mar87, BS84]. *PCN* makes no attempt to solve this problem. Rather, *PCN* provides a simple mechanism to facilitate the implementation of a mapping function, but leave the determination of that

function to the programmer. Eventually, we hope to develop a mapping tool, such as in [SH86], that can automatically generate program partitioning and processor mapping from the *PCN* program and the performance data collected by our measurement techniques.

To specify a processor mapping in *PCN*, the processor on which a composition executes is identified by annotating blocks within a parallel composition with a mapping notation:

program@location

where *location* indicates the node on which the program is to be executed. Thus the composition

{|| p1(x,y)@1, p2(y,x,z)@2, p3(z,y)@3 }

would execute program p1 on processor one, program p2 on processor two, and p3 on processor three. The semantics of parallel composition are the same as without the annotation, the definitional variables x, y and z will be usable to each program as if it had been executed on a single processor.

Program to architecture mappings are simplified by the embedding a *virtual topology* within the actual physical topology of the communications paths in the parallel computer. *PCN* supports this idea by allowing the mapping to specify a direction from the current node on the virtual topology. For example, from any processor in a ring topology, a program can be mapped forward and backward. In a mesh, mappings can be directed up, down, left or right of the current node. In addition, the annotation random can be used to specify a random mapping. The embedding of the virtual topology onto the physical topology of the parallel computer is handled by the *PCN*.

4.1.5 A *PCN* example

We conclude the overview of *PCN* syntax and semantics with two small examples. The first is shown in Figure 4.3. This program recurses over a tree and computes the sum of the values stored in the nodes. The top level composition is an implication. It checks the values of the input argument: tree. If tree is defined to be a tuple with three elements, the first is assumed to contain the value associated with the tree node and the second and third the left and write subtrees. The value of sum is defined to be the sum of the

```
/* Compute the sum of the nodes in a tree.
   Each node contains a value and the left
   and right subtrees.  A empty subtree is
   indicated by the constant "null_tree"
 */

sum_tree(tree,sum)
{? /* Are we at a tree node ? */
  tree ?= {value,left,right} ->
   {|| /* Sum is node value plus sums of left and                    10
          right subtrees */
       sum = suml + sumr + value,
     sum_tree(left,suml),
     sum_tree(right,sumr) },

  /* Empty tree terminates recursion */
  tree == "null_tree" -> sum = 0,

  default ->
   printf("Improperly structured tree\n")                            20
}
```

Figure 4.3: A *PCN* program to sum the nodes in a tree

current node and the left and right subtrees. These values are all computed in parallel. Note that the definition of **sum** can not take place until the values of **suml** and **sumr** have been defined.

The second example shows the use of mutable variables and sequential composition. Figure 4.4 shows a *PCN* program to compute the DAXPY, a double precision operation from the BLAS library [LHKK79]. The DAXPY calculates:

$$y \leftarrow \alpha x + y$$

where x and y are vectors and α is a scalar. In this program, **y** is a mutable variable. A sequential composition is used to ensure that the updates to

```
/*
   Compute the vector sum:  Y = aX + Y
   where "a" is a scalar and X and Y are vectors.
   The length of the vector is N and I is initialized
   to the first element to be computed.
*/
daxpy(res,a,x,y,I,N)
    double x[],y[],a;
{? i0 < N ->
    {; y[I] := a * x[I] + y[I],                                    10
         daxpy(res,a,x,y,I+1,N)
    }
}
```

Figure 4.4: A *PCN* program to perform the DAXPY operation

y occur in order and that all updates are complete before the composition terminates. Note that since the argument I is not declared, it is a definitional variable.

4.2 Implementation of *PCN*

In the next chapter, we present a set of techniques for measuring the performance of *PCN* programs during execution. These techniques depend not only on the structure and semantics of *PCN*, but also on the details of the implementation of *PCN*. For this reason, it is necessary to give a brief overview of how *PCN* is implemented on a parallel computer. Some simplifications are made for the purposes of this overview. However, a detailed discussion of *PCN* implementation can be found in [FT90].

There is no single way in which *PCN* must be implemented. Many tradeoffs were made in arriving at the current design. The tradeoffs were driven by four factors. These factors are:

- **Observability.** The *PCN* implementation was designed so that observability is a property of the programming language rather than the

program. Enabling the performance measurements detailed in the next chapter is a factor in the design of the *PCN* implementation.

- **Portability**. There are many different types of parallel computers currently available. These machines have different communications architectures as well as different processor architectures. It is necessary to port the *PCN* system to any of these machines with a limited amount of effort. Therefore there can be no reliance on architectural features such as shared memory, or a particular instruction set architecture.

- **Efficiency**. Decreased execution performance is often the price paid for high level programming abstractions. Minimizing this performance penalty is a major concern in designing the *PCN* implementation.

- **Simplicity**. Portability and correctness require that the implementation be as simple as possible. In some cases efficiency has been sacrificed for simplicity. The result is that a complete working *PCN* system has been implemented in six months.

4.2.1 The Program Composition Machine

A *PCN* program is a specification of a parallel computation. However to be useful as a programming language, a *PCN* program must be converted into a form which can be executed on a parallel computer. In *PCN*, as in many other programming languages, efficient implementation requires that a compiler translate operations in the language into operations in the target architecture. However, rather than compiling *PCN* directly into the instruction set of the target architecture, the compiler generates code for an architecture and instruction set specifically designed to support the execution of *PCN*. This architecture is called the Program Composition Machine or *PCM*.

The *PCM* defines a set of resources and an instruction set. The resources include a set of registers and memory for allocating *PCN* data structures. The abstract instruction set manipulates these resources to perform the *PCN* computation. The instruction set is defined to expose potential optimizations to the compiler. However, because the instruction set is tailored to *PCN* execution, it is much easier to apply these optimizations than if a native instruction sequence had been generated. In addition, all code generation and optimizations targeted to the *PCM* can be applied regardless of the

actual architecture on which the *PCN* program is to execute. A summary of the architecture and instruction set of the *PCM* is found in Appendix B.

Once a *PCN* program has been translated into *PCM* code, the *PCM* code can then be translated into the native instruction set of the computer on which the program is to execute. However, for the initial implementation of *PCN*, this approach is not taken. Rather, the *PCN* program is executed directly by emulating the actions of the *PCM* on the target architecture. The emulator is a program that runs on the target architecture and interprets a *PCM* instruction stream. If the emulator is written in a high level language such as C, a very portable system results.

When executing on a parallel computer, one copy of the *PCM* executes on each processor. To maximize portability, communication between *PCM*s is accomplished by message passing. The only architectural characteristic assumed is the ability to asynchronously send and receive messages. On multicomputers, this capability is usually provided by the operating system on the processing node. On multiprocessors, a message passing library built on shared memory is used; outside of this library, there is no shared memory or locking used in the *PCM*.

Obviously, executing *PCN* by an emulator has an impact in the execution time of a *PCN* program. The emulator must fetch and decode *PCM* instructions, operations that are not performed if the *PCM* code is translated into a native instruction sequence. However, experience with similar languages shows that the size of a native code implementation of a *PCN* program is much larger than an interpreted implementation [Tay89]. This tends to have a negative impact of the performance of the memory subsystem; in particular more cache misses and page faults can be expected. In the long run, an implementation that uses native code compilation of selected procedures yields the best tradeoff between speed and size.

4.2.2 Core *PCN* and the *PCM* execution model

To further simplify *PCN* implementation, the *PCM* is not actually capable of executing the complete *PCN* language. Rather, prior to compilation, a *PCN* program is translated to a simpler form called core *PCN*. Core *PCN* differs from *PCN* in the following respects:

- There is no sequential composition operator. Parallel compositions are sequenced explicitly with definitional variables to provide the same functionality.
- The are no nested blocks. The body of a parallel composition can only contain assignments, definitions, *PCN* and foreign program calls.
- All choice compositions have a **default** implication. The body of this implication can be a call to the program **skip()**, which does nothing.
- The body of an implication must be a parallel composition.
- Assignments and definitions only occur within a choice composition whose guard ensures that all definitional variables on the right hand side been defined.

With core *PCN*, the *PCM* only needs to be able to evaluate the guards of a choice composition and spawn the programs in a parallel composition. All other operations, such as sequential composition and nested implication, are encoded directly in core *PCN*. Not only does core *PCN* reduce complexity of the emulator, but it makes instrumenting a program for performance measurement easier as well.

Runnable instances of a composition are created by the execution of a parallel composition. Each instance, called a process, is specified by the program code and the arguments with which the program was invoked. Because many processes can be created from the execution of a single parallel composition, the state of a *PCN* program is determined by a collection of processes, called a process pool.

Because core *PCN* only has parallel composition, there is no structure to the process pool. Any dependencies and interactions between programs are determined solely by the definitional variables shared between process arguments. Thus any process in the collection can be selected for execution. This fact forms the basis of the core *PCN* execution algorithm, which is shown in Figure 4.5.

Because processes are selected from the pool in any order and interaction between processes is controlled by definitional variables, a parallel execution algorithm can select more than one process from the pool at once. By invoking multiple instances of the algorithm of Figure 4.5, parallel execution is obtained.

1. *Program selection.* Select any process from the process pool and remove it.

2. *Guard evaluation.* Evaluate the guards for each implication in the program.

   If at least one guard is successful, go to step 3.

   If none of the guards succeed but at least one suspends, go to step 4.

3. *Guard success.* Arbitrarily select one of the implications whose guard was successful.

   If the body of that implication is an assignment or definition, perform it. Otherwise, for each program call in the body, add an instance of the program to the process pool.

   Go to step 1.

4. *Guard suspend.* Place the program back into the process pool.

   Go to step 1.

Figure 4.5: The abstract execution algorithm for core *PCN*

Core *PCN* is implemented by a collection of *PCM*s, one per processing node. Within a *PCM*, the core *PCN* execution algorithm is implemented in two pieces: a reduction component and a communication component. The reduction component is responsible performing process reductions: removing a process from the pool, evaluating the guard and adding new processes to the pool. The communications component is responsible for ensuring that the values of all definitional variables are available to every *PCM*. Recall that since a variable can be defined only once, consistency of variable values within the *PCM* is not an issue.

Within an instance of the *PCM*, the execution of the reduction component and communications component alternates. The reduction component is allowed to perform some number of process reductions and then control is turned over to the communications component. The number of reductions executed is determined by a system parameter called the *timeslice*. The communications component executes until all outstanding communications requests have been processed, at which time control returns to the reduction component.

4.2.3 The reduction component

In the reduction component of the *PCM*, abstract machine code representations of *PCN* programs are executed. The basic layout of the sequence of *PCM* instructions generated by the *PCN* compiler is shown in Figure 4.6. Implications are placed one after another. Within an implication, code for the tests in the guard appears first, followed by the *PCM* code to spawn the parallel tasks in the implication body. The method used for sequencing between implication guards is discussed in detail in Section 5.5.3.

Each instance of a *PCM* has a storage area called the *heap*. All *PCN* data structures, including process records and *PCN* code, are allocated from this area. A garbage collector is used to reclaim unused data items from the heap when storage space runs low. New structures are allocated on the heap only during the execution of parallel composition, never during guard evaluation.

Definitional variables are allocated on the heap along with the other data structures. The compiler arranges to create a definitional variable the first time it is referenced during program execution. When first created, the contents of a definitional variable are set to a value which indicates that it

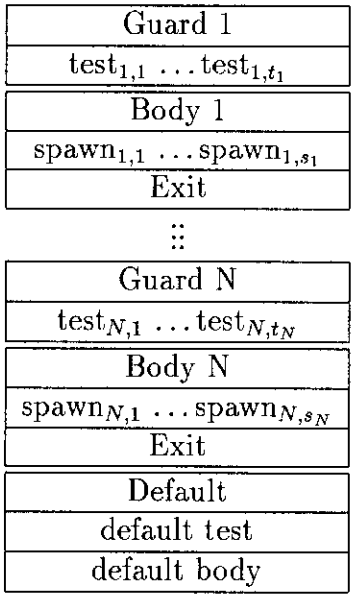

Figure 4.6: Basic structure of *PCM* code representation of a core *PCN* program

is undefined. When a definition is made, the variable contents are replaced with a pointer to the defining value. If the variable has already been defined, a runtime error is generated.

Definitional variables never move from the heap on which they were created. However, consider the following example:

```
{|| p(x),q(x)@fwd }
```

The definitional variable x is created in the *PCM* which executes the program p. However, the variable is also referenced from that *PCM* which executes q. On this processor, x is represented by a remote reference to its location on the *PCM* that creates x. When x is defined, this reference can be replaced by the value of the definition. If the definition of x is made in program q, then a request to perform the definition along with the defining value must be sent to the *PCM* executing p.

An executable instance of a program is represented by a structure called a process record. Each process record contains a pointer to the *PCM* code implementing the program and the arguments with which the process in invoked. Process records are linked into one of two lists: a *run queue* or a *suspension queue.*

If a program is not currently being executed and is ready to run, it is linked into the run queue. If a process is executed but suspends, the process must wait until the required definitional variables become defined. So as not to waste time retrying the process, the process record is pushed onto the *suspension queue*[1]. The reduction process is summarized in Figure 4.7.

4.2.4 The communications component

The communications component is entered whenever the reduction component's timeslice is over or the run queue is empty. The primary job of the communications component is to update the state of the memory of the *PCM* based on the actions of the *PCM*s executing on other nodes. These actions are 1) another node has requested the definition of a locally created variable and 2) provide the value of a definitional variable to another *PCM*.

Communication between *PCM*s is performed by sending messages. Three basic types of messages are sent: read, value and define. If during the execution of an implication guard, the value of a remote reference is required,

[1]This is a simplification of how things actually work. For details consult [CTKF90].

1. *Check timeslice.* If timeslice is over, go to communications component.

2. *Check run queue.* If it is empty, go to communications component.

3. *Install process.* Remove the first process record from the run queue copying the A process arguments from the process record into the first A abstract machine registers.

4. *Execute guard.* Execute *PCM* instructions for guards for all implications of a choice composition.

   If the guard succeeds go to step 5.

   If the guard suspends go to step 6.

5. *Execute body.* Execute *PCM* instructions for body of implication.

   For each program call in the body, allocate new process records and insert them on to run queue.

   Perform any assignments.

   If a definitional variable is defined then: 1) perform the definition, if the variable was created on another node, then this is done by requesting that node to perform the definition, 2) move any local processes from the suspension queue to the run queue and 3) notify other nodes who have requested the value that the definition has taken place.

   Go to step 1

6. *Guard suspension.* Arrange to get values of definitions when they become defined.

   Go to step 1.

Figure 4.7: The reduction process within the *PCM*

a read message is sent to the *PCM* to which the reference points. The requested value is returned in a value message. The define message is used to perform a definition when the left hand side is a remote reference. The sending of read and define messages takes place within the reduction component. Sending value messages and all processing of received messages takes place within the communications component.

An overview of the communications component is given in Figure 4.8. Note that in addition to performing the communications task, this algorithm has the side effect of implementing an *idle loop*. That is, when there is no computation to perform, the *PCM* cycles through the communications component waiting for variable definition that will enable execution to continue.

1. *Perform definitions.* If a definition of a locally created definitional variable has been requested by another processor then: 1) perform the definition, 2) move any local processes waiting for this variable from the suspension queue to the run queue and 3) inform nodes that have requested the variable value that the definition has been made.

2. *Wake processes.* Another *PCM* has executed a definition of a variable whose value is required. For each new definition value, replace the remote reference with the defined value. Move any processes waiting for this variable from the suspension queue to the run queue.

3. *Check run queue.* If run queue is empty go to step 1 else go to reduction component.

Figure 4.8: The communications component with the *PCM*

4.3 Summary

In this chapter, we have reviewed the different ways in which a parallel execution of a program can be achieved. We have selected the Program Composition Notation or *PCN* to demonstrate our approach to performance measurement. *PCN* is a parallel programming notation based on the single concept of composition. *PCN* was designed to enable multilingual programming, a programming methodology in which both *PCN* and sequential programming languages are used to represent a parallel algorithm. This approach combines the clarity of notation of a parallel programming language with the efficiency of sequential programming languages.

In the next chapter, we turn our attention to developing measurement techniques for *PCN* programs.

Chapter 5

Profiling Techniques for Parallel Programs

In Chapter 2, we discussed the advantages of performance measurement based on execution profiling over performance measurement based on event logs — the probe effect can be eliminated and the amount of data generated is not dependent on execution time. In this chapter, we examine how to make the measurements required for an execution profile. We proceed in two steps. First we develop a general framework for designing a measurement process. We then apply this framework within the context of *PCN* to obtain a set of measurement techniques for profiling.

5.1 Measurement and Execution Profiling

In an execution profile, measurements are made to determine the frequency and duration of events in the measurement model. As discussed in Section 3.3, the events of interest are program execution, interprocessor communication and processor idle time. Within the definition of a parallel execution profile, a considerable degree of freedom exists in selecting exactly which data is needed and how it is to be measured. In deciding how and what to measure for an execution profile, one trades off the cost of measurement against its advantages.

The cost of measurement is determined by the overhead in terms of execution time and storage space. The execution time of a computation can be

expressed as:

$$T = T_{PCN} + T_{foreign} + T_{communication} + T_{idle}$$

where T_{PCN} is the amount of time spent executing *PCN* programs, $T_{foreign}$ is the amount of time spent executing foreign programs, $T_{communication}$ is the amount of time a *PCM* spends communicating and T_{idle} is the amount of time the *PCM* spends idle. For clarity, the dependence of T on $\mathcal{P}$ is not shown. If $T_{measurement}$ is the amount of time spent making and recording performance measurements during the execution of a program, we define the execution time overhead of the computation as:

$$TO = \frac{T_{measurement}}{T_{PCN} + T_{communication} + T_{idle}} \tag{5.1}$$

To focus on the parallel components of program execution, $T_{foreign}$ is not included in the overhead calculation. As a result, the values of TO presented in this chapter are conservative.

Storage overhead is the fraction of the static size of a program devoted exclusively to profiling. The code size of a program is

$$S = S_{PCN} + S_{foreign} + S_{PCM}$$

where S_{PCN} is the size of the *PCN* parts of a program, $S_{foreign}$ is the size of the foreign programs and S_{PCM} is the size of the abstract machine. We define storage overhead by:

$$SO = \frac{S_{measurement}}{S_{PCN}} \tag{5.2}$$

where $S_{measurement}$ is the amount of additional storage required to make and record the measurements. Again, to focus on the parallel aspects of a program, only the storage requirements of the *PCN* programs are included in the overhead calculation.

5.1.1 Principles of design

To structure the design of the measurement procedure for execution profiling, we formulate three design principles. These principles enumerate the design

tradeoffs that are possible in the measurement process. The design parameters that fall under the control of these principles are: 1) the measurement model and method of measurement used, 2) the accuracy of the measurement required and 3) the implementation of the parallel programming system.

5.1.2 Determining how to measure

The cost of measurement depends directly on the measurement model and state decomposition. These factors determine:

- What is to be measured
- The number of measurements made
- How are the measurements are made

For example, an execution profile can be derived from an execution trace. But the storage overhead of an execution trace is always greater than the storage overhead incurred when a profile is measured directly. This tradeoff is an instance of our first design principle.

Design Principle 1 *Choose a state decomposition and measurement model that balances the requirements of the profile with the cost of measurement associated with the measurement model.*

The level of abstraction at which the state decomposition takes place determines what is to be measured and how often measurements are made. The measurement hierarchy for a *PCN* computation, shown in Figure 5.1, ranges from measuring program execution in terms of a whole computation down to the execution of machine instructions. For each performance index in a profile, the appropriate decomposition is determined by how a programmer will use the index and the cost of measurement.

In addition to directing what to measure, Principle 1 also directs how to measure. That is, what measurement model to use. We identify three classes of measurement models based on the type of measurements they require. These are: direct measurement of index values, measurement based on statistical inference and indirect measurement. We now discuss each of these alternatives and their applicability to parallel execution profiles.

Per Computation
Per *PCM*
Per Module
Per Composition
Per Implication
PCM Instruction/ *PCM* Operation
Native Instruction

Figure 5.1: The measurement hierarchy for a *PCN* computation

5.1.3 Direct measurement

The obvious way to evaluate a performance index is to directly measure its value. A program is instrumented with sensors; each sensor is capable of observing and recording the value of a performance index. The sensors are placed in a program in accordance to the mapping from program state to program code defined by the measurement model.

Implementing software sensors to count event frequency is straightforward. For each state, a storage location of suitable size is allocated on each processing node. The minimum size of the storage location needed depends on the total execution time of the program and the relative frequency of the event being measured. The instruction sequence that implements the sensor increments the contents of the storage location by one. If the storage size is one word, the addition can be performed by a single integer add instruction. Depending on the computer architecture, fetching the current counter value and storing the new value requires up to three additional instructions.

Capturing the time duration of an activity on current parallel computers is more involved. Time duration can not actually be measured. Rather it is computed by measuring the time at which the activity in question starts and stops. The accumulated time is stored into a memory location whose size depends on the accuracy of the time measurement and the maximum expected total time spent in the activity.

The way time is measured depends on the level of support for timing provided in the computer system hardware and software. Fortunately, most parallel computers provide a low overhead mechanism for determining the

current time, from which lapse time can be computed. Typical of such a mechanism is the timing facility found on the BBN TC2000 parallel computer. Each processing node on the TC2000 maintains a 32 bit counter that is incremented once a microsecond. The current contents of the counter can be obtained by a library call. The Sequent Symmetry provides similar mechanism; however, the counter can be mapped directly into a program's address space and accessed via a single move instruction.

Lapse time is computed from counter values by subtracting the value of the counter at the beginning of an action from the value of the counter recorded at the end of the action. The result is added to the storage location used for recording time duration of action being timed. Typically, the execution time overhead for computing lapse time is five to ten machine instructions.

A consideration when directly measuring lapse time is the resolution of the counter; a processor such as the TC2000 can execute about 30 instructions within a single microsecond counter tick. In practice, the effects of limited timer resolution and the overhead for computing lapse time combine to place a lower bound on the size of an action that can be observed directly. The minimum execution time of a *PCN* program falls below this bound and direct measurement is not feasible.

For our profiling techniques, most frequency measurements are made directly. The remaining frequency measurements are made using the indirect methods discussed in Section 5.1.5. For the reasons given above, we cannot use direct measurement for *PCN* programs. However, other components in the execution profile, such as the execution of foreign programs, have individual execution times long enough to make direct measurement possible.

5.1.4 Statistical methods (sampling)

Statistical inference provides a second method by which the values of performance indexes can be obtained. In this method, a histogram of states is constructed by repeatedly sampling the state of the program. If enough samples are taken and the sampling interval is independent of program activity, the histogram approachs the actual frequency distribution of states. Furthermore the time spent in each state can be estimated by:

$$\text{Total execution time} * \frac{\text{number of samples in state } s}{\text{Total number of samples}}$$

Sampling is often used for profiling execution times in sequential [Knu71, GKM83] and vector [Cra89] programs. In these, the value of the program counter is sampled and execution time is determined from the resulting histogram of program addresses. Program counter sampling in *PCN* is complicated by the existence of two program counters: the program counter in the target architecture and the program counter in the *PCM*. To determine the amount of time spent in foreign programs as well as in *PCN* programs, two measurements must be made. First the program counter of the target machine must be examined. Its value determines if the *PCM* is executing *PCN* code, is executing a foreign program, is idle or is communicating. If the program counter in the target machine is pointing to the reduction component of the emulator, a *PCN* program is being executed and the *PCM* program counter is sampled to determine its identity.

The applicability of sampling depends on having enough samples to make the histogram statistically significant. In *PCN* execution, actions of interest such as guard execution account for a small fraction of total execution time. This suggests that either high sample rates or long execution times are required to obtain meaningful information. The nature of multilingual programs exacerbates this problem since the time spent in the *PCN* component of a computation is often small compared to the time spent executing foreign programs. Because of these factors, we do not use sampling as a means of measurement in our profiling techniques.

5.1.5 Indirect methods

There are situations where, due to difficulty in placing sensors or the cost of measurement, direct measurement of a performance index is not practical. In these cases, an alternative is to express the value of a performance index in terms of measurements that are easier to make or less costly to obtain; we call this an *indirect method.*

The basis for indirect methods is the use of a performance model to relate measurements to the value of a performance index. Models can be deterministic — relating measurements to a single performance index value,

or probabilistic — relating measurements to a distribution of performance index values. An example of a probabilistic model is given shortly.

The general structure of a performance model is shown in Figure 5.2. In addition to the measurements collected during program execution, the model utilizes two other sources of information: 1) statically determined characteristics of the program and 2) characteristics of the implementation of the programming language on a specific parallel computer. As the figure indicates, different performance indexes can be obtained from a single set of measurements by using different performance models.

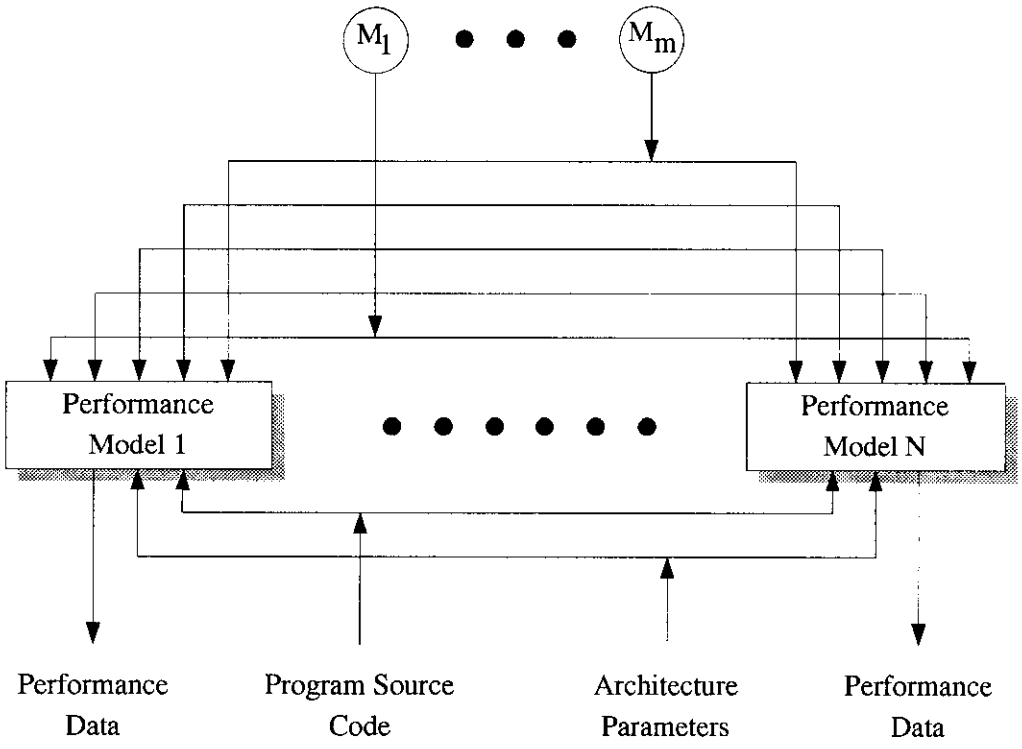

Figure 5.2: The form of a performance model

As a clarifying example, we present a indirect method to determine execution time from execution frequency. For each instruction, i_k, in the *PCM* instruction set, let n_{i_k} be the number of times the instruction executes during an execution of a program p. The execution time of the program, t_p can be determined by:

$$t_p = \sum_k n_{i_k} t_{i_k} \tag{5.3}$$

where t_{i_k} is the execution time of instruction i_k.

If the execution time of each instruction is always the same, then the model is deterministic and t_p has a well defined value. However, for a variety of reasons, such as dependence on value of the instruction's arguments, the contents of the computer's cache memory or contention for the main memory bus, an instruction can have a range of execution times. When this is the case, the execution time of an instruction can be represented by a random variable whose value is characterized by a probability density function. This implies t_p must also be a random variable and the performance model is probabilistic. A parameter of the distribution of t_p, such as the mean, can be used to represent the program's execution time. By taking the expected value of both sides of Equation 5.3, we see that a single value for execution time can be obtained by using the average instruction execution time in place of t_{i_k}.

The profiling techniques we present in this chapter rely extensively on indirect methods. The recognition of the importance of indirect methods to parallel profiling is unique to the methods presented in this dissertation.

An obvious concern with indirect methods is the potential for error in the resulting value of the performance index. In the next section, we address the issue of accuracy in performance measurement.

5.2 Accuracy in Performance Measurement

It is usually assumed that a high degree of accuracy is required when measuring the performance of parallel programs. However, we argue that the special demands of parallel systems dictate that some error in measurement be tolerated and expected. The advantage of this view is that measurement overhead can be reduced by sacrificing measurement accuracy.

By favoring an execution profile over an event trace, a tradeoff between accuracy and overhead has already been made. While there is less information in a profile, the cost of getting that information is less than that for a event trace. Going a step further, we argue that a profile is useful even when there are errors in the values of its performance indexes.

Profiles are generally not used to answer detailed questions about actual values of performance indexes. Often we are interested in locating an extremal value: the processor that has the least amount of work, the program that took the longest to run, etc. Relative values are also of interest; for example, program A spent about twice as much time idle as program B. In either case, errors of 10% or 20% do not drastically affect the course of the performance analysis. As a case in point, the Cray X-MP has a profiler based on sampling the program counter only once every 200,000 instructions. Yet the users of this system still find the profile data useful [Rei90].

Another factor in measurement accuracy is the number of processors used for a computation. As the number of processors on a parallel computer increases, it becomes more difficult to analyze and interpret performance data with respect to an individual processor. Rather, it is likely the user is presented with a statistical summary of the performance data and the value of any one data point becomes less important. In a summary, individual data values will deviate from the statistic, introducing uncertainty.

Finally, it is not always possible to associate a single "correct" value with a performance index. Instead of having one well-defined value, some performance indexes are best described by a random variable. For example, in the previous section we showed a situation in which the execution time of a program could have more than one "correct" value. In this case, the probabilistic description arose from the use of a model that did not distinguish between different instruction execution times. However, uncontrollable factors such as cache memory contents, contention for shared resources and external interrupts can all affect the execution time of a program. Because of this, execution time is often best represented by a random variable with some probability distribution.

In a profile, this distribution is represented by a statistic such as mean value. It is possible, however, that no single performance experiment can ever produce that value and some error will be present. None of this is a great surprise --- almost any measurement process is prone to some degree of error and measuring program execution is no exception.

For these reasons, we take the approach that measurement accuracy can and should be sacrificed if something is gained in return. In return for accepting "noisy" performance measurements, one is given a new freedom in how performance indexes are measured. Hence, the following design principle:

Design Principle 2 *If the cost of a measurement is too high, consider a less accurate measurement if it can be made at a reduced cost.*

5.3 Implementation and Measurement

The first two design principles focus on how the type of measurement affects the overhead of execution profiling. For the third and final design principle, we look at the interaction between the implementation of a programming system and measuring the performance of that system. This interaction is of particular interest if we accept that observability is a design criteria of the implementation of *PCN*.

There is no single way to implement a programming language. Different approachs to implementation have advantages and disadvantages with respect to speed, size, ease of implementation, etc. In addition, varying the method of implementation can have a significant impact on the performance measurement. This design tradeoff is represented in the following design principle.

Design Principle 3 *The method used to implement a programming language has an impact on the overhead of performance measurement. Consider this interaction by balancing the advantages of an implementation against the cost of measurement.*

5.3.1 Design goals

To design the measurement process for execution profiling, we have taken a qualitative approach. Rather than attempting to formulate a single design equation to be optimized, our design is based on discussions with potential *PCN* users and *PCN* implementors on the relative importance of the costs and benefits that alternative designs offer [TF].

Invariably, users rank minimizing runtime overhead over all other factors. Of secondary importance is storage overhead, followed by accuracy. Based on these considerations, our design goal is to produce an execution profile with the following characteristics (in order of importance):

- Runtime overhead of less than 3%

- Storage overhead of less than 10%
- Accuracy within 20%

5.4 Test Programs

Throughout the remainder of this chapter, we will require the use of actual *PCN* applications for various measurements. We have selected seven benchmark programs for this purpose. The seven programs used are summarized in Table 5.1.

Program Name	Task performed
similarity	Search for similar DNA sequences
2PTBVP	Solve a two point boundary problem
Composite	Solve the shallow water equations on a sphere projected onto two planer surfaces
gauss	Solve a set of linear equations using gaussian elimination
rhombus	Solve shallow water equations using control volume method based on a rhomboidal tiling of a sphere
triangle	Solve a simple puzzle by search
FLOW	Simulate Taylor vortices in an incompressible fluid

Table 5.1: Test programs used to study measurement overhead

Because *PCN* is a recent development, a large number of major applications do not exist. Even so, we were able to obtain several application programs that solve large scale problems. The programs cover a range of application domains and parallel programming techniques. The seven programs were written by six different people, with gauss and triangle having the same author. We now give a brief description of the programs.

Similarity is a program, written completely in *PCN*, that matches DNA sequences. The basic process structure is a single master and a set of work-

ers. The workers repeatly request sequences from the master, which are then matched against the worker's own sequence database with a dynamic programming algorithm. To mask the latency between requesting and receiving work, a single element bounded buffer is used to prefetch the next sequence from the master. The communications topology is a star, with the master at the center. There is no communication between the workers. The amount of parallelism in similarity depends on the number of sequences to be matched. In actual use there are thousands of sequences matched; for our tests, a small number of sequences are matched repeatly.

2PTBVP is solves a two point boundary value problem. It is written in *PCN* and Fortran. The parallelism in 2PTBVP is obtained by partitioning the problem into a set of subproblems, solving the subproblems and combining the results. The communications structure is a binary tree with computation occurring at each node. The performance of this program is limited by the decreasing amount of work available as the locus of computation moves up the tree.

Composite, written in *PCN* and **C**, solves a set of partial differential equations on a sphere. These equations, known as the shallow water equations, are a simple model of the earth's climate. To handle the singularities in the polar coordinate systems, the north and south poles are solved on a plane, and the results interpolated back onto the sphere. There are three different communication structures used in Composite. Grid cells on the sphere have nearest neighbor communications. At the poles, the interpolation from the plane to the sphere results in a communications structure which is not nearest neighbor. Finally, there is a global broadcast which is implemented by connecting grid points together in a spanning tree. The amount of parallelism in the program is determined by the partitioning of the grids.

Gauss solves a set of linear equations using gaussian elimination. The input to gauss is a randomly generated set of equations that are guaranteed to be solvable; the size of the system is specified by an input parameter. Gauss is written completely in *PCN*and uses mutable variable and sequential composition extensively. Each row in the input matrix is encapsulated by a process. Consecutive rows are mapped onto adjacent processors in a virtual ring topology. In the elimination phase of the algorithm; the diagonal elements of the input matrix are broadcast to all other row processes by a shared definitional variable. The same method is used to propagate solution values during the back propagation phase. Most of the computation

time in gauss is spent in the double precision vector calculation: $ax + y$. The amount of parallelism in gauss depends on the size of the system of equations being solved. However, its performance is limited by the algorithm since in both the elimination and back substitution phases, the number of rows to be processed decreases as the algorithm progresses.

Rhombus is another solution to the shallow water equations. However, this program solves the equations by an icosahedral tiling of the sphere. Written in *PCN* and Fortran, rhombus is designed to run on 13 processors. Each data partition is encapsulated by a *PCN* process and the computation is locally synchronized by the availability of data from the neighboring rhombuses. All communication is nearest neighbor.

Triangle is written in *PCN* and solves a simple puzzle by breadth first search. The basic structure and characteristics of triangle are very similar to that of similarity, a master/worker process structure in a star topology.

FLOW is a multilingual *PCN* and Fortran program to simulate Taylor vortices in an incompressible fluid using the three dimensional Navier-Stokes equations. FLOW is the subject of a case study in Chapter 8. The simulation is based on a grid, with parallel computation taking place on a set of grid points called a cell. Each cell is encapsulated in a process and the process mapped onto a processor. The most frequent type of communications is nearest neighbor in the grid space, implemented by a sharing a definitional variable between adjacent cells. There is also a global communications component which is implemented by a combination of merger processes and shared definitional variables. The amount of parallelism in FLOW is determined by the size of the grid being used for the simulation.

5.5 Measuring Event Frequencies

In this section, we develop methods for determining the frequency of events which occur during the execution of a *PCN* program. The sensor used to measure event frequency is a counter. The number of counters used and the frequency of their update determines the measurement overhead for execution profiling.

The objective of our measurements is to determine the number of times each *PCM* instruction in a composition is executed. From this, higher level frequency metrics can be derived. Examples of such metrics include the

number of times a program is used, how often a specific implication guard fails and how many times a program is called from a specific implication.

Although the user writes a program in *PCN*, it is core *PCN* that the *PCM* executes. Our measurement techniques are limited to core *PCN*. The execution frequency of a *PCN* program is found by summing the frequencys of the core *PCN* programs that implement it. This means that for any *PCN* program, we need only know how to measure parallel compositions, choice compositions, assignments, definitions and foreign program calls. We address each of these in turn.

5.5.1 Frequency measurement of parallel composition

A parallel composition appears either by itself or as the body of an implication in a choice composition. The operations performed in a parallel composition are limited to: 1) start the execution of a *PCN* program, 2) perform assignments and definitions, 3) call foreign programs and 4) allocate new data structures on the *PCM* heap.

The basic structure of the *PCM* code generated for a parallel composition is shown in Figure 5.3. A new *PCN* computation is started by executing a *fork* instruction which allocates a process record and appends it onto the active queue. The arguments to the process are allocated on the heap by a sequence of *put_value* instructions, which also place references to the arguments in the process record. After all the programs in the composition have been forked, the execution of the parallel composition terminates with a *halt* instruction, which dequeues the current process record.

The *fork* and *put_value* instructions are only used to call *PCN* programs. If the program is written in a foreign language, then *call_foreign* and *put_foreign* instructions are used instead. Frequency measurement of foreign programs is discussed in Section 5.5.12. Special instructions are also used to implement definition and assignment of variables. This is discussed in Section 5.5.11.

The number of times a parallel composition executes can be measured by counting the number of times the *halt* instruction in the composition is used. The operations in a parallel composition are performed in sequence. Thus if the number of times each *halt* instruction executes is known, then the number of times each instruction in the parallel composition executes is also known. The actions performed by the *PCM* during the execution of a

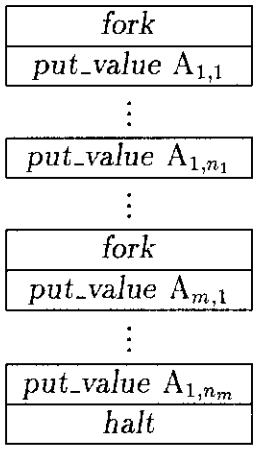

Figure 5.3: Basic structure of the *PCM* code for a parallel composition

parallel composition are completely determined by the instruction sequence executed in that the behavior of the instructions does not depend on the state of the *PCM*.

We now consider an optimization to the implementation of parallel composition and its impact on the measurement process. Tail recursion optimization is a technique that improves the performance of the last fork in a parallel composition by having it reuse the process record from the currently executing composition. This eliminates the cost of enqueuing and later dequeueing the program on the active queue. In addition, the rate of memory consumption is reduced by reusing the process record.

Tail recursion optimization only applies to parallel compositions that call at least one *PCN* program. To implement the tail recursion optimization, a *recurse* instruction is used instead of the normal *fork* $\cdots$ *halt* instruction sequence which terminates such a composition. The *recurse* instruction sets the program counter to the program specified in the instruction's argument, executing the new program with the current contents of the argument registers.

As defined, there is a problem with the *recurse* instruction: a program can recurse infinitely, violating the fairness requirement of parallel execution. To prevent this, the *PCM* has a *time slice* register which is set to a system

defined value whenever a process record is taken from the front of the active queue. Every time a *recurse* instruction executes, the contents of the time slice register are decremented. If the time slice is not zero, the execution of the *recurse* instruction proceeds as described in the previous paragraph. However, if the contents of the time slice register are zero, the *recurse* instruction does not execute the program. Rather, the argument registers of the *PCM* are copied into the current process record and the process record is enqueued on the active queue. With the time slice register, tail recursion optimization can be used and still guarantee that every runable *PCN* composition will still have a chance to execute.

These alternative behaviors cause the problem from the perspective of measurement. Depending on the value of the time slice register, the behavior the *PCM* will be radically different during execution of a parallel composition. Using Principle 3 is not feasible because tail recursion optimization is far too important to the performance of *PCN* not to use. Looking to Principle 1, we observe that altering the measurement to record how many times each behavior occurs would result in a increase in space overhead of 100% in the worst case.

The final option is to apply Principle 2 and not distinguish between the two cases in the *recurse* instruction. In the current *PCM* implementation, the initial timeslice value is fifty. Thus at most 2% of the executions of a *recurse* instruction will require enqueueing and dequeueing a process record. In practice, guard suspension and execution of *halt* instructions often reset the time slice register before it reaches zero. For example, in the test program gauss, *recurse* instructions are never executed with a timeslice value of zero. Based on the frequency of use of the non-tail recursive case, the doubling of measurement overhead is not justified. Thus, frequency measurements of parallel compositions count the number of time a *halt* or a *recurse* instruction is executed.

5.5.2 Frequency measurement of choice composition

We now turn our attention to the execution of a choice composition. In addition providing *if/then* type conditional execution, the execution of choice composition enforces the synchronization requirements between concurrently executing *PCN* programs. Recall from Section 4.1.3 that a choice composition is constructed from one or more implications:

$\{?\ \mathsf{test}_{1,1},\ \ldots\ \mathsf{test}_{1,g_1}\ \text{->}\ \{||\ \mathsf{program}_{1,1},\ \ldots\ \mathsf{program}_{1,p_1}\},$

$\vdots\quad\vdots$

$\mathsf{test}_{n-1,1},\ \ldots\ \mathsf{test}_{n-1,g_{n-1}}\ \text{->}\ \{||\ \mathsf{program}_{n-1,1},\ \ldots\ \mathsf{program}_{n-1,p_{n-1}}\},$

$\mathsf{default}\ \text{->}\ \{||\ \mathsf{program}_{n,1},\ \ldots\ \mathsf{program}_{n,p_n}\}$

$\}$

Each implication has a guard, consisting of a sequence of guard tests, and a body, consisting of a single parallel composition. The guard of the last implication in a choice composition must contain a single *default* test. Corresponding to its structure, the execution of a choice composition takes place in two phases: guard execution and body execution. Since the body of an implication is always a parallel composition, the techniques of the preceding section apply. In this section, we focus on measuring the actions taken during guard execution. To understand the tradeoffs that can be made in measurement, it is first necessary to understand how guard evaluation is implemented.

5.5.3 Implementation of guard evaluation

The semantics of guard evaluation specify that each test within the guard of an implication is tried sequentially in the order in which the tests appear in the program text. However, with the exception of the *default* implication, which must be tried last, the guards themselves can be evaluated in any order. In fact, because guard tests do not alter the state of the computation, more than one guard can be evaluated simultaneously.

These semantics allow two different strategies for implementing guard execution: linear search and decision tree. As we will see, the implementation used has a direct effect on the measurement of guard execution.

Linear search compilation is the most direct method for implementing an implication guard. In this approach, an implication is selected and its guard is evaluated. If every test in the guard evaluates to true, then the body of that implication is executed and the execution of the choice composition terminates. On the other hand, if any test in the guard fails or suspends, another implication is selected and the process repeats. Linear search optimizes guard execution by trying the next implication as soon as the first test in a guard fails.

When the selection of implications is exhausted, the *default* implication

is executed. If all the guards fail, then the *default* succeeds and its body is executed. If at least one guard test suspends, the *default* test appends the process record onto the suspension queue to be tried again later.

A flow diagram of this scheme is shown is Figure 5.4. Each circle represents a guard test and arrows indicate possible flows of control. Guard execution starts with the first test of the first clause, indicated by the arrow in the upper left corner of the figure. If a test succeeds, execution continues by following the arrow to the right. If a test fails, the downward path is followed and the guard of the next clause is tried. With the exception of the *default* guard, which is tried last, the order in which the guards are evaluated is not necessarily the order they appear in the program text. If the *default* guard fails, the search is repeated using the algorithm of Figure 4.7.

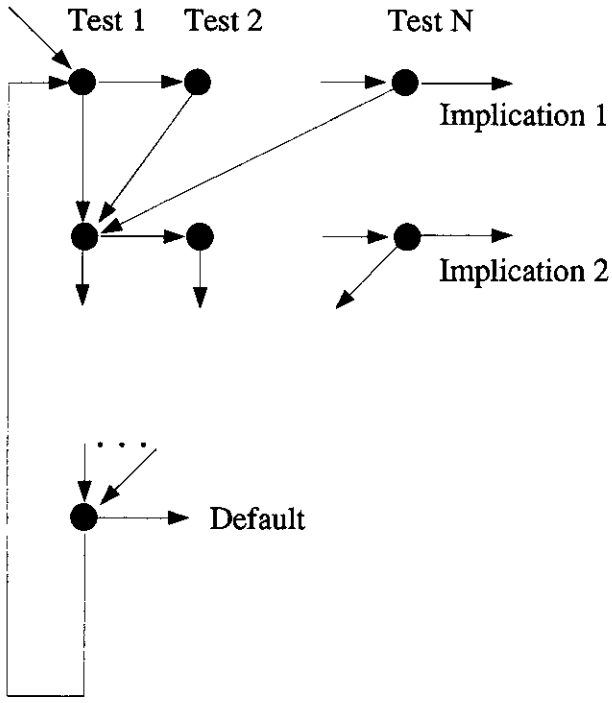

Figure 5.4: Flow graph of linear search implementation of guard evaluation

The *PCM* needs to support three operations to sequence the flow of control through the guards tests:

- jump to first test of the first clause
- execute the next test
- jump to the first test of the next clause

The first two operations are normal sequencing operations within the *PCM*. The first happens whenever a process record is taken from the active queue. The second is the normal next instruction sequencing that occurs as part of instruction execution. Jumping to the first test of the next clause is implemented by the *try* instruction, which stores the address specified in its argument *failure* register in the *PCM*. If a test fails, the contents of the failure register are copied into the program counter and execution continues.

While simple to implement, the linear search algorithm can waste time trying tests that can never succeed or are redundant. For example, consider the code fragment in Figure 5.5.

```
p(x)
{? x ?= ["a"] -> a_stuff(),
  x ?= ["b"] -> b_stuff(),
  x == [] -> c_stuff(),
  default -> skip()
}
```

Figure 5.5: A *PCN* program in which linear search executes tests that are redundant or tests that cannot succeed

The steps followed in a linear search implementation are shown in Figure 5.6. Note that the argument x is tested twice to see if it is a list. In addition, consider the situation where x is defined as a list whose head is undefined. In this case, the test in the third implication is executed, even though it is already known that it must fail.

Decision tree compilation [Tay89] is method of implementing guard evaluation that eliminates this type of inefficiency. In a decision tree implementation, tests in two or more guards are combined into a single test. The effect is to evaluate the guards in parallel. To combine type or equality tests, a combined test whose action resembles a case statement is used.

1. *Implication 1.* Check the value of x. If it is not defined to be a list, go to Step 3.

2. *Implication 1a.* Check the head of the list. If it is not defined to be the constant a, go to Step 3. Otherwise add a_stuff() to the active queue and terminate.

3. *Implication 2.* Check the value of x. If it is not defined to be a list, go to Step 5.

4. *Implication 2a.* Check the head of the list. If it is not defined to be the constant b, go to Step 5. Otherwise add b_stuff() to the active queue and terminate.

5. *Implication 3.* Check the value of x. If it is defined to be the empty list add c_stuff() to the active queue and terminate. Otherwise go to Step 6.

6. *Default.* If any test could not proceed because it required a value that was undefined, place the choice composition back onto active queue and terminate. Otherwise, just terminate.

Figure 5.6: Linear search implementation of sample program

The control structure that results from combining tests in different implications is a decision tree. At each leaf of the tree a single non-*default* implication is identified. Since it is possible to identify a single implication without trying all the tests in the guard, any remaining tests at the leaf are tried as in a linear search implementation. If a test fails at any point in the decision tree, the *default* implication is executed.

A general flow graph of the decision tree approach to guard execution is shown in Figure 5.7. The dots indicate decision points while the triangles indicate a subtree. The tests remaining after a single implication is identified are represented by the rectangles at the leaf of the tree. Note that any point during execution, control can jump to the *default* case.

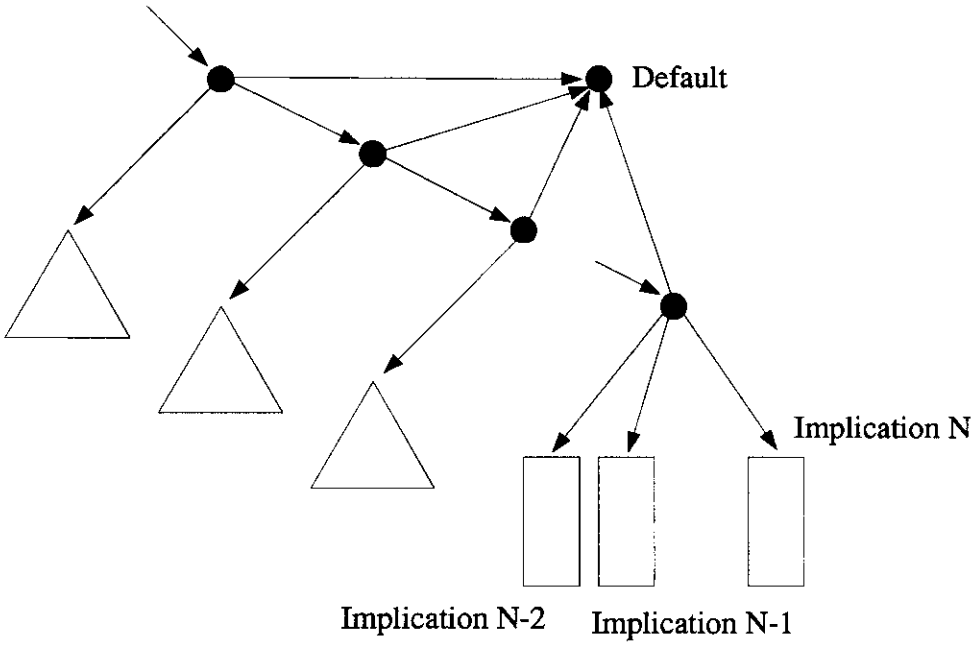

Figure 5.7: Flow graph of decision tree implementation of guard execution

Figure 5.8 shows the steps followed in a decision tree implementation of the code in Figure 5.5. The redundant tests on the first argument are eliminated by combining the two list tests and the empty list test into a single multiway branch whose target depends on the type of the first argument.

A decision tree implementation reduces the execution time of an implica-

1. *Test type.* Check the value of x. If it is an empty list go to Step 3. If it is a list go to Step 2. Otherwise go to Step 4.

2. *Test head.* We know the first argument is a list; extract the head of the list and check its value. If it the constant a or b add a_stuff() or b_stuff() respectively to the active queue and terminate. Otherwise go to Step 4.

3. *Check null.* Add c_stuff() to the active queue and terminate.

4. *Default.* If any argument was not defined, place process record back onto active queue. Then terminate execution.

Figure 5.8: Decision tree implementation of sample program

tion by reducing the number of guard tests which need to be tried. However, the sequencing operations in decision tree compilation are more complex than in linear search and correspondingly, the cost of sequencing through these tests is greater than in linear search. The control operations required for a decision tree implementation are:

- jump to first test of the first clause
- execute the next test
- based on data type or value, compute an index into a jump table, and jump to that location

In addition to the normal sequencing operations within the *PCM*, two additional instructions are used in decision tree compilation. These instructions are: *switch_on_type* and *switch_on_value*. The *switch_on_type* instruction branches to an address based on the data type of its argument. The *switch_on_value* instruction uses a hash table to map the value of its argument into an address to branch to. Clearly, the amount of computation

required to implement the *switch* instructions will be higher than that required to implement the *try* instruction.

Because of the complexity and overhead associated with the *switch* instructions, the most efficient execution is obtained by combining both compilation methods, using the decision tree method only to the extent that the decrease in execution time from reducing the number of redundant tests executed offsets the increased cost of the switch instructions [TF88].

5.5.4 Profiling guard evaluation

With an understanding of how the guards in a choice composition are implemented, we turn to the problem of profiling their execution. Our goal is to be able to measure the operations which have been performed during the execution of a choice composition. A second objective is to investigate the impact which the two different guard implementation techniques have on measurement. Understanding the relationship between implementation and measurement is important if performance observability is to be included as factor in the tradeoffs made when designing a *PCN* implementation.

Our initial approach to measurement is to view the execution of a choice composition as a unit rather than an implication at a time. From this view, the frequency profile of a choice composition determines the number of times each execution path in Figures 5.4 or 5.7 is taken. Since decision tree compilation takes a composition wide view of guard execution, starting with this high level view of a profile facilitates a comparison of the two implementation methods.

As we see from the flow graphs, there are many different execution paths that can be taken during guard execution. The number and length of these paths determines on the execution time and storage space overhead of measurement.

Consider a linear search implementation of a choice composition, C, with $i(C)$ implications: $I_1, \ldots I_{i(C)}$. Each implication, I_k, has a guard consisting of $g_C(I_k)$ guard tests. One storage location is required to record the use of each path. Therefore, the amount of storage needed, $S^{ls}_{measurement}(C)$ depends on the total number of different execution paths. For implication k, the number of paths resulting in successful guard execution depends on the number of failure paths in implications 1 through $k-1$. The number of

failure paths out of implication k are the product of the number of failure paths into I_k and the number of guard tests in I_k. Thus we have:

$$S^{ls}_{\text{measurement}}(C) = \Omega_{ls}\left[\prod_{k=1}^{i(C)} g_C(I_k) + \left(\sum_{k=1}^{i(C)}\prod_{j=1}^{k-1} g_C(I_j)\right)\right] \tag{5.4}$$

where Ω_{ls} is the amount of storage required to record the number of times a single path is used and $\prod_i^j = 1$ if $j \leq i$.

By recording the identity of each guard test that fails, the execution path through Figure 5.4 can be determined. Therefore, $T^{ls}_{measurement}$, depends on the number of tests that have failed and the number of paths executed. If a flow graph exits on implication I_k there must have been $k-1$ test failures. Thus for a single execution of a composition, the time spent in measurement is:

$$T^{ls}_{measurement}(C,k) = k\Theta_{ls} \tag{5.5}$$

where Θ_{ls} is a the amount of time it takes to record the occurrence of a guard test failure.

Now let us look at decision tree compilation. Assume that the decision tree is applied to the extent that each leaf of the tree points to a single non-default implication. The number of nodes in the tree, $n(C)$, is determined by the number of internal nodes, $n_i(C)$, and the number of leaf nodes. Because the *default* implication is not a leaf node,

$$n(C) = n_i(C) + i(C) - 1$$

Since the *default* implication is accessible from each decision point as well as after each test at a leaf node, the number of paths in a decision tree implementation is determined by the number of nodes in the tree and the number of guard tests that occur at each leaf. Therefore, the storage overhead for measuring the path taken in a decision tree implementation is:

$$S^{dt}_{measurement}(C) = \Omega_{dt}\left[2\left(n_i(C) + \sum_{k}^{i(C)-1}\lambda(I_k)\right) + i(C) - 1\right] \tag{5.6}$$

where $\lambda(I_k)$ is the number of tests remaining at the leaf for implication I_k and Ω_{dt} is the space required to record the occurrence of a single path. Since

the storage location serves the same purpose in Equations 5.4 and 5.6, it is reasonable to assume that $\Omega_{ls} = \Omega_{dt}$. We will no longer distinguish between the two and use Ω instead.

The expression for the runtime overhead for recording a path in a decision tree implementation is very simple. Because of its structure, the path taken through the decision tree can be completely determined by knowing the single guard test that failed. Thus the runtime overhead is a constant:

$$T^{dt}_{measurement}(C) = \Theta_{dt} \tag{5.7}$$

5.5.5 A comparison of the storage overhead for linear search and decision tree implementations

Let us now compare linear search and decision tree implementations in terms of measurement overhead. We start with the space requirements for measurement. In the worst case a decision tree can eliminate only one implication per level. The number of internal nodes in this tree will be $i(C) - 1$ and $n(C) = 2i(C) - 2$. Since $\lambda(I_k) < g(I_k)$ we can bound the storage requirement by:

$$S^{dt}_{measurement} \leq S\left(5i(C) - 5 + 2\sum_{k=1}^{i(C)-1} g(I_k)\right) \tag{5.8}$$

Knowing that $I_{i(C)}$ is the *default* implication and therefore $g(I_{i(C)}) = 1$, we can compare the space requirement for the two implementation methods by:

$$\delta(S) = \Omega\left[5i(C) - 5 - 2\prod_{k=1}^{i(C)-1} g_C(I_k) + \sum_{k=1}^{i(C)-1}\left(2g(I_k) - \prod_{j=1}^{k-1} g_C(I_j)\right)\right] \tag{5.9}$$

When the value of Equation 5.9 is negative, decision tree compilation will require less space to store an execution profile than a linear search implementation. This equation is dominated by the product terms and if a guard has more than one test, decision tree compilation will require less storage space for measurement than linear search. While the exact form of a program is needed to make a concrete statement, an examination of the 2300 different

implications in the test programs shows that 38% have two or more tests in their guards. From this, we can generally expect Equation 5.9 to be negative.

A true comparison of linear search and decision tree implementations can be made by computing the storage overheads for the programs of Table 5.1. Assuming that Ω is one heap cell, Equations 5.4 and 5.6 are evaluated for each choice composition in the test programs. The value of S_{PCN} for a linear search implementation is determined by compiling the programs. Obtaining S_{PCN} for a decision tree implementation is more difficult because a decision tree compiler does not exist for *PCN*. While a decision tree implementation generates fewer instructions per choice composition, the inclusion of a jump table increases the size of the individual instructions. Therefore, it is reasonable to use S_{PCN} from the linear search compiler as an approximation for the decision tree value. We can now evaluate Equation 5.2 for both implementations. The results are found in Figure 5.9.

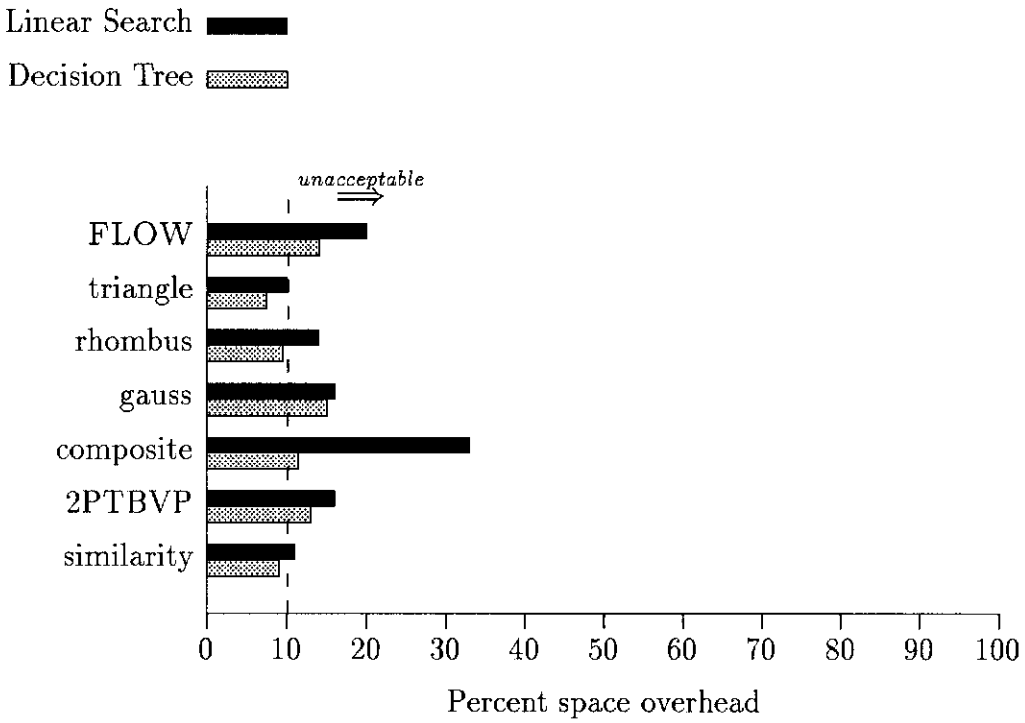

Figure 5.9: Storage overhead required to profile guard evaluation on the basis of execution path.

The minimum storage overhead for linear search is 10% while the largest shown is 33%. In Figure 5.9, the linear search data does not include the execution paths through three of the programs in FLOW. These three programs have a large number of paths and including them increases storage overhead for FLOW to over 180,000%. Thus a real program can have a storage overhead that is so high as to make execution path measurement infeasible.

The storage overhead for decision tree compilation ranges from 7.4% to 15%. The number of paths in FLOW are reduced to 1647 total. In every case, using decision tree compilation reduces the storage overhead of performance measurement. These results support our claim that decision tree compilation will in general require less storage space for an execution profile.

5.5.6 Comparing the runtime overhead of linear search and decision tree implementations

The total time spent recording the execution path through a choice composition will alway be less in a decision tree implementation than in a linear search implementation. Because it is easier to record the identity of path through a decision tree, $\Theta_{dt} \leq \Theta_{ls}$ and Equation 5.5 will bound Equation 5.7. However, since decision tree compilation can reduce the total execution time of the composition, it is not necessarily the case that the percentage of runtime overhead will be less.

To apply Equations 5.5 and 5.7, the values of Θ_{ls} and Θ_{dt} must be known. Measurement from an appropriately instrumented *PCM* is the only way to obtain meaningful values for these. Since the measurements made to determine Θ_{ls} and Θ_{dt} determine $T^{ls}_{measurement}$ and $T^{dt}_{measurement}$ as well, we measure $T_{measurement}$ instead of applying the equations.

To obtain the measurement for execution time overhead, the test programs are executed on a single processor of a Sequent Symmetry, a shared memory parallel computer based on the Intel i386 microprocessor. Because the execution is on one processor, both $T_{communication}$ and T_{idle} are zero. By executing each program with and without measurement, the values of $T_{measurement}$ and T_{PCN} are obtained. Using Equation 5.1, the results summarized in Figure 5.10 are computed.

For the linear search case, the *PCM* was modified to keep a list of the

guard tests that failed. When the flow graph is exited, this path is used as an index into a hash table and the contents of the storage location allocated for the path is incremented. Runtime overheads range from 0.87% to 3.1%.

Because a decision tree implementation does not exist, it is not possible to obtain actual decision tree overhead data. As an approximation, we recorded only the exits from a linear search flow graph. Since the objective of a decision tree implementation is to reduce execution time, it follows that this approximation provides a lower bound. The measured values range from 0.31% to 1.7%.

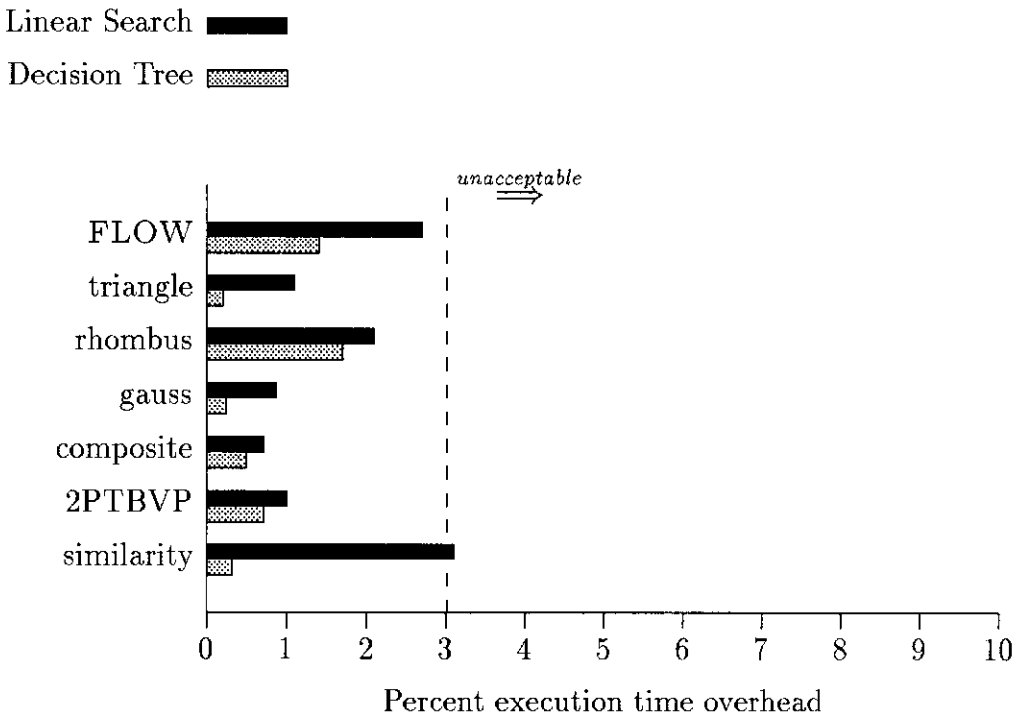

Figure 5.10: Runtime overhead for measurement of execution paths

The difference in runtime overhead between a linear search and decision tree implementation is largest for the programs triangle and similarity, with a overhead ratio of 5.5 and 10 respectively. In both of these programs, a choice composition with 5 to 16 implications is heavily used. These compositions are considered large; a choice with 5 implications is in the 99^{th} percentile with respect to choice composition size. In addition, these choice compositions

select a implication based on an equality test and the decision tree implication is very effective in these cases. This explains the high overhead ratios for similarity and triangle.

Finally we note that the values of Θ_{ls} and Θ_{dt} can be computed from the runtime overheads, the number of times each path was taken and Equations 5.5 and 5.7. We find that $\Theta_{ls} = 1.54 \times 10^{-6}$ seconds and $\Theta_{dt} = 4.28 \times 10^{-7}$ seconds. As we expected, $\Theta_{dt} < \Theta_{ls}$.

An interesting observation about the decision tree implementation can be made at this point. It has been our experience that obtaining the performance of a program is more difficult in an optimized system than in an unoptimized system. Optimization tends to make it harder to get at the details of execution. Happily, the reverse is true for an implementation based on decision trees. Because of the reduced number of alternative paths in guard evaluation, the overhead in measuring guard execution is reduced.

We have shown how the implementation of choice composition has a dramatic impact on the cost of execution profiling. Even if there were no advantages, Principle 3 would dictate that a decision tree implementation be used. However, the implementation complexity has precluded its use in *PCN* to date. We expect that as the *PCN* system matures, a decision tree implementation will be undertaken.

5.5.7 Reducing the measurement overhead

The overhead for linear search shown in Figures 5.9 and 5.10 exceeds our design goals. In this section, we investigate ways in which these costs can be reduced. Given that linear search will always be a component of any *PCN* implementation, such reductions are important. By applying Principles 1 and 2, it is possible to reduce the overhead of measuring a linear search implementation to that of decision tree compilation and less.

The decision tree implementation technique views choice composition as a unit; as such, it is amenable to producing a profile based on execution paths. In contrast, the linear search technique views a choice composition as a collection of implications where each implication is compiled as a separate entity. For this reason, a composition wide view of execution is costly to obtain.

A more appropriate approach for linear search is to measure the execution path within each implication separately. For each implication in a

choice composition, the number of times each guard test fails is measured. We call this the implication local execution path. Although the number of *PCM* instructions executed is still determined exactly, information about how implications interact is lost.

The space required for measuring the execution path within an implication depends on the number of tests in the implication guard: one storage location is required for each test. Given that implication $I_{i(C)}$ is the *default* implication and has single test, the space overhead is:

$$S^{ls'}_{measurement} = \Omega\left(1 + i(C) + \sum_{k=1}^{i(C)-1} g_C(I_k)\right) \tag{5.10}$$

Comparing Equations 5.6 and 5.10 we obtain:

$$\delta(S^{ls'}) = 2n_i + 2\sum_{k=1}^{i(C)-1} (2\lambda(I_k) - g_C(I_k)) - 2 \tag{5.11}$$

where the value of $\delta(S^{ls'})$ is negative whenever the implication local execution path through a linear search implementation requires more storage space than recording the execution path through a decision tree implementation. While

$$\sum_{k=1}^{i(C)-1} \lambda(I_k) < \sum_{k=1}^{i(C)-1} g(I_k)$$

always holds, $\sum_{k=1}^{i(C)-1} g(I_k)$ must be less than twice the size of $\sum_{k=1}^{i(C)-1} \lambda(I_k)$ in order for $S^{ls'}_{measurement} \leq S^{dt'}_{measurement}$. Our experiments show that this condition does not hold in practice.

Returning to programs of Table 5.1, we use Equations 5.10 and 5.2 to compute the storage overhead for implication local execution paths. These results are shown in Figure 5.11. The space overhead ranges from 8.2% to 16% and is within 14% of SO_{dt} in all cases.

The runtime overhead depends on the number of implications tried and Equation 5.5 still holds. However, less work is required to identify a path within an implication than within a composition. Consequently, we find that $\Theta'_{ls} \leq \Theta_{ls}$.

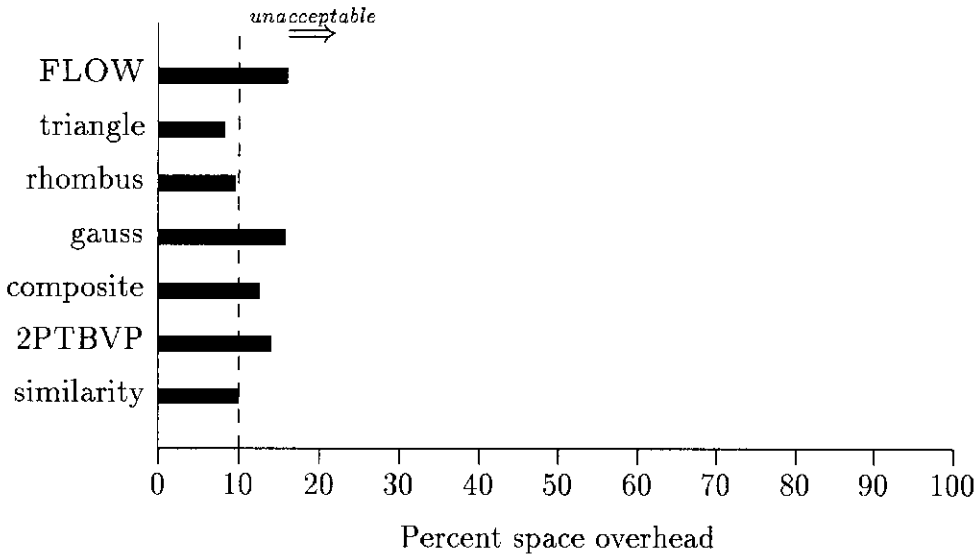

Figure 5.11: Storage overhead for measuring implication local execution path

5.5.8 Statistical models of guard execution

Although we were able to reduce the measurement overhead in a linear search implementation, the level of overhead still exceeds that of a decision tree implementation as well as the design goals. A further reduction in measurement overhead can be obtained by simplifying the structure of the linear search flow graph and applying Principle 2.

Such a simplified graph is shown in Figure 5.12. This graph is obtained from the one in Figure 5.4 by collapsing all the guard test nodes within an implication into a single node. The collapsed node appears as a rectangle in the flow graph of Figure 5.12. The last implication has only a single *default* test and it remains a circle. All arcs pointing to the right represent the flow followed by successful guard execution, while the downward pointing arcs indicate that a guard test in an implication has failed. The guard test that causes the failure is not observable from the simplified flow graph.

Extending the idea of implication local paths to the simplified graph, the measurement procedure consists of determining the number of times each arc in Figure 5.12 is traversed. In other words, the measurements count the number of successes and failures that occur in each implication guard.

As with the previous measurements, $S^s_{measurement}$, the space required

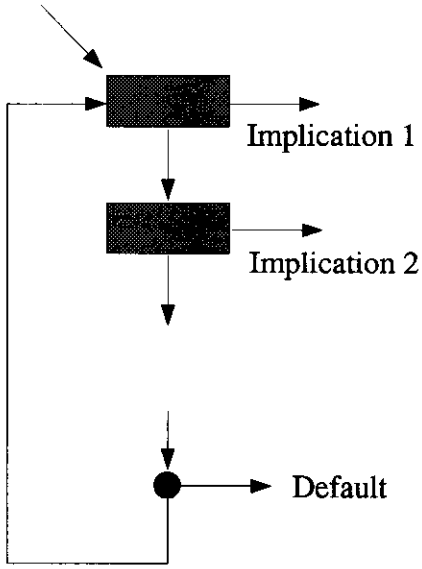

Figure 5.12: A simplified flow graph for linear search implementation of guard evaluation

for measuring the execution path through the simplified flow graph, is dependent on the number of paths through the flow graph. This would lead one to the conclusion that the space overhead for measurements based on 5.12 is:

$$S^s_{measurement} = 2\Omega i(C)$$

However, as a result of the structure of the simplified flow graph, the number of traversals of some arcs can be deduced from other arcs. Storage locations do not need to be allocated for those arcs whose traversal count can be deduced.

For implication I_k of choice C let $\nu^s_C(k)$ and $\nu^f_C(k)$ be the number of traversals of the right pointing (successful guard execution) and down pointing (failed guard execution) arcs respectively. Also let $\nu^s_C(0)$ be the number of traversals of the entry arc in the upper left hand corner of the flow graph. An examination of the flow graph of Figure 5.12 yields the following recurrence:

$$\nu^f_C(i(C)-1) \;=\; \nu^f_C(i(C)) + \nu^s_C(i(C)) \qquad (5.12)$$

$$\nu_C^f(i-1) = \nu_C^s(i) + \nu_C^f(i) \quad (5.13)$$

$$\nu_C^s(0) = \nu_C^s(1) + \nu_C^f(1) - \nu_C^f(i(C)) \quad (5.14)$$

From this recurrence, we see that if for every implication the number of successful guard executions is known, the number of failed executions can be determined and vice versa. In either case, both the number of successful and unsuccessful executions of the *default* test must be available.

However, if the number of procedure entries is known, only $\nu_C^s(i(C))$ or $\nu_C^f(i(C))$ is needed, not both. The value of $\nu_C^s(0)$ can be computed by summing the number of invocations of all parallel compositions that call C. Based on this analysis, we see that the space overhead for the measurement based on the simplified flow graph is actually:

$$S^s_{measurement} = \Omega i(C) \quad (5.15)$$

Comparing Equation 5.15 with Equation 5.6, we see that

$$S^s_{measurement} \leq S^{dt}_{measurement}$$

This fact is borne out by Figure 5.13, which shows the storage overhead in the test programs for measurement based on a simplified flow graph. The overhead measured ranges from 3.8% to 6.3%. This falls within the range of storage overhead in the design goals for the measurement procedure.

From the perspective of storage overhead, it does not matter if the number of guard failures or guard successes are measured. However, the choice of measurement does make a difference when runtime overhead is being determined. If failures are measured, then the runtime overhead for the simplified graph is the same as that in Equation 5.5. One failure must be recorded for each implication tried. Alternatively, if successes are recorded, only one measurement is made per traversal of the flow graph. In this case the execution time overhead is a constant:

$$T^s_{measurement} = \Theta_s \quad (5.16)$$

With the simplified model, we have achieved our goal in making the measurement overhead of linear search as low as for decision tree compilation. The measured runtime overhead for the test programs, which ranges from 0.24% to 1.7%, is shown in Figure 5.14. The highest runtime overhead is

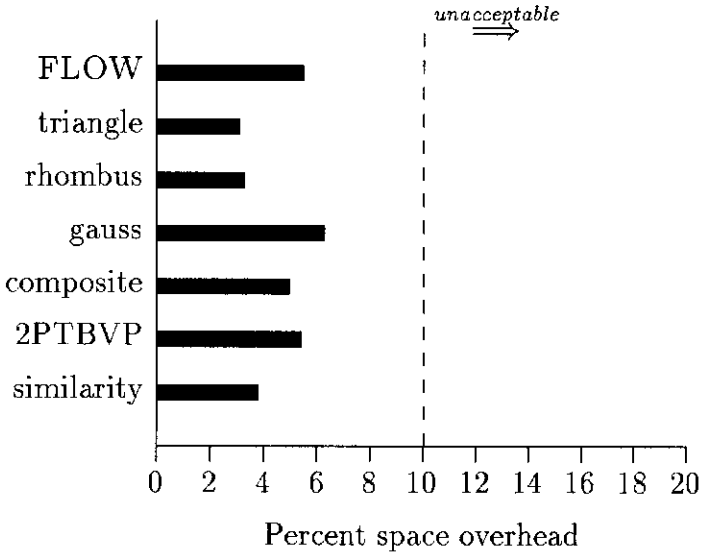

Figure 5.13: Space overhead with measurements based on a simplified flow graph

1.7% for the rhombus program. We recall that the overhead statistics are computed only from the *PCN* components of the multilingual programs. When the sequential components are included, the runtime overhead will decrease by at least one order of magnitude.

5.5.9 Accuracy of the simplified performance model

In the previous section, a method was developed to reduce the measurement overhead of linear search to that of a decision tree implementation. This method also limits the accuracy with which the measurement can be made. In this section, we address the issue of measurement accuracy.

The degree to which a measurement process based on a simplified flow graph will be in error depends on two factors: 1) the number of tests executed in guards that fail compared to the total number of guard tests executed overall and 2) the manner in which we account for activity within a guard that fails. We investigate each of these factors in turn.

The first consideration is how much error is present in a measurement based on the simplified flow graph. If the total number of guard tests executed by a program is g_t, the number of tests executed in a guard that

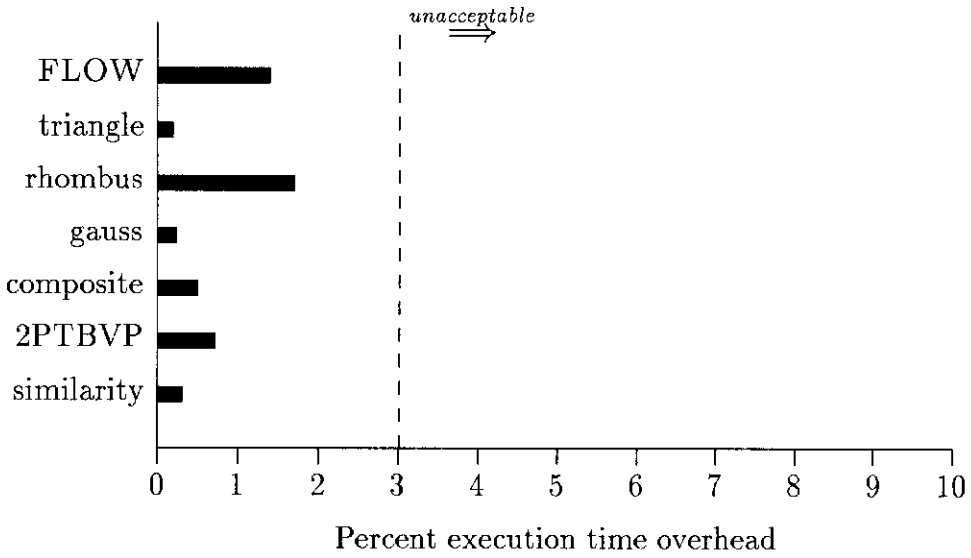

Figure 5.14: Time overhead with measurements based on a simplified flow graph

eventually fails is g_f and the number of tests assumed executed in failed guards is $\hat{g}_f$, then the measurement error is defined by:

$$Error = \frac{g_f - \hat{g}_f}{g_t}$$

Assume that the k^{th} test of an implication guard fails. If $k < i(C) - k$, the maximum possible measurement error is obtained by assuming that the first guard test failed; if $k \geq i(C) - k$, the maximum measurement error is obtained by assuming that the last guard test has failed. Using these values for the number of tests executed in a failed guard, the maximum possible measurement error for a computation can be determined.

To obtain the maximum measurement error for the test programs using the simplified flow graph, the per implication execution path must be measured. Because guard failure is affected by the availability of data, the programs must be run in parallel for the results to be meaningful. This is accomplished by a specially instrumented *PCM* executing on a shared memory parallel. Each *PCM* in the computation shares a small amount of memory with a monitor process which executes concurrently on another processor. For every guard failure, an integer that identifies the test that failed is writ-

ten into the shared memory. The monitor program observes the data value and collects the test failure statistics. Interfacing to the monitor process increases the size of the *PCM* by 0.2% and increases the execution time of the programs by less than 2%. With the *PCM*/monitor pair, guard failures can be measured with minimum impact on the behavior of the program.

The maximum error calculation was performed for the seven test programs and is summarized in Figure 5.15. The maximum error of 47%, which occurs in FLOW, exceeds the design goal of 20% accuracy. In the next section, we will describe a technique to reduce the measurement error.

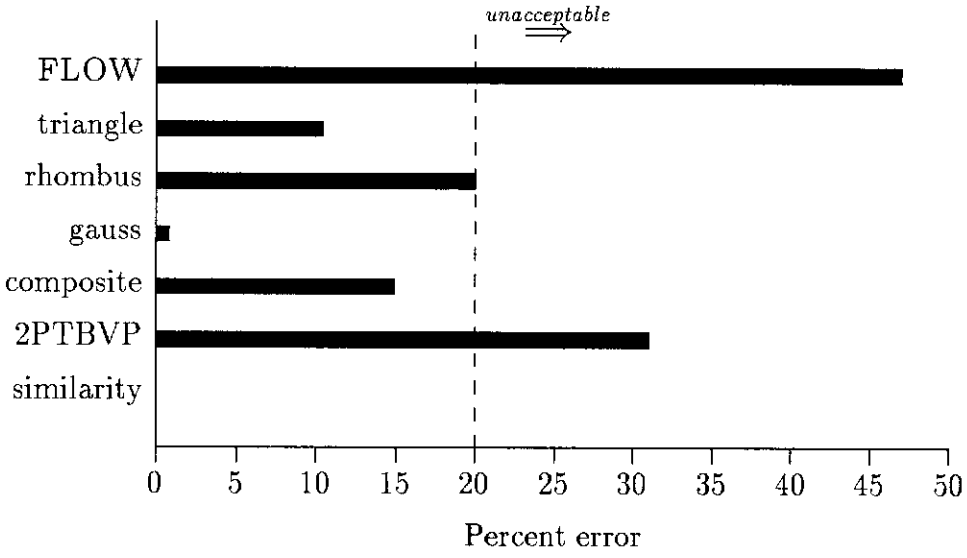

Figure 5.15: Maximum possible error in guard execution

5.5.10 Reducing the measurement error in the simplified flow graph

The amount of error in a measurement based on the simplified flow graph is determined by the method used to account for activity within failed implication guards. Generally speaking, less then 50% of the guard tests executed occur in a failing guard. As illustrated in Figure 5.15, this limits the maximum error to less 50% as well. In this section, we investigate a method to reduce the measurement error to under 20%.

One approach to reducing measurement error is to take advantage of static program analysis or *a priori* knowledge of program functionality or input. For example, if the frequency of successful execution is known for all of the implications in a choice composition, the compiler can arrange to try the most successful guards first. This reduces the number of tests executed in a failed guard and consequently reduces the measurement error. This type of improvement is especially interesting because the frequency of successful execution is a performance index included in the execution profile — profiling can be used by the compiler to improve the accuracy of profiling. However, because of its reliance on prior knowledge, we consider a different approach.

The approach we now present reduces the measurement error by using a probabilistic model of guard behavior to make an guess at which test has failed. In order for this model to be generally applicable, it cannot rely on any information about the program other than the instruction sequence generated by the compiler and properties of *PCN* in general.

In the simplified flow graph model of Figure 5.12, the number of times a guard fails is known. If $\nu_C^f(k)$ is the number of times implication I_k fails and $\nu_{g_C(I_k)}(j)$ is the number of times the j^{th} guard test in I_k fails then:

$$\nu_C^f(k) = \sum_{j=1}^{g_C(I_k)} \nu_{g_C(I_k)}(j)$$

Our objective is to devise a function $\hat{\nu}_{g_C(I_k)}(k, j)$ to approximate the actual value of $\nu_{g_C(I_k)}(j)$.

For a single failed execution of I_k, the test that caused the failure is represented by an integer valued random variable **g**. The value of **g** ranges from 1 to $g_C(I_k)$ and is described by the discrete probability density function $f_{\mathbf{g}}(g)$, where g is the parameter of the function. In reality, the exact form of $f_{\mathbf{g}}(g)$ depends on the program in which the guard occurs and the values of the arguments that the implication will process. However, in order to be practical as a measurement tool, we construct an approximation of the density function based solely on the structure of the guard and a general characterization of *PCN* execution.

To obtain a density function for guard failure, we characterize the behavior of the each guard test in *PCN*. By combining the characterizations of the tests in a guard, the density function for the entire guard is determined. Our method relies on two assumptions:

- The behavior of any use of a test in a guard can be characterized in terms of the behavior of that test in all guards.
- The behavior of tests in a guard is independent.

The behavior of a test is characterized by the probability that the test fails whenever it occurs within a guard that fails. In other words, the behavior of a guard test in *any PCN* program is determined by the conditional failure probability:

$$P\{\text{test } g \text{ makes } I \text{ fail} | I \text{ fails}\}$$

Using the assumption that the conditional failure probabilities for guard tests are independent, the probability that a guard fails on the j^{th} test is equal to the probability that test $j-1$ executes successfully *and* test j fails. Thus we define $f_{\mathbf{g}}(g)$ by:

$$f_{\mathbf{g}}(g) = \left(\prod_{i=1}^{g-1} (1 - P\{i^{\text{th}} \text{ test fails} | I_k \text{ fails}\}) \right) P\{g^{\text{th}} \text{ test fails} | I_k \text{ fails}\} \tag{5.17}$$

Using this density function, an estimate of the distribution of test failures can be obtained by:

$$\hat{\nu}_{gc(I_k)}(k, g) = f_{\mathbf{g}}(g) * \nu_C^f(k) \tag{5.18}$$

A few comments on the estimation process are in order. The assumption that the conditional failure probabilities of the tests in a guard are independent is not always true. In particular, guard tests used to extract elements from structured data are likely to be dependent on one another. Equation 5.17 can be extended to correctly handle such cases. However, as we shall see shortly, the equation produces adequate results without the additional complication of introducing dependent probabilities.

The other important assumption we are making is that a single failure probability can characterize the behavior of any test in any implication guard. While undoubtedly there are circumstances where this assumption will not hold, we find that many uses of tests are stereotypical. This makes their behavior predictable validating our assumption. Loops are a primary source

of predictability. In addition, many *PCN* programs are constructed using a small number of programming techniques [TF89]. This adds to the ability to characterize guards. A precedent to our use of branching probabilities can be found in compilers such as [Ell85], where a statistical characterization of test behavior is used to improve the code being generated.

It is important to remember that the job of the estimator is not to predict when a guard will fail, but rather to predict where it failed given that a failure has occurred. Knowing that clause guard failed or succeeded limits the range of possible behavior available to a guard and reduces amount of variation caused by data dependence.

A technique similar to ours is reported in [ME69]. In this work, branching probabilities are used to calculate the mean path length through a graph representation of a program. Although solving a similar problem, the restricted structure of our graphs leads to a simpler formulation of the density function.

The conditional failure probabilities of the test instruction in the *PCM* must be known to apply Equation 5.18. Using the instrumented *PCM* described in Section 5.5.9, the failure statistics for each test instruction were measured and the conditional failure probability computed. The results of these measurements can be found in Appendix C.

Returning to the test programs, Equation 5.18 is applied to the simplified flow graph measurements obtained when the programs execute in parallel. Figure 5.16 shows the resulting measurement error which ranges from .001% to 19.9%. In all cases, the error is within the accuracy goal for our measurement procedure.

The highest measurement error occurs in the FLOW program. Most of the measurement error can be attributed to a single implication in the program. The implication in question has eight tests, which places it in the 98.8% percentile in terms of guard length. Equation 5.18 overestimates the number of failures of the first guard test by a factor of two and the exceptional length of guard magnifies the effect of this error. A closer examination reveals that the first test in the guard is a continuation test for a loop. Since *PCN* does not have an explicit iteration notation, this use of a test is not distinguished. However, if this use of a test can be detected, a lower failure probability would have been obtained and the total error reduced.

Situations with the potential for large amounts of measurement error can be detected by computing the maximum measurement error from the

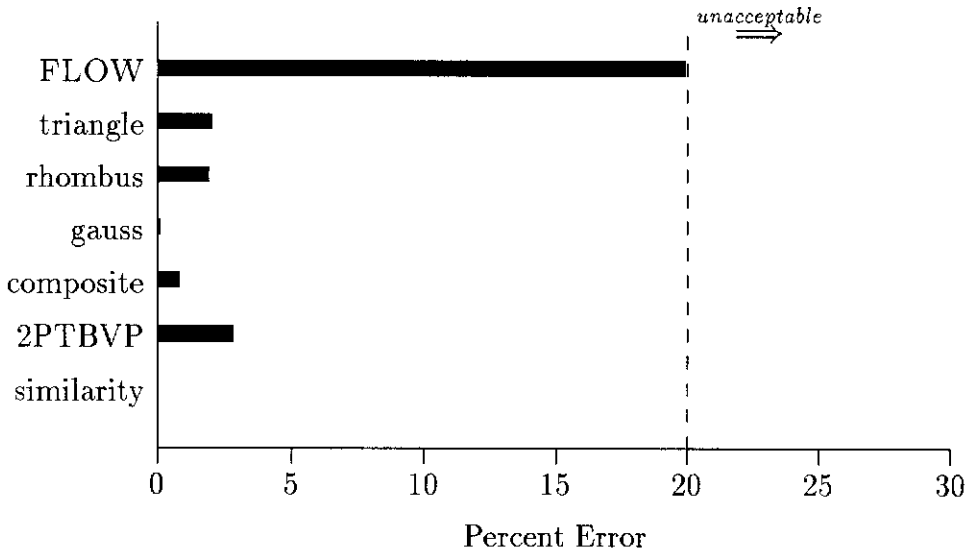

Figure 5.16: Measurement error using conditional failure probabilities to estimate the guard test that fails

simplified flow graph measurements and the probabilistic model of guard failure. For each implication, the probabilistic model is applied and the procedure of Section 5.5.9 is used to determine the maximum error in the number of tests executed within a failed guard. Dividing this number by the minimum number of tests executed in the implication yields the maximum possible relative error.

This concludes the discussion of frequency measurement of choice composition. In the next section, we consider the remaining parts of the core *PCN* language.

5.5.11 Assignments and definitions

Assignments and definitions always occur within the context of a parallel composition. The number of times a specific assignment or definition executes is determined via the measurement techniques of Section 5.5.1. There are, however, additional characteristics of assignment and definition that are of interest.

If the right side of a definition or assignment is a mutable variable, then as described in Section 4.1.3, a snapshot of the variable's value must be taken.

The *copy_mutable* instruction makes a snapshot by creating a new copy of a mutable variable on the *PCM* heap.

In some instances, it is possible to determine the size of a mutable to be copied by analyzing the text of the *PCN* program. When this is the case, the behavior of an assignment or definition can be completely determined by measuring the number of times it executes. However, because definitional variables are not typed, it is not generally possible to determine the amount of data that must be copied prior to actually performing the operation. In such cases, an important aspect of assignment and definition is not known.

Because the cost of snapshotting has an impact on deciding to make a variable mutable or definitional, the amount of copying performed needs to be recorded as well as the number of copy operations. A single storage location is allocated to record the number of heap cells copied during an assignment or definition. When the copy is performed, the size of the data structure on the right side is determined and recorded in the storage location. Thus copying can be measured with a constant space cost of Ω and a constant time cost of Θ_c.

5.5.12 Foreign programs

The final component of core *PCN* is foreign language programs. The semantics of *PCN* views a foreign program as a black box; there is no concern with its inner workings. This philosophy is extended to profiling as well; our measurements do not attempt to capture what goes on within a foreign program. From the perspective of a frequency measurement, this limits us to determining the number of times a foreign program is called. Since foreign programs are alway called from a parallel composition, the execution frequency of a foreign program can be determined indirectly from the execution frequency of compositions that call the foreign program. No additional measurements are required.

This concludes the discussion of measuring a program's execution frequency. We now consider how to measure the execution time of a program.

5.6 Measuring Execution Time

In this section, we examine methods for measuring the time spent executing program code. Techniques for measuring the amount of time a computation spends communicating and idle are discussed in Section 5.7. We proceed with a discussion of measuring the time spent executing *PCN* code followed by a discussion of how to measure the time spent executing foreign programs.

5.6.1 *PCN* procedures

Our first concern in measuring the execution time of *PCN* code is to choose between a direct or indirect methods of measurement. The execution of *PCN* code can be split into the time spent executing compositions and the time spent executing assignments and definitions. Recall from Section 5.1.3 that if the time duration of an activity too small, direct measurement is not applicable. The size of the smallest possible *PCN* program, one *PCM* instruction, falls below this threshold. In additional consideration is that direct measurement effectively doubles the measurement's runtime overhead. For these reasons, the execution time of *PCN* code is measured indirectly. The method used is based on Equation 5.3. Approaches similar to ours can be found in [GK87] and [Sar89].

The time spent executing a composition depends on the number of times each instruction in the composition executes. Restating Equation 5.3, if n_{i_k} is the frequency of occurrence of instruction k in a composition, and there are K different instructions in the *PCM*, then the execution time of the composition is given by:

$$t_C = \sum_{k=0}^{K} n_{i_k} t_{i_k} \tag{5.19}$$

The execution time of a definition or assignment will depend on two factors: the fixed cost associated with starting the copy operation and a variable part whose cost will depend on amount of data to be copied. The fixed cost part of definition or assignment is already accounted for in Equation 5.19. Adding the variable part to the model we have:

$$t_p = n_c t_c + \sum_{k=0}^{K} n_{i_k} t_{i_k} \tag{5.20}$$

where n_c is the amount of data copied and t_c is the per data element cost of copying.

5.6.2 Validation of the *PCN* execution time model

The cost parameters t_c and t_{i_k} in Equation 5.20 must be obtained before the cost model can be used. The values of these parameters will depend on the parallel computer on which the computation executes. We note that the architectural component of a performance experiment, $\mathcal{A}$, is comprised of these cost parameters. In this section, we describe the process by which the values of the parameters in the execution time model are determined.

The basic approach is to execute a set of calibration program and measure the execution time of each program. Each calibration program is a parallel *PCN* application. After subtracting the time spent in foreign programs, the parameter values can be determined from Equation 5.20 and the instruction frequencies via linear regression. Once the parameter values have been obtained, the execution time of any *PCN* program can be determined by applying Equation 5.20.

There are 38 different instructions in the *PCM*. Including the copying cost parameter, a total of 39 parameters must be determined in the execution time model. Since regression requires an overdetermined system of equations, at least 39 test programs must be run. In practice, we find that closer to 100 programs are required for good results. Obtaining and executing such a large number of test cases is awkward and it is advantageous to reduce the number of parameters in the execution time model.

By grouping instructions with similar execution times into an *instruction class*, the number of parameters in the execution time model can be reduced. The following procedure is used. A rough estimate of the execution time for each *PCM* instruction is obtained with a specially instrumented *PCM*. The *PCM* determines the average execution time of each instruction with a microsecond timer. This measurement is performed on a single processor and measurement overhead is not an issue. The estimated times are only used to for instruction classification; they can *not* be used to determine execution

time.

Classes of instructions are formed using the *k-means* clustering algorithm [JD88]. For a given number of instruction classes, the k-means algorithm assigns instructions to classes in a manner that minimizes the variance of the execution times within an instruction class. In Figure 5.17, the class variance for different numbers of instruction classes is shown. The error in the final execution time measurement is directly related to the variance in the instruction classes. Taking the results of Figure 5.17 and the number of test programs available into consideration, we choose to use four instruction classes. Thus the execution time model has five parameters.

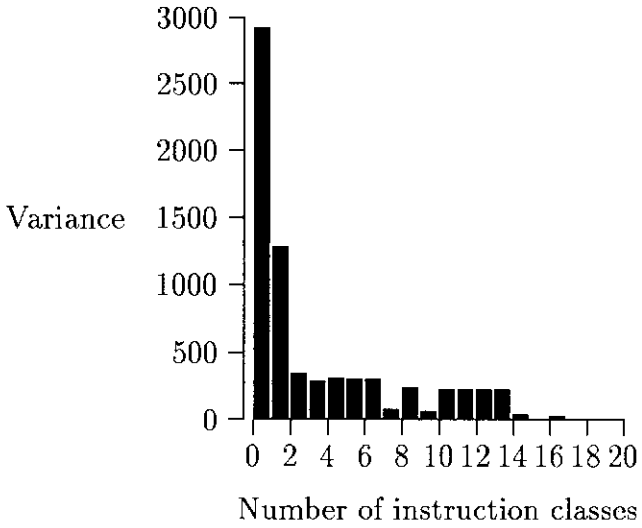

Figure 5.17: Variance versus the number of instruction classes

The regression was performed on eight test programs executed with two to four different inputs. To reduce the variance in the measurement of total execution time, each test program and input combination was executed five times. Figure 5.18 shows the distribution of relative error for the test programs. This figure shows that the maximum measurement error is 17% and in most cases the error in execution time is less than 5%. If the time spent in foreign programs is included in this figure, the measurement error would decrease further.

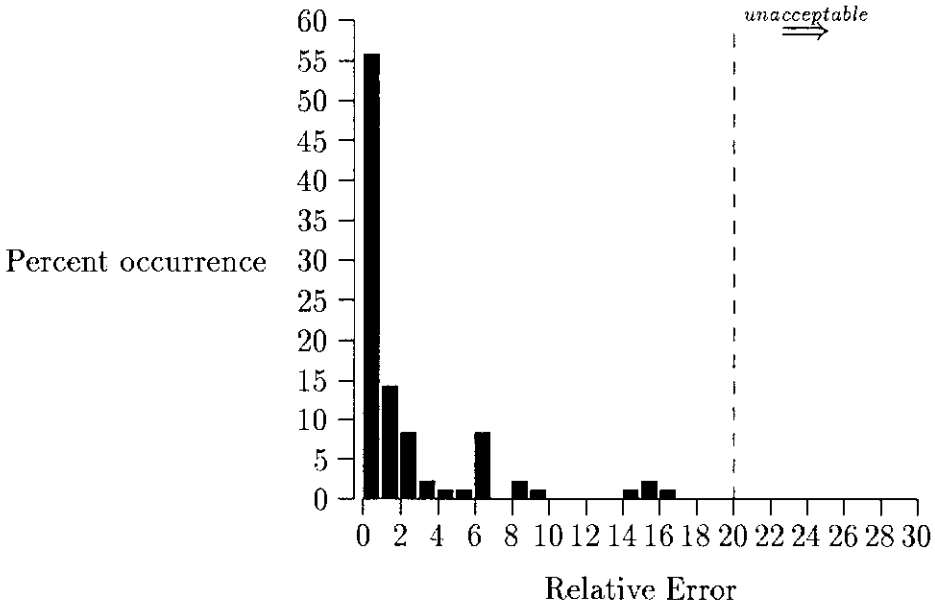

Figure 5.18: Distribution of errors in execution time

5.6.3 Execution time of foreign programs

In keeping with the black box view of foreign program execution, measurements of the execution time are limited to the total time spent in a foreign program. The use of time in a foreign program is not relevant in explaining the behavior of the parallel parts of program execution.

Performance measurements which detail the actions taken within a foreign program can be obtained with a sequential performance measurement tool. It is also possible to extend the indirect measurement techniques used for *PCN* programs to sequential languages. However, the complexity of languages such as **C** or Fortran puts this beyond the scope of our research.

Experience with multilingual programming has shown that the time spent in a single invocation of a foreign program is typically much greater than the time spent executing a *PCN* composition [FO90]. The granularity of foreign computation makes direct measurement practical. Every call to a foreign program is timed and the lapse time added to the content of the appropriate storage location.

There are situations where the amount of computation in a foreign program will be so small that direct measurement will not be able to capture

its execution time. Such a situation is encountered Chapter 8. Occurrences of foreign programs of this type are easy to detect; they have a non-zero execution frequency but an average execution time close to zero. No special provisions have been made for foreign programs with small execution times. We assume that their impact on the performance of the program is negligible and knowing of their existence and their execution frequency is adequate for the purposes of performance improvement.

This concludes our discussion of execution time measurement. We have seen that by combining direct and indirect measurement methods, we are able to obtain the execution time and frequency data required for a parallel execution profile and meet our design goals. In the next section, we discuss the remaining components of a parallel execution profile: interprocessor communications cost and idle time.

5.7 Profiling Multiple Processor Execution

Execution time and frequency are only one part of a parallel execution profile. A parallel execution profile has two other components: communication cost and idle time. We now show how to account for the time spent in these activities.

5.7.1 Profiling idle time

A *PCM* becomes idle when it is waiting for the value of a definitional variable and the active queue is empty. When the value of a definitional variable is received by an idle *PCM*, it leaves the idle state, moving the programs waiting for the received value into the active queue. The amount of time spent idle is an important performance index and is included as a part of a parallel execution profile.

Measuring idle time in a *PCN* computation is problematic in that idle time is not the amount of time spent waiting for a data value. Rather, it is that portion of the time spent waiting that can not be masked by executing some other program. As such, idle time is disassociated from the composition that is its cause. One would be tempted to say that idle time is a processor-wide attribute of program execution and the execution profile should contain a single idle time figure for each *PCN* executing the computation.

However, a processor-wide measure provides no insight as to the cause of idle time. The reason a processor becomes idle is that it is waiting for an arc in a precedence graph of a program to be satisfied. Ideally, a parallel execution profile would identify this precedence arc by its source and destination compositions and attribute the idle time to the arc.

However, because a definitional variable can be embedded into any non-mutable data structure, the precedence arcs do not correspond to the program call graph and determining the precedence arcs prior to program execution is difficult. Furthermore, determining the precedence arc at runtime is also hard, for at the time a definition is made, one must know if another processor is idle waiting for the value of that definition. This requires knowledge of the global state of the computation and as such, is specifically excluded from our measurement model. To overcome these problems, the following definition of idle time is used:

Definition 2 *Assume that the* k^{th} *time a* PCM *enters the idle state, it remains in that state for* $t^k_{\sigma^i}$ *seconds and exits the idle state with the active queue containing the programs,* $\mathcal{Q}_k$. *The idle time* $t_{\sigma^i}(p)$ *for a* PCN *program* p *is defined as:*

$$t_{\sigma^i}(p) = \sum_{\{k|p\in Q_k\}} t^k_{\sigma_i} / \mid Q_k \mid$$

If more than one precedence arc is active at one time, the idle time is distributed equally to all the programs at the destinations of the arcs. Ideally, we would prefer that idle time be assigned to the precedence arc that represents the critical path in a computation's precedence graph. However, given that the source programs are not known, determining which destination to credit with the idle time is not possible. Dividing the time equally between all activated programs is a good compromise and has proven useful in practice.

A drawback to Definition 2 is that a program can be charged for idle time that it is not responsible for. An example of such a situation is shown in Figure 5.19. Program T and program A execute in parallel. Program A has just requested a data value that is produced on another processor and will take some time to arrive. Program T is a timer; it waits on a stream that will receive a new value periodically and then starts again. When the data value for program A actually arrives, there is no idle time left to attribute

to A, and T is given full responsibility for the time spent idle. While this result is misleading, we have not seen this problem occur in any of the *PCN* programs we have examined.

Figure 5.19: Example of how assignment of idle time to programs can be fooled

5.7.2 Measuring idle time

Two methods for measuring idle time have been investigated: direct measurement and indirect measurement. Direct measurement is well-suited for a computer with low overhead access to an accurate timer. The time of entry to and exit from the idle loop is recorded and the resulting lapse time is attributed to the appropriate programs. Computing the lapse time will delay to the execution of the first program in the active queue by the time it takes to perform the subtraction.

The storage overhead for this measurement will be one location for every composition that can become suspended waiting for a variable being defined on another processor. The runtime overhead will be dependent on the total number of programs whose execution is enabled when exiting the idle state.

An alternative is to measure the idle time indirectly. Referring to Figure 4.8, when a processor is idle, it executes a loop within the communications component of the *PCM*. The idle time can be inferred by measuring the number of times the loop is taken and the amount of time required for one pass through the loop. With indirect measurement, the lapse time computation is no longer needed, reducing the execution time overhead of the measurement.

Experience with *PCN* thus far indicates that idle times are measured in milliseconds rather than microseconds. For this reason, the overhead advantages of the indirect method are negligible. Since the direct method is more accurate and somewhat simpler to implement, we choose this approach to measure idle time.

5.7.3 Profiling communication time

Concurrently executing *PCM*s communicate by sending messages to one another. There are eight different types of messages in the *PCM*; five deal with garbage collection and system shutdown and the remaining three are used to define and distribute the values of definitional variables. Six of the message types have a fixed size, while the two messages that send the value of variables have variable size. A parallel execution profile includes data on the number of messages sent, the amount of data sent and how much time was spent communicating.

From the point of view of an execution profile, the cost of communication is strictly in terms of the processing time required to send or receive a message and not the time actually required to transfer the data between processors. The transfer time is either not seen because the processor can execute some other *PCN* program while the transfer is taking place or the transfer time is already being measured as idle time.

As with the measurement of idle time, a decision must be made as to what level to profile interprocessor communications: the program level or the processor level. Measuring communication cost at the processor level is straightforward: the cost depends on the messages sent and the messages received. Attributing a communication to a specific program is more difficult. Certainly, when the value of a remotely defined variable is requested, it is possible to attribute the cost of making the request to the sending program. However, if prior to receiving the value another program requests the same variable, it will register its requirement and wait for the response to the message sent earlier. A question arises as to whether the cost of sending the message should be redistributed to both programs.

Attributing the cost of processing a received message causes more problems. If the message contains the value of a definitional variable, it would make sense to assign the cost to the program that was waiting for the value. In general, this will not be possible as pointers from variables to the programs that are waiting for their value are not always kept.

There is even a more fundamental problem. Consider the code of program `p` of Figure 5.20 executing on processor 1. The variable `x` is created on processor 1, defined on processor 2 and used on processor 3. This means that processor 1 will process a message defining `x`, and a message requesting and providing the value of `x` to processor 3. It is not at all clear to what

program the cost of these communications should be assigned. Any reference to p is long gone as the program has completed execution. Likewise, there is no information about programs f or g on processor 1. Because of these problems, we have chosen not to try to assign a communications cost to an individual *PCN* program. Rather, communications costs are accumulated on a per processor basis.

```
p(x) {|| f(x)@2, g(x)@3 }

f(x) {|| x = "a" }

g(x)
{? x ==  "a" -> h(),
  default -> h()
}
```

Figure 5.20: Code sample showing how communication is unrelated to a program

We now consider what to profile. A detailed profile would record the total number and size of each type of message sent and received. This data would be collected separately for each pair of processors in the parallel computer. Obtaining such a degree of detail can result in committing a large amount of memory to measuring communications: on a 192 processor computer with 3 Mbytes of memory per processor, such as the one used for the case study in Chapter 8, 75% of total system memory would be required to record this data.

The storage requirement can be reduced by collecting data about the communications activities of a single processor without regard as to which processor the message is going to or coming from. This simplification makes the total storage requirement proportional to the number of processors in the computer rather than the square of the number of processors.

For every message sent or received on a processor, a profile records: 1) that the message was processed, 2) the size of the message and 3) the amount of time required to send or receive the message. While each of these quantities can be measured directly, there are advantages to using an indirect

measurement for communications time. We expect message processing to occur frequently and be completed rapidly, making direct measurement difficult. Furthermore, with direct measurement of communications time, three separate profiling operations must take place for each message processed. Reducing the number of operations will reduce the execution time overhead of profiling communications.

To the first order, there are only two factors that contribute to the cost of processing a message: a fixed, per message startup cost and a per byte transfer cost [Dun88]. Using this simple linear model, communications time is determined indirectly from the number of messages processed and the total message volume. Because time does not need to be recorded, the indirect measurement will have less runtime overhead than a direct measurement.

Runtime overhead can be further reduced by making the following observation. Six of the eight message types have a fixed size. For these messages, there is no need to record the volume of data sent, as this information is implicit in the message type. The storage overhead for these messages is a single storage location while the runtime overhead is one counter update per message sent or received. The remaining two messages move the values of definitional variables between *PCM*s and their size varies with the amount of data being moved. Two storage locations are needed to measure variable sized messages, one for the number of messages sent and one for the total number of memory cells transferred. The runtime overhead is two counter updates per message sent or received.

The runtime overhead observed when executing the test programs in parallel is shown in Figure 5.21. The overhead ranges from .04% to 1.5%[1]. The storage overhead of this method will be 12Ω per processor, where Ω is the amount of storage required for a counter. This overhead is insignificant when compared to the size of all but the most trivial *PCN* programs.

The two cost parameters for the execution time model for communication are computed along with the program execution time cost parameters using the procedure discussed in Section 5.6.2. The input data to the regression procedure is obtained by executing the test programs in parallel, varying the input data as well as the number of processors used. The calibration was

[1]The C compiler used to build the *PCM* for these measurements generates a redundant add instruction at one of the instrumentation points, increasing the measurement overhead for a frequently sent message type by about 33%.

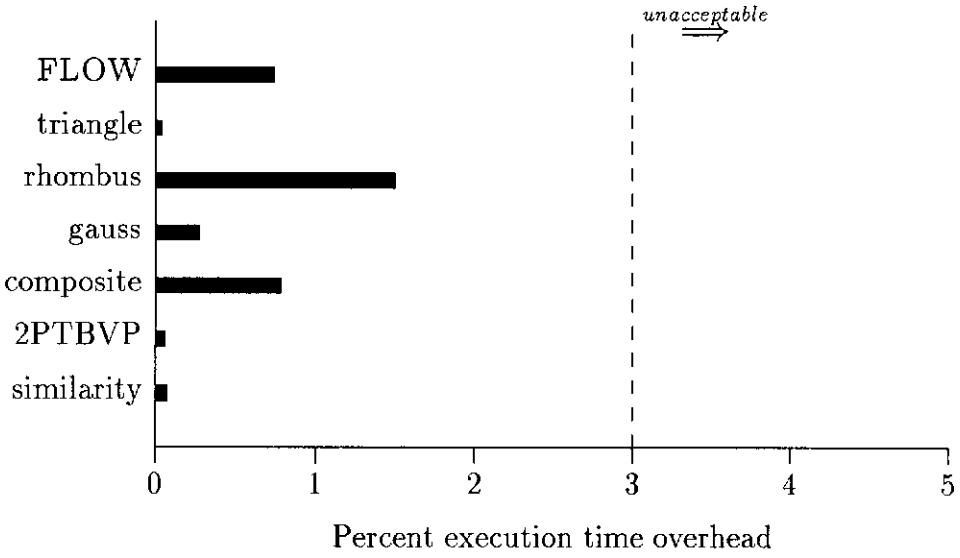

Figure 5.21: Runtime overhead for communications measurements

done on a 10 processor Sequent Symmetry with 32 Mbytes of shared memory. The number of processors used varied from three to seven. Figure 5.22 shows the results of the calibration procedure. The measurement error is on the horizontal axis while the vertical axis shows the fraction of test cases that exhibit that error. The error is calculated from T_{PCN} and $T_{communication}$, not just the communications component.

In 4.1% of the test cases, the measurement error exceeds that of our design goal. We suspect that the cause of these errors is the limited number of test cases used as input to the regression procedure. For some programs, varying the number of processors does not alter the amount of processing done on a *PCM*, exacerbating the problem. This hypothesis is supported by examining the range of the singular values obtained by the singular value decomposition of the data matrix. Although none of the singular values are zero, we find a difference of over three orders of magnitude between the largest and smallest singular value. This indicates that there are linear combinations of the input data that do not contribute additional information to the regression [PFTV88]. The solution is to have a more extensive set of calibration programs from which the parameter values are obtained. Unfortunately, due to the newness of the *PCN* system, obtaining more test cases is not possible at this time.

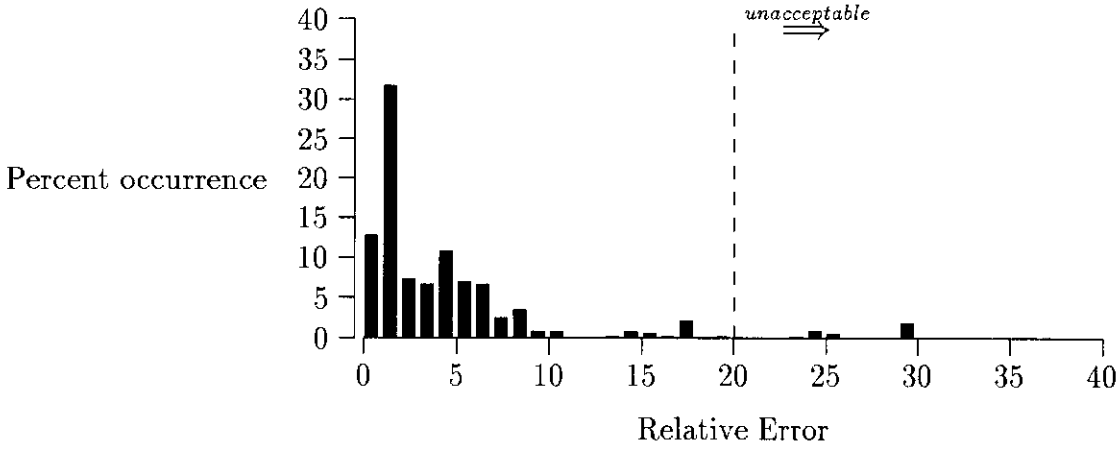

Figure 5.22: Distribution of errors in times including communications measurements

5.8 Summary

We opened this chapter with a discussion of three basic principles for designing methods for profiling the execution of a parallel program. These principles are general and can be applied to any parallel programming system. We then showed how the principles are applied in the design of a profiling system for *PCN*.

The result is a set of measurement techniques that provide a wide range of alternatives to the system designer. The tradeoffs in cost and accuracy were demonstrated for a set of parallel applications. In the next chapter, we show how the techniques developed in this chapter are integrated into the *PCN* system to form a practical performance measurement system.

Chapter 6

Implementation of a Parallel Execution Profiler

The focus of Chapter 6 is the design and implementation of our profiler for *PCN*. A profiler is a system whose function is to collect profile data from a running program and present it to the user in a meaningful form. In our profiler, data collection is based on the measurement techniques developed in the preceding chapter. The methods used to integrate our measurement techniques into the implementation of *PCN* are discussed in this chapter.

6.1 Profiler Overview

An overview of the *PCN* profiler is shown in Figure 6.1. The profiler has two components: the *PCN* system, which we will refer to simply as *PCN*, and *Gauge*. *PCN* is responsible for obtaining the basic data needed in an execution profile, while *Gauge* is responsible for generating a profile from the raw data and presenting it to the user.

The *PCN* system consists of the *PCN* compiler, the *PCN* environment and the *PCM*. The environment, which has not been previously discussed, is the interface between the *PCM* and an application program; its functions include loading code into the memory of the *PCM*, starting new computations and providing an interactive top level. Profiling starts with the compiler where *PCN* programs are instrumented in accordance with the requirements of Sections 5.5.1 and 5.5.8. As the program executes, profile data is col-

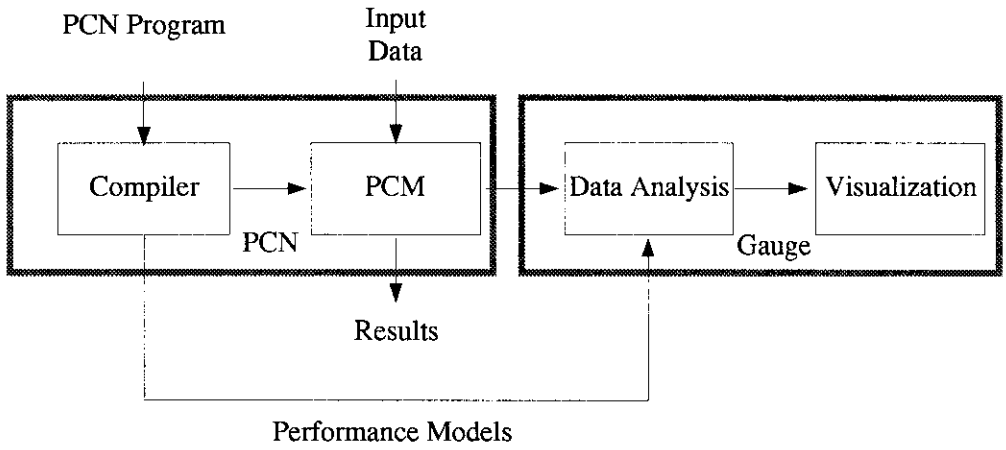

Figure 6.1: An overview of a performance analysis system

lected by the *PCM*. On program termination, the environment extracts the raw performance data from the *PCM* and outputs it in a form that can be interpreted by *Gauge*.

Gauge is an interactive tool which processes the raw performance measurements into an execution profile and then graphically presents the profile data to the user. To obtain an execution profile from the data output by the *PCN* environment, the data analysis component of *Gauge* applies the performance models of Section 5.6.1. Because of the complexity inherent in parallel performance data, *Gauge* presents the profile to the user graphically. A unique aspect of *Gauge* is that the user can dynamically construct and display alternative views of the profile data. A detailed discussion of *Gauge* is found in Chapter 7.

In the remainder of this chapter, we examine how profiling has been integrated into the implementation of the *PCN* system. The *PCM* will be discussed first, followed by the compiler and finally the environment.

6.2 Runtime Support for Profiling

As discussed in the previous section, measurements for a profile are made by the *PCM*, which accumulates measurement values in its memory until they are extracted by the environment. Recall that a measurement consists

of counting or timing the occurrence of an event and that all measurements corresponding to the same event are summed into one storage location. The first problem we consider is how the storage location for measurements are allocated.

6.2.1 Storage for counters and timers

Storage for profile data is allocated by the compiler as part of a module. This decision was motivated by the manner in which modules are used. The storage space for modules is allocated from the *PCM* heap and therefore the absolute location of a module is not know until runtime. Consequently all *PCM* code must be position independent. It follows that references to profile data from within *PCM* code must be position independent as well.

To obtain position independence, we include the storage space needed to profile the programs in the module along with the program code. In Figure 6.2, the format of a module generated by the *PCN* compiler is shown. The storage space for profile data is found in the counter section, which is located after the *PCM* code for the programs in the module. Since the code and instrumentation are allocated from the same data structure, the offset from an instruction to a counter or timer is constant regardless of the module's location. Hence, a reference to a counter or timer from the program section is encoded by the offset from the point of reference to the counter or timer being referenced.

An advantage of this module layout is that the profile data is in a contiguous block on the heap. Using the offset to the counter section and the number of counters and timers, the profile data for an entire module can be obtained with a single block copy operation. Thus collecting profile data is both simple and fast. This is important if snapshots of the profile data are to be made during program execution.

The amount of *PCM* memory used to store the value of an individual counter or timer determines the maximum time a program can execute and still produce a meaningful profile. As discussed in Section 2.3.4, parallel applications can run for extended periods of time. The *PCN* profiler is designed to accommodate programs that execute for up to one week. Practical considerations further constrain the amount of memory used to store a profile measurement; sizes of 8, 16, 32 or 64 bits long map most efficiently onto the memory of the target architectures on which the *PCM* is hosted.

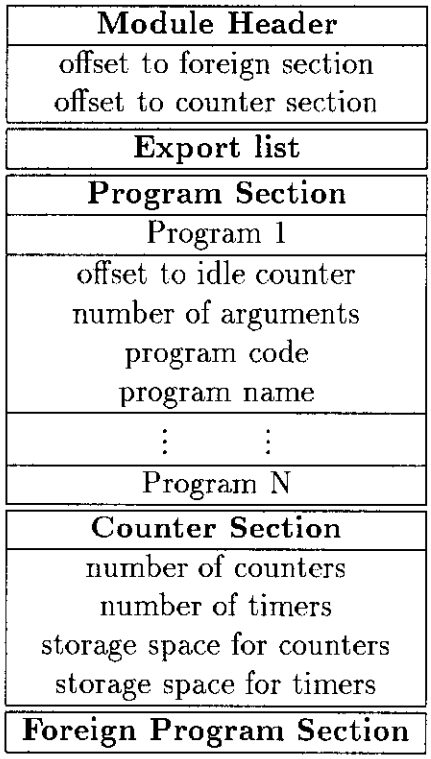

Figure 6.2: The layout of sections and fields in a *PCM* module

For counters, the amount of memory required is determined by the maximum frequency at which a counter update can occur. When profiling choice or parallel composition, the worst case occurs in a computation consisting of a single parallel composition that does nothing but call itself recursively. When executed on a Sun Sparcstation 1, such a program can execute for about 285 hours before overflowing a 32 bit storage location. Based on this data, we conclude that 32 bits is large enough to store a counter.

The value of a timer is the number of clock ticks that occur during the event being timed. On the current generation of parallel computers, clock ticks occur at one microsecond intervals. With a 32 bit timer, the maximum time that can be recorded is 71 minutes — a value that is far short of the maximum times we wish to measure. Thus 64 bits of storage are used for timers in the profiler. Unfortunately, many of the architectures on which *PCN* executes do not directly support addition of 64 bit integers. Consequently updating a 64 bit wide timer will be more than twice as slow as updating a 32 bit timer.

6.2.2 Measurement in the *PCM*

In this section, we discuss how performance measurement is integrated into the *PCM*. In *PCN*, two types of measurements are made: 1) measurements which record the activity of a program and 2) measurements which record activities in the *PCM* that are not direct consequences of executing *PCM* instructions.

The method used to integrate measurement into the *PCM* differs for the two types of measurements. Measurements of the first type are integrated into the *PCM* via the instruction set. The measurements are generated explicitly by the instruction sequence produced by the *PCN* compiler. The second type of measurement is integrated into the *PCM* via a set of profile registers included in the machine definition. The contents of the registers are manipulated implicitly by the communications component and garbage collector during *PCM* operation.

6.2.3 Measuring program activity

The first type of measurement is triggered by the execution of specific *PCM* instructions. Specifically, to profile choice and parallel composition, a mea-

surement is made each time a *halt*, *recurse* or *default* instruction executes. Measurement of foreign programs is indicated by the execution of a *call_foreign* instruction and snapshotting in assignments and definitions is indicated by a *copy_mutable* instruction.

The measurement of program activity is integrated into *PCN* through the *PCM* instruction set. Two different approaches to integration are considered. The first is to have special purpose instructions to make measurements. The second is to add measurement to the basic functionality of existing *PCM* instructions and have the instructions which trigger a measurement make the measurement.

The required measurements can be implemented by three special purpose *PCM* instructions. Counter measurements are made by a *update_counter* instruction and lapse time measurements are made by pairing *start_timer* and *stop_timer* instructions. The *update_counter* instruction has two arguments: the location of the counter to be updated and the value by which the counter is incremented. The *start_timer* instruction has no arguments while the single argument of the *stop_timer* instruction points to the storage location of the timer.

Because these instructions are completely orthogonal to the rest of the *PCM* instruction set, optimizations to reduce the number of counters and timers needed are possible. For example, interprocedural analysis can identify counters whose value is expressible in terms of other counters. The redundant *update_counter* instruction can be eliminated, reducing the execution time and storage overhead of the measurement. Another potential optimization is found in a choice composition whose implication guards consist only of data tests. Compositions of this type are often produced by the compiler in converting a *PCN* program to core *PCN*. In such programs, the *default* implication can never succeed and *update_counter* instruction can be eliminated.

The disadvantage of using separate measurement instructions is that the runtime overhead of a measurement is increased by the cost of the instruction decode. Because of this cost, we choose the alternative technique: measurement is made part of the functionality of the *halt*, *recurse*, *copy_mutable* and *call_foreign* instructions. In essence, the *halt* instruction becomes *halt_and_update_counter*, and so on.

Combining measurement with other instruction functions has one drawback. If optimization to eliminate unnecessary measurements is desired, ver-

sions of the instructions that don't perform the measurement are also needed. These additional instructions increase the size of the emulator, which is not desirable. With the current instruction encoding, measurement optimization requires the addition of four instructions, a 13% increase in the number of instructions in the *PCM*.

When an instruction makes a measurement, the value is added to the a storage location passed to the instruction by an argument. The *halt* and *recurse* and *default* instructions have a counter as an argument. Following the procedure of Section 5.5.1, the counter of a *halt* and *recurse* is incremented by one every time the instruction executes. From Section 5.5.8, we find that the counter of a *default* instruction is incremented by one only when the *default* test fails.

Snapshotting of mutable data values during an assignment is performed by the execution of a *copy_mutable* instruction. This instruction has three arguments: a source register, a destination register and a counter. The model of Section 5.5.11 requires recording the number of heap cells copied during the execution of a *copy_mutable*. The size of the data structure being copied, is determined before the copy starts and is added to the contents of the counter by a single addition.

The *call_foreign* instruction is the only instruction in the *PCM* that updates a timer. The function of this instruction is to call a program written in a foreign language. As per Section 5.5.12, the time spent in the foreign program is directly measured as part of the *call_foreign* instruction. The lapse time between the entry and exit of the foreign program is added to the timer and passed to the instruction as one of its arguments.

To summarize, we integrate profiling into the instruction set of the *PCM* by adding measurement to the functionality of four instructions. Measurement optimization is not currently supported, so unmeasured versions of the instructions are not required. In total, the fraction of the emulator devoted to profiling accounts for about 3% of its size. The cost of profile measurement in execution time was reported in Section 5.5.

6.2.4 Measuring *PCM* activity

We now turn to the measurement of activities in the *PCM* not associated with the execution of *PCM* instructions. Within the communications component of the *PCM*, measurements of idle time and interprocessor communications

are made. Measurements are also made in the garbage collector, which measures the amount of time spent by a processor in garbage collection.

Idle time is measured according to the model developed in Section 5.7.1. To store idle time, a timer is allocated for each *PCN* program when it is compiled. This timer is located a known distance from the beginning of the program. Since a process record points to the start of the code to be executed, the idle timer for a program can be located from its process record.

Idle time is measured as follows. The total time spent in any one instance of the idle loop is directly measured and the value is stored in a register internal to the communications component. After exiting the idle loop, the number of process records on the active queue is counted and the contents of the idle time register is scaled by this amount. The active queue is scanned again. This time, the scaled contents of the idle time register are added to the idle timer for each program on the queue.

A set of ten counter registers has been added to the *PCM* into which measurements for the communications model are stored. Updates to these registers are directly coded into the implementation of the inter-*PCM* communications component of the *PCM*. The procedure for counter updates follows the communications model of Chapter 5.

The last measurement records the amount of time a *PCM* spends in garbage collection. Each *PCM* has single timer register to store the amount of time spent in garbage collection. As with the communications component, the update of this timer is coded directly into the garbage collector.

6.3 Compiler Support for Profiling

The job of the *PCN* compiler is to take a file containing *PCN* source code and convert it into a file containing *PCM* code. The *PCN* compiler supports profiling in three ways, it:

- allocates the storage space for counters and timers used to measure program activity
- generates *PCM* instructions with counters and timers as arguments
- builds performance models for the *PCN* programs in a module.

Allocating storage for timers and counters was discussed in Section 6.2.1. We now focus on the second and third points. Figure 6.3 shows an overview of the compilation process. The shaded boxes represent compiler steps that support profiling in some way. Files containing *PCN* source code are input on the left and the compiler produces a file containing *PCM* code on the right. In addition to *PCM* code, the performance models for the module are output into a separate file.

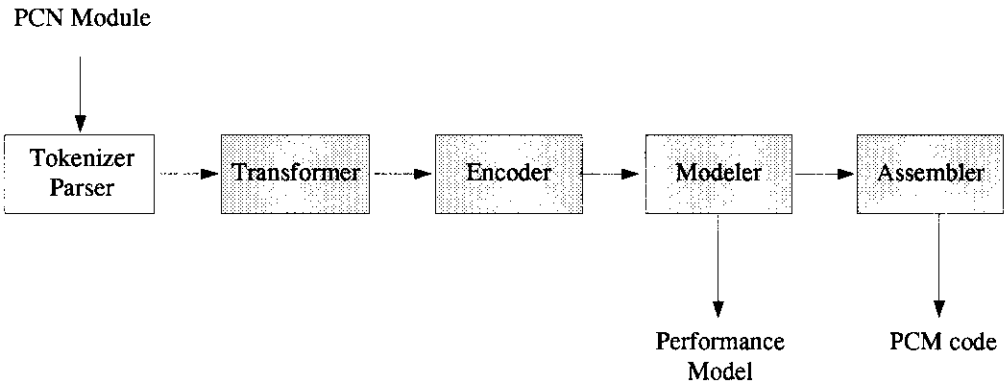

Figure 6.3: The *PCN* compilation process

The first steps in compilation are the tokenizer and parser. The output of the parser is a list of parse trees, one tree for each program in the module being compiled. The transformer then converts these trees into a format compatible with core *PCN*. As part of this process, the compiler maps a single program written by a programmer into a collection of compiler generated core *PCN* programs. The transformed parse trees are then encoded into *PCM* instruction sequences. Next, the modeler scans each instruction sequence and builds a cost model to be used for profiling. Finally the assembler converts the *PCM* assembly language programs into binary format, resolving references to labels, counters and timers. The resulting binary executable module is put into a file. A step-by-step discussion of how profiling affects each compiler stage follows.

The transformer converts a single user defined *PCN* program into one or more core *PCN* programs. Although knowing which programs are generated by the user and which programs are generated by the compiler is inconsequential to the operation of the rest of the compiler, this information

is needed by the profiler. When a profile is presented to the user, it should be in terms of the original programs — compiler generated programs should not be included. To enable the profiler to make this distinction, each parse tree is tagged to indicate its source. The tag is passed through the encoder to the modeler where the information is recorded for *Gauge*.

The instruction encoder generates a sequence of *PCM* instructions for each parse tree received from the transformer. Within the instruction stream, each occurrence of a *halt*, *recurse*, *default*, *copy_mutable* and *call_foreign* contains an unresolved reference to a counter or timer. These references are resolved to locations within the counter segment of the module in the *PCN* assembler.

The modeler produces performance models from the instruction sequences generated by the encoder. In addition to knowing the instruction sequence, generating a performance model requires knowing which instructions are part of an implication guard and which instructions are part of an implication body. This information is provided to the modeler by having the encoder annotate the instruction stream to indicate the location in the instruction sequence where each implication guard and implication body starts. The format of the output of the encoder is shown in Figure 6.4. The boxes containing *start_guard* and *start_guard* are the markers put into the instruction stream so the modeler has enough information. In addition to locating the position of implication guards, the type of composition the instruction stream implements is also identified by the markers: a parallel composition has only one *start_guard* marker and it is followed immediately by a *start_body* without any intervening instructions.

The modeler is the only component in the compiler that is exclusively dedicated to profiling. The input to the modeler is a list of encoded programs. For each program, the modeler builds a performance model and outputs it along with other information needed by the *Gauge*. After removing the *start_guard* and *start_body* markers, the modeler passes the encoded programs on to the assembler.

For each program in a module, the modeler outputs the data listed in Table 6.1. The program's name and source are determined from the information passed on by the transformer. To determine the composition type, the first two *PCM* instructions of the program are examined. If they are *begin_guard* and *begin_body* markers, then the program is a parallel composition; otherwise it is an implication. The program code is then scanned

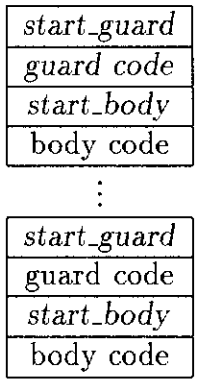

Figure 6.4: The structure of the *PCM* code generated by the encoder

for *copy_mutable* and *default* instructions. Once the counter arguments to these instructions have been resolved, the resulting offsets into the counter segment are stored into the *program copy* and *default failure* fields of the model. The *default* counter is included with the composition specific data because there is only one per composition and it is interpreted differently from any other implication counter. If the *composition type* is *parallel*, then the *default* counter field is empty.

The idle timer for a program is used to store the results of the idle time model. The idle timer is unique in that it is not referenced from an instruction in the program; updates to the idle timer are performed directly by the *PCM*. The reference to the idle timer is generated by the modeler and explicitly passed to the assembler. The assembler resolves the reference and outputs the offset to the idle timer as part of the composition header. The offset of the resolved timer is stored as part of the per composition data.

In addition to the per composition data, there is a set of data associated with each parallel composition in a program. Because the body of a core *PCN* implication is a parallel composition, we call this *per implication* data. The contents of the per implication data are summarized in Table 6.2. The *compositions called* and *foreigns* fields are used to build the call graph of the programs in a module. The call graph serves two purposes. First, the execution frequency of a foreign program is computed from the execution frequency of its callers. For this calculation the call graph is needed. Second,

Per Composition Data	
Element Name	Value
program name	string
program type	user/system
composition type	parallel/choice
program idle	timer offset
program copy	counter offset
default failures	counter offset

Table 6.1: Model components for each *PCN* program

the call graph is needed to collapse performance data from compiler generated programs back into the user specified program from which they were generated.

The performance model is split into two parts: 1) a *failure model* which applies to the execution of an implication that terminates in guard failure and 2) a *success model*, which applies to all other situations. Each model is a vector whose length is equal to the number of instruction classes defined in Section 5.6.1. The value of each element in the vector is the weight of the corresponding class in the model. In the success model, the weight is the number of times an instruction of that class appears in the code. The weight of an instruction class in a failure model is determined by summing the probability of execution for each occurrence of the instruction class in the implication guard.

The actual allocation of timers and counters is left to the assembler. For each program encountered, the assembler allocates an idle timer and a copy counter. Although more than one *copy_mutable* instruction can occur in a program, for space reasons we have chosen to use a single counter for all instances of the instruction within a composition. The offsets to these locations are passed back to the modeler to be stored with the per program data.

When assembling the instruction sequence for a composition, the assembler allocates one counter for each *halt*, *recurse* or *default* instruction encountered. When a *copy_mutable* instruction is assembled, a reference to the

Per Implication Data	
Element Name	Value
compositions called	list of strings
foreigns	list of strings
foreign timers	list of timer offsets
success counter	counter offset
failure model	instruction class vector
success model	instruction class vector

Table 6.2: Model components for each *PCN* implication

previously allocated copy counter is placed in the counter argument. Finally, a new timer is allocated for each *call_foreign* instruction assembled. To resolved counter and timer references into program counter relative offsets, two complete passes thorough the module are required. An interesting feature of the assembler is that since it is written in *PCN* itself, both passes are executed concurrently using parallel composition.

This concludes the discussion of compiler support for profiling. We now discuss support for profiling in the *PCN* runtime environment.

6.4 Collecting the Profile Data

A feature of our approach to performance measurement is that every computation is instrumented. While measurements are always made during program execution, there is no requirement to collect the data once the program terminates. Performance data is only collected if the programmer explicitly specifies that the program execution is part of a performance experiment. Performance experiments were discussed in Section 2.2.1. To support profiling, we extend *PCN* runtime environment with the ability to conduct a performance experiment.

A performance experiment is initiated by entering a *profile* command into the environment. The parameters are required to specify an experiment are:

- The computation being profiled

- A termination signal
- The modules that are part of the experiment
- The processors participating in the experiment
- The name of the performance experiment

The computation parameter of a performance experiment is specified by the module name, program name and program arguments. One of the arguments to the program must be a termination signal, a definitional variable that is defined to a constant when the computation has completed. Because of the cost of detecting global termination in a distributed environment, *PCN* does not provide a general mechanism for termination detection. To ensure that profile data is not collected until the computation is done, the programmer is required to provide an explicit termination signal. If no compositions are suspended when a computation terminates, the termination signal can be generated automatically via a source to source transformation [HSS87].

To facilitate modular programming, *PCN* allows the module component of an intermodule call to be a variable. This means that in general it is not possible to determine all the modules used in a computation. Thus, the specification of a performance experiment must include a list of modules from which to collect performance data. There is no requirement that the module list include every module used in the computation.

The default behavior is to collect performance data from every processor participating in a computation. However, there are situations where it is desirable to only collect performance data from a subset of the processors. Therefore, the range of processor from which to collect performance data is specified as part of the performance experiment.

While naming a performance experiment is a simple idea, we can find examples, such as the Unix profilers gprof [GKM83], where this capability is not provided by the profiler. Performance evaluation studies can require that many experiments be performed and keeping track of the individual experiments can become difficult. Naming the experiment provides a simple means to organizing the performance data.

Performance experiments are conducted in the environment by a *profile server*. When a profile request is forwarded by the environment to the profile server, the following steps are taken: 1) initialize the counter segments of the

modules being profiled, 2) start the computation, and 3) collect and output the raw performance data. The performance data output by the profiler for each processor is summarized in Figure 6.5. This data is in a raw form and is not an execution profile until it is converted by *Gauge*, a process that is described in Section 7.1.1.

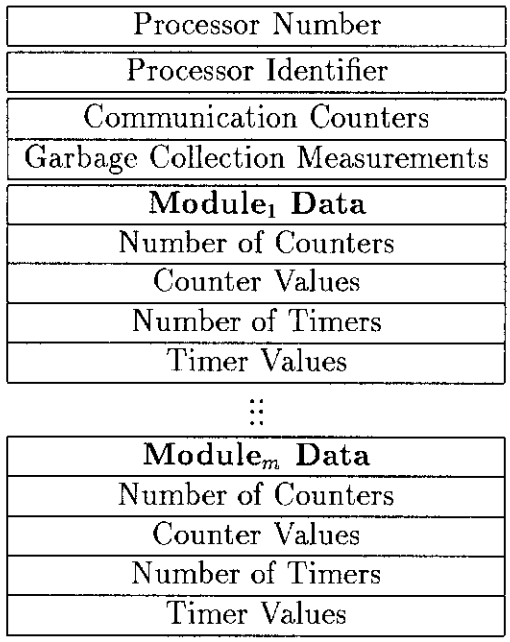

Figure 6.5: The format of the data output for each processor by the profile server

The profile server is split into two parts: a master profiler and a node profiler. There is only one master profiler, typically executing on the same processor that dispatches user requests. In contrast, a copy of the node profiler executes on every processor in the parallel computer.

The actions of the master profiler are summarized in Figure 6.6. As we see, its responsibilities are to start the node profilers, initiate the computation and coordinate the collection of the resulting performance data. It is the node profiler that obtains the values of the counters and timers from a module. The complete activities of a node profiler are summarized in Figure 6.7.

1. *Initialize node profilers.* Inform the server on each node of the list of modules to be profiled and the variable that will be used to indicated termination of the computation.

2. *Reset modules.* Request each server to reset the modules on its module list. Wait for acknowledgment from all servers that reset is done.

3. *Execute computation.* Start a lapse timer to compute total execution time. Spawn the computation to be performed.

4. *Wait for termination.* Stop lapse timer, record the total execution time under the name of the experiment.

5. *Record data.* For each node, request profile data. Format data and save it under the name of the experiment.

Figure 6.6: The master profiler algorithm

1. *Find modules.* For each name in the list of modules, obtain a pointer to the actual module on the heap. Ensure that the module has been loaded into the *PCM*.

2. *Reset modules.* For each module in the module list, reset the module timers and counters to zero. Indicate to the master profiler when the initialization has been done.

3. *Wait for termination.* Wait for the done variable to become defined.

4. *Collect node specific data.* Record the communications counters and the time spent in garbage collections for the node.

5. *Collect module data.* For each module in the module list, snapshot the module timers and counters.

6. *Record processor data.* When requested by master profiler, provide the node specific data and the profile data for each module in the profile.

Figure 6.7: The node profiler algorithm

6.5 Summary

A number of conclusions can be drawn from our experience in implementing a profiling system for *PCN*. Profiling has some impact almost every component of the *PCN* system. Most of the code to support profiling is found in the profiler, the node profilers and the modeler. Outside these components, the amount of code added to the system is quite small.

In terms of object code, the percentage of the *PCN* implementation dedicated to profiling is shown in Table 6.3. Compiler support is completely written in *PCN*while the support within the abstract machine is written in C. Low level data collection and timer manipulation in the node profiler is written in C, while the top level coordination of the node profiler is written in *PCN*.

Component	Percent of code
Abstract machine	2.3
Compiler	8
Environment	18

Table 6.3: Fraction of *PCN* devoted to supporting execution profiling

While the code required for profiling is minimal, it must be integrated into the *PCN* system at a fundamental level. We have carefully examined the interaction of profiling with the semantics of *PCM* instructions, the layout of program memory, the garbage collector and virtually every other aspect of the *PCN* implementation. The lesson is clear: the only way to cleanly achieve the low level integration necessary for efficient profiling is to design the profiler into *PCN* from the beginning, not to add it as an afterthought.

Chapter 7

Interactive Performance Visualization for Parallel Execution Profiles

In this chapter, we address the problems associated with presenting performance data for a parallel program to the user. In our system, there are two parts to this process: the raw performance data is converted to an execution profile and then the profile is graphically presented to the user. Both tasks are performed by *Gauge*, our performance visualization tool for presenting parallel profile data.

7.1 *Gauge*

Gauge[FKT90] is a tool that we have developed to present a parallel execution profile to the user in an intelligible manner. It is not part of *PCN* proper and is used after the performance experiment has completed. The functions of *Gauge* are:

- To convert the raw performance data output by *PCN* into a parallel execution profile
- To graphically display profile data
- To enhance a programmer's understanding of multidimensional execution profile data thorough interactive data analysis.

The output of *Gauge* is a graphic representation of the parallel execution profile, an example of which can be seen in Figure 7.1.

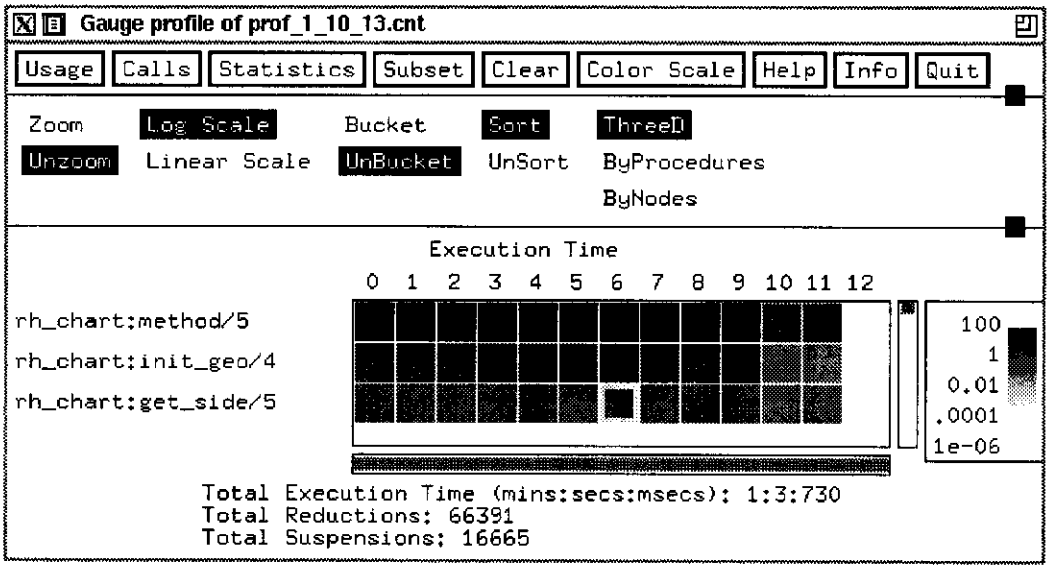

Figure 7.1: A flat 3D display of execution time

7.1.1 Obtaining an execution profile from measurement data

There are three sources of input to *Gauge*: the performance models generated by the compiler, the raw measurement data collected by the environment and the architecture specific model parameters determined by the system calibration of Section 5.6.1. When invoked, *Gauge* combines these inputs to build an internal representation of the parallel execution profile. This representation puts the performance data into a form that is convenient for the interactive visualization component of *Gauge*. The data structure, shown in Figure 7.2, is built in three stages: 1) the models for each module in the computation are read in, 2) the performance data is input and 3) performance models are used to convert the raw data into an execution profile. We examine this process in detail.

At the top level, the data structure *Gauge* builds consists of the list of modules used by the profiled computation. The models, which consist of

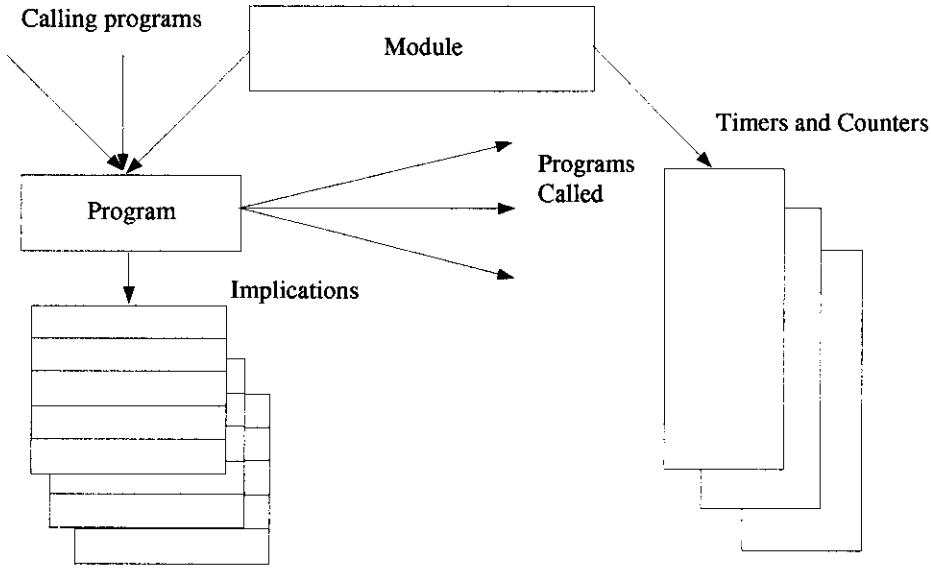

Figure 7.2: The representation of profile data in *Gauge*

the data from Tables 6.2 and 6.1, are read in one program at a time. Using the *compositions called* and *foreigns* fields from Table 6.2, the call graph for programs in a module is built. The nodes in the graph are linked in both upward and downward directions, allowing easy access to both the callers of a composition and the programs called from a composition. The data from the tables is stored in the call graph node corresponding to the program.

The profile data is input next. For each processor, a data structure is created to hold the communications and garbage collection data. The processor identifier field from Figure 6.5 is used as an index into a database of cost parameter values. Each entry in the database contains the model parameters determined by the calibration procedure for the different types of processors on which *PCN* can run. Since a *PCN* computation can execute on different types of processors simultaneously, the parameter values must be accessed for each processor in the computation.

The values for the counters and timers for a module are in the form of a vector, one vector per processor. When they are read in, the values are stored in the module data structure indexed by the processor number. The value of a specific counter is determined by using the processor number to obtain the

appropriate counter vector and using the offset stored in the program model as an index into the vector.

With the performance data input, *Gauge* generates an execution profile. The first step is to determine the execution frequencies of all programs in the computation. This is done first for *PCN* programs and then for foreign programs. If a *PCN* program is a parallel composition, it has only one *success counter* field. The value of the counter indicated by this field is the execution frequency of the program.

In a choice composition, we compute the execution frequency of each implication and from this, the execution frequency of the composition as a whole. The execution frequency of an implication is equal to the number of times the implication succeeds and fails. Using the recurrence relation of Equations 5.12 through 5.14, these profile values are computed from the *success counters* of each implication and the *default failures* of the program.

The call frequency of foreign programs is determined next. The execution frequency of a foreign program is computed by summing the execution frequency of each composition that calls that program. These compositions are found by following the reverse links in the call graph from the foreign program.

To obtain execution time from the frequency data, the performance models, counter values and architectural parameters must be combined. Recall that the success and failure model for a program is a vector of weights. The number of times each instruction class executes in a program is determined by scaling the success model by the contents of the success counter and adding to this, the failure model scaled by the contents of the failure counter. Execution time is obtained from this sum by taking the inner product with the cost parameter values obtained from the parameter database.

The final step in this process is to compute the execution time of the foreign programs. Recall that the timers for a foreign program are associated at the call site in the *call_foreign* instruction. For each program that calls a foreign program, the contents of the timer in the appropriate *call_foreign* instruction are added to the total time spent in the foreign program.

7.2 Presentation of Parallel Execution Profile

Once an execution profile is derived from raw performance data, *Gauge* presents the profile to the user. Because of the inherent complexity of parallel performance data, the method used to convey the data to the user takes on special significance. Regardless of how much information is available in a profile, it is useless unless the user can understand and draw conclusions from it.

In a sequential profile, performance data is often presented in a table showing each procedure along with the percent of the total execution time spent in it. This tabular approach quickly breaks down with parallel profile data. The reasons for this are:

- In addition to execution time and frequency data, we must also contend with communications and idle time. Understanding the interaction between all of these indexes is important. Recognizing the interrelationships from a table is difficult.

- With one data set for each processor, either the number of tables or size of each table will grow to an unmanageable size.

For the above reasons, we find that tabular representation is not generally used for parallel performance data. Rather, most parallel performance measurement systems, including *Gauge*, present performance data to the user graphically. This technique, called performance visualization, takes advantage of the fact that the human visual system is capable of processing large quantities of information at one time [DBM89].

Gauge further aids in data interpretation by incorporating mechanisms for interactive data analysis, or dynamic graphics [BCW87] as it is sometimes called. With dynamic graphics, the user interacts with the data, examining data values, constructing new views of the data and relating data points between views. These techniques enable a clearer understanding of the data than is possible with a static display [Hub83].

7.3 Visualizing Parallel Profile Data

The goal of performance visualization is to provide a clear concise visual representation of the performance data. The user should be able to comprehend relations between the data and easily identify anomalous behaviors. The use of performance visualization is a recent occurrence and many different graphical representations have been proposed. Some systems offer only one type of display while others provide a range of different display types.

7.3.1 Related work in performance visualization

Because performance visualization of systems other than *Gauge* are geared toward data in the form of event traces, displaying the data with respect to time is usually the focus of these systems. While some of the systems use animation to show the relationship of the data to time [Cou88, WH90, GHPW90], the bulk of the performance visualization systems being developed focus on a static display of data.

The most obvious type of display is a graph which plots the value of a performance index on the vertical axis and time on the horizontal axis. In IPS-2 [MCH+90], the user can combine several data sets into a single graph, even if the units being measured are different; different data sets are plotted with different line styles. Clearly, analysis of performance data from more that a few processors is difficult with this type of display.

More data can be displayed at once by using a different timeline for each processor and positioning the timelines one next to another. Both Gist [BBN] and PIE [GS85] use this approach. Rather than showing the value of a performance index, these tools display how the program state changes with time on a processor. Each different state is represented on the time line by a different color. The number of processors that can be displayed using this technique is far greater than possible with the graphs of the preceding paragraph. However, because each program state must have a nonzero width, there is a limit to how much time can be displayed on the horizontal axis. To overcome this problem, both Gist and PIE display a subset of the timeline, and provide the ability to scroll over the data.

7.3.2 Performance displays in *Gauge*

Gauge can display seven types of profile data:

- The execution time on a per program, per node basis.
- The idle time on a per program, per node basis.
- The number of invocations on a per program, per node basis.
- The number of communications on a per node basis.
- The time spent in communication on a per node basis.
- The volume of communication on a per node basis.
- The time spent in garbage collection on a per node basis.

Parallel profile data is essentially three-dimensional, a profile value being associated with every pairing of a program and a processor. By imposing an ordering on the programs in a profile, parallel profile data can be displayed as a surface in three-dimensional space. While this approach displays the performance data in a form that many users are familiar with, it has disadvantages. The primary drawback is that in order to fully comprehend the data, the user must look at it from all angles; a single view is not sufficient. Another problem is that surfaces represent a continuum of values although profile data is discrete in two dimensions: the processor and the program. A three dimensional dimension surface contains more information than we require.

In *Gauge*, parallel profile data is displayed in a form based on histograms or bar charts. The basic *Gauge* display, which we call a 3D-histogram, is a rectangular grid. Each row of the grid represents a different program and each column of the grid represents a different processor. Based on the profile data and the mapping of the processors and programs onto the grid, each grid cell is be assigned a value. The value of the cell is represented by coloring it, with each different color denoting a range of data values.

An example of a 3D-histogram showing execution time can be found in Figure 7.1. The histogram is displayed in a window consisting of three panes. The top two panes are used in interactive data exploration, the subject of the next section. The histogram itself is located in the center of the bottom

pane. The names of the programs being shown are on the left side of the histogram and the processor numbers are indicated along the top of the histogram. The values of the profile data are represented by different shades of gray; the darker the gray, the larger the value. The mapping of the gray scale into value is shown on the right side of the histogram. The rectangular regions located to the immediate right and below the histogram are scroll bars which are used to move other parts of the data set into the window of observed values.

In practice, we have found the 3D-histogram to be an effective means of display. Data from more than 64 processors and 30 programs can be displayed at one time on a typically sized workstation display. In general, the view of the data is less cluttered and more comprehensive that which we would get from a three dimensional surface. Additionally, a 3D-histogram is quickly rendered on a color workstation without requiring special graphics hardware.

The color scale for the 3D-histogram must be carefully chosen. In addition to grayscale, we have experimented with other color scales, with mixed results. With the exception of shades of gray and red, there is no intuitive sense of order implied by a color scale [Tuf83]. Although visually less striking than other color scales, gray scale seems to be the best choice from the point of view of transferring information to the user.

Displays similar to 3D-histograms can be found in other performance visualization tools. The matrix display in Hypertool [MAA+89] represents values by coloring cells on a grid, although the values mapped are between two processors rather than between a processor and program. Similar to the matrix display is the performance cell plot described in [RPW89].

Some of the data collected by *Gauge*, such as interprocessor communication, is not three dimensional. For this data, *Gauge* displays standard two dimensional histograms with either node numbers or program names plotted on the independent axis. In addition to communications data, garbage collection time is displayed on a standard histogram. Two dimensional histograms are also used in *Gauge* to display the result of projecting three dimensional performance data onto the processor or program axis. The use of projections is discussed further in Section 7.4.3.

Data from two or more standard histograms can be combined into a single stacked histogram. An example of a stacked histogram which shows the amount of idle time and execution time for the processors in a computation

is shown in Figure 7.3. Stacked histograms with programs on the independent axis are also available.

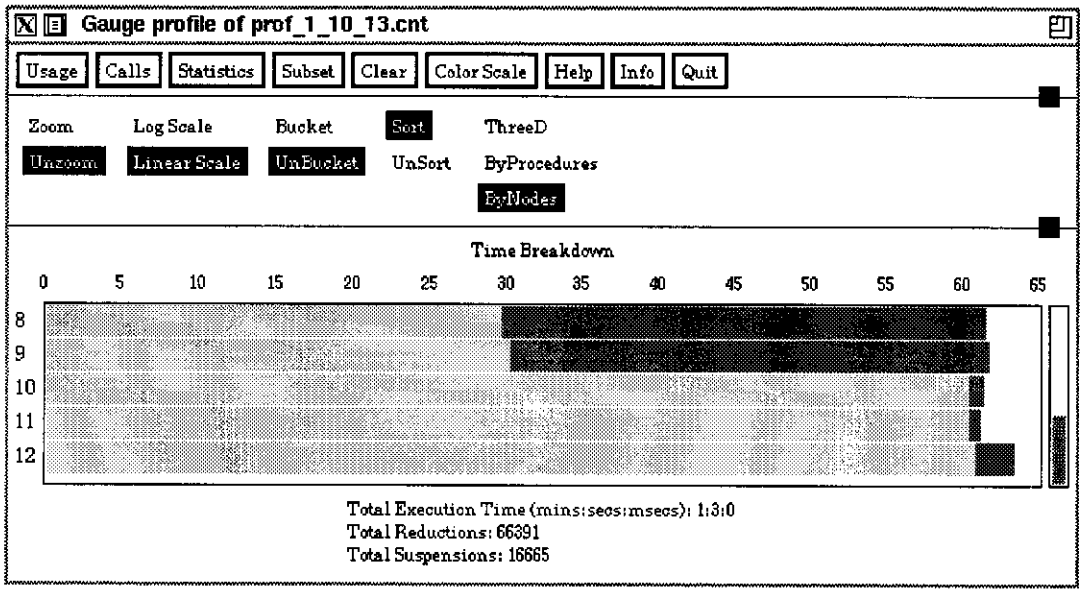

Figure 7.3: A stacked histogram showing the idle time and execution time

The final display type available in *Gauge* is a binned histogram. The independent axis of the histogram is a range of values which cover all the values present in a profile. The length of each bar is equal to the number of times a value in the range occurs in the profile. For example, if an execution time profile with values ranging from zero to 100 seconds is displayed in a 10 bin histogram, the length of the first bar is equal to the number of programs whose execution time is between zero and five seconds.

7.4 Interactive Data Analysis

A unique aspect of *Gauge* is its incorporation of interactive data analysis techniques. Interactive data analysis is a means with which the user can gain a better understanding of complex data sets. In *Gauge*, it allows the user to understand which elements of the profile data are of interest and to focus on those elements without being distracted by other data. It also aids

in understanding the relationship between different elements of the execution profile.

7.4.1 Related work in interactive data analysis

In trace oriented performance measurement systems, refining the data set is often a function of data collection. Specifically, each sensor has a filter component that determines if an event is to be recorded or not. The user must decide on the filter criteria before the program executes. Filters can be implemented and specified in a number of different ways. In IIE [SSS+83], a relational query language is defined and the data to be collected is specified in terms of a query in that language. The query compiler inserted sensors with the appropriate filter functions. A simpler approach is found in PIE [LSV+89] where sensors are enabled by the programmer specifying nodes of interest in a graphical representation of the program. Another option is to define a filter specification language and associate filter function written in this language with a sensor as in [Lin90]. The limitation of these techniques is that they all depend on *a priori* determination of which data will be of interest.

Some tools offer limited interactive control of the data display during visualization. Two commonly used techniques are zooming and scrolling. Rather than displaying all of the performance data at once, the data is viewed as through a window. Scrolling moves the window around and zooming adjusts the amount of data that can be seen through the window at one time. These techniques allow the user to interactively specify how much data is to be displayed; but they offer limited control over which data is to be displayed.

More general techniques are required if one wishes to display arbitrary subsets of the data, a capability that is important for performance visualization if we are to assume no *a priori* knowledge of the performance data. For example, one might first want to examine the execution time of every program on every node. A follow up question would be to ask how many times the top five time consuming programs were called. General interactive techniques that support on the fly construction of displays are needed.

The Seeplex [Cou88] performance visualization tool facilitates interactive analysis with a technique called projection pursuit in which views of the data are refined by defining a series of projections. Examples of projections include sorting, joining data sets, threshold filtering and a range of visual

presentations. A graphical editor is provided to specify the projections to be used and how data flows between them. Because Seeplex gives the user a great deal of control over which data is to be used and how it is to be presented, using the tool for simple tasks can be difficult [Cou].

7.4.2 Interactive data analysis in *Gauge*

Interactive data analysis in *Gauge* is based on the two functions.

- The ability to dynamically construct new views of performance data from an existing display
- The ability to interactively query the data and perform on the fly statistical analysis.

Essential to both of these functions is the concept of a *selection set*. A selection set specifies the input to a display or query operation.

A selection set is a subset of all possible processor/program pairs. The method used by *Gauge* to build a selection set is called *selection*, an interactive technique which uses a pointing device such as a mouse. An element of a histogram display is selected by positioning the mouse cursor over the object on the display, then clicking (pressing and releasing) a mouse button. The user is shown that the selection has been made by highlighting. For example, in Figure 7.1, the program **rh_chart:get_side** on processor six has been selected. If the selection is made from a 3D-histogram, then the selected program/processor pair is added to the selection set. If the selection is made from any of the other displays, all the processors or programs represented by the selected bar are added to the selection set. For example, assume the current display is two dimensional view of idle time per node. If the bar for node one is selected, then for each program p_i in the computation, the pairs $< p_i, 1 >$ are added to the selection set.

If the mouse is moved while the mouse button pressed, each element in the display the mouse cursor passes over is selected. If the mouse is over a selected point when the mouse button is pressed, then the operation becomes *unselect* and, until the button is released, the points passed over by the mouse are removed from the selection set.

There are three shortcuts for frequently used selections. All the programs on a processor can be selected by positioning the mouse cursor over the pro-

cessor number on the horizontal axis and clicking the mouse button. Likewise, a program can be selected on every node by clicking the mouse button when the mouse is positioned over the program name. The third shortcut, which unselects all current selections, is provided by the *clear* button in the top panel of the display.

The selection mechanism used in *Gauge* has much in common with a technique from interactive statistical analysis called brushing [BCW87]. With brushing, a variable sized box, or "brush", is swept over scatterplot matrices to select points of interest. The two methods differ in that with brushing, the size of region selected at one time can be changed; in *Gauge* the selection is always one element at a time. Using a long skinny brush, the data can be conditioned on a specific variable. In *Gauge* the same effect is achieved by selecting on a program name or processor number. While a variable sized brush can make certain section operations easier, the simpler approach used in *Gauge* is sufficient for the vast majority of selection sets needed for profiling.

7.4.3 Creating alternative views of performance data

Gauge provides the user with three mechanisms for constructing a new display of profile data from an existing display: subsetting, pivoting and sorting. The input to each of these operations is the selection set defined by the items currently selected in the display. If no items are selected, a default selection set containing all the points in the current display is used.

Subsetting is the simplest of the three operations. Subsetting creates a new histogram containing only the data in the selection set. By subsetting the data set, the user reduces the volume of data that must be examined, eliminating points that are not of interest. A second benefit of subsetting is that the mapping of the color scale onto the values being displayed is rescaled to fit the new range of data values. This is helpful when the original display is dominated by one or two extreme values. This is often the case with execution time displays of multilingual programs; the execution time of the foreign programs is much larger then that of *PCN* programs. By eliminating the large programs and their large execution times, the scale can be expanded, making detail that was previously obscured visible. Subsetting is initiated by pressing the *subset* button in the top pane of a histogram window.

Pivoting is a mechanism which projects three dimensional histogram data into a two dimensional histogram. Two pivots are possible. A pivot toward the node axis produces a two dimensional histogram displaying node number and profile data. Likewise, a pivot toward the program axis produces a two dimensional histogram displaying program names on one axis and profile data on the other. Each pair in the selection set defines two possible slices through the profile data: one parallel to the processor axis and one parallel to the program axis. When a pivot is made, the value displayed is the sum of all of the slices parallel to pivot axis.

Sorting is the third mechanism in *Gauge* used to alter the way data is displayed. Either axis can be sorted in order of decreasing value of the currently displayed profile data. In addition, the node axis can be unsorted back to node number order and the program axis can be unsorted back to the order in which programs are defined in their modules. In a 3D display, the sort is applied to the program axis. The value used for the sort is determined by summing the currently displayed data over the nodes in the selection set. In a 2D display the sort takes place on the axis displayed.

A sort is initiated by pressing the *sort* button in the middle panel of the histogram display. When a view of the data changes, the order in which the data is presented is preserved from view to view. To show how sorting is used, we present an example. A view of the execution time of all the programs in a computation presented in order of the amount of idle time on processor six is constructed by following these steps:

1. Display the idle time on a 3D histogram.

2. Select processor six.

3. Press the sort button. The program axis will be sorted so that the data on processor six is in decreasing order.

4. Switch to the execution time display.

.

The above example shows how specialized displays of performance data can be constructed. By combining all three viewing mechanisms: subsetting, pivoting and sorting, the user has a range of different methods to examine a performance experiment.

7.4.4 Querying values in a histogram

In addition to graphical displays, *Gauge* offers three ways to directly observe profile data: *program usage*, *program calls* and a statistical summary. We call these query operations, for their effect is to directly query the profile data. Query operations are used to obtain detailed information about specific data points in a profile.

A query is initiated by pressing the appropriate button in the top pane of a histogram window. The results of a query are displayed textually in a window separate from the histogram, as shown in Figure 7.4. This figure shows that any number of query results can be displayed simultaneously. This is convenient for making comparisons between different queries.

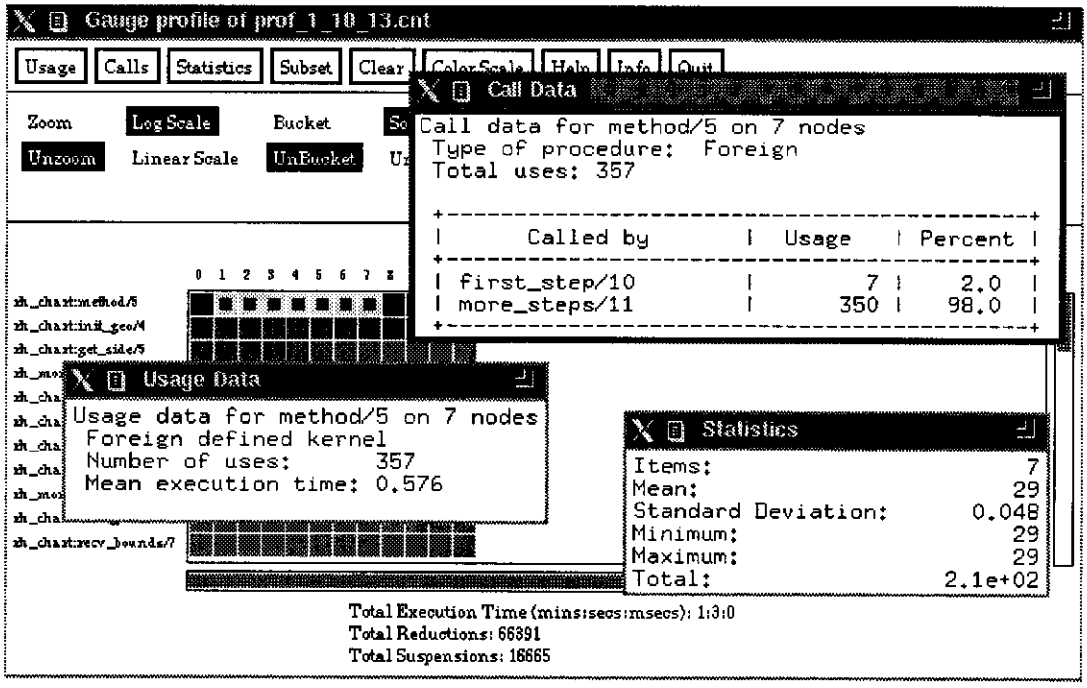

Figure 7.4: A direct query of profile data from a histogram

The input to a query operation is a selection set. The selection set for the *usage* and *calls* query is restricted in that only one program can be in the set. Any selection set is valid input to the *statistics* query.

The *usage* query presents summary information about a program on the

selected nodes. If the program is a foreign, then the total number of program uses and the mean execution time per use is displayed. If the program is written in *PCN*, the program type and total number of uses is displayed. Additionally, the number of times each implication guard in a choice composition succeeds and fails is shown.

The *calls* command identifies the callers of a program. Unlike tools such as gprof, we make no attempt to propagate profile information through the call graph. Statistical assumptions generally used to assign execution time to a caller are considered to be unreliable [PF88]. However, by selecting a single composition, a new window is created which contains a list of the compositions that call the selected program and the frequency of calls.

The last facility in *Gauge* displays first and second order statistics of selected data points. For example, knowing the variance of execution time for a specific program executing on all processors is important when evaluating how well balanced the computation is. This information is obtained by selecting the program and then requesting statistics using the command button in the upper pane of the histogram window.

7.5 Summary

Visualization tools are becoming the accepted approach to presenting performance data from parallel programs. In this chapter, we described *Gauge*, a tool for visualizing parallel execution profiles. *Gauge* is unique in its simple but effective means of presentation and its focus on interactive methods for exploring performance data.

In designing *Gauge*, we strove to make it simple to use. In combination with *PCN*, *Gauge* forms an environment for developing efficient parallel programs. When performance measurement tools are easy to use and tightly integrated, performance measurement and performance improvement become part of the normal program development cycle, enabling programmers to produce better parallel programs.

Chapter 8

Profiling to Improve Program Performance: A Case Study

In this chapter, we demonstrate how the performance visualization tools and techniques developed in this dissertation are used in practice. To this end, we study the performance of a nontrivial parallel application. We show how the profile data is interpreted and used to ultimately improve the performance of the application. As part the discussion, we highlight the advantages of profiling in general and our approach to performance visualization in particular. We will also point out where profiling limits our analysis.

8.1 The Application

The program whose performance we will study is a computational fluid dynamics program called FLOW [Lin], which computes wavy Taylor vortices. Taylor vortices are a particular type of fluid movement that occurs in an incompressible viscous fluid trapped between two concentric cylinders, one rotating with respect to the other. In FLOW, Taylor vortices are computed by numerically solving a system of three dimensional Navier-Stokes equations. The only input to the program is the problem size: the number of grid points in each dimension on which the solution is calculated.

The solution method used by FLOW is a Gauss-Seidel relaxation algorithm. The body of the program consists of a double nested loop with the Gauss-Seidel iterations being the inner loop. The algorithm is controlled

by two convergence criteria: one to terminate the inner loop and one to terminate the outer loop.

For a given grid spacing, the number of iterations needed by the relaxation algorithm to converge depends on the convergence criteria for relaxation and the size of the problem being solved. The number of iterations required in the outer loop depends on the amount of error acceptable in the solution and the problem size. Because the relative amount of time spent in the two loops of FLOW changes with problem size, performance measurements should be made during the execution of a realistic problem size and cover the entire duration of the program's execution. This is precisely the situation where profiling is superior to event tracing.

8.1.1 Parallel implementation

In FLOW, parallel execution is obtained by decomposing the grid space in two dimensions [FJL+88]. With a problem size of $I \times J \times K$ grid points, the domain decomposition produces lines of length I parallel to the i axis at each point (j, k). Operations in the i dimension are done sequentially; operations in each (j, k) take place in parallel.

About 3500 lines of Fortran code implement sequential operations in FLOW. These operations include the Gauss-Seidel iteration over a line, computation of boundary points and calculating the residual error in the solution.

The parallel component of FLOW consists of about 700 lines of *PCN* code. The *PCN* code is organized as a master and a number of workers, all of which execute concurrently. The computation starts with the master, which starts a worker on each of the remaining processors. After the workers have started, the master distributes the lines in the grid to the workers and, using shared definitional variables, interconnects the lines in the communications pattern shown in Figure 8.1. The master also serves as the collection point for the global values used to determine convergence: maximum residual for each relaxation step and average pressure for the outer loop.

Each worker is responsible for calculating the solution for a set of lines. All the lines are computed in parallel. The computation for a line is a double nested loop. At the end of each inner loop iteration the residual value on a worker is sent to the master. The master computes the global maximum residual value and broadcasts it back to all of the workers. This value is com-

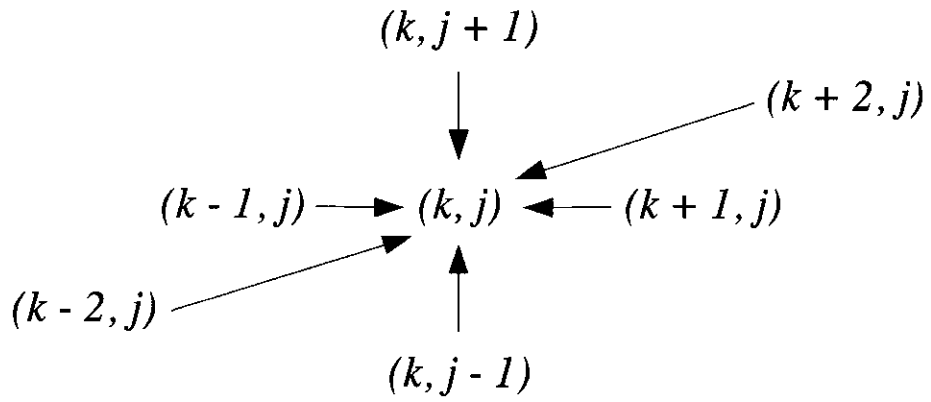

Figure 8.1: The communications pattern between lines in FLOW

pared to the convergence criteria to determine if another inner loop iteration is needed. A similar process takes place with the outer loop and the average pressure value.

8.1.2 Execution environment

In this study, FLOW was run on a Symult S2010 parallel computer [Sym89]. Each processing node on the S2010 consists of a Motorola 68020 microprocessor with a 68881 floating point coprocessor, memory and a routing chip which implements message passing between processors. Each router is connected to its four nearest neighbors, forming a rectangular mesh interconnection topology.

The S2010 used has a total of 192 nodes — 64 of the nodes have 4 Mbytes of memory and the remaining nodes have 3 Mbytes. When FLOW runs, all the data structures for the grid points computed on a node must fit into the available node memory — the size of the largest problem which can be solved is therefore limited by the amount of memory available on a node after the FLOW code has been loaded. This situation makes efficient storage of performance data very important. If tracing were used, the buffer space required to store the event log will force us to run a smaller problem size. From the discussion of Section 8.1, we know that changing the problem size will alter the characteristics of the program, placing the validity of the

performance data in question.

FLOW executes under the standard *PCN* system. As discussed in Chapter 6, profiling support is integrated into *PCN* and thus no special preparation of FLOW was required to conduct this performance improvement study. The measurement overhead for profiling FLOW is shown in Table 8.1. These measurements differ from those in Chapter 5 in that the cost of the Fortran code is included in determining the total program size and execution time. The statistics of Table 8.1 show that in actual use, the costs of profiling FLOW are an order of magnitude below the design goals for the profiler.

Measurement Overhead	
Type	Overhead
Storage	2.6 %
Run time	0.18 %

Table 8.1: The measurement overhead for FLOW (including Fortran procedures)

8.2 Performance Improvement of FLOW

In this section, we demonstrate how to use *Gauge* to: 1) understand the behavior of FLOW, 2) determine if there are aspects of the behavior that degrade the program's performance; and 3) guide the process of identifying and correcting the cause of a performance problem. Performance evaluation with *Gauge* is a dynamic, interactive process. Unfortunately, describing the users interaction with *Gauge* is difficult. In this chapter, we show the path taken to improve the performance of FLOW. What we cannot show is the high degree of interaction with *Gauge* that enables us to find that path.

Performance improvement is an iterative process which repeats until the programmer is satisfied with the resulting performance of the program. With *Gauge*, the steps to improve performance are:

1. Conduct a performance experiment.

2. Examine the profile data from the experiment, looking for aspects of the profile that indicate a performance problem. Continue on to the next step if a significant problem is found.

3. Formulate a model of the program's actions which accounts for the performance bottleneck. Alternative views of the profile and interactive statistical queries are useful in this step.

4. Based on the model of program activity, propose an alteration to the program to correct the performance problem.

5. Conduct a performance experiment with the altered program.

6. Examine the new profile, comparing it with the original. Of interest is: 1) how the behavior of the program has changed, 2) how that change affects the performance bottleneck and 3) the performance of the program overall. If the alteration is successful, the performance bottleneck is eliminated and the program performance improves. Program improvement continues with Step 2. A failure to improve program performance can be caused by incorrectly identifying a performance bottleneck, having an erroneous understanding of what the program is doing or altering the program in a manner that does not actually fix a correctly identified performance problem. A detailed examination of profiles from the original and altered program can often reveal which of these situations is true. Again, interactive statistical analysis and constructing different views of the data are important parts of the examination. We return to Step 3 and try again.

8.2.1 Preliminary analysis

FLOW was executed on 160 nodes with a problem size of $32 \times 32 \times 32$. Figure 8.2 is an execution time profile for the execution of FLOW. The horizontal axis is labeled with processor numbers from 0 to 159 and the vertical axis is labeled with the names of the different programs in the computation. The programs have been sorted in order of decreasing total execution time. The scale of the profile is in seconds.

The summary data at the bottom of the profile shows that the program completed in 88 minutes and 38 seconds. Over twenty million *PCN* programs

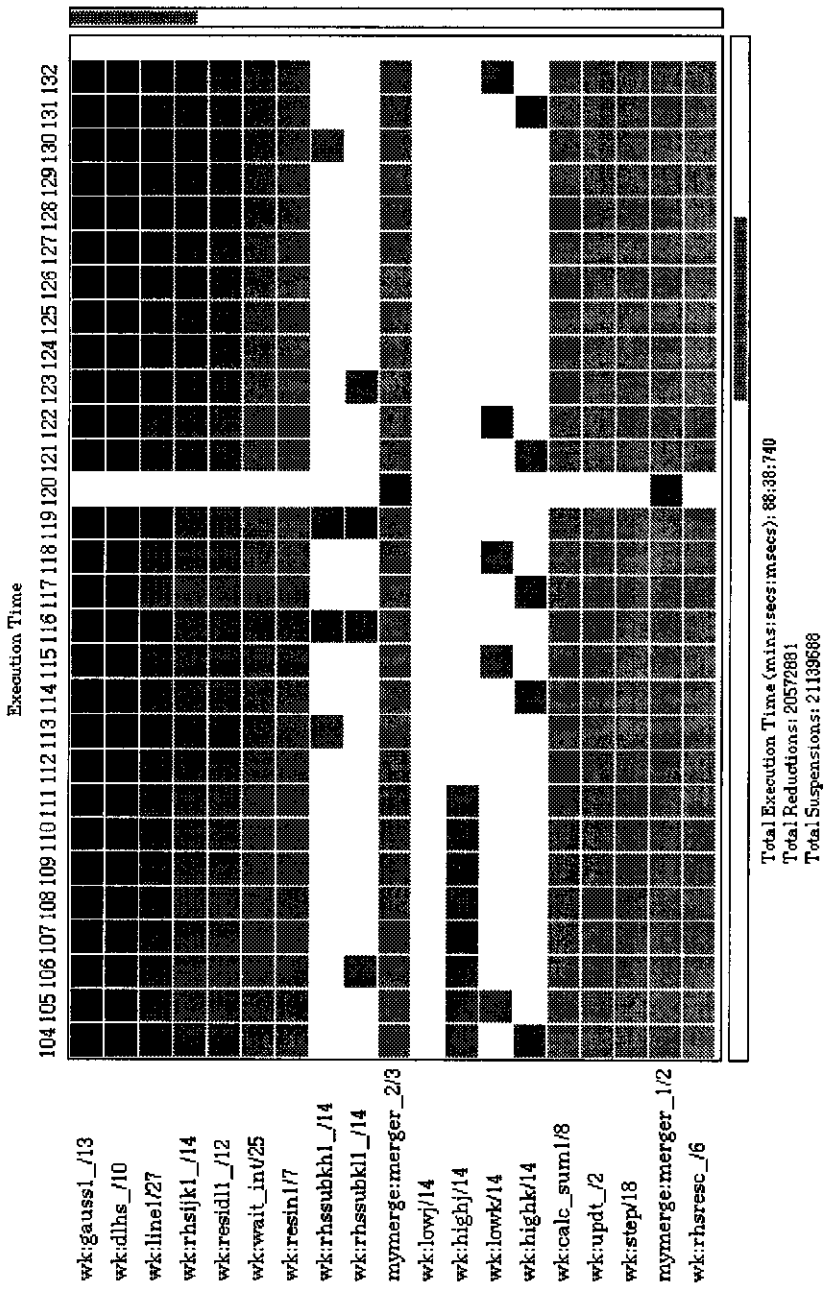

Figure 8.2: The execution time profile of FLOW

were executed. If an execution trace were to be used to record just these events, over 100 Mbytes of data would be recorded. The *Gauge* profile for the entire computation, including the performance models, is less than 1 Mbyte long.

The profile itself shows that FLOW spends more time in a Fortran procedure called **gauss1** than in any other procedure in the program. In **gauss1**, a Gauss-Seidel sweep on a line is performed. The first *PCN* program to appear in the profile is **line1**, which collects the values from neighboring lines and calls **gauss1**. These programs comprise the inner loop of FLOW and their position in the profile is not surprising.

The master is executed on node 120. The fact that node 120 is doing something different from the other nodes in the computation is clearly shown in the profile. Another area of interest is found in the programs **lowj**, **highj**, **lowk** and **highk**. These programs implement the boundary conditions on the solution. We can see that although a pattern is present, the boundaries are not mapped evenly onto the processing nodes. However, their total computation is in the one second range and as we shall see shortly, this is inconsequential.

8.2.2 Improving the load balance in FLOW

The first question to be answered by *Gauge* is: how well does FLOW perform? Without a performance measurement system, the an answer to this questions is obtained by executing the complete program several times and computing speedup. From speedup the efficiency of the computation can be obtained and used to determine if a performance bottleneck exists. With *Gauge*, we look to utilization rather than efficiency. Processor utilization for FLOW is determined from the execution time breakdown shown in Figure 8.3. In this view of the profile, processors are on the vertical axis and the horizontal axis shows time in seconds. The light gray part of each bar is the amount of time a processor spends idle; the dark gray area is the amount of time a processor spends computing. The distance between the end of a bar in Figure 8.3 and the total execution time of the FLOW is the amount of time a processor spends in communication and garbage collection.

The peak idle time is on node 120, the master, as would be expected. However, almost 2000 seconds of idle time can be seen on processors performing the relaxation computation. Using the interactive statistical capability

of *Gauge*, the average idle time and average execution time are determined. From these numbers we can determine how much time FLOW spends executing code, communicating and idle. This breakdown in shown in Table 8.2. The processor utilization in FLOW is about 60% and we investigate further into the behavior of the program to try to improve this.

Execution Time Breakdown	
Activity	Percent
Computing	61
Idle	27
Communicating	12

Table 8.2: Breakdown of program activity during the execution of FLOW

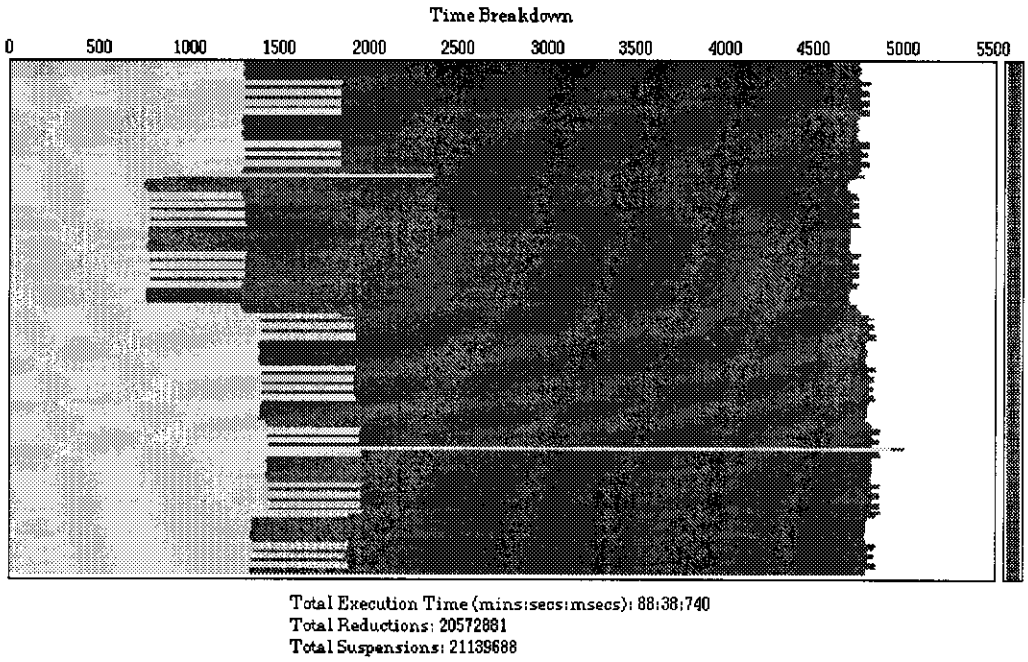

Figure 8.3: Breakdown of time spent in FLOW

A peculiar aspect of Figure 8.3 is that, excluding the master, idle time

on a processor has tree distinct values. If we pivot the display of Figure 8.2 into the processor axis, shown in Figure 8.4, we see that the variation in idle time is caused by a variation in computation on a node. There are two possibilities: different numbers of tasks are mapped to the nodes or the amount of computation varies from task to task. Given the presence of three discreet execution times, we suspect a problem with the function that maps lines onto processors.

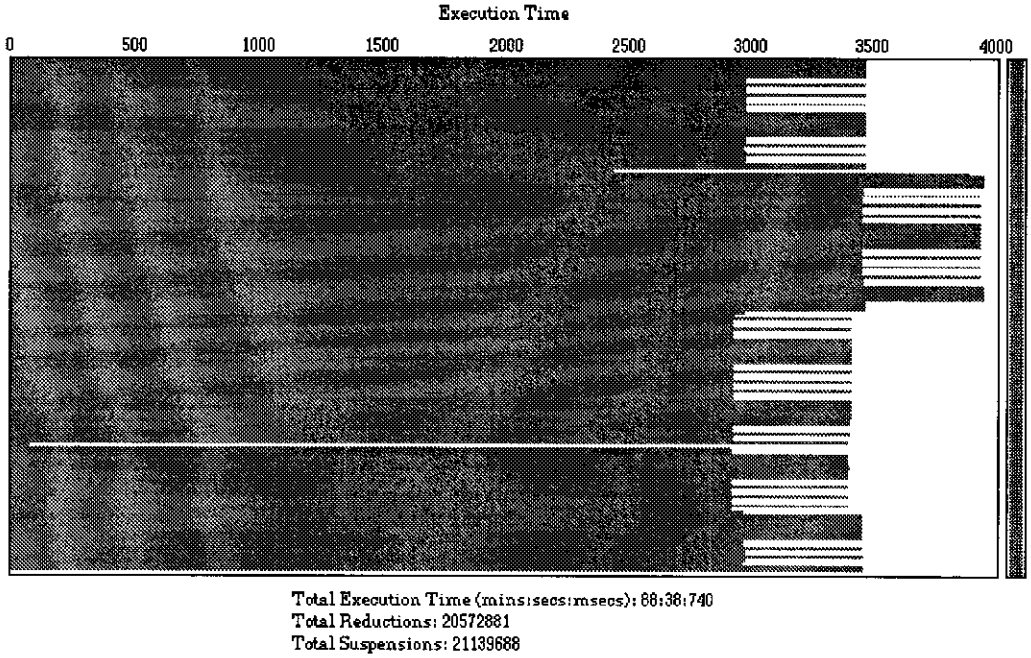

Figure 8.4: Load balance for FLOW

This conjecture can be confirmed by looking at the execution frequency profile of **line1** or **gauss1**, programs which execute once per task per iteration. Selecting **gauss1** in Figure 8.2, pivoting into the node axis and displaying execution frequency, we obtain Figure 8.5. From this display we see that the variation in execution time can be explained by different numbers of lines being mapped to a processor, not by variation in the time required to compute a line.

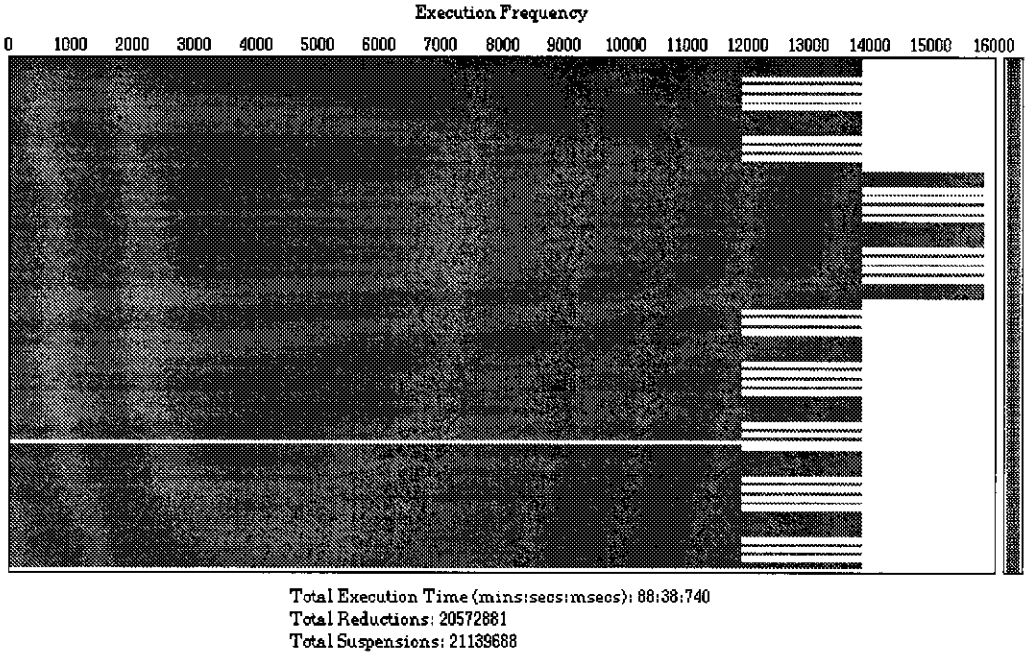

Figure 8.5: Per processor execution frequency for **gauss1** in FLOW

FLOW maps a line defined by the point (j,k) onto a processor n by:

$$n(j,k) = (j * (K+1) + k) \bmod \mathcal{N}$$

where $\mathcal{N}$ is the total number of processors and K is the problem size in the k dimension. This mapping was chosen by FLOW's author because of it's simplicity. Given the caveat against optimizing too early in the design of a program [KP78], this choice is justified. However, the profile clearly shows that this mapping causes an extra 10% in idle time due to load imbalance.

FLOW1 is a version of FLOW in which the original mapping function is replaced by:

$$\begin{aligned} L &= \lfloor (J * K)/\mathcal{N} \rfloor \\ E &= (J * K) \bmod \mathcal{N} \\ n(j,k) &= \begin{cases} \frac{j*K+k}{L+1} & \text{if } j*K+k < E*(L+1) \\ \frac{(j*K+k)-(E*(L+1))}{L} + E & \text{otherwise} \end{cases} \end{aligned} \tag{8.1}$$

Equation 8.1 ensures that the first E processors will execute $L+1$ and the remaining processor will execute L lines. For our 33×33 problem, there are 1089 lines. The new mapping function assigns the 7 lines to the first 130 processors and 6 lines to the remaining processors. Another advantage of the mapping function of Equation 8.1 is that it assigns consecutive values of k to the same processor, reducing the amount of communication needed. While the mapping is more complex, it is a one time cost that is incurred only during program startup.

To see how well the new mapping works, the same data from Figure 8.5 is shown for FLOW1 in Figure 8.6. As we see, there are only two execution frequencies for **gauss1**. The new mapping works as expected.

The amount of execution time spent in **gauss1** on each processor is shown in Figure 8.7. Node 35 presents a puzzle: although it executes **gauss1** the same number of times as nodes 0 though 135, its execution time is 500 seconds less. This is probably caused by an interaction between the location of boundaries in the grid, the number of processors used and the mapping function.

The execution time of FLOW1 is 15% less than that for FLOW. The source of the improvement can be seen in the execution time breakdown for

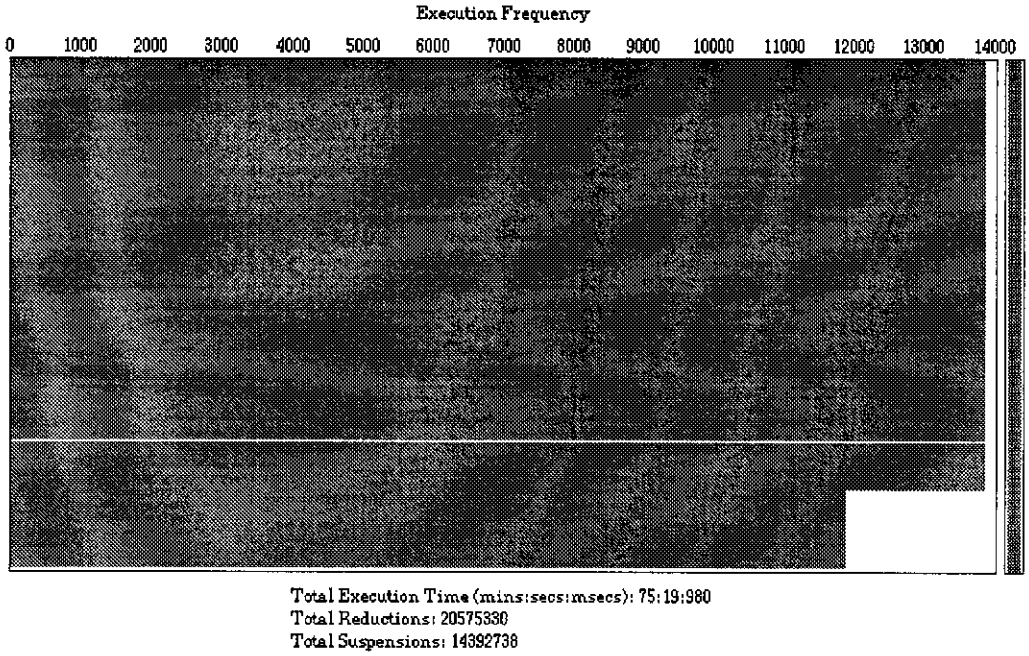

Figure 8.6: Execution frequency of **gauss1** in FLOW1

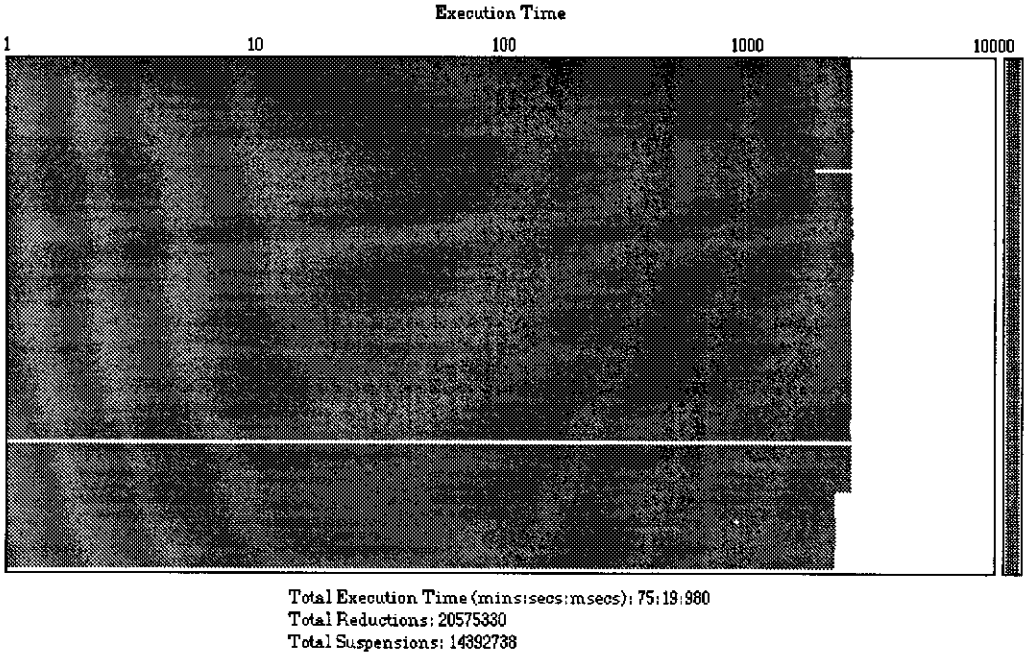

Figure 8.7: Execution time for **gauss1** in FLOW1

FLOW1 shown in Figure 8.8. The breakdown of execution time in FLOW1 is shown in Table 8.3. Although the computation time is the same in FLOW and FLOW1, we have increased the utilization to 73% by decreasing the idle time by 41% and the communications time by 35%.

Execution Time Breakdown	
Activity	Percent
Computing	73
Idle	18
Communicating	9

Table 8.3: Breakdown of execution time for FLOW1

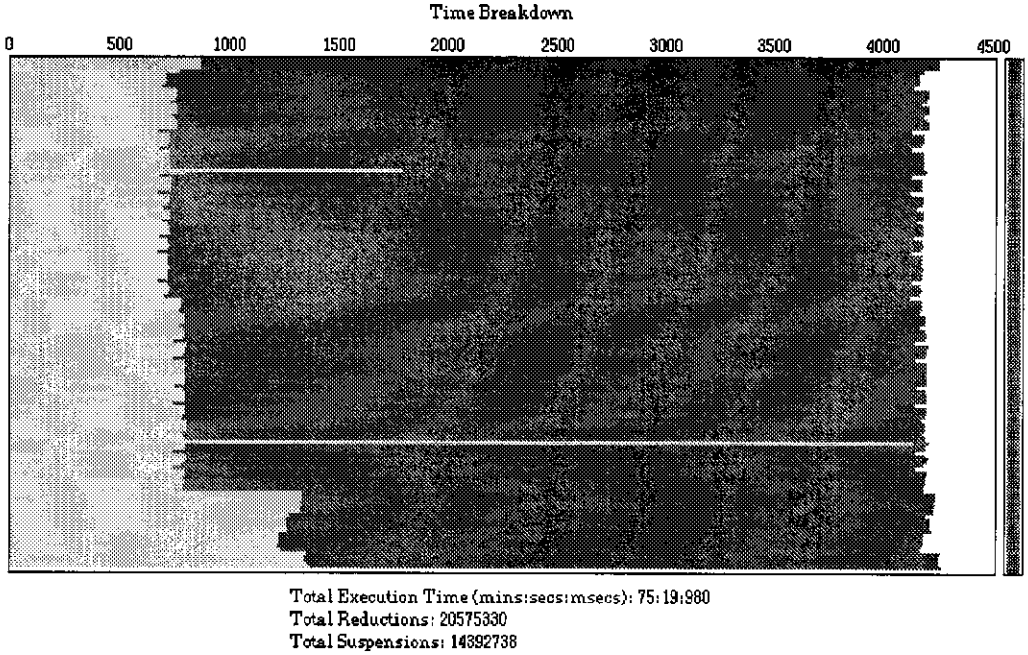

Figure 8.8: Breakdown of execution times in FLOW1

Figure 8.9 shows the idle time for FLOW1 on a per node basis. Subsetting has been used to remove the master node from this figure. The effect of

processing one more line on nodes 0 through 135 than on 136 though 159 is apparent in the difference of 500 seconds between the idle times on the two sets of nodes. In addition to the idle time caused by the unequal mapping of nodes to processors, there is an additional 800 seconds of idle time at each node. We focus on this idle time next.

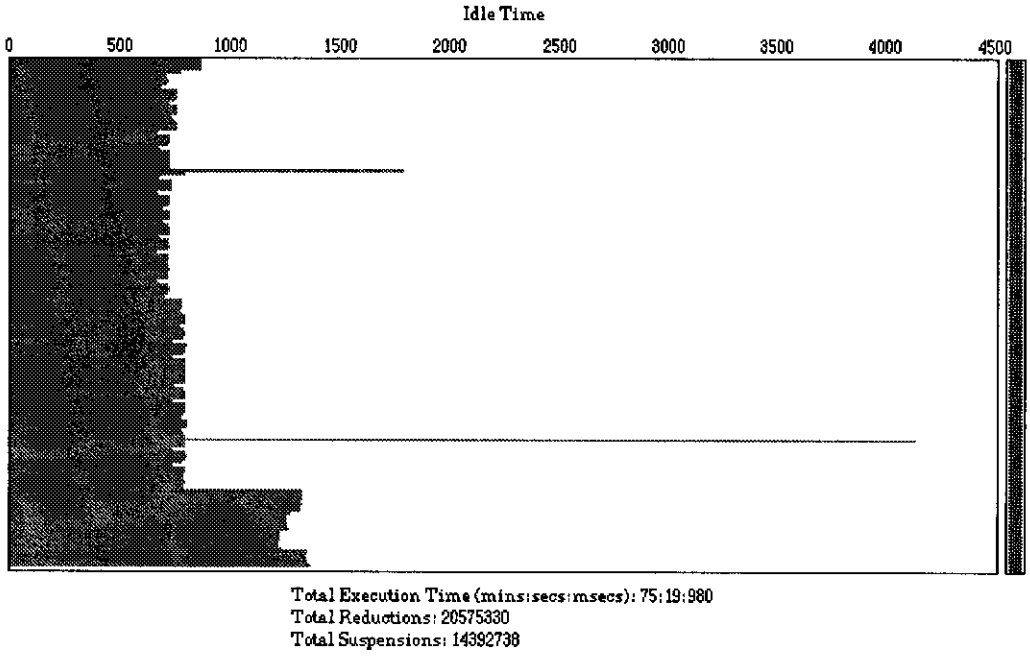

Figure 8.9: The per node idle times for FLOW1

8.3 Decreasing the Cost of Global Synchronization

In Figure 8.10, the idle time of each program and node is shown. Two programs are responsible for 90% of the idle time on a node: wait_int and line1. Because of their prominence in the profile, we focus only on these two routines.

The *PCN* code for line1 is shown in Figure 8.11. There is one invocation of line1 for each line in the computation. For each execution, the variable

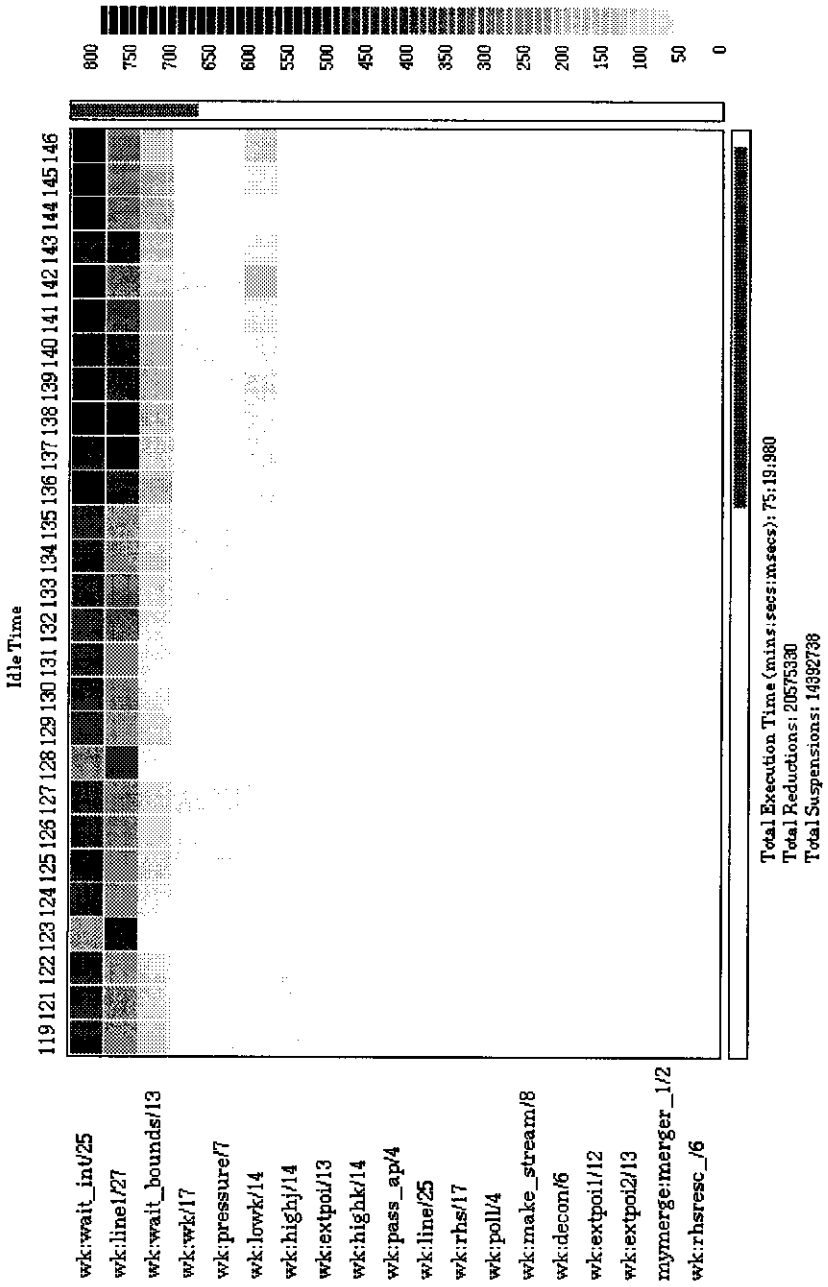

Figure 8.10: The per node, per program breakdown of idle times in FLOW1

Sweep is checked. If it is greater than zero, the current value of lines $(j-1,k)$, $(j+1,k)$, $(j,k+1)$, $(j,k-1)$, $(j,k+2)$ and $(j,k-2)$ are obtained from the variables Jp, Jm, Kp, Km, Kpp and Kmm and the line (j,k) is updated by calling gauss1. The assignment of DQQJK to the head of the list ME makes the current value of the line available to other invocations of line1. Finally, line1 is called recursively to perform the next update of the line. If Sweep is zero, the current value for the four adjacent grid points is obtained and the residual for the line is calculated. The assignment to RSin sends the residual to a process which computes the maximum residual value over all lines. Then the program wait_int is called to perform the convergence test.

The variables Jp, Jm, Kp, Km, Kpp and Kmm are streams on which the current value of adjacent lines are communicated to line1. If a neighboring line is computed on a different processor and the value has not been sent, then line1 must wait for the value to be communicated. If no other compositions are on the active queue, this wait shows up as idle time. Since all the other variables in line1 are mutable or locally produced, all the idle time of line1 is due to waiting for one or more of the above variables.

The convergence check at the end of a relaxation iteration is performed by wait_int, whose *PCN* code is found in Figure 8.12. The convergence test requires the maximum line residual, which is broadcast to all instances of wait_int in the variable RESIN. If the calculation has converged, wait_int informs the process responsible for starting another outer loop iteration. If relaxation has not converged then a new line1 computation is started. There are two possible causes of idle time in wait_int: obtaining the value of RESIN in both implications and obtaining the value of Kpp and Kmm in the first implication.

Since the idle time of wait_int is the larger of the two, we start here. Our initial hypothesis is that the bulk of the idle time is caused by waiting for RESIN. Our reasoning for suspecting RESIN over Jp, Jm, Kp, Km, Kpp or Kmm is:

- The calculation of RESIN requires global communication
- The profile shows that the second implication of wait_int executes 16 times as often as the first. The second implication does not need the value of any neighboring points.
- The second implication of line1, from which wait_int is called, uses all

```
line1(j,k,PC,LC,Sweep,IMAS,JMAS,KMAS,Jm,Km,Jp,Kp,ME,Kmm,Kpp,DQQJK,
        RSFRE,GHSFRE,G,Wk1,Str,RESIN,RESIN1,RSin,RSin1)
    double G[],DQQJK[],RSFRE[],GHSFRE[],resin;
    int IR,JR,KR,MR;

{? /* Get values from neighboring grid cells */
  0 < Sweep, Jm?=[X2|X2s], Km?=[X3|X3s],
  Jp?=[X5|X5s], Kp?=[X6|X6s],Kmm?=[_|X1s], Kpp?=[_|X4s] ->
    {; /* relax the line */
        gauss1_(PC,LC,DQQJK,X2,X5,X3,X6,RSFRE,IMAS,JMAS,KMAS,4,G),
        R1=DQQJK, ME=[R1|Rs],
        line1(j,k,PC,1-LC,Sweep-LC,IMAS,JMAS,KMAS,X2s,X3s,X5s,X6s,Rs,
            X1s,X4s,DQQJK,RSFRE,GHSFRE,G,Wk1,Str,RESIN,RESIN1,RSin,RSin1)
    },
  /* Last sweep through the data? */
  Sweep==0,Jm?=[Xjm|_],Km?=[Xkm|_],Jp?=[Xjp|_],Kp?=[Xkp|_] ->
    {; /* Calculate residual */
        dlhs_(GHSFRE,DQQJK,Xjp,Xjm,Xkp,Xkm,IMAS,JMAS,KMAS,G),
        resid1_(j,k,GHSFRE,RSFRE,IMAS,JMAS,KMAS,IR,JR,KR,MR,resin),
        /* Send the residual to master */                          20
        RSin=[resin|RSin2],
        /* Wait for maximum residual value from master */
        wait_int(RESIN,RESIN1,RSin2,RSin1,Wk1,IMAS,JMAS,KMAS,
                Jm,Km,Jp,Kp,Kmm,Kpp,ME,DQQJK,RSFRE,GHSFRE,G,Str,PC,j,k)
    }
}
```

Figure 8.11: *PCN* code for line1

```
wait_int(RESIN,RESIN1,RSin2,RSin1,Wk1,IMAS,JMAS,KMAS,
         Jm,Km,Jp,Kp,Kmm,Kpp,
         ME,DQQJK,RSFRE,GHSFRE,G,Str,PC,j,k)
    double DQQJK[],RSFRE[],GHSFRE[],G[];
{?
  /* Inner loop value has converged.
     Get values of neighboring grid cells */

  RESIN ?= [H|T], H<=0.005,
  Jm?=[_|A], Km?=[_|B], Jp?=[_|C], Kp?=[_|D], Kmm?=[_|E], Kpp?=[_|F] ->
    {|| /* Prepare for the next iteration of the outer loop */
         Wk1={Str,A,B,C,D,ME,E,F}, RESIN1=T,RSin2=RSin1 },

  /* Inner loop value has not yet converged */

  RESIN ?= [H|T], H>0.005 ->
    /* Start a new reduction step */
    line1(j,k,PC,0,3,IMAS,JMAS,KMAS,Jm,Km,Jp,Kp,ME,Kmm,Kpp,DQQJK,
          RSFRE,GHSFRE,G,Wk1,Str,T,RESIN1,RSin2,RSin1)
}
```

Figure 8.12: *PCN* code for wait_int

neighboring points except Kpp and Kmm. Thus idle time can occur only while waiting for these two variables. However, since consecutive k points are mapped to the same processor, we would expect 83% of the Kpp and Kmm values needed to be generated locally.

Limitations in the amount of detail available in a profile force us to infer what is going on in FLOW1. In situations such as this, an event trace can be useful. By defining trace events to capture when wait_int is waiting for RESIN and when it is waiting for Kmm or Kpp, we can find out exactly what the cause of the idle time is. Guided by the profile, we can limit the number of trace events generated on a node to about 100,000.

If RESIN is the cause of the idle time in wait_int, increasing the speed at which the value of RESIN is sent to each processor should reduce the idle time. In FLOW1, the value of RESIN is not needed until wait_int is executed. Since RESIN is produced on the master, this means that wait_int must wait until the request for the value of RESIN has been sent to the master, processed and sent back before execution can continue. Because wait_int is called when the iteration terminates, there is no other work available to mask this communications latency and idle time results.

A solution to this problem is to start a computation executing in parallel with line1 which requests the next value of RESIN as soon as the current one arrives. This allows the latency of the request to be masked by the computation line1. Because RESIN values are generated one per iteration, no flow control is needed in the process that performs the prefetch. This version of FLOW is called FLOW2.

The idle time profile for FLOW2 is shown in Figure 8.10. This figure shows the result of an interactive statistical query of the idle time in wait_int on the first nine processors. The results are disappointing; there is virtually no change in the idle time of wait_int and the execution time of FLOW2 is within a few seconds of FLOW1.

At this point, we must revise our interpretation of what is holding up the execution of wait_int. If global communication are not the problem, then the values of Kmm and Kpp must be the source. The idle time in wait_int and line1 turn out to be caused by the same problem.

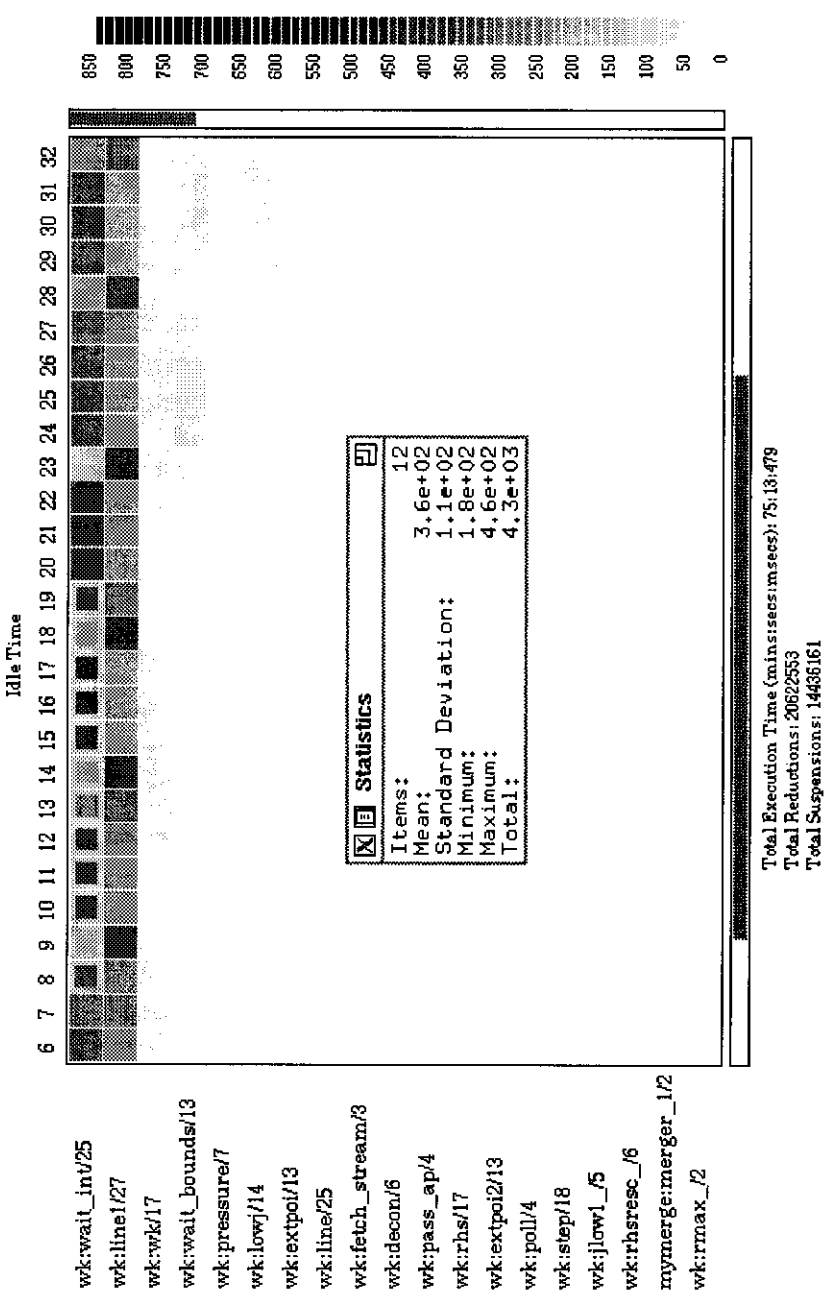

Figure 8.13: Breakdown of idle times in FLOW2

8.4 Decreasing the Cost of Relaxation Step

We now look at reducing the idle time associated with line1. Our hope is that in reducing this time, the idle time of wait_int will decrease as well. Every execution of line1 will require at least two values from another processor: Jp and Jm. Given that there are at least 6 lines per processor, we would expect that there would be enough computation to mask the latency of most communications of neighbors. Thus the prefetch technique that we applied in the previous section will probably fail here as well.

Something must be preventing the values of neighboring points from being communicated in a timely fashion. We suspect the problem is caused by a peculiarity of the *PCN* implementation. While a Fortran computation is taking place, the *PCM* is in the reduction component and therefore cannot process any communications. We see from the execution profile that if a *PCM* is executing gauss1, a processor waiting for a value can end up waiting up to .19 seconds before the *PCM* will respond. If we assume that only the first line1 execution of an iteration on a node will have to wait, the per invocation waiting time for line1 is around .14 seconds. The data supports the hypothesis that line1 is being forced to wait for a gauss1 call on another processor to complete.

The solution to this problem is simple. The single Fortran procedure gauss1 is replaced by three procedures: 1) init_gauss, which starts a Gauss-Seidel iteration, 2) gauss_loop, which updates a subset of the points in a line and 3) end_gauss, which finishes the update on a line. Thus gauss1 becomes:

```
{; init_gauss,
  gauss_loop, ... gauss_loop
  end_gauss
}
```

The number of gauss_loop calls used are specified by an argument to init_gauss.

When composed sequentially, these smaller routines perform the same function as gauss1 while allowing communication with other *PCM* to occur more frequently. We will call this version of the program FLOW3. In Figure 8.14, we can see the idle time resulting from splitting gauss1 into four pieces. Instead of processing one 33 point line at once, the line is processed in eight point chunks.

As we expected, the idle time of both wait_int and line1 have decreased. The overall performance achieved is shown by the profile in Figure 8.15. The

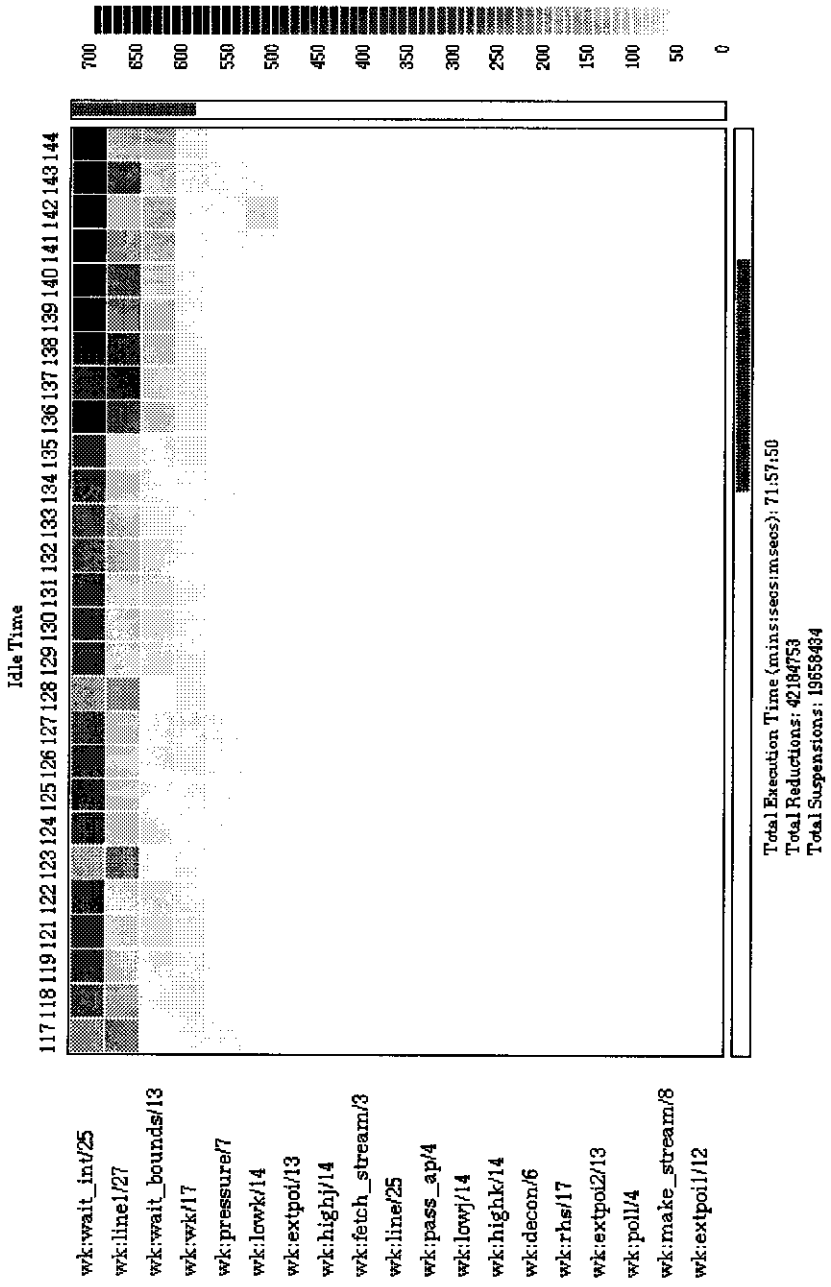

Figure 8.14: The idle time in FLOW3

breakdown of execution times for FLOW3 is shown in Table 8.4. We see that the processor utilization is now up to 79% and the performance of FLOW3 is 4.4% better than FLOW1 and 18.8% better than FLOW.

Execution Time Breakdown	
Activity	Percent
Computing	79
Idle	12
Communicating	9

Table 8.4: The breakdown of execution time for FLOW3

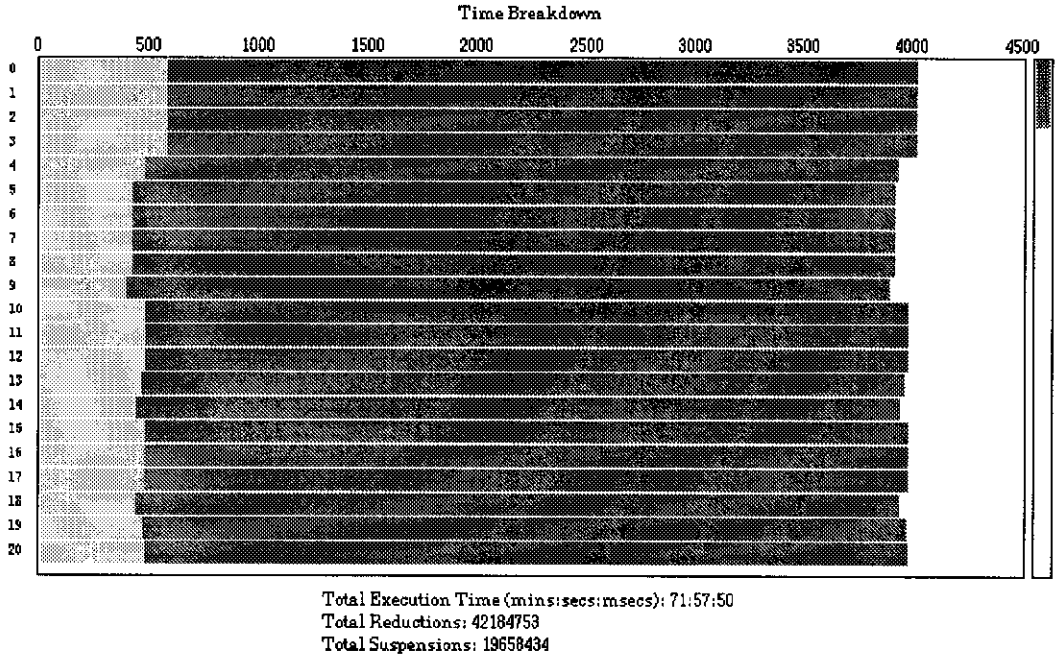

Figure 8.15: Breakdown of idle times in FLOW3

It is of interest to know what the cost of splitting up **gauss1** is in terms of execution time. The execution time profile for FLOW3 is shown in Figure 8.16. In this figure, we show the usage statistics for each of the elements

in the split version of **gauss1**: initialization, the main loop, and post loop calculations. The time spent in **init_gauss** and **end_gauss** fell below the minimum value that can be recorded on the Symult. The loop itself took .048 seconds per chunk, or .192 seconds per line, within 1% of the time it takes to process a line all at once. The execution time of **line1**, which calls **gauss**, increases by 26%. This increase is due to the overhead of the additional sequential composition.

There is still room for improvement in FLOW. Further analysis proceeds as we have shown and we conclude the case study here.

8.5 Summary

In this chapter, we have shown how the *PCN* profiler and *Gauge* are used to improve the performance of a parallel application. In the process, we outlined a general procedure that can be used as the basis for any performance improvement study.

The analysis presented here relied heavily on facilities that are unique to *Gauge*. Dynamic statistical analysis, subsetting, and pivoting were all extensively used.

A final note on the virtues of profiling: at no point during this study did we concern ourselves with the mechanics of generating performance data. No matter what modifications were made to the program, the profile was alway generated and available. Since we did not have to worry about measurement, we were able to concentrate on the program.

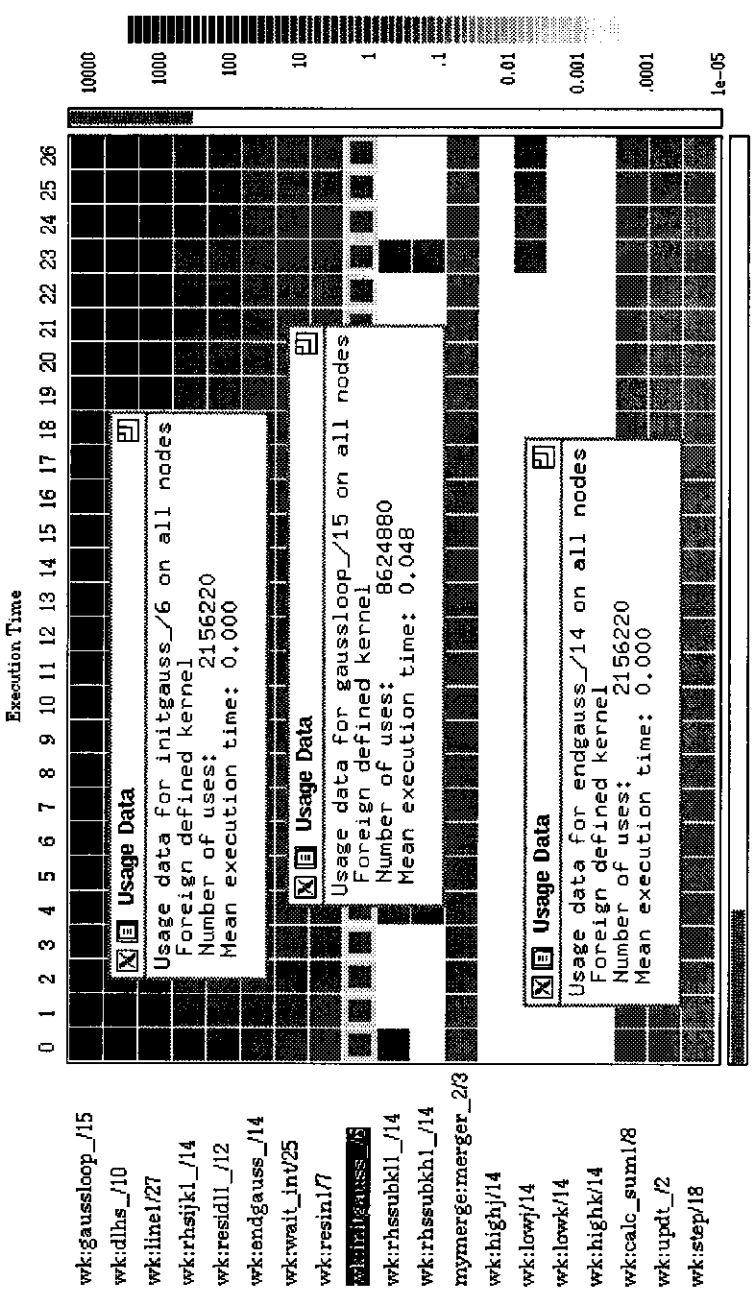

Figure 8.16: Breakdown of execution times in FLOW3

Chapter 9

Summary and Conclusions

We started this dissertation with the thesis that execution profiling is an important means to understanding and improving the behavior of parallel programs. In Chapter 8, we demonstrated this to be true by improving the performance of a large parallel program executing on 160 processors.

In addition to introducing profiling as an important tool, our research makes the following contributions.

- We have developed novel, low overhead measurement techniques to obtain execution profiles from parallel programs. Only simple counters and a microsecond timer are required. These techniques can be used on any commercially available parallel computer.

- Our measurement techniques were demonstrated by integrating them into the *PCN* parallel programming system.

- We have developed a new approach for visualizing performance data obtained from parallel programs. This approach is demonstrated in *Gauge*, a performance visualization tool we designed and built.

- We have developed a methodology for performance understanding and improvement based on parallel execution profiling. This methodology is demonstrated on a large parallel program.

The tools and techniques we have developed are an integral part of the *PCN* system. In total, about four man-years of effort have gone into the

development of *PCN*. Of this, developing the facilities for performance measurement account for about one quarter of this effort.

9.1 Future Work

As they stand, the methods developed in this dissertation have been used to implement a practical profiler for parallel programs. Every day, *PCN* and *Gauge* are used to develop parallel programs that solve difficult computing problems [TF]. However, there are a number of directions that this work can be taken.

9.1.1 Alternative uses of performance data

In Chapter 7, we showed how performance visualization is used to present the performance data to a user. However, performance data can also be used by other components of the parallel processing system, for example the compiler and runtime system.

Of particular interest is the use of profile data to partition, map and schedule tasks with a program. Some work in this area has already been done [GT88, SH86, Cyt84]. Because of the continuously monitored nature of programs developed under our system, the entire execution history of a program can be kept and used as a basis for compile time resource allocation.

The profile data for a program is kept on the processing node as the program executes. An interesting question is: can this data be put to use as the program executes? It is possible that by examining the profile data collected up to a point in the computation, the future behavior of a program can be predicted. This prediction can be used to for load balancing, repartitioning the computation, etc.

9.1.2 Snapshots and profiling

Profiles do not contain any temporal information. However, during program execution, it is possible to capture a snapshot of the profile data that has been collected up to that point. By looking at the difference between successive snapshots, a sequence of profiles can be obtained. Taken to the limit, this sequence is equivalent to a statement level program trace. A technique similar

to this is applied in [KCHC89] in which snapshots of summary statistics are kept by the operating system on a processing node and periodically dumped over a network to another machine for display.

Snapshotting poses difficulties due to the problem of determining the global state of a distributed computation. Chandy and Lamport [CL85] have developed methods for doing this. A more practical approach would be to forego the idea of a global snapshot and assume that profile sequences are only valid within a node.

Snapshots would most likely be of use in computations that have several phases. A different profile would be collected for each phase. Phase based display of profile data has been explored in the context of trace based measurement systems in [MCH+90]. For extremely long computations, animation of profile data is also a possibility. However, the overhead of snapshotting will limit the frequency at which data can be collected.

9.1.3 *Gauge* and performance visualization

Gauge can display performance data from about 100 processors at once on a typical color display. However, there is an immediate need for an order of magnitude increase in this number. Some method of compressing the data in the processor axis is needed.

One method of data compression which we are considering is to use a classification algorithm to group together data from sets of processors that perform similar activities. For example, in figure 8.2, the activities of processor three and four are much alike and could have been combined into a single processor group.

9.1.4 Application level performance measurement

As defined, *PCN* has three basic composition operators. However, a goal in designing *PCN* is to be able to provide high level composition operators specialized to specific application domains [FS90]. For example, the FLOW code discussed in Chapter 8 could have been expressed in terms of a Gauss-Siedel composition operator.

As new operators are introduced, new measurement methods will be needed. These methods can be integrated into the domain specific composition operators much as we have integrated measure into the basic *PCN*

compositions. The result would be the ability to express performance at in terms of an application domain rather than in terms of the programming language. This will simplify understanding the performance of an application.

9.2 Final Thoughts

Parallel processing is a reality: commercially available parallel computers are being used daily to solve problems in application areas from physics to banking. The future success of parallel computing lies in the ability for nonspecialists to take advantage of the technology; this has been one of the goals of *PCN* from the start. By integrating performance measurement into *PCN*, hopefully we have put the ability to write efficient parallel programs within the reach of any individual who cares to try.

Appendix A

A Summary of Notation

$\mathcal{A}$	The architecture parameter of a performance experiment
C	A choice composition
$f_{\mathbf{g}}(g)$	The probability density function of random variable $\mathbf{g}$
$g_C(I_k)$	The number of guard tests in implication I_k of composition C
i_k	A *PCM* instruction
$i(C)$	The number of implications in choice composition C
$\mathcal{I}$	The input data for a performance experiment
I_k	An implication
$n(C)$	Total number of nodes in a decision tree
$n_i(C)$	Number of internal nodes in a decision tree
n_{i_k}	The number of times instruction i_k execute
n_c	The amount of data copied by during an assignment
$\mathcal{N}$	The number of processors used for a computation
P	The performance of a system
p_i	A performance index of a system
$\mathcal{P}$	The program executed during a performance experiment
$P\{x\|y\}$	The probability of x given y
$\mathcal{Q}_k$	The bag of programs in the active queue when the *PCM* exits the idle state
$S_{\mathcal{P}}(\mathcal{N})$	The speedup of program $\mathcal{P}$ when executing on $\mathcal{N}$ processors
S_{PCN}	Size of the *PCN* components of a program
$S_{foreign}$	Size of the foreign components of a program
S_{PCM}	Size of the *PCN* abstract machine
$S_{measurement}$	Amount of storage required for measurements

SO_{dt}	Storage overhead for decision tree measurements
SO_{ls}	Storage overhead for linear search measurements
t	The execution time of a program
t_s	The sequential execution time of a program
t_c	The per data element cost of copying data during an assignment
t_{i_k}	The cost of executing instruction i_k
$t_{\sigma_i}(p)$	The idle time for a *PCN* program p
$\mathbf{g}$	An integer valued random variable representing which test in an implication guard fails
T_{PCN}	The amount of time a computation spends executing *PCN* programs
$T_{foreign}$	The amount of time a computation spends executing foreign programs
$T_{communication}$	The amount of time a computation spends communicating
T_{idle}	The amount of time a computation spends idle
$T_{measurement}$	The amount of time a computation spends making and recording measurements
TO_{dt}	Execution time overhead for decision tree measurements
TO_{ls}	Execution time overhead for linear search measurements
$\delta(S_1, S_2)$	Difference in the amount of storage required for measurements S_1 and S_2
$\lambda(I_k)$	The number of tests left at the leaf of a decision tree for implication I_k
$\nu_C^s(k)$	The number of times the k^{th} implication of choice composition C executes successfully
$\nu_C^f(k)$	The number of times the k^{th} implication of choice composition C fails
$\nu_{gc(I_k)}(j)$	The number of times the j^{th} guard test in I_k fails
σ	A program state, σ member of the set Σ
σ_c^e	Component c of a program is executing user code
σ_c^i	Component c of a program is waiting (idle)
σ_c^c	Component c of a program is communicating
Σ	The set of states representing activities of program execution
Θ	The amount of time required to make a single measurement
Ω	The storage space required for a single measurement

Appendix B

A Summary of the Program Composition Machine

This appendix is a summary of the *PCN* abstract machine. The information presented here is abstracted from [FT90]. Some of the more esoteric details of the *PCM* have been omitted in this discussion.

B.1 Machine Registers

The *PCM* has 256 general purpose registers and six special purpose registers. A summary of the registers follows.

The program counter (PC). Points to the next *PCM* instruction to be executed. The contents of PC are incremented by the *PCM* after each instruction executes.

The failure label (FL). Holds the address to load into the program counter if a test instructions fails. FL is loaded by the *try* instruction.

The current process (CP). Contains a pointer to the current process record. A process record is a data structure on the heap. The contents of CP are set by *halt* and *recurse* instructions as well as by the communications component of the *PCM*.

The argument registers (A1 – A256). The *PCM* has 256 general purpose registers that can serve as an argument to an instruction.

The structure pointer (SP). This register is used to point to the next element

to be filled when building a structure on the heap. SP is initialized by the *fork* and *build_static* instructions.

The heap pointer (HP). Points to the first available cell on the heap. HP is incremented as part of the *build* and *put* instructions and reset by the garbage collector.

The foreign pointer (FP) points a set of foreign argument registers. FP is reinitialized to point to the first argument register on the return from a foreign program call.

B.2 Instruction Summary

The following is a summary of the instruction set for the *PCM*.

fork(Procedure,A) creates new process record with *A* arguments and pushes it onto the run queue. The *program* field of the process record is initialized to point to the *PCN* program *P*.

recurse(Procedure,A,Counter) starts execution of program *Procedure* with *A* arguments if the *timeslice* register is nonzero. If the contents of the *timeslice* register is zero, a new process record is created and initialized as in the *fork* instruction. In either case, the contents of the timeslice is decremented and the counter is incremented.

halt(counter) removes the current process record from run queue and prepares the *PCM* to execute the next process on the queue. The contents of the counter are incremented.

default(A,Counter) suspends the current process if any test instruction in the current process have suspended. Otherwise execution continues. The contents of *Counter* are incremented only if the process suspends.

try(Label) stores *Label* in the failure register.

build_static(Reg,Byte,Type,Size) builds a data structure of type *Type* and size *Size* on the heap. The header of the data structure is set to *Byte*. A pointer to the data structure is stored in *Reg*. If the data type is a tuple, the structure pointer is set to point to its first element.

build_dynamic(Type,Reg1,Reg2) is similar to *build_static* except the size of the structure to be build is specified in *Reg1* and a reference to the structure is put into *Reg2*.

build_def(Reg) creates a definition on the heap and puts a reference to it in *Reg*.

put_data(Reg,Byte,Size,Value) builds a constant on the heap and put a reference to in into *Reg*. The *Byte*, *Size* and *Value* arguments indicate the constant to be used.

put_value(Reg) copies the contents of *Reg* to the storage location pointed to by SP. SP is incremented.

copy(Reg1,Reg2) copies the contents of *Reg1* to *Reg2*.

get_tuple(Reg1,A,Reg2) tests *Reg1* to see if it points to a tuple. If the contents of *Reg1* are not defined, then the instruction suspends.

equal(Reg1,Reg2) tests to see if the data structures referenced by *Reg1* and *Reg2* are equal. If the contents of either register are not defined the instruction suspends.

neq(Reg1,Reg2) tests to see if the data structures referenced by *Reg1* and *Reg2* are defined to constants that are not equal. If the contents of either register are not defined the instruction suspends.

type(Reg,Tag) tests to see if the contents of *Reg* has the type specified by *Tag*. If *Reg* is not defined, the instruction suspends.

le(Reg1,Reg2) tests to see if the contents of *Reg1* is less than the contents of *Reg2*. The appropriate type conversion is performed. If the contents of either register is not defined, the instruction suspends.

lt(Reg1,Reg2) tests to see if the contents of *Reg1* is less than or equal to the contents of *Reg2*. The appropriate type conversion is performed. If the contents of either register is not defined, the instruction suspends.

data(Reg) tests to see if the contents of *Reg* is defined. The test fails otherwise.

define(Reg1,Reg2) defines *Reg1* to be the contents of *Reg2*. It is an error for *Ref1* to be anything other than a definitional variable or a remote reference.

get_element(Reg1,Reg2,Reg3) copies an element from the array referenced by *Reg2* into the data structure referenced by *Reg3*. The index into *Reg2* is referenced by *Reg1*.

put_element(Reg1,Reg2,Reg3) performs the inverse of *get_element*.

sizeof(Reg1,Reg2) places into *Reg2* a reference to the size of the term referenced by *Reg1*.

copy_mutable(Reg1,Reg2,Counter) copies the value of the data structure referenced by *Reg1* into the structure referenced by *Reg2*. The counter pointed to by *Counter* is incremented by the number of heap cells copied.

put_foreign(Reg) puts a pointer to the data structure referenced by *Reg* into the foreign argument register pointed to by FP. FP is incremented.

call_foreign(Address,Timer) calls the foreign program at specified by *Address*. Upon return from the foreign program, FP is reinitialized and the contents of *Timer* are incremented by the length of time spent in the foreign program.

add(Reg1,Reg2,Reg3) places into *Reg3* a reference to the sum of the values referenced by *Reg1* and *Reg2*. The appropriate type conversions are performed.

sub(Reg1,Reg2,Reg3) places into *Reg3* a reference to the result of subtracting the value referenced by *Reg2* from the value referenced by *Reg2*. The appropriate type conversions are performed.

mul(Reg1,Reg2,Reg3) places into *Reg3* a reference to the result of multiplying the value referenced by *Reg2* by the value referenced by *Reg2*. The appropriate type conversions are performed.

div(Reg1,Reg2,Reg3) places into *Reg3* a reference to the result of dividing the value referenced by *Reg2* by the value referenced by *Reg2*. The appropriate type conversions are performed.

mod(Reg1,Reg2,Reg3) places into *Reg3* a reference to the result of the value referenced by *Reg2* modulo the value referenced by *Reg2*.

Appendix C

Failure Probabilities of Guard Test Instructions

The application of the Equation 5.18 requires that the conditional failure probability of each guard test be known. Given that a guard has failed, the conditional failure probability is the probability that a specific test instruction is responsible for the failure.

The probability values are determined experimently by comparing the number of times a test instruction causes a guard failure to the total number of times a test instruction appears in a failing guard. Both of these numbers are obtained by executing the test programs from Table 5.1 in parallel on an instrumented *PCM*. The instrumentation is described in Section 5.5.9. The results of this measurement are summarized in Table C.1. Note that because the *default* test always appears alone in a guard, its conditional failure probability is one.

Conditional Failure Probabilities	
Test	Probability
get_tuple	.415465
equal	.656528
neq	.176910
type	0.001060
le	.558756
lt	.609249
data	.337033
default	1.0

Table C.1: Failure probabilities of guard tests

Bibliography

[AG88] Ziya Aral and Ilya Gertner. Non-intrusive and interactive profiling in parasight. In *Proceedings of the ACS/SIGPLAN PPEALS*, pages 21–30. ACM, June 1988.

[AI83] Arvind and R. A. Iannucci. A critique of multiprocessing von Neumann style. In *Proceedings of the Tenth Symposium on Computer Architecture*, pages 426–436, 1983.

[Bac78] John Backus. Can programming be liberated from the von Neumann style? a functional style and its algebra of program. *Communications of the ACM*, 16(8):613–641, August 1978.

[BBN] BBN Advanced Computers Inc., Cambridge, MA. *nX Software Tools.*

[BBN90] BBN Advanced Computers Inc. *nX Programmers Reference Manual*, 1990.

[BCW87] Richard A. Becker, William S. Cleveland, and Allan R. Wilks. Dynamic graphics for data analysis. *Statistical Science*, 2(4):355–382, November 1987.

[Ben82] Jon Louis Bentley. *Writing Efficient Programs.* Prentice-Hall Software Series. Prentice-Hall Inc., 1982.

[BGK89] A.P.W Bohm, J.P Gurd, and M.C. Kallstrom. Monitoring experimental parallel machines. In *Instrumentation for Future Parallel Computing Systems*, chapter 7, pages 121–141. Addison-Wesley, 1989.

[Bok81] Shahid H. Bokhari. On the mapping problem. *IEEE Transactions on Computers*, C-30(3):207-214, March 1981.

[BS84] Francine Berman and Lawrence Snyder. On mapping parallel algorithms into parallel architectures. In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 307–309, August 1984.

[CL85] K. Mani Chandy and Leslie Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[CM88] K. Mani Chandy and Jayadev Misra. *Parallel Program Design.* Addison-Wesley, 1988.

[Col79] James S. Collofello. *Effect of Program Modifications on Software Performance.* PhD thesis, Northwestern University, June 1979.

[Cou] Alva Couch. Personal Communication.

[Cou88] Alva L. Couch. *Graphical Representations of Program Performance on Hypercube Message-Passing Multiprocessors.* PhD thesis, Tufts University, April 1988.

[CRA] CRAY Research, Inc. *CRAY X-MP Computer Systems Mainframe Reference Manual.*

[Cra89] Cray Research Inc. *UNICOS Performance Utilities Reference Manual*, May 1989.

[CT89] K. Mani Chandy and Stephen Taylor. The composition of parallel programs. In *Proceedings of Supercomputing 89*, November 1989.

[CT90] K. Mani Chandy and Stephen Taylor. A primer for program composition notation. Technical Report Caltech-CS-TR-90-10, California Institute of Technology, 1990.

[CTKF90] K.M Chandy, S. Taylor, C. Kesselman, and I.T Foster. The program composition project. Technical Report Caltech-CS-TR-90-03, California Institute of Technology, Pasadena, California, 1990.

[Cyt84] Ron Cytron. *Compile-time Scheduling and Optimization of Asynchronous Machines.* PhD thesis, University of Illinois at Urbana-Champaign, 1984.

[DBM89] Thomas A. DeFanti, Maxine D. Brown, and Bruce H. McCormick. Visualization: Expanding scientific and engineering research opportunities. *Computer*, 22(8):12–25, August 1989.

[Dun88] T. H. Dunigan. Performance of a second generation hypercube. Technical Report ORNL/TM-10881, Oak Ridge National Laboratory, November 1988.

[Ell85] John R. Ellis. *Bulldog: A Compiler for VLIW Architectures.* MIT Press, 1985.

[EZL89] Derek L. Eager, John Zahorjan, and Edward D. Lazowska. Speedup versus efficiency in parallel systems. *IEEE Transactions on Computers*, 38(3):408–423, March 1989.

[Fer79] Domenico Ferrari. *Computer System Performance Evaluation.* Prentice-Hall, 1979.

[FJL+88] G. C. Fox, M. A. Johnson, G. A. Lyzenga, S. W. Otto, J. K. Salmon, and D. W. Walker. *Solving Problems on Concurrent Processors*, volume Volume 1. Prentice-Hall, 1988.

[FKT90] Ian Foster, Carl Kesselman, and Stephen Taylor. Concurrency: simple concepts and powerful tools. *The Computer Journal*, 33(6):501–507, June 1990.

[FO90] Ian Foster and Ross Overbeek. Experiences with bilingual parallel programming. In *The Proceedings of the Fifth Distributed Memory Computer Conference*, 1990.

[FO91] Ian Foster and Ross Overbeek. Bilingual parallel programming. In *Proceedings of the Third Workshop on Parallel Computing and Compilers*. MIT Press, February 1991.

[Fos90] Ian Foster. Personal Communication, 1990.

[Fre89] James C. French. A global time reference for hypercube multicomputers. In *The Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, 1989.

[FS90] Ian Foster and Rick Stevens. Parallel programming with algorithmic motifs. In *Proceedings of the International Conference on Parallel Programming*, 1990.

[FT90] Ian Foster and Stephen Taylor. A portable run-time system for PCN. Technical Memorandum No. 137, Argonne National Laboratory, January 1990.

[Gai86] Jason Gait. A probe effect in concurrent programs. *Software—Practice and Experience*, 16(3):225–233, March 1986.

[GHPW90] G. A. Geist, M. T. Heath, B. W. Payton, and P. H. Worley. PICKLE: A portable communications library. Technical Report ONRL-TM-11130, Oak Ridge National Laboratory, 1990.

[GK87] Michael Gorlick and Carl Kesselman. Timing prolog programs without clocks. In *Proceedings of the Symposium on Logic Programming*, pages 426–431. IEEE Computer Society Press, 1987.

[GKM83] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusik. An execution profiler for modular programs. *Software—Practice and Experience*, 13:671–685, 1983.

[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):609–638, 1988.

[GS85] Francesco Gregoretti and Zary Segall. Programming for observability support. Technical Report CMU-CS-85-176, Carnegie-Mellon University, 1985.

[GT88] Annoop Gupta and Andrew Tucker. Exploiting variable grain parallelism at runtime. In *Proceedings of the ACS/SIGPLAN PPEALS*, pages 212 - 221, July 1988.

[HSS87] Michael Hirsch, William Silverman, and Ehud Shapiro. *Concncurrent Prolog, Collected Papers*, volume Volume 2, chapter Computation Control and Protection in the Logix System, pages 28–45. MIT Press, 1987.

[Hub83] Peter J. Huber. Statistical graphics: History and overview. In *Proceedings of the Fourth Annual Conference and Exposition of the National Computer Graphics Association*, pages 667–676, 1983.

[JD88] Anil K. Jain and Richard C. Dubes. *Algorithms for Clustering Data*. Advanced Reference Series. Prentice Hall, 1988.

[KBC+74] D. J. Kuck, P. B. Budnic, S. C. Chen, E. Davis Jr., J. Han, P. Kraska, D. Lawrie, Y. Muraoka, R. Stebendt, and R. Towle. Measurements of parallelism in ordinary FORTRAN programs. *Computer*, C-23(1):36–46, January 1974.

[KCHC89] David W. Krumme, Alva L. Couch, Barton R. House, and Jon Cox. The triplex tool set for the NCUBE multiprocessor. Technical report, Tufts University, 1989.

[Knu71] Donald E. Knuth. An empirical study of FORTRAN programs. *Software—Practice and Experience*, 1(2), 1971.

[KP78] B. W. Kernighan and P.J. Plauger. *The Elements of Programming Style*. McGraw-Hill, second edition edition, 1978.

[KT86] C.F Kesselman and S Taylor. The partitioning of logic languages for parallel execution. Aerospace Technical Report ATR-86(8463)-1, The Aerospace Corporation, El Segundo, California, 1986.

[LHKK79] C. Lawson, R. Hanson, D. Kincaid, and F. Krogh. Basic linear algebra subprograms for Fortran usage. *ACM Transactions on Mathematical Software*, 5:308–329, 1979.

[Lin] Dong Lin. Personal communication.

[Lin90] Laura Bagnall Linden. *Performance Instrumentation and Visualization*, chapter Parallel Program Visualization Using ParVis, pages 157–187. Addison-Wesley Publishing Company, 1990.

[LSV+89] Ted Lehr, Zary Segall, Dalibor F. Vrsalovic, Eddie Caplan, Alan L. Chung, and Charles E. Fineman. Visualizing performance debugging. *Computer*, 22(8):38–51, 1989.

[MAA+89] Allen D. Malony, James W. Arendt, Ruth A. Aydy, Daniel A. Reed, Dominique Grabas, and Brian K. Totty. An integrated performance data collection, analysis, and visualization system. In *The Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, 1989.

[Mal89] Allen D. Malony. Multiprocessor instrumentation: Approaches for cedar. In *Instrumentation for Future Parallel Computing Systems*, chapter 1, pages 1–33. Addison-Wesley, 1989.

[Mar87] Joanne L. Martin. Mapping applications to architectures. In *Proceedings Supercomputing '87*, 1987.

[MBC86] M. Ajmone Marsan, G. Balbo, and G. Conte. *Performance Models of Multiprocessor Systems.* Computer Systems Series. The MIT Press, 1986.

[MCH+90] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 1(2):206–217, April 1990.

[McK88] Phillip McKerrow. *Performance Measurement of Computer Systems.* Addison-Wesley Publishing Company, 1988.

[ME69] David F. Martin and Gerald Estrin. Path length computations on graph models of computations. *IEEE Transactions on Computers*, C-18(6):530–536, June 1969.

[MH89] Charles E. McDowell and David P. Helmbold. Debugging concurrent programs. *ACM Computing Surveys*, 21(4):593–622, December 1989.

[MLR90] Allen D. Malony, John L. Larson, and Daniel A. Reed. Tracing application program execution on the Cray X-MP and Cray 2. In *Proceedings Supercomputing '90*, pages 60–73, November 1990.

[PF88] Carl Ponder and Richard J. Fateman. Inaccuracies in program profilers. *Software—Practice and Experience*, 18(5):459–467, 1988.

[PFTV88] William H. Press, Brian P. Flannery, Saul A. Teukolsky, and William T. Vetterling. *Numerical Recipes in C.* Cambridge University Press, 1988.

[Rei90] Matthew H. Reilly. *A Performance Monitor for Parallel Programs.* Academic Press, Inc, 1990.

[RPW89] D. T. Rover, G. M. Prabhu, and C. T. Wright. Characterizing the performance of concurrent computers: A picture is worth a thousand numbers. In *The Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, 1989.

[Sar89] Vivek Sarkar. Determining average program execution times and their variance. In *Proceedings of the SIGPLAN '89 Conference on Programming Language Design and Implementation*, pages 298–312. acm, ACM Press, 1989.

[Seq99] Sequent Computer. *Sequent Hardware overview*, 1999.

[SH86] Vivek Sarkar and John Hennessy. Compile-time partitioning and scheduling of parallel programs. In *Proceedings of the SIGPLAN '86 Symposium on Compiler Construction*, pages 17–26. ACM, 1986.

[SSS+83] Z Segall, A. Singh, R. T. Snodgrass, A. K. Jones, and D. Siewiorek. An integrated instrumentation environment for multiprocessors. *IEEE Transactions on Computers*, C32(1):4–14, January 1983.

[Svo76] L. Svobodova. *Computer Performance Measurement and Evaluation Methods: Analysis and Application.* Elsivier, 1976.

[Sym89] Symult Systems Corporation. *Series 2010 System Fortran 77 Programmer's Guide*, April 1989.

[Tay89] Stephen Taylor. *Parallel Logic Programming Techniques.* Prentice-Hall, Englewood Cliffs, 1989.

[TF] Stephen Taylor and Ian Foster. Personal Communication.

[TF88] Stephen Taylor and Ian Foster. STRAND: Language reference manual. PAR 88/10, Imperial College of Science and Technology, August 1988.

[TF89] Stephen Taylor and Ian Foster. *STRAND: Concepts in Parallel Programming.* Prentice Hall, 1989.

[Tuf83] Edward R. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, 1983.

[WH90] Dieter Wybranietz and Dieter Haban. *Performance Instrumentation and Visualization*, chapter Monitoring and Measuring Distributed System, pages 27–40. Addison-Wesley Publishing Company, 1990.