Performance observability

Malony, Allen Davis, Ph.D.

University of Illinois at Urbana-Champaign, 1990

U·M·I
300 N. Zeeb Rd.
Ann Arbor, MI 48106

PERFORMANCE OBSERVABILITY

BY

ALLEN DAVIS MALONY

B.S., University of California, 1980
M.S., University of California, 1982

THESIS

Submitted in partial fulfillment of the requirements
for the degree of Doctor of Philosophy in Computer Science
in the Graduate College of the
University of Illinois at Urbana-Champaign, 1990

Urbana, Illinois

UNIVERSITY OF ILLINOIS AT URBANA-CHAMPAIGN

THE GRADUATE COLLEGE

SEPTEMBER 1990

WE HEREBY RECOMMEND THAT THE THESIS BY

ALLEN DAVIS MALONY

ENTITLED PERFORMANCE OBSERVABILITY

BE ACCEPTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS FOR

THE DEGREE OF DOCTOR OF PHILOSOPHY

[illegible]
Director of Thesis Research

[illegible]
Head of Department

Committee on Final Examination†

[illegible] Chairperson

[illegible]

[illegible]

[illegible]

[illegible]

† Required for doctor's degree but not for master's.

O-517

PERFORMANCE OBSERVABILITY

Allen Davis Malony, Ph.D.
Department of Computer Science
University of Illinois at Urbana-Champaign, 1990
D. Reed, Advisor

Performance observability is the ability to accurately capture, analyze, and present (collectively *observe*) information about the performance of a computer system. Advances in computer systems design, particularly with respect to parallel processing and supercomputers, have brought a crisis in performance observation — computer systems technology is outpacing the tools to understand the performance behavior of and to operate the machines near the high-end of their performance range. In this thesis, we study the performance observability problem with emphasis on the practical design, development, and use of tools for performance measurement, analysis, and visualization.

Tools for performance observability must balance the *need* for performance data against the *cost* of obtaining it (environment complexity and performance intrusion) – too little performance data makes performance analysis difficult; too much data perturbs the measured system. We discuss several methods for performance measurement concentrating specifically on mechanisms for timing and tracing. We show how minor hardware and software modifications can enable better measurement tools to be built and describe results from a prototype hardware-based software monitor developed for the Intel iPSC/2 multiprocessor.

Any software performance measurement perturbs the measured system. We develop two models of performance perturbation to understand the effects of instrumentation intrusion: time-based and event-based. The time-based models use only measured time overheads of instrumentation to approximate actual execution time performance. We show that this model can give accurate approximations for sequential execution and for parallel execution with independent execution ordering. We use the event-based model to quantify the perturbation effects of instrumentations of parallel executions with ordering dependencies. Our results show that this model can be applied in practice to achieve accurate approximations. We also discuss the limitations of the time-based and event-based models.

The potentially large volume of detailed performance data requires new approaches to presentation that can show both gross performance characteristics while allowing users to focus on local performance behavior. We give several examples where performance visualization techniques have been effectively applied, plus discuss the architecture and a prototype of a general performance visualization environment.

Finally, we apply several of the performance measurement, analysis, and visualization techniques to a practical study of performance observability on the Cray X-MP and Cray 2 supercomputers. Our results show that even modest improvements in the existing set of performance tools for a particular machine can have significant benefits in performance evaluation capabilities.

To my Father, my Mother, Kimberly, and Lindsay

Acknowledgments

The journey to complete my doctorate work during these last five years was long and difficult. Fortunately, there were many people who helped me along the way. I would like to thank Duncan Lawrie for first offering me a research assistantship through the Department of Computer Science with the Center for Supercomputing Research and Development.[1] David Kuck gave me the opportunity to be a full-time professional at CSRD and has always treated me with respect. George Cybenko has supported my completion of the Ph.D. dissertation and I appreciate his understanding.

I would like to thank the members of the Picasso group. Ruth Aydt's technical assistance in many capacities has helped smooth out the rough spots. Dirk Grunwald was always available for enlightening discussions. The rest of the boys have provided many a laugh to relieve the tensions of the moment.

I give credit to Justin Rattner of Intel for the idea of the HyperMon hardware monitor. Paul Close, also of Intel, was instrumental in securing support for HyperMon's development and has provided much encouragement during the project. Tracy Tilton, John Andrews, and Dan Lavery provided invaluable technical expertise during HyperMon's construction.

What can I say about my advisor, Daniel Reed. He has been my mentor, my colleague, and my friend. His guidance has been helped me mature as a researcher and his respect for my ideas has made working with him very rewarding. I look forward to many fruitful interactions with Dan in the future.

My Father and Mother have always inspired me to do my best. Their lives and achievements have set examples for me to follow. I proudly share this accomplishment with them.

Finally, I would like to thank most of all my wife Kimberly. Her love and encouragement have both supported and sustained me. I could not have achieved this without her.

[1]The thesis research was supported by the Department of Computer Science under grants from the National Science Foundation, NSF CDA-87-22836 and NSF CCR-87-06653; and by the Center for Supercomputing Research and Development under grants from the National Science Foundation, NSF ASC-84-04556 and NSF MIP-88-07775, and the NASA Ames Research Center, NCC-2-559.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

Introduction

Wisdom is the principal thing; therefore get wisdom, and with all thy getting, get understanding.

— Proverbs, Chapter IV, Verse 7, "Old Testament"

The technology to design and build high-performance computer systems, combining advances in architecture, hardware, systems software, and applications algorithms, is rapidly outpacing the tools and technology to both understand the performance behavior of these systems and to operate the machines near the high-end of their performance range. No known, general purpose methods can predict the performance of a computer system. Moreover, for high-performance systems, seemingly minor perturbations of the architecture, system software, or application algorithms can induce large changes in performance behavior. In numerical analysis, such problems are called ill-conditioned — small changes in the input x of a function $f(x)$ yield large changes in $f(x)$ (i.e., $f(x+\epsilon) \not\approx f(x)$). Clearly, approximating $f(x)$ by $f(\overline{x})$, by $\overline{f(x)}$, or by $\max_x f(x)$, is dangerous, if not wrong. Yet, peak performance ratings in MIPS (millions of instructions per second) or MFLOPS (millions of floating point operations per second) are precisely such an approximation. Furthermore, these peak performance ratings obscure the importance of interacting *performance levels.* Simply put, computer system performance is the complex product of its component interactions — their frequency and their type. These interactions often cannot be predicted, but they can be observed given the proper tools.

What combination of performance tools is appropriate for high-speed computer systems? How do the architecture, hardware, and system software affect how performance data is collected? What performance events can and cannot be observed? How do the performance evaluation tools affect the performance being measured? How should performance information be conveyed to the performance analyst? At present, there exists no model of performance *observability* for high-performance computer systems that can answer these questions. That is, no formal approach exists to determine, given a performance evaluation problem, how to accurately "observe" (capture, analyze, and visualize) system operation to produce the required performance results.

1.1 A Performance Evaluation Crisis

In any field, experimental progress is inextricably coupled with technological advances; the latter provide the requisite tools to more accurately measure and analyze known phenomena and to test hypotheses that predict the existence of heretofore undetected phenomena. Although the scientific method's systematic measurement and hypothesis testing is both appropriate and desirable for computer system performance evaluation [51], during the last ten years, the tremendous growth in computing power has been due to advances in component technology (density and speed) and new computer architectures (RISC and multiprocessors), and not the discovery of new insights into computing through advanced performance tools. Regrettably, during this period, a complacency arose concerning the development of performance measurement and analysis systems. The major reason for this was, of course, economic — there was little or no commercial incentive for manufacturers to include such tools. As a result, with the existence of extremely powerful computer systems there also is a dearth of experimental performance tools for users interested in determining the most effective way of exploiting the high-performance processing capabilities of these machines.

Many of the performance evaluation activities in this decade of computing power growth were centered on choosing fair and appropriate metrics for computer comparison (i.e., benchmarking). Although this work was important to the manufacturers for system marketing and to the users for system procurement (and still is), it contained little technical contribution with respect to systems design and application code tuning. For these latter purposes it is not as important to ask, "Who won?"; but "How did they win?" (the architecture, system software and application code techniques used); and "Why did they win?" (an in-depth analysis of the interaction of the algorithm, architecture and system software that demonstrates why the techniques used were successful). The pursuit of answers to the last two questions is greatly facilitated by the inclusion of an integrated performance analysis and evaluation system as early as possible in the design stage of a high-performance computer system. Moreover, as computers grow more complex, it will be increasingly difficult to retrofit any system capable of yielding significant performance information.

1.2 Performance Observability

Performance observability is the ability to accurately capture, analyze, and present (collectively *observe*) information about the performance of a computer system. Tools for performance observability must balance the *need* for performance data against the *cost* of obtaining it (environment complexity and performance intrusion) – too little performance data makes performance analysis difficult; too much data perturbs the measured system. The major goal of this thesis is to understand the constraints on the design, development, and use of tools for performance observation in each of these areas.

The complexity of computer systems, particularly parallel computer systems, makes *a priori* performance prediction difficult and experimental performance measurement crucial. A complete characterization of software and hardware dynamics, needed to understand the performance of high-speed machines, requires execution time performance instrumentation and data collection. However, the efficient implementation of performance measurement systems is non-trivial. In Chapter 2, we describe several approaches to performance measurement and discuss requirements for future measurement systems. We present results from a prototype hardware-based software monitor developed for the Intel iPSC/2 multiprocessor to emphasize the problems of performance measurement.

Computer system performance evaluation is subject to the same instrumentation pitfalls facing any experimental science, notably uncertainty and instrumentation perturbation. Instrumentation, no matter how unobtrusive, introduces performance perturbations, and the degree of perturbation is proportional to the fraction of the system state that is captured — volume and accuracy are antithetical. Formal models of performance perturbation are needed that permit quantitative evaluation of perturbations given instrumentation costs, measured event frequency, and desired instrumentation detail. We develop two models of performance perturbation to understand the effects of instrumentation intrusion: time-based and event-based. In Chapter 3, time-based models are used to approximate actual execution time performance by removing measured time overheads of instrumentation. We show that this model can give accurate approximations for sequential execution and for parallel execution with independent execution ordering.

In Chapters 4 and 5, we use event-based models to quantify the perturbation effects of instrumentations of parallel executions with ordering dependencies. Chapter 4 focuses on performance perturbation of synchronization constructs. During perturbation analysis, the partial ordering relationships of the original computation must be preserved. By applying the semantics of the synchronization operators, these relationships can be maintained in the approximated execution. However, the question still remains whether the approximated execution, although a "feasible" execution, is a "likely" execution of the original program. Chapter 5 examines the question of approximation accuracy in the context of parallel loops. In particular, we consider DOALL and DOACROSS loops. Although it is difficult to prove that the approximations are likely, we present empirical evidence that shows accurate approximations can be achieved in practice. The main conclusion drawn from Chapters 3, 4, and 5 is that a systematic application of performance perturbation analysis techniques will allow more detailed, accurate instrumentation than traditionally believed possible.

Even when the perturbations of a performance instrumentation environment are reduced to acceptable levels, a computer system can quickly generate vast quantities of performance data. This data must be presented in ways that emphasize important events while eliding irrelevant details. The acceptance of scientific data visualization is testimony to the remarkable ability of the human visual system to interpret and identify graphical data. Just as visual presentation of scientific data can provide new insights, a performance visualization environment would permit the performance analyst to explore and compare interesting data components by dynamically interconnecting new performance displays and data analysis tools. In Chapter 6, we describe several approaches to performance visualization, and discuss the architecture and a prototype of a general performance visualization environment.

Finally, Chapter 8 concludes with a synopsis of what we have learned and directions for future research.

Chapter 2

Performance Measurement

A person with one watch knows what time it is.

A person with two watches is never sure.

— Chinese proverb

Performance measurement is the foundation of performance observability. If an experiment cannot be constructed, even in principle, to measure a phenomenon, it cannot operationally be said to exist. If a phenomenon cannot be measured in practice, it cannot be observed. The complexity of computer systems, particularly parallel computer systems, makes *a priori* performance prediction difficult and experimental performance measurement crucial. A complete characterization of software and hardware dynamics, needed to understand the performance of high-speed machines, requires execution time performance instrumentation and data collection.

Despite the manifest need for dynamic performance instrumentation and data capture, their efficient implementation is non-trivial. Instrumentation, no matter how unobtrusive, introduces performance perturbations, and the degree of perturbation is proportional to the fraction of the system state that is captured. Also, some systems are more conducive to instrumentation than others. In addition to the measurement uncertainty resulting from instrumentation intrusion, the hardware and software accessibility of performance events of interest will determine the scope of realistic performance observation.

Ideally, computer systems would be designed to facilitate performance measurement. In addition to the hardware and software components needed for standard operation of a computer system, a *performance instrumentated computer* would include additional resources for perfor-

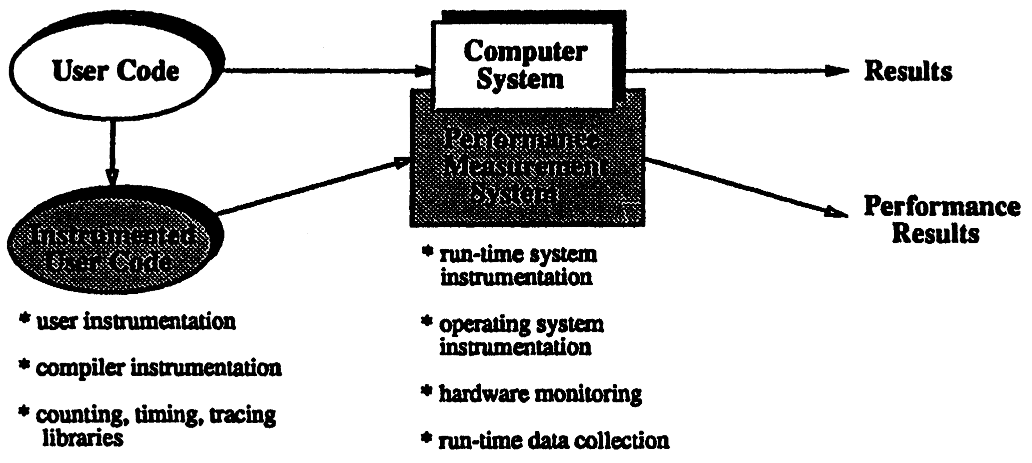

User Code
Computer System
Results
Performance Measurement System
Performance Results
Instrumented User Code
* run-time system instrumentation
* operating system instrumentation
* hardware monitoring
* run-time data collection
* user instrumentation
* compiler instrumentation
* counting, timing, tracing libraries

Figure 2.1: Performance Instrumented Computer

mance instrumentation and data collection [101, 102]; see Figure 2.1. Although these resources could be added to a system after it has been designed and developed, such modifications will become increasingly difficult as systems become more integrated and complex. Rather, a performance instrumented computer promotes the goal of including performance instrumentation and data collection as early as possible in the computer system hardware and software design.

Whereas we argue that performance instrumentation and data collection support should be a main component of any high-performance computer system, the degree and type of support depends on its intended purpose — the nature of the performance experiments to be conducted defines the needed capabilities of the performance measurement system. Also, the economic considerations cannot be overlooked. Although many measurement systems have been designed and studied in a variety of contexts [49, 59, 60, 79, 94, 135, 167, 170, 171, 180]), few exist on high-performance systems — the principal exception being the Cray X-MP and Y-MP systems [107]. In the past, there has been little commercial incentive for manufacturers to include such facilities. However, with the advent of complex computing systems, which combine advances in architecture, hardware, system software, and applications software to achieve high performance, there is renewed interest in the development of performance measurement and analysis capabilities to aid in improving the dismal ratio of the performance achieved by ordinary users to the potential performance of the machine [129, 144].

Our goal in this chapter is to present some insights into the simple, though difficult, problem of performance instrumentation and data collection, and to propose measurement techniques that might have practical application. In §2.1, we begin with a discussion of the general performance measurement problem in the context of different performance observability requirements. We describe a simple event-based model of performance measurement in §2.2 that will serve as a reference for the remainder of the chapter. The timing of events is a fundamental form of instrumentation that *often* is not well supported on high-performance systems. In §2.3, we focus on the timing issue. Although performance data can take many forms, including samples, counts, and running sums, event traces provide the greatest flexibility; given a trace, one can compute counting and timing statistics, in addition to showing dynamic execution profiles. In §2.4, we discuss issues that concern tracing as a general technique for software performance measurement.

Although software recording of performance data suffices for low frequency events, capture of detailed, high-frequency performance data ultimately requires hardware support if the performance instrumentation is to remain efficient and unobtrusive. However, designing a machine independent data capture system is exceedingly difficult — the variety of system interfaces limits generality. Because message-based parallel systems typically lack both a global clock for event timestamps and a common memory for event data buffering, they pose particularly vexing, and interesting, instrumentation problems. Without hardware support, event trace extraction either must compete with system and application programs for access to network communication bandwidth, or the traces must be buffered locally in individual node memories, limiting both trace size and application program memory. Given these problems, hardware support for event data capture and recording is crucial to minimizing instrumentation perturbations. But hardware data collection is not a panacea, since practical issues such as hardware development and system accessibility must be considered. In §2.5, we present results from a case study of performance instrumentation and data collection on the Intel iPSC/2 message passing system.

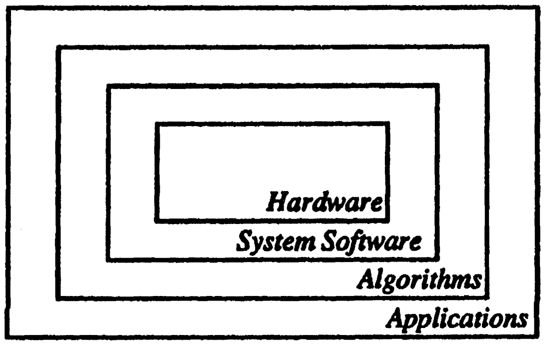


Figure 2.2: Performance Measurement Levels

2.1 Performance Measurement and Observability

The appearance of any new computer system raises many questions about its performance, both in absolute terms and in comparison to other machines of its class. Unfortunately, computer systems are among the most complex of man's creations, making satisfactory performance characterization difficult. Despite this complexity, there are strong, indeed, almost irresistible, incentives to quantify computer system performance using a single metric. The fallacy lies in succumbing to such temptations — complete characterization of computer system performance encompasses more than operations executed per second. Peak performance ratings in MIPS (millions of instructions per second) or MFLOPS (millions of floating point operations per second) obscure the importance of interacting *performance levels* and *dynamic equilibrium*.

As Figure 2.2 illustrates, there are four levels in the hierarchy of performance measurements. The answer to the oft-asked question, "How fast is it?" depends on the intended use of the performance data. At the lowest level lies the performance of the hardware design. Determining this performance provides both a validation of and directives for the system software design. Only by understanding the strengths and weaknesses of the hardware can system software designers develop an implementation and user interface that maximizes the fraction of the raw hardware performance available to the end user. Given some characterization of the available processing resources and the services provided by the system software, users can develop algorithms that are best suited to the computer system's capabilities. Finally, the best mix of key algorithms will maximize the performance of user applications.

A complete performance characterization requires not only an analysis of the system's constituent levels, it also requires both *static* and *dynamic* characterizations. Static or average behavior analysis may mask transients that dramatically alter system performance.[1] The history of virtual memory research offers a classic example of transient behavior and its importance. The slow drift model [41] predicted that program reference locality changed slowly. Later, more detailed measurements showed that reference localities change swiftly and catastrophically. Most page faults and associated overhead occur in small time intervals, and a phase-transition model more accurately reflects observed behavior.

A combination of static and dynamic characterizations is also needed to understand the interaction between performance levels. Repeated studies have shown that a system's performance is maximized when the components are balanced (i.e., there is no single system bottleneck) [42]. As an example, optimizing the performance of message passing systems [158] requires a judicious combination of node computation speed, message transmission latency, and operating system software. However, these must be considered in the context of the applications running on the machine — high speed processors connected by high latency communication links restrict the classes of algorithms that can be efficiently supported.

Regardless of the performance level, performance characterization requires the specification of the desired measurements and the implementation of instrumentation and data collection mechanisms to carry out those measurements. Although computer system performance is inextricably tied to the performance of its constituent hardware and software levels, it is less clear that performance instrumentation and data collection techniques for one level, or even one system, are rarely applicable to other systems or other levels. As an example, Table 2.1 shows a subset of the important performance measurements for three levels (hardware, system software, and algorithm) and three systems (the Cray X/MP, the University of Illinois Cedar system [99], and the Intel iPSC/2 hypercube [10]). The diversity of underlying technology and system architecture makes it difficult to develop a single set of performance instrumentation techniques. Memory bank conflicts on the Cray X/MP have no analog on the distributed memory Intel iPSC/2. Moreover, the event time scales differ by six orders of magnitude. Similarly,

[1]By analogy, biological researchers have long recognized the importance of both *in vitro* and *in vivo* measurements. Laboratory measurements of isolated cells or biological molecules often differ from similar measurements in natural environments.

Level	*Cray X/MP*	*Illinois Cedar*	*Intel iPSC/2*
Hardware	**vector startup** **memory conflicts**	**network contention** **vector/cache interaction**	**processor speed** **communication latency and bandwidth**
Software	**compiler**	**compiler**	**OS support**
Algorithm	**vectorization**	**shared memory access**	**communication pattern**

Table 2.1: *Performance Level Comparison*

the shared memory access patterns of Cedar application algorithms may cause interconnection network conflicts, but these patterns are not predictors of performance degradation due to network contention.

At all performance levels there exists a minimal set of required events that must be captured. Measurement of events should be enabled by signals to a hardware monitor, operating system calls, or flags to a compiler preprocessor, for example. In addition to standard events, certain others must be enabled selectively, either to minimize the performance perturbations of instrumentation or to reduce the data volume to tractable levels. The next section describes a simple model of performance measurement based on events.

2.2 Event-Based Performance Measurement Model

In order to measure the performance of a computer system, certain aspects of its behavior must be made observable. A computer system's operation can be regarded as a sequence of *actions* representing some significant physical or logical activity performed. In the context of a program's execution, an action might be an instruction in the program code or a routine. In general, actions represent the computational aspects of a computation one wishes to observe.

The execution of an action generates an *event* — an encoded instance of an action. Events are dynamic and each instance of an action defines a separate event. A performance measurement can be viewed as the collection of a (possibly infinite) set of events. Each event in the measurement indicates the action type (or identity), the time when the action occurred, information about where the action occurred, and any additional data that further defines the action state.

Event-based models have been widely used to describe program behavior and to define techniques for performance measurement [13, 14, 86, 141, 180]. For instance, the *behavioral abstraction* approach by Bates [14] characterizes a system in terms of the observable effects and interactions of system components as represented by a stream of characteristic atomic behaviors (i.e., events). Using an *event definition language* [13], an event recognition tool can interpret the stream of measured events that occur during execution and present the user with an abstract view of the program's behavior in terms of a sequence of hierarchically-defined events. Snodgrass [181] also uses an event model as the basis of a relational approach to monitoring in which an historical database formalizes the information processed by the monitor. The benefits include a simple, consistent structure for the information, the use of powerful declarative query languages, and the availability of a catalogue of optimizations. In this approach, the user is presented with the conceptual view that the dynamic behavior of the monitored system is available as a collection of historical relations, each associated with one or more actions in the subject system.

However, before events can be analyzed, they must be detected and captured by some monitoring system. If the event is physically-based (e.g., a memory access), a measurement device consisting of software and/or hardware probes must be used to observe the corresponding action. Higher-level, logical events would be defined from a combination of physical (or "primitive" [119]) events. Any performance characterization can be accomplished by defining a sufficiently sophisticated set of logical or physical events, $E = \{X, Y, \ldots, Z\}$, and performing two primitive measurements: i) how many times e occurs, $n(e)$, and ii) how long e lasts, $t(e)$, where $e \in E$.[2] In practice, the specification of complex system states and the problems associated with their detection necessitates measurement solutions that capture a large volume of time-based event data (e.g., tracing) for later analysis.

The following sections discuss issues concerning event measurement. In particular, we focus on the problems of effective timing and tracing.

[2]The proof is based on modeling the entire system as a huge finite state machine where all possible system states are represented as events. We assume proof by intimidation.

2.3 Timing

The ability to record timing information, both instantaneous (when an event occurred) and elapsed (the time between two events), is a fundamental performance measurement requirement of any computer system. Unfortunately, the technology commonly used to maintain and access timing information has not evolved with advancements in computer system complexity, particularly with respect to parallel systems. The standard UNIX timing approaches used for the scheduling of system resources and multiprogramming are often considered good enough for process-level time accounting. As the performance of computer systems increases due to faster clock speeds and the ability to support multiple execution streams, the need for accurate timing measurements can no longer be met by standard techniques. New approaches that emphasize high-resolution, globally consistent timing measurements with low-overhead access at both the user and operation system level must be developed.

2.3.1 Clocks and Timers

The support for timing measurements begins with the existence of a *real-time clock* (RTC). Clearly, every computer system generates clock signals to drive the hardware components of the machine. The real-time clock is commonly implemented as a binary counter that increments at a specific rate defined by the frequency of some hardware clock signal. The clock signal driving the RTC can be separate from or a derivative of another hardware clock signal. The main purpose of the RTC is to establish a reference for timing measurements. The RTC may or may not be accessible from software and may or may not be used directly for timing calculations.[3] The RTC defines the *resolution* of timing measurements. That is, the elapsed time of any performance phenomenon lasting shorter than one tick of the RTC cannot be measured and the minimum measured time difference between any two events cannot be less than one RTC period (excluding zero).

However, the RTC does not define timing *accuracy*. We define a *timer* to be different from a *clock* (i.e., the real-time clock) in that a timer is used directly to make timing measurements. The distinction is subtle but important [162]. Conceptually, a timer can measure the elapsed

[3] If the RTC is accessible from software, it will normally be implemented as a memory-mapped location in the user or system address space.

time between two successive events. The timer must detect the first event, then measure the time until the second event occurs. It is the implementation of the timer that determines how precisely the occurrence of the events is detected and, thus, how accurately the elapsed time can be measured. Any timer will have the RTC as its fundamental time reference. However, timing accuracy depends on the variability and magnitude of the costs of timer implementation.

2.3.2 Timer Implementation

All performance timing measurements are made by timers and not by clocks, but timers are derived from clocks. Thus, the RTC defines timing resolution and, hence, the limits of timing accuracy. We define *software timers* and *hardware timers* as timers used to measure software and hardware events, respectively. In the discussion of timing implementation that follows, we mainly focus on software timers.

Software timers are typically implemented in the operating system for the purpose of measuring process execution time. Two approaches have been developed: *sampled* process timing and *measured* process timing. The UNIX operating system was originally designed to use sampled process timing [163]. In this case, a hardware interval timer generates periodic "timer" interrupts; normally, every 10 to 20 milliseconds. These interrupts cause various scheduling decisions to occur as well as system and process resource utilization statistics to be computed. Process timing statistics are commonly separated into *user* time (the time spent executing in user mode) and *system* time (the time spent executing in system mode). With sampled process timing, whatever mode the running process was in at the time of the interrupt, the corresponding time is increased by the timer interrupt period. The process timing measurements are based on sampling because the timer interrupt results in a sample being taken of the current process state, and this state determines how the timing statistics are updated.

Because the timer interrupt signal serves as the clock base for process time calculations, the resolution of sampled-based process timing is the timer interrupt period (e.g., 10 to 20 milliseconds). The accuracy is harder to determine because a process can change states several times between timer interrupts, and can even be context switched off the machine before the timer interrupt occurs. In the first case, adding the entire period to the time of the interrupted execution mode (e.g., user mode) over-estimates this time while under-estimating the execution time of the other mode (e.g., system mode); the error can be as much as an entire timer period.

In the second case, the process is not accounted any of the time it was executing during the timer interrupt period and the times are under-estimated; again, the error can be as great as a single timer period.

The acceptability of sampled process timing depends on the requirements of the performance experiment. In general, however, as the clock speeds of the machines increase, finer resolution and accuracy will be required in process timing measurements; thus, the timer interrupt period must correspondingly decrease. However, interrupt processing speeds might not scale the same as processor speeds — context changes in operating systems can influence pipelining, cache misses, and page faults [113]. Therefore, there will be resistance to increasing the timer interrupt rate. Indeed, as processor execution speeds have increased 10 to 100 times in the last 10 years, the timer interrupt periods for systems using sampled process timing have stayed roughly the same — correspondingly, the relative accuracy of sampled timing measurements has decreased.

Measured process timing directly measures the time spent in different states of process execution. A separate timer is logically maintained for each state of each process. A particular process state timer is active only when the process is executing in that state. No timer interrupts are generated for process timing[4] and, hence, no "artificial" state changes occur. Rather, process state changes are identified by the operating system. At a process state transition, the elapsed time that the current process state was active is added to that state time. Usually, timestamps are taken from the RTC to establish when a process state begins and when it ends. The elapsed time is easily computed as the difference between these two timestamps. Thus, the resolution of the timing measurements is that of the RTC. The accuracy depends only on the overhead of accessing the RTC and updating the timing statistics at process state transitions.

Measured process timing has clear advantages over the sampled approach. The natural process state transitions are used to make timing measurements, thus avoiding the artificial timer interrupts. The resolution and accuracy of measured process times are significantly improved, and scale more cleanly with faster clock speeds. Although measured process timing does require operating system instrumentation to capture process state transitions, it also allows more distinction in process timing measurements; for instance, system execution time can be refined and process idle time can be kept [118]. One drawback to measured process timing is that the overhead for timing measurement increases as the frequency of state transitions

[4]Timer interrupts can still occur for scheduling purposes, for instance.

increase. However, efficient implementation of the simple timing calculations can help minimize the overhead incurred.

Given the advantages of measured process timing, it is surprising that only recently has it begun to gain acceptance. We conjecture that the need for operating system modification is the principal reason measured process timing is not more prevalent than it is. However, as the performance complexity of computer systems increase, especially as it relates to parallel processing, we maintain that sampled process timing will be unacceptable for many performance timing measurements.

2.3.3 User-Level Timing and Profiling

User-level timing measurements differ from process-level measurements because the events of interest are defined with respect to user, not process, states. The most common user timing measurement is the profiling of routine execution in programs. Basically, profiling measurements produce statistics for each routine indicating the number of times a routine is called and the amount of time spent executing code in a routine, including time spent in calls to other routines. Counting measurements are trivially made by instrumentation inserted at the beginning of each routine. Timing measurements, however, are more complicated.

The UNIX *prof* [17] and *gprof* [72] tools have become accepted standards for profiling measurement. In keeping with standard UNIX process timing, the time accounting in these profiling tools follows a sampled approach. Periodically (during the same timer interrupt used for sampled process timing), the program counter (PC) is sampled by profiling code in the operating system. A PC histogram for the program is kept that, at the end of the program's execution, represents the frequency distribution of PC samples. Using the program's load image and knowledge of the PC sampling period, the profiling analysis tools calculate the time spent in each routine as the number of PC samples in a routine's code range times the sampling period. Thus, the profile time measurement assumes the routine associated with a sampled PC value was executing for the entire sample period.

Sampled profile measurements generate statistical approximations to actual execution time profiles. The resolution and accuracy of the time profiles suffer from the same problems as sampled process timing. Because many routine transitions can occur during a sample period, the potential for timing errors is significant [155]. This will become even more serious as the machine speeds increase. However, profiling is often used to show relative execution time distribution. As such, errors of a few percent might not be important. Furthermore, from results of statistical sampling, as the program execution time increases, so will the accuracy of the sampled profiling measurements.[5]

As with measured process timing, profile times can also be measured. In this case, timestamps of entry and exit from routines are dynamically kept during execution, and routine times are updated at routine transitions. Timing resolution and accuracy are greatly improved over sampled profiling, but time measurement overheads (particularly to access the user execution timer normally kept by the operating system) can result in observable program slowdown when the routine calling frequency is high. Although sample profiling is more common, there have been profilers developed using a time measured approach [28].

Any advantages of measured profiling are arguably minor given the way profiling statistics are commonly used. Instead, we described the measured time approach to highlight a more problematic user-level timing measurement. Quite often, a performance analyst will want to time a section of code, which may or may not be a routine. Although some version of PC mapping might be applied to compute a sampled-based time measurement of the code section, if the actual elapsed time is on the order of the sample period, large timing errors could result. In part, this is a sampling error problem, but it is also a timing resolution problem. A measured time approach can solve the first problem, but the second must rely on measured process timing with support for low-overhead user-level timer access.

User-level timing in UNIX is mainly done through system calls that return the current value of process user time. Because the process user timer advances only when the process is executing in user mode, we can think of the timer as defining a *virtual* time reference as opposed to the real time reference of the RTC. Timestamps from this virtual time domain are used to make user-level timing measurements. However, if the process time is updated by sampling, user-level timing resolution is limited by the sampling period (e.g., 10 to 20 milliseconds). For many user-level timing needs, a resolution of 10 to 20 milliseconds is insufficient. Often code

[5]Remarkably, users of Cray X-MP systems have found that even with a sample rate of one sample every four milliseconds, program execution profiles are still adequate for directing performance optimisation efforts. During one sample period, approximately two hundred thousand instructions are issued.

sections of smaller duration need to be timed, forcing the performance analyst to use timing loops in order to achieve the needed statistical accuracy.

Of course, if the RTC is accessible from user code, RTC timestamps could be used in the time calculations. The resolution would improve and the accuracy would be limited only by the overhead for RTC access. However, the RTC is not a virtual timer. It is possible that between two successive RTC timestamps, a process can be context switched off the processor. In this case, the time difference between the two RTC timestamps would over-estimate, sometimes by a large factor, the time spent in the code section. Why? Because the time difference includes the time the process is not executing. To get around the virtual timer problem, performance analysts often resort to trial-and-error, hoping context switches do not occur during the measurement, or to running the code in dedicated mode. Neither of these approaches is a satisfactory general solution.

User-level time measurement based on measured process timing is a compromise approach. The time calculations are made with virtual timestamps and have better resolution than with sampled process timing (i.e., RTC resolution). The accuracy is limited by the system call overhead to access the process user time.[6] This timer access overhead relative to the code section time will determine the granularity of practical timing measurements.

Ideally, a virtual timer for each process state based on a high-resolution RTC and accessible to user code with low-overhead would be available for user-level timing measurements. The ELXSI 6400 multiprocessor is interesting in this respect because it allocates a 64-bit hardware virtual timer for every process, accessible by a single user-level instruction, that is incremented once every 25 nanoseconds when the process is executing [28, 149]. However, to our knowledge, no other current commercially available machines have this level of timing support.

2.3.4 Context Switch Tracing and Timing

One method has proven effective on the Cedar multiprocessor for providing low-overhead, high-resolution user-level timestamping support [118]. The problem solved is one of removing the real-time when a process is not executing from time measurements made with RTC timestamps. In Cedar, the RTC can be accessed through a single instruction with an overhead less than the

[6]Because the process times are measured, the user time will be updated by the system call as the process transitions between user and system state.

RTC resolution of 10 microseconds. In contrast, accessing the virtual times from the operating system cost approximately 200 microseconds, significantly reducing the accuracy of timing measurements. The problem, as discussed above, is that the real-time values do not reflect virtual process times.

If information could be saved, in addition to the RTC timestamps, that reflected when a process was not running, we might be able to adjust the user-level time measurement to remove the non-execution time. We instrumented the Xylem operating system [47] to record context switches in a user-supplied buffer. Each context switch, whether on or off the processor, is stored in the buffer with an identifier and a RTC timestamp. With this information, any context switches occurring between two user timestamps can easily be detected by comparing RTC timestamp values with those of the time measurement — any non-execution time can then be removed from the time measurement.

Context switch tracing allows the RTC to be used for user timestamps, thereby giving a low-overhead timestamping access, while maintaining virtual timing measurements with high-accuracy. Similar techniques could be easily applied to machines other than Cedar. Context switch tracing is also useful in the more general case of user event tracing for establishing a virtual time domain of execution. Finally, context switch tracing can be applied to understanding the scheduling relationships between multiple tasks of execution. This application has been studied by Lehr in the PIE system [113].

2.3.5 Timing and Parallel Execution

The timing of parallel programs is fundamentally different from that of sequential programs. We distinguish between two timing values: *elapsed* time and *CPU* time. Elapsed time is the total execution time between any two program events along a virtual timeline. It measures, in a real sense, how long after the first event the second event occurred, adjusting for the time a program is not executing. In contrast, CPU time is a measure of the total amount of processing resources consumed between two program events. CPU time is a measure of total work and is computed by summing the amount of time a processor is used by the program between the two events for all processors.

The elapsed time value will always be less than or equal to the CPU time value. For sequential execution, elapsed time equals CPU time. However, this is not the case for parallel execution. The basic question is what time measurements are important for parallel execution. To determine speedup, elapsed time should be measured. However, to determine execution efficiency, we also need to be able to determine the amount of work performed.

If we consider a parallel program to be composed of multiple tasks, the process timing approaches discussed above can be used to determine task execution times. The sum of the task execution times represents the total CPU time of the program. Within a task, elapsed timing measurements can be made in a similar manner.

The problems occur when making inter-task timing measurements. The time reference used for local task measurements (i.e., the task virtual time maintained by the operating system) will not suffice, because a separate virtual timeline will be defined for each task. The first event may occur on task A at task virtual time t_A and the second event at task virtual time $t_B < t_A$ on task B. Instead of virtual timing, the real-time clocks of each processor could be used for timing measurements, but they would have to be synchronized, and there would be the risk of including time when none of the program tasks are executing. This latter problem could be resolved by running the program in dedicated mode, but this severely restricts the general applications of a parallel timing facility.

Under the assumption of globally synchronized RTCs, the approach adopted for the Cedar system is an attempt to solve the inter-task timing problem [118]. Effectively, the ability to make inter-task timing measurements depends on the existence of a program-global virtual time reference. We extended the measured process timing capabilities in the Xylem operating system to apply to each task of a multitask program (referred to as a Xylem *process*). In addition, we defined a virtual timer for the Xylem process. This process virtual timer differs from the individual task virtual timers in that it "ticks" whenever at least one task of the program is executing. By accessing the current process virtual time, globally consistent virtual timestamps can be used for inter-task timing measurements.

Because of the additional overhead incurred in the operating system, the process virtual time approach used for Cedar could only be practically applied in a shared memory system. Even then, as the number of tasks increase, the corresponding increase in total task transitions will make maintaining the process virtual timer unmanageable.

2.3.6 Global Clock Synchronization

Approaches that generalize RTC timestamping with context switch tracing to multiple tasks hold promise for supporting inter-task timing measurements, but depend on globally synchronized real-time clocks for correct timestamp analysis. Rather ingenious solutions have been developed in the case of parallel and locally distributed machines to solve the global clock synchronization problem [54, 104, 165]. The foundation for many of these software solutions is due to Lamport [104]. The techniques address the problem of establishing temporal relationships between events on different processors. Essentially, they all use some form of local clocks and timestamps to create a partial ordering of events. Unfortunately, the timing accuracy of many of these approaches is poor.

Other mechanisms attempt to improve timing accuracy by measuring the time difference and drift between individual processor clocks [165]. Generally, some method is used to establish an initial time reference for all processors.[7] Unfortunately, after this initial synchronization, the individual processor clocks can drift apart at different rates. Similar synchronization mechanisms can be used to measure the amount of clock drift over a period of time. This drift factor could be made available to each processor to be used during timing measurements. Again, depending on the variability of the synchronization mechanisms, the accuracy of the drift approximations, and the sensitivity of the hardware clocks, these techniques can result in poor timing accuracy.

Our main interests are in multiprocessing systems used to support parallel processing. Thus, we view processors in systems of this type to be in close proximity (e.g., within the same room). In this respect, we regard the issue of providing a high-resolution, globally synchronized clock as an engineering problem. Well-known techniques, such as phase-locked loops, can be applied to hardware clock distribution and synchronization. In fact, separate processor clocks can be synchronized to an arbitrary degree of resolution using these methods.[8]

We advocate the inclusion of a globally synchronized, real-time clock sub-system in all multiprocessor machines that potentially require inter-processor timing measurements. This view is shared by many researchers [176]. Although the implementation considerations of such

[7]For example, Rudolph [165] uses the global RS-422 connection to all Intel iPSC/2 nodes for simultaneously requesting every node to reset its clock to zero.

[8]The μTRAMS performance measurement chip of Mink and Carpenter has been designed to provide global real-time clock synchronization at resolutions up to 100 nanoseconds [138].

a sub-system are not complex, it must be thought out in the context of the whole system design. The difficulty, of course, is in convincing multiprocessor manufacturers to build these capabilities into their systems.[9] Some arguments cite the need to support processors of different speeds, with each processor running off its own clock. However, the real-time clock sub-system should not be viewed as the main processor clocking mechanism but as a separate component, supporting performance timing measurements. Unfortunately, only when the performance measurement support is considered a main component of the overall system design, will globally synchronized, real-time clocks become commonplace.

2.3.7 Remarks

The ability to make high-resolution, accurate timing measurements is critical for good performance observability. Although it was said at the beginning of the chapter that the performance experiment dictates the needed capabilities of the performance measurement system, timing is so fundamental a performance measurement that it should be given more than just casual consideration during computer system hardware and software design. Many of the high-performance systems today have poor timing capabilities. Ironically, timing solutions, some of which are discussed above, are often significantly less complex to implement than most of the components used to achieve increases in raw performance. Yet, the faster these systems become, the more problematic will be timing measurement. In particular, high-resolution, globally synchronized clocks is an idea whose pervasive implementation is long overdue.

2.4 Tracing

The ability to observe time-dependent performance behavior requires some form of tracing measurement. Although execution summary profiles (mainly of program routines) show where performance occurs (i.e., the performance of individual execution modules), they do not reflect performance behavior over time. However, the advantage of profiles is that they can be calculated at run-time and require only a small amount of storage. On the other hand, the periodic sampling of performance information to determine run-time means or historical distributions of

[9]Some commercial multiprocessor manufacturers, such as Alliant and Cray, already provide support for globally synchronised real-time clocks.

performance (as in the case of Unix timing measurements of processes) lack the measured data to correlate these values with the logical operations of a program's or the operating system's execution. These approaches show when the performance is being achieved (because they are time-based) but provide no information as to where. Time-based event traces capture both what performance is being achieved when and where in the program's or operating system's execution the performance occurs.

The tracing of events has a long history in performance measurement and debugging. Although many of the early computer performance measurement devices were designed to capture resource utilization statistics [148], their use in collecting traces was also considered [49, 60, 167, 175]. With the growth of distributed processing, the tracing of software events became the dominant technique for performance monitoring [1, 14, 79, 92, 104, 135, 180]. For multiprocessor computer systems, software event tracing is the foundation for many parallel debugging techniques [13, 61, 48, 81, 90, 110, 112, 130, 136]. Also, several activities have demonstrated the utility of tracing in multiprocessor performance measurement [27, 57, 59, 66, 88, 115, 133, 137].

While the literature is rich with papers discussing the application of trace-based measurements for debugging and performance analysis, little has been written on the more fundamental problem of designing an efficient event collection facility. Below we consider several issues in this regard. We focus our discussion on the tracing of software events using software-based and hardware-based approaches. In particular, we will consider the problem of tracing parallel program execution. First, we described the general tracing approach.

2.4.1 The General Tracing Approach

Conceptually, tracing is a simple operation that saves data associated with an event X in a buffer whenever X occurs during program execution. To be able to interpret the data, it must be coded with an event identifier. If a parallel program is being traced, the events should be further typed by the thread of execution producing the event and by the physical processor identifier where the event occurred. The processor identifier is often required to perform parallel performance analysis (e.g., processor utilization). Each event must additionally be timestamped to properly order its time of occurrence with other events. A *trace*, τ, therefore, is defined to be the time-ordered sequence of events occurring during program execution where each event is a tuple of the form

event id	*timestamp*	*thread id*	*processor id*	*optional data*

The optional data field is for event-specific data; for instance, if the event is an entry into a routine, the optional data could be used to store the name of the entered routine.

An efficient implementation of the general tracing approach must solve four problems:

1. event logging
2. timestamping
3. trace buffer allocation
4. trace I/O

Event logging involves mechanisms used to produce an event in the trace. This includes software instrumentation to generate the event (a *sensor* in [113]) and any run-time system support to store the event in a trace buffer (a *notifier* in [113]). The problem of timestamping is to provide a high-resolution, globally synchronized time value that will allow detailed event timing measurements and provide a basis for time-ordering multiple event streams. We have covered the issues related to timing and timestamping in §2.3 — in this section we assume the existence of a high-resolution, globally synchronized time source that additionally has low-overhead access. Lastly, a tracing system must solve the problems of trace buffer allocation and trace I/O. The mechanisms used to store the trace often determine the performance and versatility of the tracing system. Not only do issues of trace buffer size have to be considered (this determines where event data can be stored), but the dynamic event bandwidth into the tracing system must be compared to its output bandwidth to determine where trace buffering should be place and what limits exist on real-time trace output.

Below, we consider these problems in the context of software-based and hardware-based solutions to software event tracing.

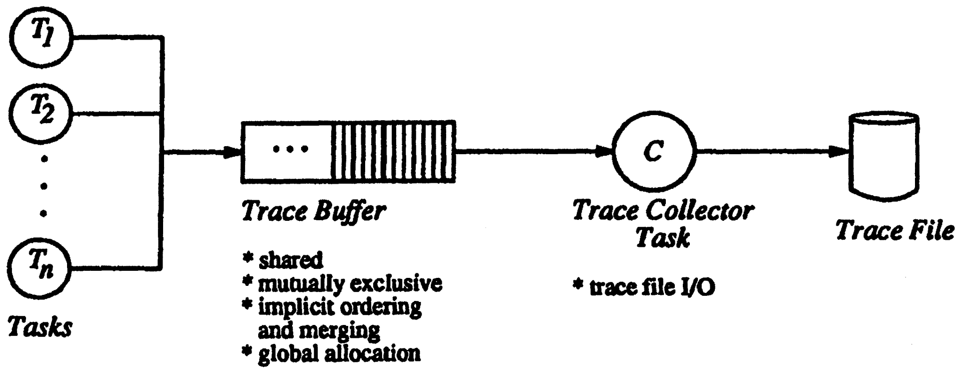
2.4.2 Software-Based Solutions

Software-based solutions to tracing use the resources of the computer system to log events, to buffer the trace, and to output the trace. The process of logging the events involves event construction and trace buffer access. Although hardware-based solutions also depend on software instrumentation for event generation, information such as the timestamp and processor identifier could be off-loaded to the hardware tracing system. In software approaches, this data must be retrieved by software and packaged with the event, causing logging delays. Access to trace buffers depends on the trace buffer allocation strategy. Ideally, each thread of execution would be assigned a separate, static trace buffer so that trace buffer contention is eliminated. However, to efficiently use trace buffer memory, some form of dynamic trace buffer sharing is often required; see below. With respect to the logging of events, dynamic trace buffer allocation and trace I/O necessitate *mechanisms for the error-free use of trace buffer resources.*

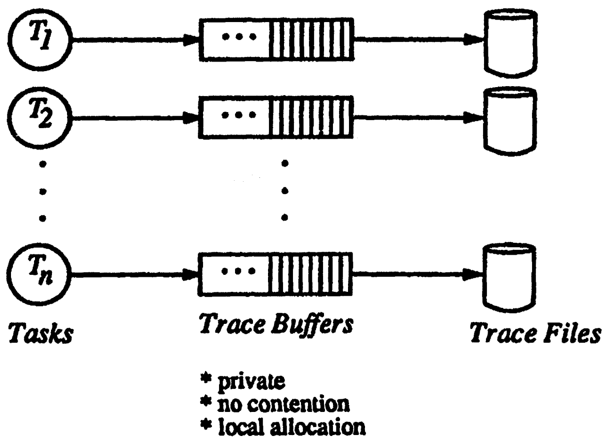
There are two general approaches to software-based trace buffer allocation: shared and private; see Figure 2.3. Shared trace buffer allocation is only possible on shared memory multiprocessors. The most restrictive approach requires that all tracing tasks gain exclusive access to a common trace buffer with every event logging operation. Although this increases the overhead of event logging, it has advantages in maximizing buffer memory utilization and in implicitly merging multiple events streams — the timestamp can be generated after buffer access has been granted. To reduce the overhead of buffer access, various forms of dynamic buffer allocation (where buffer blocks are allocated from a shared pool) are possible. For this discussion, we consider these dynamic approaches to be shared buffer allocation policies. However, the use of buffer blocks for parallel event logging gives up the implicit merging capability of the basic shared queue and this merging operation must explicitly be done later on.

To completely remove the overhead of trace buffer contention, tasks can be allocated private buffers. For distributed memory systems, private buffer allocation is the only scheme possible. In shared-memory systems, private trace buffers are also an option, but the drawback is less efficient use of trace buffer memory. In addition to removing the contention overhead, private buffer allocation has the benefit in hierarchical memory machines of being able to "cache" portions of the event trace buffer for faster access. Because most of the benefits of locally caching shared buffers would be lost to the synchronization overheads, shared buffers are normally allocated in globally shared memory regions.

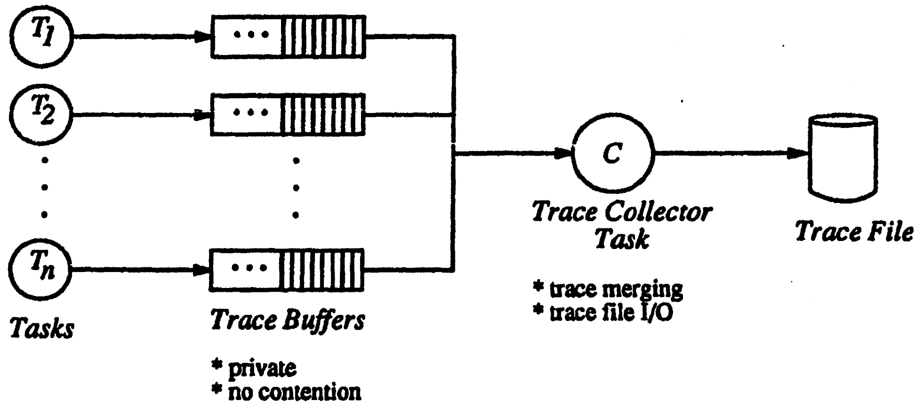
The requirements for trace I/O depend on the need to free buffer space (in the case that traces are larger than can be completely stored in buffer memory) and the need to analyze the trace data as it is being produced. Clearly, if trace buffers fill, event logging must wait



Shared Trace Buffer



Private Trace Buffers



Private Trace Buffers with Trace Collector

Figure 2.3: Alternative Trace Buffering and Output Strategies

until buffer space is released.[10] Depending on the buffer allocation scheme, several trace I/O alternatives are possible. In the shared buffer case shown in Figure 2.3, a *trace collector* task checks the trace buffer periodically for overflow conditions, writing trace data to a trace file when necessary. The disadvantage of this approach is that the collector task competes for processing resources with the other tasks. If there are fewer tasks than processors or the average parallelism is less than the number of processors, the intrusion of collector tasks might not be significant;[11] otherwise, the performance might be perturbed [113]. It could also be the responsibility of the tracing tasks to check for overflow and perform the I/O themselves when overflow occurs. Although this removes the problems of processor sharing with a collector task, it might significantly intrude on program execution. Two trace I/O examples are shown in Figure 2.3 for private buffer allocation. The first has each task performing its own I/O to its own trace file. Obviously, the task traces are not merged in this case and must be done after program execution. The second example again uses a collector task. In addition to the trace I/O, the collector task must merge the event streams; a single, merged trace file is generated. For distributed memory systems, the collector task would reside on a (host) processor and would access the task's private buffers via the interconnection network.

The tracing facilities used in PIE [113, 170] and Cedar [119, 172] exemplify tracing approaches for shared memory multiprocessor systems. In PIE [113, 170], tasks are allocated separate, fixed-size trace buffers. A collector task is responsible for removing events from all task trace buffers and writing the combined traces to a file. The Cedar tracing system [119, 172] allows the user to select among several alternatives for trace buffer allocation and trace I/O. Each tracing task can either have statically allocated fixed size trace buffers or dynamically allocated trace buffers using linked buffer blocks. Independently, the user can choose where the trace buffers will be stored in Cedar's memory hierarchy. Finally, run-time trace I/O can be selected, causing a parasitic "snooper" task to run concurrently with the program task — the trace I/O can be sent to a file or to a remote computer over a network.

[10]For virtual memory systems, very large trace buffers can be created in virtual space. Although virtual memory is not infinite, in some cases, explicit software to handle trace buffer overflows can be avoided by just allocating more virtual memory.

[11]If unused processors exist, the collector task could be bound to a processor for the duration of the computation.

The software-based tracing approaches for distributed memory systems are restricted mainly by the system architecture. Trace buffers cannot be shared and must instead be allocated in the each processor's memory. Buffer overflow can only be detected by tasks local to a processor and message passing operations must be used to off-load the trace to a central point for trace I/O.[12] Furthermore, the generation of globally, synchronized timestamps in these systems is a problem. Rudolph and Reed [161] give an excellent discussion of these issues in the context of the Intel iPSC/2 multiprocessor. The tracing software developed by Rudolph [165] for the iPSC/2 reflects the design constraints imposed by the architecture.

2.4.3 Hardware-Based Software Event Tracing

The purpose of considering hardware support for software event tracing is to reduce the overheads associated with event logging, to perform timestamping externally, to eliminate the need for software trace buffering, and to improve file I/O, including real-time trace access. A hardware tracer exists either as a memory-mapped device to which tasks can write events using memory operations [7, 59, 79], or as an I/O device accessible by I/O operations [125, 139, 140, 164]. As a memory-mapped device, the hardware tracer monitors the address bus for addresses falling within its range. If a valid address is detected, the address and the data (on the data bus) are logged. As I/O devices, hardware tracers reside on special I/O ports implemented as part of the standard machine. Hardware tracers can be centralized [59, 125, 139], where all event streams come to a central monitoring system, or distributed [79, 138, 164], where event streams are captured locally at each processor.

Event logging is generally faster on memory-mapped hardware tracers because memory operations can be used. However, when generating user-level events, it is important to have the tracer visible in the user memory space. I/O operations must be used for tracers configured as I/O devices, resulting in greater event logging overheads due to system call delays. Hardware tracers typically support the timestamping of events with a high-resolution clock. Because the monitor is external, implementing a globally synchronized time source can take advantage of standard hardware solutions to clock distribution [138].

[12]The existence of concurrent I/O facilities, such as provided on the Intel iPSC/2 system, alternately allows I/O to be done from each processor node.

The most challenging aspect of designing a hardware tracer is the trace buffering system. Although hardware can be designed for any level of tracing support desired, cost is often a major design consideration. The approach taken for trace buffering, in particular the choice of trace buffer size, is the prime determinant of the total cost. To the extent possible, the trace buffer size should be expandable, allowing extra buffer capacity to be added if cost permits. However, assuming that there are traces whose size will always be greater than the hardware tracer's buffer memory, the hardware tracer must also provide solutions to secondary memory buffering. Whereas file access is commonly provided by computer systems and, thus, can naturally be used for software tracing, the hardware tracer will not normally be able to take advantage of it. However, approaches often support standard I/O interfaces (e.g., the tracer in [7] uses the VME bus interface), allowing standard disk drive technology to be used. Perhaps the most important problem related to trace buffering is its bandwidth. The hardware tracer must be able to support the capture of event data at its nominal rate.

Hardware-based software event tracing for a large number of processors must look to logic integration to be both economically and technically feasible. The design of VLSI components for hardware tracing [138] can help to lower the unit cost but at the expense of reducing trace buffer size (e.g., the buffer size in the μTRAMS chip [138] is 512 bytes). This places additional burden on the supporting infrastructure used to extract the trace data from each processor tracer. This support system serves the function of merging the trace streams, and must rely on a combination of high-bandwidth networks and a hierarchical organization to handle the potentially large amount and rate of trace data [160].

2.4.4 Remarks

The development and deployment of trace measurement tools would be facilitated by the adoption of standards for trace format, software interfaces for software-event tracing, tracing libraries for software-based tracing, and instrumentation interfaces for hardware-based tracing [123]. A standard trace interchange format (STIF) [89] promotes the development of trace analysis and visualization software that can be used with traces from a variety of machines. Similarly, standard software interfaces to tracing systems provide a portable instrumentation interface between parallel systems [123]. This helps minimize the code modifications needed to port instrumented codes between machines. Standards for hardware-based tracing might seem unrealistic. However, in the area of defining an external hardware tracing interface (EHTI) between the computer system and the hardware tracer to support software event tracing, both the memory-mapped and I/O-mapped approaches have standards potential. The EHTI specification would define the address size, data size, control signals, and the timing characteristics of the external interface. In this case, simplicity is the key — a 32-bit address port, 32-bit data port, and one control signal to indicate valid address and data have been suggested [123].

The adoption of standards depends on the significance of the problem and the need for a set of standard solutions. In the area of tracing for performance measurement, it is likely standards will never be adopted. However, as more tools are made available and components of a tracing system are developed (e.g., the μTRAMS chip), a *de facto* consensus on approaches may be reached.

2.5 A Hardware-Based Monitor for the Intel iPSC/2

Although we can compare different performance measurement techniques and argue for basic levels of instrumentation support, ultimately performance measurement solutions will be evaluated in terms of their expense versus their perceived importance. Because of the design time and cost involved in building performance measurement systems, often these projects exist as research efforts where tools are either implemented on prototype machines (e.g., [22, 60, 79, 162]) or retrofitted onto existing systems (e.g., [94, 120, 137, 140]). It is a sad fact that many of the commercial computer systems contain minimal performance measurement support — this is especially true in the case of high-performance systems where performance measurement needs are more profound. Two notable exceptions exist. Encore's Parasight tool [8] provides an integrated software environment for parallel program debugging and performance evaluation. However, Parasight is completely software based, limiting it use in obtaining high-frequency performance data. The hardware performance monitor on the Cray X-MP and Y-MP supercomputers [107] has been beneficial for capturing performance data about hardware operation,

[13]Portions of §2.5 appear in the papers "A Hardware-Based Performance Monitor for the Intel iPSC/2 Hypercube" in the Proceedings of the 1990 International Conference on Supercomputing [125], and "An Integrated Performance Data Collection, Analysis, and Visualization System" in the Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications [126].

but it cannot be used in practice for parallel programs and its software support is based only on profiling.[14]

To answer the questions of cost effectiveness of a performance measurement system, it is important to understand the design factors and the trade-offs that must be considered when developing one for a particular machine. Reilly [162] describes such issues in detail for the hybrid monitor he developed for Digital Equipment's experimental M31 multiprocessor. He concludes that while the monitoring system resulted in an extremely versatile measurement tool, its implementation was too complex and expensive to be adapted to a commercial multiprocessor. In this section, we consider similar issues with respect to a performance measurement system for message passing multiprocessors, namely the Intel iPSC/2.

Message passing systems pose particularly acute measurement problems. First, detectable events occur locally at each processor. Identifying global events requires associating two or more events from different processors. The obvious approach imposes a total order on events, based on event timestamps, and identifies global events from temporally proximate event groups. Unfortunately, the second problem complicates solutions to the first: most message passing systems, including the Intel iPSC/2, lack a globally synchronized clock to create event timestamps — this clock is needed to maintain event causality. With a global time reference, the distributed event data still must be collected for analysis and presentation.

Our measurement solution approach for the Intel iPSC/2 hypercube was the design of HyperMon, a hardware system to capture and record software performance traces. HyperMon represents a compromise between fully-passive hardware monitoring and software event tracing; software generated events are extracted from each node, timestamped, and externally recorded by HyperMon. Using an instrumented version of the iPSC/2 operating system and several application programs, we present a performance analysis of an operational HyperMon prototype and assess the limitations of the current design. Based on these results, we suggest design modifications that should permit capture of event traces from the coming generation of high-performance distributed memory parallel systems.

[14]Chapter 7 describe a software event tracing system we developed for the Cray X-MP with the capability of sampling Cray hardware performance monitor statistics with each event.

2.5.1 Intel iPSC/2 Description

The design of hardware support for performance data capture necessarily depends on the underlying parallel system — system software and hardware determine the requirements for and limitations of the data capture interface. Thus, an instrumentation environment must be understood in the context of the intended architecture — the Intel iPSC/2, a second generation, distributed memory parallel system.

The Intel iPSC/2 hypercube [10, 33] incorporates evolutionary advances in technology, including an Intel 80386/80387 microprocessor pair, a 64K byte cache, and up to 16 megabytes of memory on each node. The iPSC/2 includes an autonomous routing controller to support fixed path, circuit-switched communication between nodes. This communication system eliminates most of the store-and-forward latency that existed in earlier distributed memory systems.

The software development interface for the iPSC/2 is a standard Unix system that transmits executable programs to the nodes, accepts results from the nodes, and can, if desired, participate in the computation. Finally, the Unix host supports node file I/O to its local disk and to remote disks via a network file system protocol.

To reduce dependence on the Unix host and to provide input/output performance commensurate with computing power, the Intel iPSC/2 nodes also support link connections to I/O nodes. Each I/O node is identical to a standard compute node, with the exception of an additional daughter card that provides a SCSI bus interface. The SCSI bus supports up to seven peripherals and has a peak transfer rate of 4 megabytes/second. Physically, the I/O sub-system can be packaged as a separate cabinet with cable connections to the compute nodes.

Because the I/O nodes provide a superset of the compute node functionality, software support for disk and file access is realized by augmenting the NX/2 node operating system on both the compute and I/O nodes. The Concurrent File System (CFS) allows application programs to create, access, or modify files on both the hypercube host and the individual hypercube disks. As we shall see, this provides the mechanism for archiving performance data after its capture by our HyperMon instrumentation support hardware.

Although each iPSC/2 node contains a local clock with one microsecond resolution, the node clocks are not globally synchronized and can drift apart at a measurable rate. Consequently, event timestamps on different nodes may violate casuality (e.g., a message might appear to be received before its transmission). Although software techniques can ameliorate the effects of clock drift by synchronizing the clocks and reordering timestamped events, a high-resolution, global time reference is the simplest and most desirable solution.[15] Even with a global time reference, the distributed event data still must be collected for analysis and presentation.

2.5.2 HyperMon Design

The absence of a global, accurate, and consistent time exacerbates the already difficult measurement and recording of distributed events [104]. The constraints on measurement resolution created by distributed clock synchronization, coupled with the overheads of software tracing, limit the range of performance behavior that can be accurately observed. The design of the HyperMon architecture attempts to circumvent these problems. Below, we describe the design and operation of HyperMon, followed in §2.5.4 by a summary of results obtained from HyperMon bandwidth experiments and performance tests using real message passing programs.

2.5.2.1 iPSC/2 Event Visibility

The hardware basis for HyperMon is a little-known, though standard, feature of the iPSC/2 that makes possible external access to software events generated by each node. Five "performance" bits from a port in the I/O address space on each iPSC/2 node board are routed via the system backplane to an empty slot in the system cabinet. The standard cabinet holds up to 32 iPSC/2 nodes and all 160 signals are present at the spare node slot. However, the current HyperMon prototype supports data capture from only the first 16 nodes.

Because the software performance instrumentation generates event data by writing to an I/O port, one bit must be reserved as a software strobe to signal that the remaining four event data bits are valid. The five data bits written to the I/O port are interpreted as follows.

4	3 0
Strobe	*Event Data*

A software event is generated by writing the event data to the 5-bit port, first with a strobe of zero then with a strobe of one. The 16 MHz 80386 microprocessor in the Intel iPSC/2 requires

[15]See [161] for a description of software causality maintenance and its limitations.

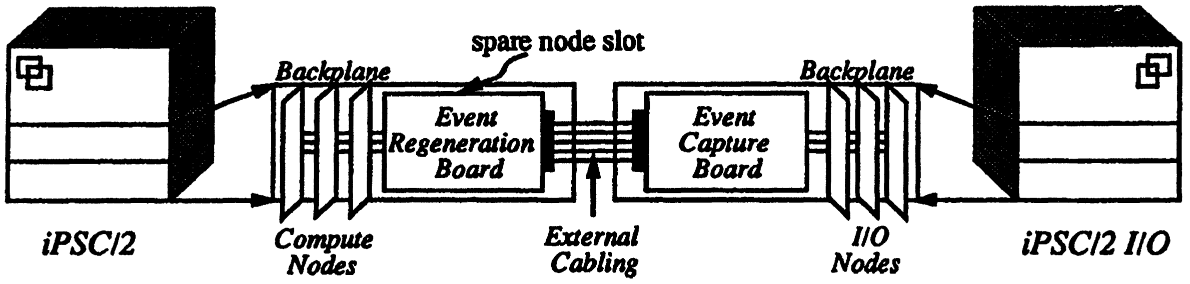


Figure 2.4: HyperMon Architecture

approximately six cycles to complete an I/O write operation, versus two for a standard memory access. Thus, a minimum of twelve cycles are needed to write a 4-bit software event.

Clearly, the sixteen possible events representable with a 4-bit event identifier greatly restrict the range of possible instrumentation. If additional events are needed, or, equally likely, if data are associated with an event, multiple I/O write operations will be required. A performance instrumentation of the Intel NX/2 operating system [165] produces a range of event sizes, ranging from two to thirteen bytes plus a timestamp. Moreover, an analysis of these operating system traces shows that most events are four or more bytes, excluding the software timestamp.

Unfortunately, transmitting larger events is expensive. In addition to the twelve cycle memory access penalty for each 4-bit I/O operation, there is software shift and mask overhead to extract 4-bit items from event words. Clearly, a larger I/O event field is desirable to increase event bandwidth and to reduce the cost of bit field extraction. However, backplane hardware constraints limit the number of available backplane signals. Despite these limitations, hardware support for data collection permits real-time extraction of performance data and, consequently, capture of larger traces than otherwise possible with node memory trace buffering.[16]

2.5.2.2 Hypermon Architecture

Figure 2.4 shows the primary components of the HyperMon architecture and their physical relation to the Intel iPSC/2. Reflecting the physical packaging of the iPSC/2, HyperMon is partitioned between the cabinet that contains the iPSC/2 compute nodes and the cabinet

[16]In §2.5.4 and §2.5.5, we will return to the question of node hardware support for data extraction. The current HyperMon design reflects the 5-bit iPSC/2 data extraction interface.

containing the I/O nodes and disks. As just described in §2.5.2.1, each iPSC/2 compute node independently sends 5-bit (four bits and a strobe) event data to HyperMon.

In the compute node cabinet, the HyperMon *event regeneration board* (ERB) converts the 5-bit TTL-level event signals from each node into differential form before transfer to the *event capture board* (ECB) residing in the I/O cabinet. The ECB captures the events, generates global timestamps, and stores the resulting event data in internal memory buffers for access by I/O nodes. To prevent disruption of event data capture and perturbation of user I/O requests, one or more I/O nodes are dedicated to event data recording.

In principle, the I/O nodes can be used for preliminary analysis of the event data. In practice, however, the desirability of real-time data reduction, with a possible decrease in disk I/O requirements for data recording, must be balanced against the probability of ECB data buffer overruns if too many I/O node compute cycles are diverted from disk service. Clearly, the most efficacious mix of real-time data reduction and disk recording, with post-mortem analysis, depends on the event data rate and instrumentation requirements.

Event Capture

Figure 2.5 shows the functional design of the event capture board, the primary hardware component of HyperMon. There are five major parts: event data queueing, event strobe synchronization, timestamp generation, event frame construction, and I/O node interface; each is described below.

After signal regeneration by the HyperMon ERB board, the 4-bit event data from each iPSC/2 node are placed in a separate FIFO buffer that is clocked by the corresponding software event strobe signal (i.e., the fifth bit from each node). Each node FIFO is 64, 4-bit entries deep and provides buffering during the event frame construction process; see below. Figure 2.6 shows the timing diagram for the two writes needed to assert valid event data from a node.

Because the individual node clocks are asynchronous, there is no direct timing relationship among the valid event data in FIFOs for different nodes. Thus, early in the HyperMon design, we were forced to decide where to synchronize the externally received event data with the internal ECB clock. Because the event data FIFOs provide implicit synchronization of the data bits, only the event strobe signals need be sampled relative to the internal ECB clock. In the HyperMon design, the software event strobes are sampled in successive 800 nanosecond time

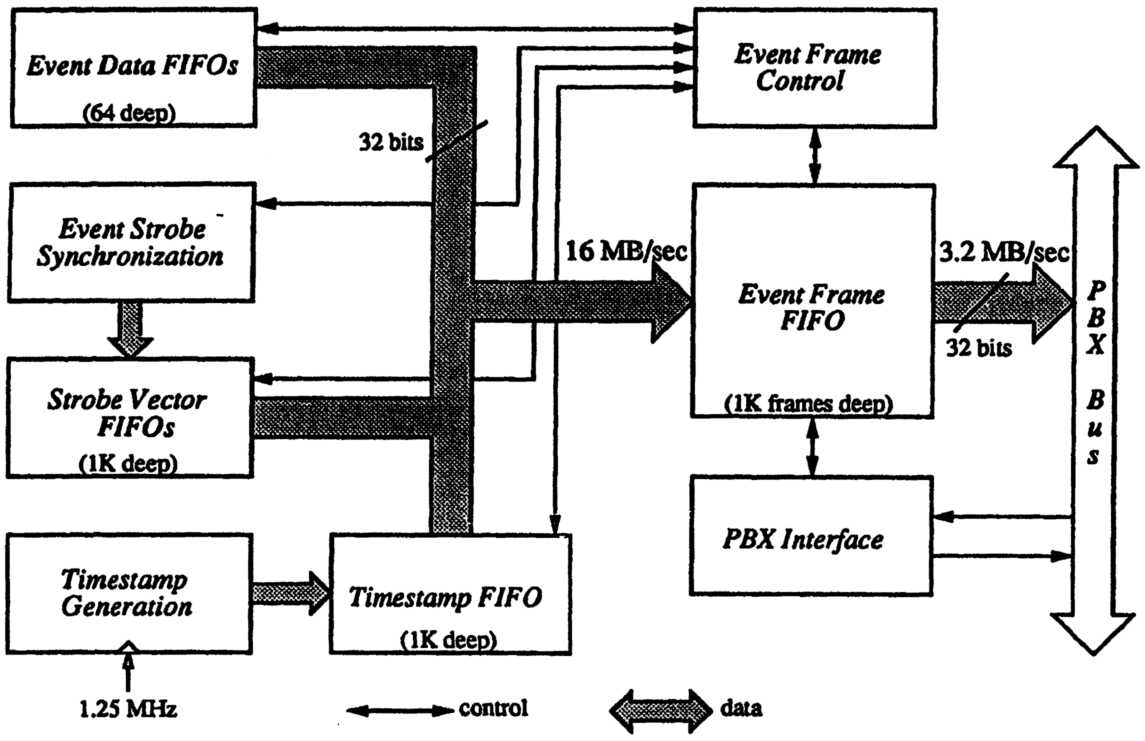


Figure 2.5: HyperMon Event Capture

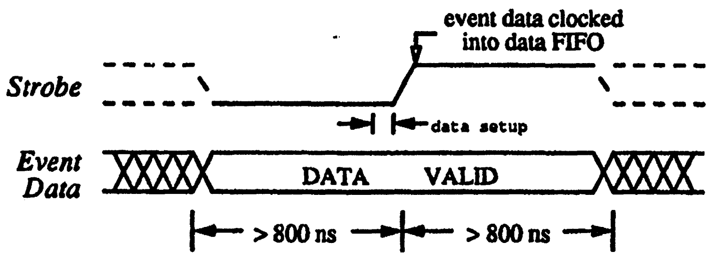


Figure 2.6: Event Data Timing

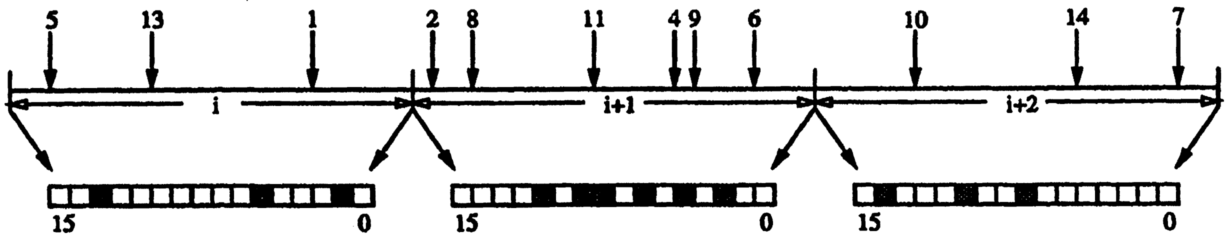


Figure 2.7: Strobe Vector Example

windows.[17] Valid event strobes within a time window are marked in an *event strobe vector* and stored in the strobe vector FIFO.

Figure 2.7 illustrates event strobes from different nodes and the corresponding strobe vectors during three successive time windows. The shaded bits in each strobe vector indicate which event data FIFOs have valid data from compute nodes.

Separate from the synchronization issue is the generation of event data timestamps. Although high-level events can be of any length, and one need generate only one timestamp for each event, the HyperMon hardware must assume that every four bits of event data represent a separate software event. Why? The HyperMon hardware embodies no notion of event types or lengths. Thus, each event data nibble must be assigned a 32-bit timestamp.

To avoid the expense of separate timestamp hardware for each node, a single timestamp generator is used. For each generated strobe vector, a timestamp also is stored in the timestamp FIFO. In general, the choice of timestamp resolution need not be dependent on the event strobe synchronization period. However, in the current ECB hardware, they are equal.

The design motivation for strobe vector and timestamp unification is to reduce hardware complexity. As a design alternative, each event nibble, a 4-bit node identifier, and a 32-bit timestamp could be sent to the I/O node. However, in this approach, 80 percent of the information would represent timestamp data. Moreover, for events occurring within the same time 800 nanosecond window, the timestamps for these events would be redundant. Instead, we adopted a strategy that packages event data as *event frames*. These event frames are the unit of transfer to the I/O node.

[17]The synchronisation period can be shortened to 400 nanoseconds or 200 nanoseconds through jumpers on the ECB. This allows faster event generation rates to be accommodated in the future.

Event 15	Event 14	Event 13	Event 12	Event 11	Event 10	Event 9	Event 8
Event 7	Event 6	Event 5	Event 4	Event 3	Event 2	Event 1	Event 0
Unused				Strobe Vector			
Timestamp							

Figure 2.8: Event Frame Format

In the event frame approach, the strobe synchronization period is the time basis for event frame construction. Each such event frame consists of four 32-bit words; see Figure 2.8. Four event data bits from each node FIFO always are placed in the frame. However, only those FIFO's with valid data, as determined by the strobe vector, for this time window, will be shifted into the event frame; the other event data fields in the frame are undefined. The strobe vector is recorded as part of the frame to identify the valid event data fields. Finally, the corresponding 32-bit timestamp is saved with each frame. Once an event frame is constructed, it is saved in the ECB's frame FIFO. To eliminate useless data, an event frame is constructed for a time window only if at least one of the nodes produces an event during the window.

If we compare separately transferring each event nibble to the use of event frames, it is clear that when there is only one valid event nibble in a frame, the overhead is substantial. In this case, only three percent of the frame is data. Only when four or more nodes have valid event data will transfer of event frames require fewer bits. The motivation for merging event data from multiple nodes is that the efficiency of event transfer is more important when more nodes are producing event data. In this case, the likelihood of multiple nodes producing event data within the same time window increases, and the ratio of valid event data to overhead also increases. When few nodes are generating event data, the need for efficient transfers to the I/O node is not as great.

Event Processing

The ECB supports a parallel bus interface (the PBX bus) to an iPSC/2 I/O node. Via this bus, event frames can be transferred to the I/O node's memory. These PBX bus transfers are mapped through the I/O node's memory space. In addition, the ECB provides a writable 8-bit control register to reset the board. There is a 4-bit status register used to signal error

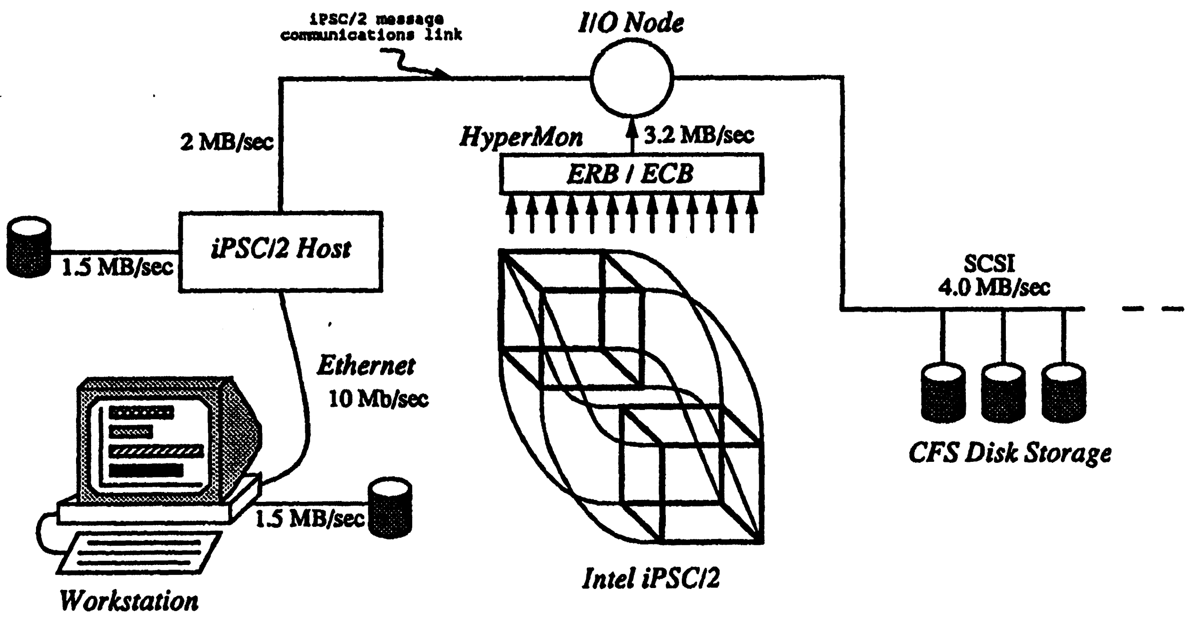


Figure 2.9: Event Trace Processing/Storage Options

conditions, most often FIFO overruns. The number of event frames generated since the last reset is accessible through a 12-bit frame count register. Finally, the event frame can be accessed using a single frame FIFO address. Referencing this PBX address will transfer one 32-bit frame word to the I/O node.

Once in the I/O node's memory, event frames can be decomposed into separate event streams for each instrumented node. Additionally, the I/O node's processor can be used to compress the event trace by computing statistics directly from the event data. Finally, the event trace data can be stored on the I/O node CFS disks for post-mortem analysis or transferred to the iPSC/2 host and associated workstations for analysis and presentation.

The ability to record trace data on local CFS disks or on remote disks attached to either the iPSC/2 host or a workstation, coupled with real-time or deferred trace analysis, provides a wide variety of trace storage and analysis configurations with distinct costs; see Figure 2.9. Selection of a trace processing mode depends on event frequency, density, and complexity. If the mean time interval between valid event frames is small, and we have observed that this often is true for operating system event traces, real-time event processing (e.g., statistical analysis) may not be possible — the I/O node minimally must record event data without loss.

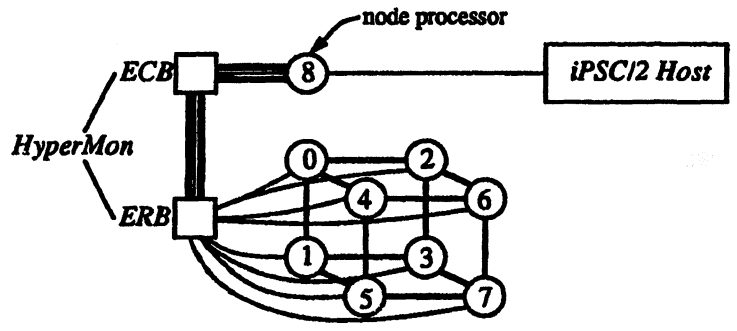


Figure 2.10: Current HyperMon Configuration

2.5.3 Current Configuration

The HyperMon prototype is implemented as a wire-wrapped, two-board set consisting of 115 integrated circuits — 20 on the ERB and 95 on the ECB. The experimental results reported here were obtained with eight nodes. We currently are testing HyperMon with sixteen nodes.

Figure 2.10 shows the current HyperMon configuration. At present, we are using a PBX-equipped node processor to communicate with HyperMon. Disk access must be done through the iPSC/2 host using the hypercube message communication links. Upon addition of an I/O node with PBX support, we will be able to test I/O transfer to CFS storage.

2.5.4 HyperMon Evaluation

The value of the HyperMon design can only be determined through experiments with real applications and software instrumentation support. Architecturally, HyperMon has the advantages of external event capture and real-time data access, but this must be weighed against the substantial overheads in event production; see §2.5.2.1.

In the remainder of this section, we describe the results of a set of experiments conducted with the HyperMon prototype. First, we used a series of synthetic benchmarks to measure the raw data bandwidth of the current configuration. Here, the goal was to determine potential data recording bottlenecks. Second, we used a software instrumentation [161] of the Intel iPSC/2 NX/2 operating system as a source of event data for HyperMon. This instrumentation generates a detailed event trace of operating system and application program interactions. The results of these experiments are discussed below.

2.5.4.1 Bandwidth Tests

A cursory examination of the HyperMon architecture suggests several points where measurement of raw bandwidth might reveal fundamental performance constraints. Below, we examine three: event generation from the node processors, ECB internal event frame construction, and PBX event frame transfers.

Given the interface constraints on the individual iPSC/2 nodes (software event strobing, a three-fold increase in access time to an I/O port, and software extraction of event nibbles), the overhead to send event data to the HyperMon ECB is substantially greater than that needed to record event data in a node's memory. To quantify this overhead, we began our tests with a simple, synthetic benchmark that generated events at the maximum possible rate on a single node. With software event recording, 3.9 seconds were needed to produce 1,000,000 events (assuming four bytes per event). This translates to a maximum software event recording rate of 1.02 Mbytes/second. In contrast, HyperMon recorded an equal amount of event data in 88.5 seconds, a hardware data recording rate of 45 Kbytes/second.[18] Simply put, the Intel iPSC/2 interface to HyperMon transfers data at a rate roughly 22.7 times less than that for software recording. Although the timestamp generation is done automatically by the HyperMon hardware, this savings in data transfer is greatly overshadowed by the costs of nibble extraction, software event strobing, and the use of an I/O instruction rather than a memory MOVE operation.

Internally, the HyperMon ECB can sustain a high event frame production rate. This value can be calculated directly from the internal ECB timing. The finite state machine controlling event frame generation operates at 20 MHz and requires eight cycles to build a frame. When all sixteen nibbles of event data in a frame are valid, this translates to a peak event data bandwidth of 10 Mbytes/second. As the number of valid data nibbles decreases, the event bandwidth decreases proportionally.

Finally, an I/O node must retrieve the event data from the ECB via the PBX bus. The PBX is an asynchronous, master-slave bus with a peak bandwidth of 18 Mbytes/second. However, when synchronization, address decoding, and data enabling overheads are included, the potential transfer rate across the PBX interface drops to 8 Mbytes/second. In practice, the

[18]Recall that the total amount of data recorded by HyperMon is much larger and includes the strobe vectors and invalid data nibbles in each event frame.

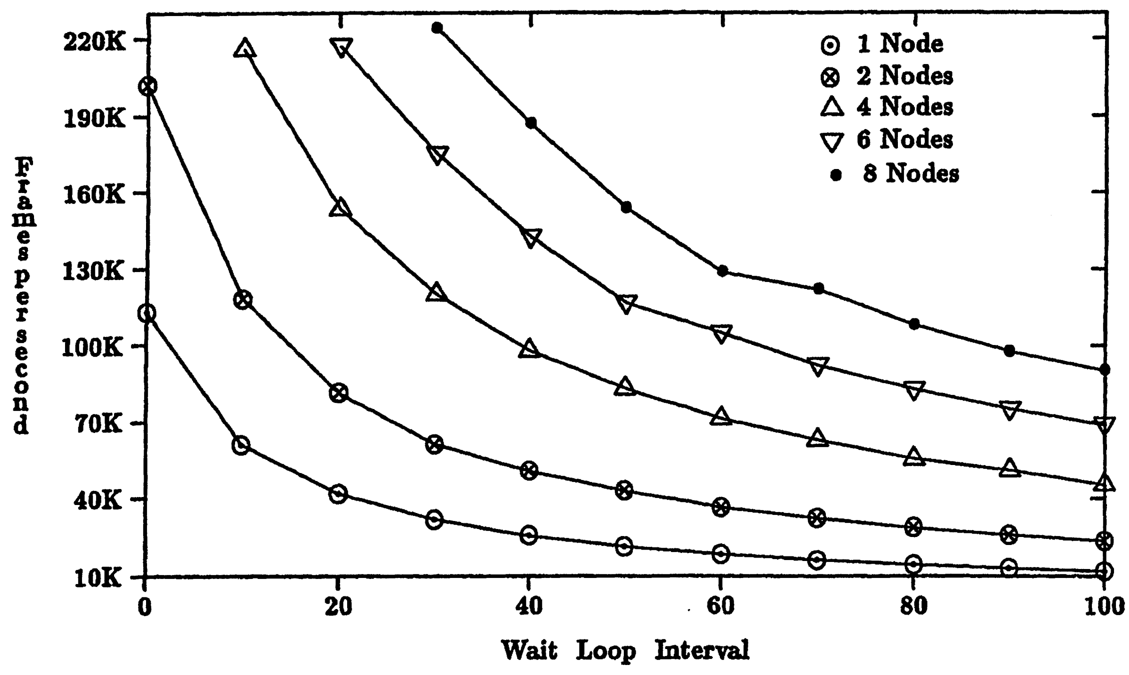


Figure 2.11: HyperMon Bandwidth Results

software overheads for PBX transfers to the I/O node (the master) degrade bandwidth performance further. Using optimized PBX interface software, we have achieved PBX transfer rates of 5 Mbytes/second to an I/O node.

To assess the interactions of the bandwidth limitations just described, we constructed a synthetic benchmark that generates a specified volume of event data for a user-selectable number of nodes. Figure 2.11 shows the maximum experimental bandwidths obtained via HyperMon when running this benchmark. In the benchmark, a delay parameter controls each node's event nibble generation rate; this delay parameter is the number of iterations of a null wait loop on the 16 MHz 80386. As expected, when the delay interval decreases, the bandwidth through HyperMon increases.

When only one node generates event data, every four bits of event data forces the creation of a separate event frame. A single node can generate significant frame bandwidth (113,000 frames/second, or equivalently, 1.8 Mbytes/second at sixteen bytes per frame). However, when two nodes produce event data at their maximum rates, the PBX bandwidth limits are challenged (202,000 frames/second or 3.2 Mbytes/second). Beyond two nodes, the ECB reports a frame

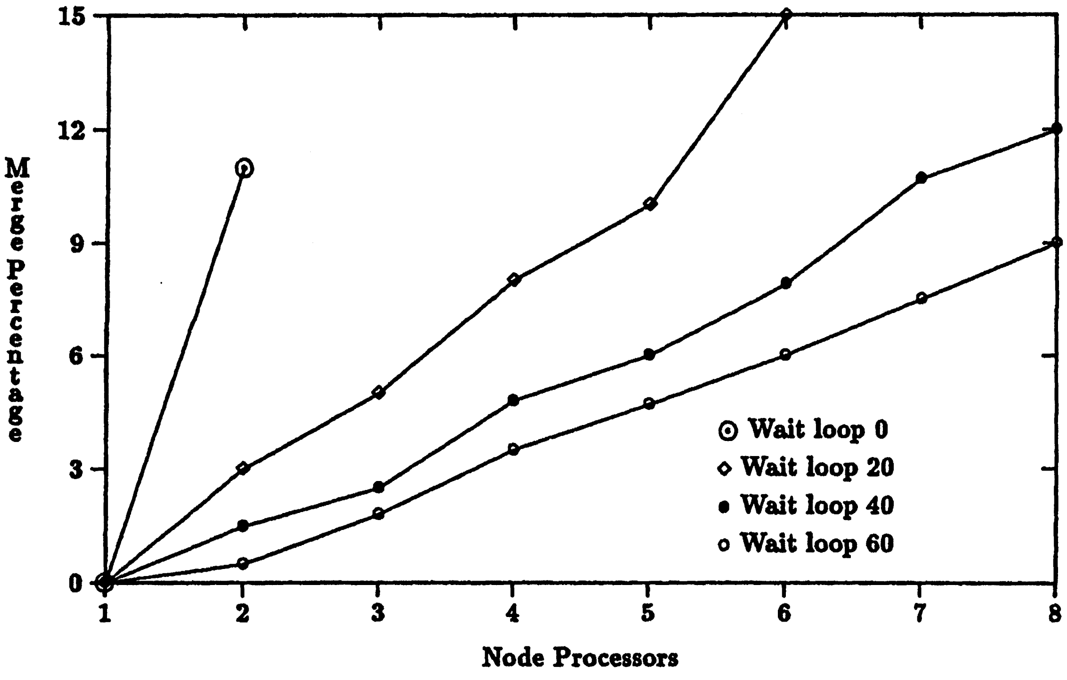


Figure 2.12: Event Frame Merging

FIFO overrun condition, and no experimental data can be obtained unless nodes delay the creation of successive event nibbles.

We indicated earlier that HyperMon was designed to be more efficient as the demand for event data bandwidth increased. This requires increased frame merging and amortization of the strobe vector and timestamp overhead over move valid event nibbles (i.e., the event frame creation rate should increase sublinearly with the number of nodes that simultaneously generate event data). Unfortunately, our experimental results suggest that the event data bandwidths must be very high to achieve effective merging. Figure 2.12 shows the percentage of merged event data as a function of the number of active nodes. Although the figure suggests that a high merging percentage should occur with high event data rates, merging does not increase quickly enough to offset the increase in the total event frame bandwidth seen at the PBX interface.

Although the high event data rates that cause PBX bandwidth saturation (evidenced by frame FIFO overruns) are beyond the sustained rate requirements for the HyperMon design, such conditions can exist during event data bursts; see §2.5.5. Overrun conditions depend on the length of these bursts relative to the size of data buffers in the ECB. Our prototype

Code	Description
Simplex	a parallel linear optimization program based on a column-wise Simplex algorithm [187]
Life	a parallel implementation of Conway's famous cellular automaton [70]
MatMul	a simple distributed memory matrix multiplication code
Place	a standard cell placement algorithm based on simulated annealing [24]

Table 2.2: Application Test Programs

implementation limits the frame FIFO size to 1K frames. As our results below suggest, this buffer size makes HyperMon susceptible to event data bursts of moderate duration.

2.5.4.2 Monitoring Real Applications

Ultimately, the software instrumentation level required to capture the execution behavior of a parallel computation will dictate the frequency and volume of data that must be recorded by the monitoring system. Not surprisingly, certain instrumentation levels can exceed the capabilities of any data recording system. Event processing and analysis requirements impose additional constraints on feasible performance experiments. As a consequence, performance observability mandates a balance between the rate of event data generation and analysis, and the fundamental limitations of the monitor's operation.

To understand HyperMon performance in an instrumentation environment for real applications, we compared the execution of four programs in three monitoring environments: no event data generation (raw), software event recording to node memory, and hardware event recording with HyperMon. The execution data for the latter two cases was restricted to that captured by an instrumentation of the iPSC/2 NX/2 operating system source code [166]. This system generates three classes of operating system events: message transmissions, process states transitions including context switches, and system calls. Below, we describe the characteristics of the application programs, the observed performance data, and HyperMon's performance.

Execution Statistics

Table 2.2 shows the four application programs used in our study of data capture for operating system instrumentation. Each of these applications was run on 1, 2, 4, and 8 nodes for each of the three data capture scenarios. To prevent event data bursts that might saturate the PBX

Application	Total Time (seconds)			Logged Data (bytes)		Logging Rate (bytes/second)	
	Raw	Soft	Hard	Soft	Hard	Soft	Hard
Simplex							
1 node	151.059	151.261	154.838	63437	2302144	419	14868
2 node	77.850	78.396	86.524	417476	10273040	5325	118730
4 node	42.068	42.561	54.338	1245712	27763296	29269	510937
8 node	24.277	24.931	41.045	3190873	67493104	127988	1644368
Life							
1 node	152.222	152.335	154.157	34103	1204608	224	7814
2 node	111.869	112.274	120.280	277359	10155584	2470	84433
4 node	78.505	79.112	93.048	752373	27607776	9510	296705
8 node	43.981	44.741	58.202	1670188	60352768	37330	1036954
MatMul (256x256)							
1 node	245.660	245.840	248.808	54706	1928096	223	7749
2 node	123.118	123.210	124.696	55215	1947792	448	15620
4 node	61.975	62.045	62.677	56277	1985376	907	31676
8 node	31.369	31.423	31.859	58456	2069136	1860	64947
Place							
1 node	318.012	318.473	322.325	74225	2628192	233	8154
2 node	82.000	85.071	143.356	2018749	74003616	23730	516223
4 node	69.668	72.851	137.145	4147564	149195680	56932	1087868
8 node	38.498	41.427	102.221	7977226	(overrun)	192561	(overrun)

Table 2.3: HyperMon Application Results

bandwidth and overrun the event frame FIFO, we set the delay interval for hardware event recording to twenty (i.e., writing of successive event nibbles was separated by twenty iterations of a null loop).

Not surprisingly, Table 2.3 shows that hardware data recording using HyperMon consistently causes greater perturbations than software recording in the node memories. This was expected from our earlier analysis of performance penalties imposed by the node interface to HyperMon. However, the degree of perturbation differs for each application program and number of nodes. Simply put, differences in program behavior are manifest as differences in the time varying demands placed on the data recording system. For instance, the minor perturbations of the *MatMul* code contrast sharply with the substantial slowdown of the *Place* code when data are recorded from eight nodes with HyperMon. Unlike matrix multiplication, which generates only a small number of instrumentation events, the cell placement code is highly dynamic and the variance in its event generation rate is high; see §2.5.5.

Table 2.3 further shows that the difference in total data volume between software and hardware data recording is large. With software event recording in individual node memories, each recorded event includes the associated data and a 16-bit timestamp delta. In contrast, the total data volume recorded using HyperMon is calculated from the total number of event frames transferred across the PBX interface. With software data recording, only one timestamp is assigned to a multiple byte event; HyperMon must timestamp each data nibble.

Clearly there is greater overhead with hardware data recording, but one might expect merging to increase the efficiency at higher event rates by amortizing timestamps across multiple event nibbles. Unfortunately, the software recording rates indicate that the potential for merging is small. Thus, we conclude that the large difference in the volume of recorded data reflects the fact that most event frames contain only one valid event data field.

Even when each node delays for twenty iterations of a null loop between transfer of event data nibbles, data overruns occur in real applications (e.g., the eight node *Place* execution). Although the sustained hardware recording rate of 2.5 Mbytes/second for this program (as extrapolated from the software recording rate) does not exceed the PBX bandwidth, the burst event rate is higher. In this case, the only alternative is to increase the interval between the output of successive event nibbles.

Dynamic Monitoring Requirements

As just noted, sustained recording rates do not reflect instantaneous demands on the monitoring system. Understanding the dynamics of event creation is important for two reasons. First, it suggests where buffers in HyperMon are most needed to ameliorate the effects of event data bursts. Second, it identifies those portions of a parallel computation where program execution might be most susceptible to performance perturbation from performance instrumentation.

For each of the four application programs of Table 2.2, we used HyperMon to record and compute statistics on the time varying rate of operating system event generation. For each application program, we recorded in the PBX node's memory the number of event frames and the elapsed time between the first and last frame for each group of event frames read.[19]

To generate Figures 2.13–2.16, the elapsed time for each program was divided into one hundred intervals of fixed size and the average event frame rate was computed for each interval. To show differing numbers of nodes on a single graph, we show normalized time intervals (i.e.,

[19]Recall that the HyperMon interface to the PBX I/O node includes a counter of the number of buffered event frames. This count defines the number of events read in each "group." Due to PBX node memory limits, only the first 100,000 event rate distribution samples were recorded.

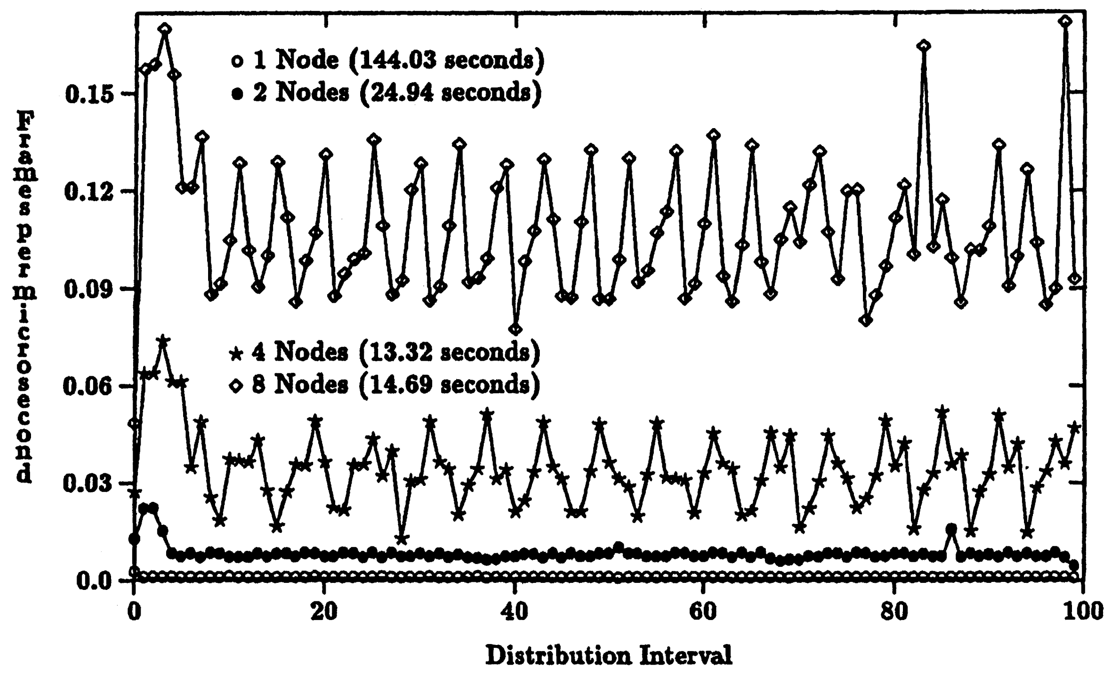


Figure 2.13: *Simplex* Event Frame Rate Distribution

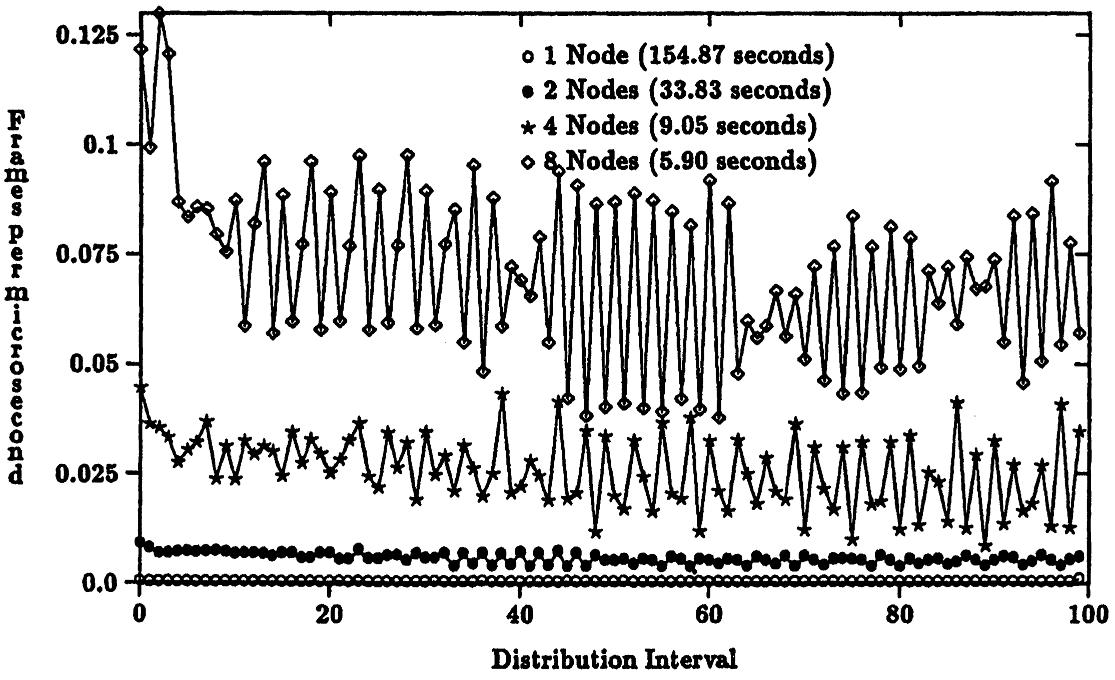


Figure 2.14: *Life* Event Frame Rate Distribution

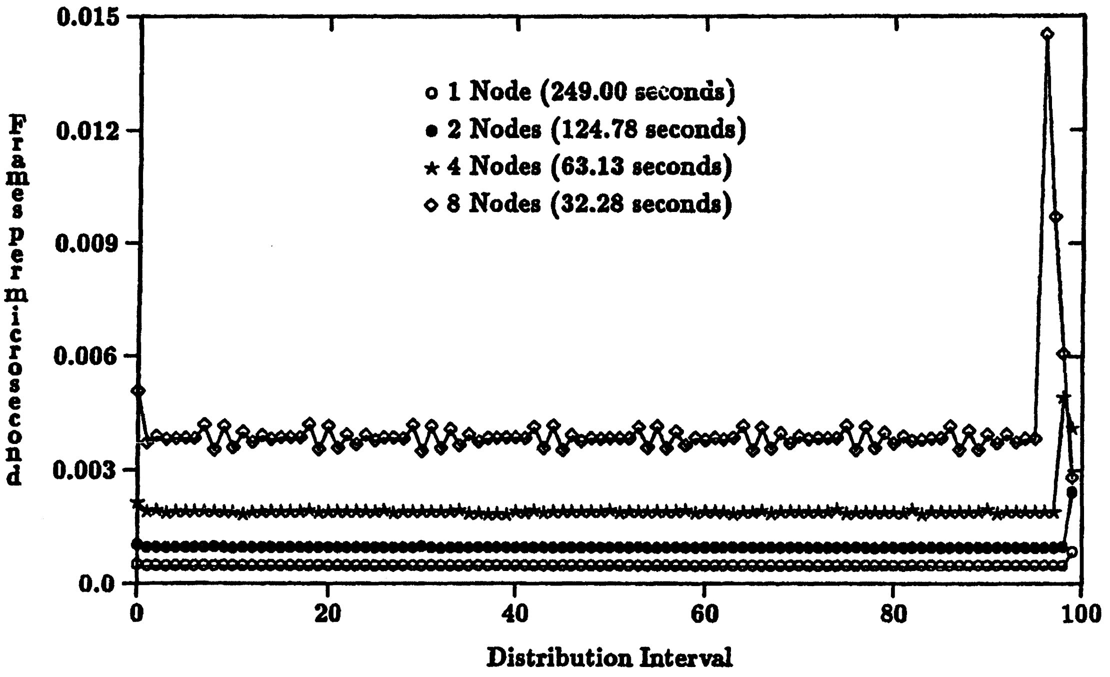


Figure 2.15: *MatMul* Event Frame Rate Distribution

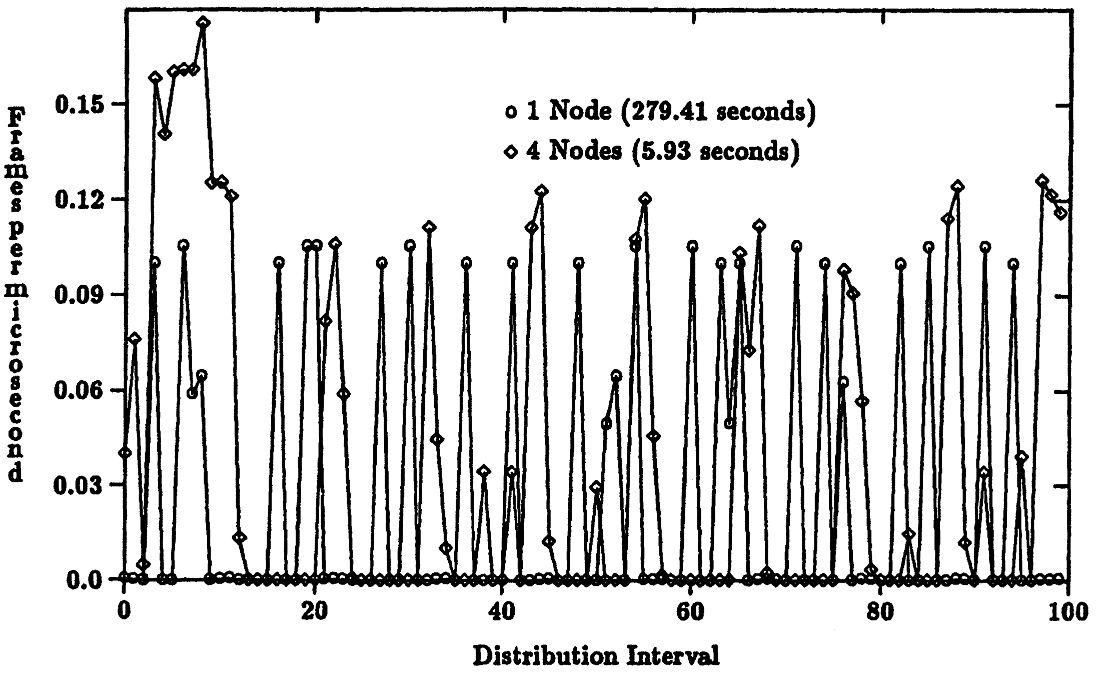


Figure 2.16: *Place* Event Frame Rate Distribution

for each number of nodes, an interval represents a different absolute amount of time). The total time range for each curve is shown in the legend.

Figure 2.13 shows the time varying event frame rate for the *Simplex* code. Clearly, the event frame rates follow a periodic pattern, and analysis of the code shows a regular cycle of computation and global data exchange [187]. Communication generates a burst of operating system instrumentation events (e.g., context switches, message buffering, and message transmission) [126], and this is reflected in the event frame rate. As the number of nodes increases, the ratio of communication to computation increases and the event frame rate increases commensurately.

Unlike the *Simplex* or *MatMul* codes, where the amount of computation on each node varies little during successive computation cycles, the *Life* code updates a grid of cells whose sparsity changes over time. Figure 2.14 shows that the data recording requirements of such codes can change substantially as the computation load balance changes.

The *MatMul* event frame rate distribution in Figure 2.15 reflects the simple structure of the application code. The computation first distributes the matrix to the nodes, where they compute independently until returning their partial results to the host. The initial matrix broadcast is not shown in Figure 2.15, but the transmission of sub-matrices to the host is clearly visible. Because no communication occurs during the computation phase, most recorded events are node time slice context switches. Finally, the event rates for *Place* application, shown in Figure 2.16, are random, bursty, and high. In §2.5.5, we describe the underlying reasons for this behavior and the implications for hardware event data recording.

As Figures 2.13–2.16 show, the dynamics of event frame rates are closely tied to application behavior and can vary widely across application types. This disparity in burst rates has important implications for capture hardware design, the subject of the next section.

2.5.5 Lessons Learned

The design of HyperMon was subject to the engineering constraints imposed by the iPSC/2 system: the 4-bit I/O event data interface, the physical separation of event regeneration from event capture, and the PBX I/O node interface. Although the larger overhead for recording event data via HyperMon was expected, and we knew that many applications exhibited cyclic communication behavior, we did not foresee all the implications of bursty event data rates.

The lesson regarding decreased execution time perturbations with hardware data recording is clear. External interfaces used to record event data via hardware should have sufficient bandwidth to avoid delaying the computation processors. Ideally, the access time to the interface should be no larger than that needed to write the event words to memory (i.e., hardware event recording should have less overhead than that for software buffer management and data recording).

Regarding bursty event data rates, further investigation of event data bursts using software event traces reveals significant variances in event data rates during the lifetime of most computations. Figure 2.17 shows the event data volume generated by our NX/2 operating system instrumentation [161] in one millisecond intervals for the *Place* application on four, eight and sixteen nodes. Although the average event data rates are 82 Kbytes/second, 276 Kbytes/second, and 629 Kbytes/second, respectively, event data bursts significantly exceed these rates. In particular, the data rate for the sixteen node *Place* execution can reach two to three Mbytes/second in twenty millisecond bursts. To support this type of software performance instrumentation, a hardware data recording system must be designed with sufficient buffer capacity to accommodate event data bursts. Analysis of software event traces can be instrumental in defining buffer requirements. At present, we are using the software traces as input to simulation models of monitor designs to understand dynamic buffering requirements.

An important decision in the HyperMon design was to treat each 4-bit event datum as a potentially unique event. This determined timestamp generation and motivated the notion of event frames to amortize timestamp overhead. In practice, our NX/2 operating system instrumentation produced logical events composed of multiple event data nibbles. Significant reductions in the volume of data recorded by HyperMon would have been possible had we chosen to timestamp larger data units (e.g., 32-bit quantities). In this case we would accumulate a 32-bit word on each 4-bit input port before storing it in the event data FIFO. The size of timestamped quantities should be chosen so that only a small fraction of the available bandwidth is lost. Ideally, there should be support for selective timestamping of event data such that timestamps are produced only when directed by the software.

The experiments conducted with the instrumented NX/2 operating system, described in §2.5.4.2, represent HyperMon stress tests. Clearly, there exists a spectrum of data recording and data analysis alternatives. No reduction of event data occurred in our experiments prior to

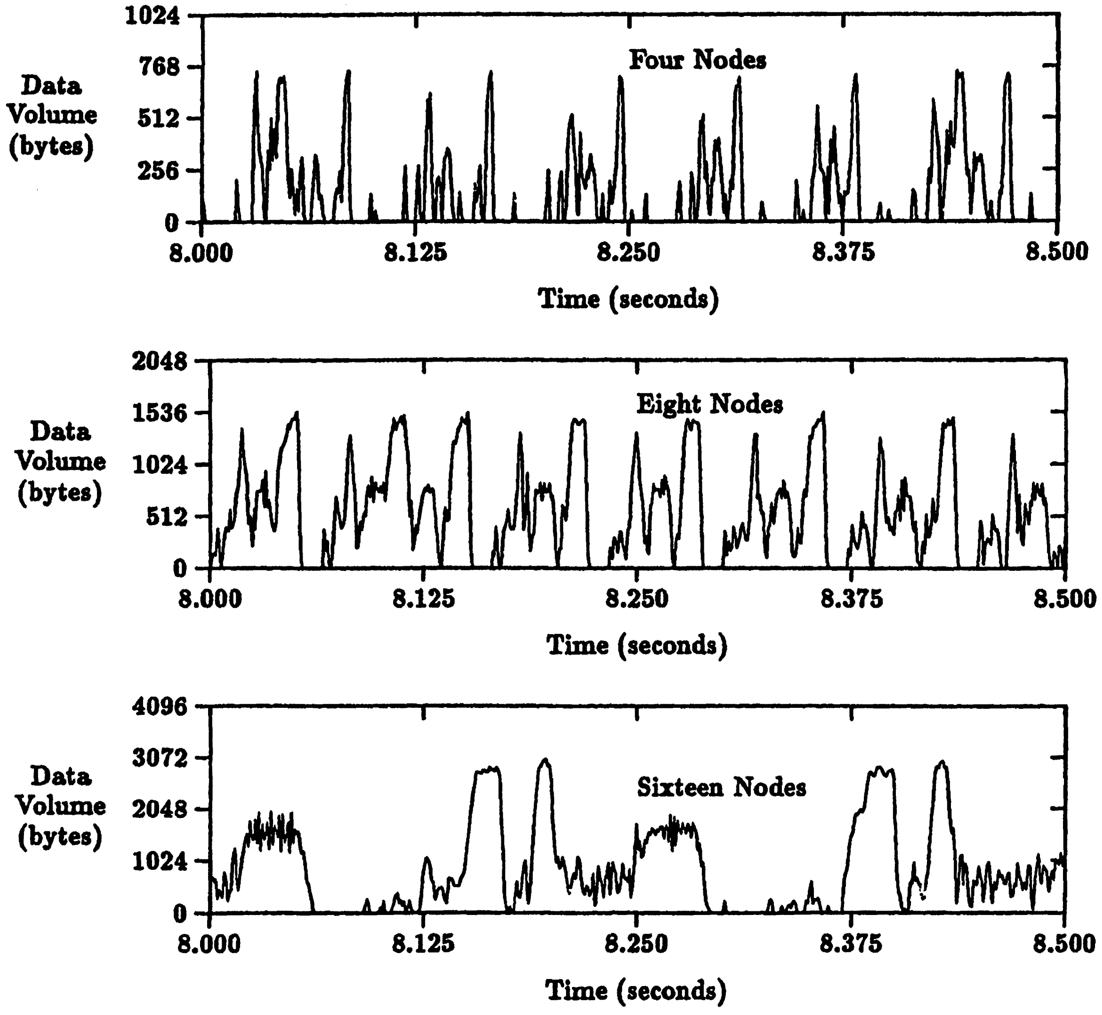


Figure 2.17: Place Event Data Volume (One Millisecond Intervals)

writing data to HyperMon. The use of parallel, on-the-fly data reduction, possibly in the form of periodic statistical summaries, would eliminate many of the problems encountered during our stress tests of HyperMon operation. Although improvements in the HyperMon design can extend its operational range, there are many performance experiments that can take advantage of the current prototype's real-time monitoring capabilities.

2.6 Comments

As technology makes possible the creation of ever more complex computing systems, which combine advances in architecture, hardware, system software, and applications algorithms to achieve high performance, we believe that the capture of detailed performance data will become increasingly important to both system designers and application software developers. Although the design of performance measurement systems is non-trivial and it is difficult determine monitoring requirements without some rudimentary understanding of the performance data bandwidth produced by a measured system, their implementation is not overly complex — the lack of sophisticated performance measurement facilities in commercial computer systems attests more to economic considerations rather than engineering.

Thus, in the future, one might imagine every computer system with a fully-integrated performance measurement environment supporting both timing and tracing measurements for post-mortem or real-time analysis. In particular, it is intriguing to consider the use of performance measurement tools for adaptive control of computer system operations. In this case, the performance measurements would be automatically analyzed at run-time and used to guide future execution decisions. Performance-guided adaptive control could have immediate application to problems such as load balancing and scheduling.

Chapter 3

Time-Based Performance Perturbation

Things are seldom what they seem,
Skim milk masquerades as cream.
— Gilbert and Sullivan, "H.M.S. Pinafore"

Many advances in modern science have been achieved through the systematic application of the scientific method. The experimental measurement of phenomena in nature to test the predictions of a model or validate a hypothesis is central to the advancement of operational theories of how systems function. Experimental scientists have long understood the relationship between the need to observe finer levels of operational behavior (e.g., to test the limits of a theoretical framework) and the technological capabilities of the measurement tools to deliver accurate observations. In physics, the Heisenberg uncertainty principle bounds the attainable measurement certainty.[2] Clearly, instrumentation and phenomenon must be commensurate to maintain instrumentation perturbations at acceptable levels and to achieve reliable observations.

The problems of uncertainty and instrumentation perturbation in computer system performance analysis are no less profound than in other experimental sciences. The terms "Heisenberg

[1]The results presented in this chapter appear in the paper "Performance Measurement Intrusion and Perturbation Analysis" submitted for publication to the IEEE Transactions on Parallel and Distributed Computing [128].

[2]For example, in measurements of a particle's position and momentum, the product of the measurement uncertainties is constant. Attempting to measure one quantity with greater certainty increases the uncertainty in measurements of the other.

Uncertainty" [112] and "probe effect" [62] have been used to describe the error introduced in the performance measurement due to a monitor's intrusion on computer system behavior. With the exception of passive hardware performance monitors, performance experiments rely on software instrumentation for performance data capture. Such instrumentation mandates a delicate balance between volume and accuracy.[3] Excessive instrumentation perturbs the measured system; limited instrumentation reduces measurement detail — system behavior must be inferred from insufficient data. Simply put, performance instrumentation manifests an *Instrumentation Uncertainty Principle*:

- Instrumentation perturbs the system state.
- Execution phenomena and instrumentation are coupled logically.
- Volume and accuracy are antithetical.

The primary source of instrumentation perturbations is execution of additional instructions. However, ancillary perturbations can result from disabled compiler optimizations and additional operating system overhead. These perturbations manifest themselves in several ways: execution slowdown, changes in memory reference patterns, event reordering, and even register interlock stalls. From a performance evaluation perspective, instrumentation perturbations must be balanced against the need for detailed performance data. Regrettably, there are no formal models of performance perturbation that would permit quantitative evaluation given instrumentation costs, measured event frequency, and desired instrumentation detail. Given the lack of models and the potential dangers of excessive instrumentation, detailed performance measurements, mainly in the form of software event traces, often are rejected for fear of corrupting the data (i.e., a small volume of accurate, though incomplete, instrumentation data is preferred). We hypothesize that this restriction is, in many cases, unduly pessimistic.

Our approach to understanding the problem of performance perturbation involves both the creation of perturbation models and the testing of those models through empirical studies. The perturbation models we developed are based on timing and event analysis. Time-based

[3] In contrast to other experimental disciplines, computer systems instrumentation does permit the epistemological trickery of declaring instrumentation part of the system. The perturbation is then, *ipso facto*, null. However, users of computer systems, particular those of high-performance machines, are rarely willing to sacrifice performance for permanently enabled instrumentation. Thus, our analysis here and in Chapters 4 and 5 excludes such possibilities.

perturbation models attempt to recover accurate timing of trace events from knowledge of instrumentation overhead, assuming event independence. Event-based perturbation models focus on removing the effects of instrumentation on the ordering of events in parallel execution. For both time-based and event-based perturbation analysis, we conducted a series of instrumentation experiments to determine the validity of the models in an actual execution context. The results of these experiments suggest that a systematic applications of performance perturbation analysis techniques will allow more detailed, accurate instrumentation than traditionally believed possible.

This chapter concentrates on time-based perturbation analysis. Depending on the characteristics of the execution environment, instrumentation perturbation manifests itself in different ways. In §3.1, we describe an example execution environment, the Alliant FX/80, a shared memory, vector multiprocessor, which embodies a variety of execution modes. In §3.2, we discuss the range of possible instrumentation timing perturbations on the Alliant FX/80. In §3.3, we develop several time-based models of performance perturbation that permit removal of perturbations from application program traces. The trace instrumentation support for the FX/80 system used in the empirical testing of the perturbation models is described §3.4. In §3.5 and §3.6, we validate the models using experimental data obtained from execution of the Livermore Loops on the Alliant FX/80. Finally, §3.7 discusses the limits of time-based perturbation analysis and illustrate the need for the event-based perturbation models of Chapters 4 and 5.

3.1 Experimental Environment

The Alliant FX/80 contains a combination of hardware and software features that makes it an interesting system for characterizing the types of perturbations that might occur in practice. As such, it offers a rich environment for experimental perturbation analysis. Below, we briefly describe the FX/80 architecture; see Figure 3.1.

The Alliant FX/80 consists of up to eight computational elements (CEs), each containing a vector processor [5, 150]. The CEs are connected via a concurrency control bus that permits dispatching of small computation granules to cooperating CEs. Using this bus, parallel loop iterations can be directly allocated to the CEs through a hardware self-scheduling mechanism.

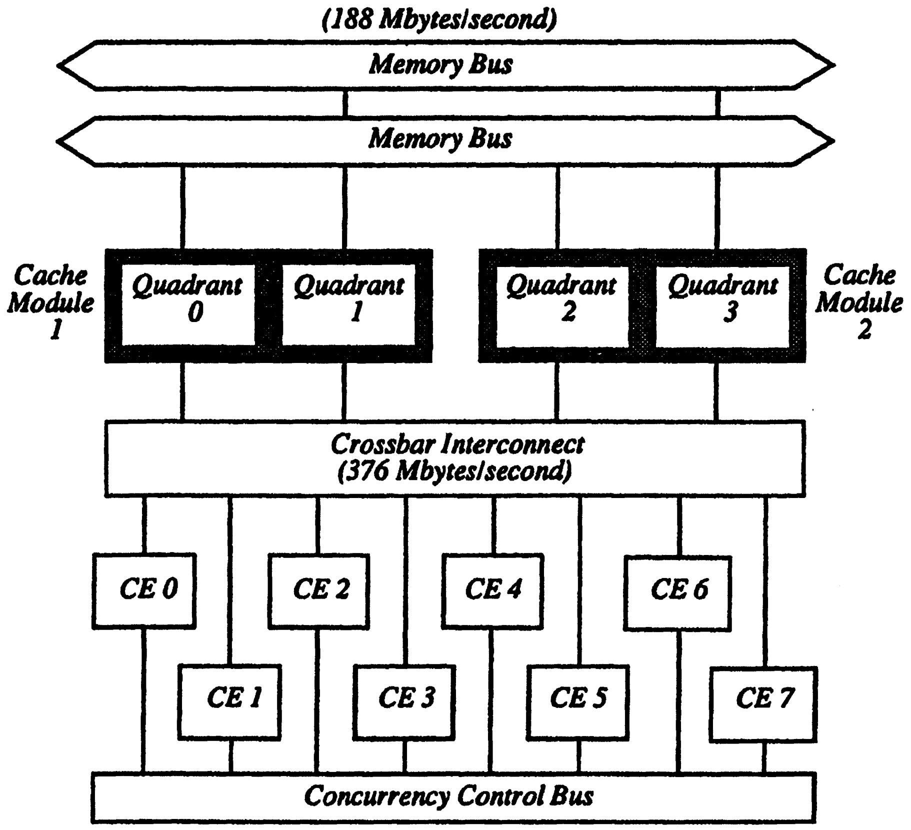


Figure 3.1: Alliant FX/80 Architecture

The memory system of the Alliant FX/80 combines parallel data access with a hierarchical memory structure. It is organized as three levels: a large main memory, a 512K byte cache shared by all CEs, and scalar and vector registers private to each CE. Each vector register contains 32, double precision (64-bit) words and is accessed by each CE's vector processing unit. The 64K word, write-back cache contains four banks that are connected to the eight CE's via a crossbar switch. The cache can service up to eight simultaneous accesses per cycle. The cache and the four-way interleaved main memory are connected via a main memory bus with a peak transfer rate of four words per cycle. Therefore, the peak bandwidth between main memory and the CEs is half that between the cache and the CEs.

The CE instruction set is a variation of the Motorola 68020 with certain extensions (e.g., vector and concurrency instructions). The Alliant vector instructions are of two types:

- *Internal*: all operands are contained in vector and scalar registers
- *External*: one operand involves a memory request

Within each type, most vector instructions have similar timing behavior, typically differing only in their startup time. Because the internal, register-register instructions do not depend on conditions extrinsic to each CE, their timings are deterministic. In contrast, the timing behavior of external vector instructions depends on memory activity. Access contention, either from the CE's previous requests or from requests issued by other CEs, and data location, either cache or main memory, both contribute to delays.

The memory hierarchy, multiple CEs, vector memory operations, and the concurrent execution modes all make the Alliant FX/80 a complex experimental environment. Successful hypothesis validation in this environment would provide strong evidence that our timing perturbation models are applicable to parallel systems with simpler execution environments (e.g., the multiple processor Cray X-MP [29] or the distributed memory Intel iPSC/2 [10]).

3.2 Performance Perturbation

The number and complexity of the Alliant FX/80's execution modes, and those of other high-performance computer systems, are equaled in both number and complexity by perturbation mechanisms for performance instrumentation. Unfortunately, perturbations are not additive, nor can perturbation magnitudes easily be deduced from measurement data without knowledge of perturbation mechanisms. Thus, understanding these perturbation mechanisms is a prerequisite for analysis of experimental performance data and application of sequential and parallel performance perturbation models. Because the possible perturbations during sequential execution are but a subset of those possible during parallel execution, we begin with an analysis of the former.

3.2.1 Sequential Perturbations

Although the range of possible instrumentation perturbations depends on the complexity of the underlying architecture and system software, the single stream of control flow in sequential programs localizes most perturbations about the instrumentation point. The localization of perturbation effects means that the computation's performance behavior is only affected within a local region of the instrumentation. Although the timing error introduced by the perturbation accumulates during the performance measurement, if the perturbations are assumed local, we can empirically characterize different instrumentation perturbations in isolation and apply models that remove the timing errors at the instrumentation source; see §3.3. Below, we discuss the possible perturbations in the context of our example execution environment, the Alliant FX/80. Many of the issues are equally relevant in other sequential execution environments from complex RISC-based platforms, such as those built with the Intel i860 microprocessor, to supercomputer systems, such as the Cray X-MP, the Cray Y-MP and the Cray 2.

As discussed in §3.1, the pipelined Alliant FX/80 processors are connected to a complex hierarchy of registers, cache, and primary memory. A sophisticated compiler generates code to maximize access frequency to the smaller, faster components of this hierarchy. For instance, at the lowest level of the memory hierarchy, pipelined register dependencies arise when an instruction accesses a register whose value has not yet been produced by a previous instruction. Although these dependencies stall the processor or functional unit until the requisite value is produced, optimizing compilers can reduce the number of stalls by judicious register allocation and instruction scheduling.

At this level, instrumentation perturbations need not increase execution time; instrumentation can both add and remove register dependencies. The former can occur when the prologue of the instrumentation code uses registers that have STORE operations pending. Typically, most instrumentation points first save the active registers on the local stack, generate the desired performance data, and then restore the active registers. Because most registers are addressed during save and restore, dependencies between instructions just before or after the instrumentation point are highly probable. Conversely, register dependencies can be removed by inserting instrumentation between two instructions that have an existing register dependency. Execution of the intervening instrumentation code will decouple the dependent instructions.

Even if instrumentation does not perturb register dependences, it can change both the spatial and temporal patterns of cache and memory references, with both positive and negative effects. Consider an application code fragment that contains a loop with a high density of memory references. If the instrumentation generates large volumes of data, the resulting cache and memory traffic may flush most application data from the cache. When application execution resumes, the data cache will be reloaded by a sequence of cache misses. The overhead for this "cold start" [178] may be comparable to that for a context switch.

If an application's memory references generate bank conflicts in the interleaved memory [179], instrumenting the code will distribute the application memory references across a larger interval of time, decreasing the memory access time as perceived by the application code. During execution of the instrumentation code, all outstanding memory references, including those with bank conflict stalls, will compete. Because time spent in instrumentation code is not charged to the application, the apparent memory access time decreases. The converse is also true. If application data access patterns are structured to minimize bank conflicts, inserted instrumentation code can disrupt the access pattern, perhaps creating a degraded, steady state memory access pattern with bank conflicts.

As the magnitude of direct instrumentation perturbation grows (e.g., added register dependencies or modified memory reference patterns), the probability of indirect perturbation grows proportionately. For example, the probability of context switches is higher for instrumented applications because they execute longer. Although the cost of these context switches can be measured by instrumenting the operating system, identifying "real" and induced context switches is non-trivial.

Perhaps more subtle than either direct perturbations or induced context switches are changes in the application program's executed code. Although the object code for a program with source code instrumentation (i.e., instrumentation inserted in the application source code) clearly differs from the object code for the same program without instrumentation, removing the instrumentation from the object code does not result in identical codes. For example, source code instrumentation can prevent certain optimizations and can change register allocation.[4] For vectorizing and parallelizing compilers, the potential for code perturbation is greater — inserting instrumentation in a vectorizable loop can easily prevent vectorization.

3.2.2 Parallel Perturbations

As noted earlier, the class of possible perturbations during sequential execution is but a subset of those possible for parallel programs. Parallel programs often stress their execution environment. For example, a single processor of the Alliant FX/80 cannot generate contention at the shared cache, nor can it saturate the memory bus. In concurrent mode, however, both the cache and memory bus can be performance bottlenecks [64]. This also is true of other parallel machines with high-performance memory systems [179]. Instrumentation that causes only small perturbations in sequential mode (e.g., memory traffic to save instrumentation data) can create unacceptable perturbations in concurrent mode, including perturbations of the task execution order.

With the exception of asynchronous input-output, the execution states of sequential programs form a total order. Sequential trace instrumentation may change event times, but it rarely changes the event order. In contrast, the states of parallel programs inherently form a partial order. Consequently, the reordering of instrumented states is not only possible but likely. The number of reordered events depends on both the task scheduling algorithm and the frequency of parallel task synchronization.

If parallel tasks are assigned to processors statically (i.e., the mapping of tasks to processors is known *a priori*), the sequence of tasks executed by each processor cannot change as a consequence of instrumentation. However, the lengths of the respective tasks can change, and this may reorder events across tasks. Consider a parallel program with two tasks, where task A reaches a synchronization point before task B. If the instrumentation overhead for task

[4]We return to the interaction of compiler and instrumentation in §3.4.

A exceeds that for task B, the order that the tasks reach the synchronization point will be reversed, and the recorded event order will differ from nominal.[5]

In general, some tasks are dynamically assigned to processors. Indeed, the Alliant FX/80 permits dynamic assignment of single loop iterations. If instrumentation changes task execution times by disproportional, data dependent amounts, the sequence of tasks executed by each processor, and the order of inter-task events cannot be predicted without knowledge of the task scheduling algorithm.

Given the diversity and complexity of possible instrumentation perturbations, both direct and indirect, software instrumentation must be designed to ameliorate or eliminate as many perturbations as possible. However, timing perturbations cannot be removed solely by efficient instrumentation, and perturbation analysis must be applied to resolve timing errors. An instrumentation design for the Alliant FX/80 is the subject of §3.4. The following section develops a theory of time-based performance perturbation and constructs perturbation models for removing timing perturbations during sequential and concurrent execution.

3.3 Timing Perturbation Models

Models to capture and remove timing perturbations due to instrumentation must be based on a particular instrumentation approach. Because tracing is the most general form of instrumentation, allowing both static and dynamic analysis, we derive time-based perturbation models for trace instrumentation. Given an understanding of possible performance instrumentation perturbations (see §3.2), measures of *in vitro* trace instrumentation costs in an execution environment (see §3.4), and an instrumentation trace, our goal for perturbation analysis is to recover the "true" trace of events as they would have been generated during an execution with instrumentation. There are two phases in this perturbation analysis:

- **Execution Timing Analysis** – Given the measured costs of instrumentation, adjust the trace event times to remove these perturbations.

[5]Different parallel execution orders can occur, even without instrumentation, due to asynchronous task operation. The reproducibility of parallel executions is another aspect of uncertainty separate from instrumentation. For our purposes, we assume the differences between possible event orders from executions without instrumentation is minimal.

- **Event Trace Analysis** – Given instrumentation perturbations that can reorder trace events, adjust the event sequence based on knowledge of event dependencies, maintaining causality.

In both phases, models are needed that describe observed behavior as a perturbation of true behavior. For timing analysis, one must approximate true times of event occurrence, either for each trace event or for the total execution time. That is, the timing model must describe how the perturbations affect measured execution times. Event analysis models are more difficult; program or system semantic information is needed to determine if the relative event order is incorrect and, if so, generate a better approximation to the true order. We restrict our attention here to two classes of timing analysis models; event-based perturbation modeling is the subject of Chapters 4 and 5. The first, simpler model predicts total execution time given trace data. The second adjusts the times of individual trace events. For both models, we discuss, where appropriate, the perturbations that might be removed by appropriate event analysis models. We begin, however, with a formal description of the instrumentation problem.

3.3.1 Definitions

Given a program P composed of a sequence of statements $S_1, S_2, \ldots, S_n$ and a set of instrumentation points $I_1, I_2, \ldots, I_n$, an instrumentation of P is defined as

$$\mathcal{I}(P) = I_1, S_1, I_2, S_2, \ldots, I_n, S_n \ ,$$

where some I_j may be null (i.e., no instrumentation). Thus, we define instrumentation on a statement basis, where an event represents the execution of a statement.

A *logical event trace*, τ, is a time-ordered sequence of events $e_1, \ldots, e_m$ where each e_i is of the form $\{t(e_i), eid_i\}$, eid_i is the event identifier for the i^{th} event representing the statement S_{eid_i} in the program, and $t(e_i)$ is the time when the event occurred. If the program is instrumented, we use the notation τ_m to denote a *measured event trace.* Because a program can have both sequential and concurrent components, we define the sequential event trace, τ^s (τ_m^s), as the subsequence of events $e_p, e_q, \ldots, e_r$ generated in sequential mode.[6] Similarly, the concurrent event trace for processor i, τ^i (τ_m^i), is the sub-sequence of events $e_p^i, e_q^i, \ldots, e_r^i$ executed in concurrent mode on processor i.

[6]Because sequential executions form a total ordering of events, $\tau^s = \tau_m^s$.

Given these definitions and letting $T(S_{eid_i})$ be the *true* execution time of statement S_{eid_i}, the total execution time of a sequential program P is

$$T^s(P) = \sum_{e_i \in \tau^s} T(S_{eid_i}) .$$

The *measured* program execution time of an instrumentation of P is

$$T_m^s(\mathcal{I}(P)) = \sum_{e_i \in \tau_m^s} [T(S_{eid_i}) \oplus T(I_{eid_i})] ,$$

where $T(I_{eid_i})$ is the true execution time overhead of the instrumentation point I_{eid_i}. The coupling of execution times for program statements and instrumentation, represented by $\oplus$, denotes perturbations not included in individual instrumentation and statement timings (e.g., the disruption of memory reference patterns).

For concurrent execution time one must determine the critical path during concurrent computation. Let $\tau^s = e_p, e_q, \ldots, e_r$ represent the locical trace of sequential events and $\tau^{cp} = e_s, e_t, \ldots, e_u$ the logical trace of concurrent events along the critical path, respectively. The total true execution time of a concurrent program P is

$$T^c(P) = \sum_{e_i \in \tau^s} T(S_{eid_i}) + \sum_{e_j \in \tau^{cp}} T(S_{eid_j}) .$$

Similarly, the measured program execution time of an instrumentation of P is:

$$T_m^c(\mathcal{I}(P)) = \sum_{e_i \in \tau_m^s} [T(S_{eid_i}) \oplus T(I_{eid_i})] + \sum_{e_j \in \tau_m^{cp}} [T(S_{eid_j}) \oplus T(I_{eid_j})] .$$

where $\tau_m^{cp} = e_x, e_y, \ldots, e_z$ represents the critical path of concurrent events in the instrumented program. Unfortunately, the concurrent event sequence identified as the critical path in $T^c(P)$, τ^{cp}, may differ from that for $T_m^c(\mathcal{I}(P))$, τ_m^{cp}; this is the motivation for event trace analysis models of Chapter 4 and 5.

From these execution time definitions one can define a series of instrumentation perturbation metrics. We will use $T(P)$ and $T_m(\mathcal{I}(P))$ to represent true and measured execution times for both sequential and concurrent timing measurements. The simplest metrics, absolute and relative error for measured execution time, are defined in the standard way. The absolute error AE is

$$AE = T_m(\mathcal{I}(P)) - T(P) , \tag{3.1}$$

and the relative error RE is

$$RE = \frac{T_m(\mathcal{I}(P)) - T(P)}{T(P)} . \tag{3.2}$$

The direct perturbation DP

$$DP \;=\; DP^s + DP^c \tag{3.3}$$

is the increased execution time directly caused by instrumentation; its sequential and concurrent components, DP^s and DP^c, respectively, are

$$DP^s = \sum_{e_i \in \tau_m^s} T(I_{eid_i})$$

and

$$DP^c = \sum_{e_j \in \tau_m^{cp}} T(I_{eid_j}) .$$

Although (3.1) and (3.2) estimate the instrumentation perturbation, they do not estimate actual execution time from trace data. The *approximate* execution time, $T_a(P)$, is the difference between the measured execution time and direct perturbation,

$$T_a(P) = T_m(\mathcal{I}(P)) - DP . \tag{3.4}$$

That is, $T_a(P)$ is the approximated execution time after applying a timing analysis model that includes only direct perturbations.

Finally, the relative approximate error, RAE, estimates the accuracy of a timing analysis model (i.e., how accurately one can determine actual execution time from an instrumentation trace):[7]

$$RAE = \frac{T_a(P) - T(P)}{T(P)} = \frac{T_m(\mathcal{I}(P)) - DP - T(P)}{T(P)} . \tag{3.5}$$

3.3.2 Execution Time Analysis

The test of a timing analysis model's veracity is its ability to predict a program's actual execution time given an instrumentation trace. In the remainder of this section, we present execution time models for both sequential and concurrent execution.

[7] In the remainder of the paper we will use the measure $\frac{T_a(P)}{T(P)} = RAE + 1$ to express the accuracy of the model.

3.3.2.1 Sequential Execution

During sequential execution, the principal perturbation is direct — execution of additional instrumentation instructions. Furthermore, instrumentation does not perturb the total order of program events. Thus, our sequential perturbation model assumes that *all* perturbations are direct (i.e., $AE = DP$) and that the cost for instrumentation is decoupled from statement execution. Simply put, the model approximates actual execution time by the difference between measured execution time and all direct instrumentation costs. More formally, the model's assumptions imply the following:

1. The actual cost $T(I_{eid_i})$ for each instrumentation point I_i is approximated by a constant α.
2. $DP = \sum_{e_i \in \tau^*} T(I_{eid_i}) = \alpha N$, where N is the number of instrumentation points.
3. $T_a(P) = T_m(\mathcal{I}(P)) - DP = T_m(\mathcal{I}(P)) - \alpha N$.
4. $RAE = \frac{(T_a(P) - T(P))}{T(P)} \simeq \frac{(T_m(\mathcal{I}(P)) - \alpha N)}{T(P)}$.[8]

As the approximate equality above suggests, the accuracy of our assumption depends on the interaction of instrumentation perturbations and statement execution. With source code instrumentation, compiler register optimizations can invalidate the assumption of a constant instrumentation perturbation; see §3.4. To remove these indirect perturbations, we first apply the simple model above to approximate the actual execution time $T_a^1(P)$. This prediction reflects direct perturbations from instrumentation execution. We then assume the instrumentation code is removed from the instrumented program's assembly code (or object code) and measure the execution time of the resulting program, producing a second execution time estimate $T_a^2(P)$ that measures only the code perturbations. The final execution time approximation in the case of source code instrumentation is given by

$$T_a^{source}(P) \simeq T_a^1(P) * \frac{T(P)}{T_a^2(P)} \, . \tag{3.6}$$

The desired approximation in this case would be based on a non-uniform perturbation model (i.e., one that considered the effects of each individual instrumentation instance). However, the

[8]The astute reader may notice that we are assuming we know the actual execution time of P, $T(P)$, in the calculation of the relative approximate error. In actuality, $T(P)$ must be determined from measurement. This will become clear in §3.5.

number of different cases to consider can become large and requires an analysis of the differences in the code generated with and without instrumentation. In light of these complications, the scaled approximation above is reasonable when source code instrumentation is necessary.

3.3.2.2 Concurrent Execution

During concurrent execution, multiple threads of control may simultaneously reach trace instrumentation points. Intuitively, a critical path analysis would identify the set of instrumentation points needed to compute total execution time [198]. Unfortunately, perturbation mechanisms such as those described in §3.2.2 make this difficult. Events can be reordered, and the critical path identified from the instrumentation trace may not be the critical path in the real code.

Without resorting to the event analysis models of Chapters 4 and 5, we can assume that events are *not* reordered and that the concurrent thread with the longest execution time (after direct perturbations have been removed) is the critical path. If most threads execute similar instruction streams (i.e., there is little data dependent code), this assumption is accurate.[9] Like the sequential execution time model, our base assumption implies the following:

1. The actual cost $T(I_{eid_i})$ for each instrumentation point I_i is approximated by a constant α.

2. $DP = DP^s + DP^c_{max} = DP^s + \alpha N_{max}$, where
   - $(T_m(\mathcal{I}(P_{max})) - \alpha|\tau^{max}|) \geq (T_m(\mathcal{I}(P_i)) - \alpha|\tau^i|) \;\; \forall i \;\; 0 \leq i \leq p$,
   - p is the number of processors,
   - $T_m(\mathcal{I}(P_i))$ is the measured concurrent execution time on processor i,
   - $N_{max} = |\tau^{max}|$ is the number of instrumentation events in trace τ^{max}.

3. $T_a(P) = T_m(\mathcal{I}(P)) - DP^c = T_m(\mathcal{I}(P)) - DP^s + \alpha N_{max}$.

4. $RAE = \frac{(T_a(P) - T(P))}{T(P)} \simeq \frac{(T_m(\mathcal{I}(P)) - DP^s + \alpha N_{max})}{T(P)}$.

In simpler terms, the concurrent perturbation model chooses as the critical path the sequential execution path plus the execution path along the concurrent thread that has the greatest accumulated execution time after the direct perturbation has been removed; see Figure 3.2.

[9] If not, an event analysis model is needed; see Chapters 4 and 5. However, in §3.6, we show that timing analysis alone can yield significant insight in many practical cases.

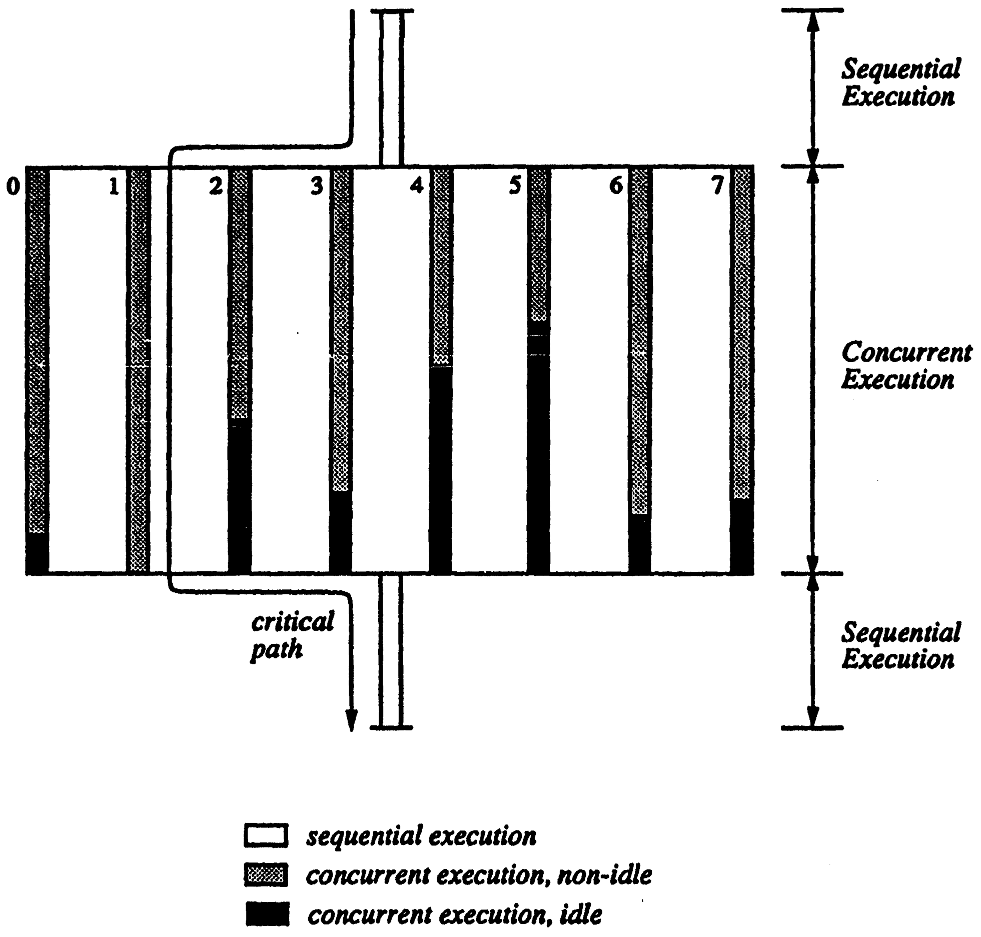


Figure 3.2: Critical Concurrent Path Selection

3.3.3 Event Trace Timing Analysis

To recover the true sequence of trace event times, one must consider not only the total execution time but all possible inter-event dependencies and associated perturbations. Even given careful analysis and a predictive model, one cannot directly determine the accuracy of the predicted event times.[10] Instead, one must infer the stability of the event timing model by comparing its trace predictions with varying levels of trace instrumentation. As with execution time models, we begin with the simpler, sequential case.

3.3.3.1 Sequential Trace Timing Analysis

Each trace event identifies a unique spatial and temporal state (i.e., a code location at a specified time). In a sequential trace, each event is perturbed by the instrumentation for all previous events. Thus, we iteratively calculate each event time, given the perturbations of previous events.

For a trace τ^s of sequential events $e_1, \ldots, e_m$ where each e_i is of the form $\{t_m(e_i), eid_i\}$, we approximate the actual time of event e_i by $t_a(e_i)$,

$$t_a(e_i) = t_m(e_i) - (i-1)\alpha \; , \tag{3.7}$$

where α is the mean time for each trace instrumentation point and $t_m(e_i)$ is the measured time of occurrence of e_i from a trace of an instrumented execution.[11]

Unfortunately, it is possible that two events e_i and e_j occur so close together that $t_m(e_i) \leq t_m(e_j)$ but $t_a(e_i) > t_a(e_j)$. Simply put, software instrumentation may be unable to separate closely spaced events because the hardware timer lacks the resolution to measure instrumentation overhead and elapsed time with sufficient accuracy. For example, on the Alliant FX/80, the timer resolution is 10 microseconds, but the machine cycle time is 170 nanoseconds. In a 10 microsecond window, several events may occur. Although, for sequential programs, these events form a total order, their times have a 10 microsecond uncertainty. As a consequence, we must assume simultaneity for all events whose estimated times differ by less than 10 microseconds.

[10]If this were possible, event tracing would be unnecessary.

[11]For example, if e_{101} is (854, 10) in the trace, the approximation of the actual time of occurrence would be $t_a(e_{101}) = 854 - 10.9 * 100 = 745$, if $\alpha = 10.9$.

3.3.3.2 Concurrent Trace Timing Analysis

Approximating event times for concurrent traces is more difficult than for sequential traces. The perturbation of each event depends on the perturbation of all events on the critical path to the event. In the worst case, a complete characterization of the execution dependencies between concurrent threads of execution is required. To simplify analysis, we assume that events on separate concurrent threads are independent and that the program contains only a single level of fork-join concurrency. With the requisite event analysis model, these assumptions can be relaxed.

Given a trace τ_i^c of concurrent events $e_1^i, \ldots, e_n^i$ for each concurrent thread i, and a trace τ^s of sequential events $e_1^s, \ldots, e_m^s$, we approximate the actual time of a concurrent event e_k^i as follows.

1. If e_k^i is the first concurrent event after a sequential event e_j^s in the time ordered trace, then
$$t_a(e_k^i) = t_m(e_k^i) - t_m(e_j^s) + t_a(e_j^s) .$$
We use the measured and approximated times of the last sequential event occurrence as the *time basis* for computing the execution time of the first concurrent event of a concurrent phase of computation.

2. If e_k^i immediately follows a concurrent event in the trace on thread i, then
$$t_a(e_k^i) = t_m(e_k^i) - t_m(e_j^s) + t_a(e_j^s) - \alpha c_k^i ,$$
where c_k^i is the number of events in concurrent thread i after the last sequential event e_j^s in the trace. Along a sequence of concurrent events, we use the last sequential event as the time basis for approximating the time of occurrence of e_k^i, but the direct perturbation along thread i also is removed.

We approximate the actual time of a sequential event e_k^s as follows.

1. If e_k^s is the first sequential event in the trace after the last concurrent event from a concurrent phase of computation, then
$$t_a(e_k^s) = t_m(e_k^s) - t_m(e_j^i) + t_a(e_j^i) ,$$

where $t_a(e^i_j) > t_a(e^m_n)$ for all n and m such that e^j_i and e^m_n appear before e^s_k in the trace. It is here that we determine the critical concurrent path in the instrumented execution. The concurrent event appearing before e^s_k in the trace with the greatest approximated timestamp is used as the time basis to approximate the sequential event occurrence.

2. If e^s_k follows a sequential event in the trace, then

$$t_a(e^s_k) = t_m(e^s_k) - t_m(e^i_j) + t_a(e^i_j) - \alpha c^s_k \ ,$$

where c^s_k is the number of events that have occurred in sequential mode since the last approximated concurrent event e^i_j in the trace (or the beginning of the trace). Along a sequence of sequential events, we again use the last approximated concurrent event as the time basis for approximating the sequential event occurrence. Additionally, we remove the direct sequential perturbation.

3.4 Performance Instrumentation

To validate our time-based pertubation models against the Alliant FX/80 execution environment, we constructed a performance instrumentation facility for the target machine. Below, we describe the performance instrumentation implementation for the FX/80 and its instrumentation overhead, both in execution time and memory bandwidth. Measures of *in vitro* instrumentation costs are necessary for practical application of the perturbation models.

For our experiments on the Alliant FX/80, we constructed a tracing library that includes static trace buffer declarations and a set of assembly language tracing routines. The library maintains a trace buffer for each of the eight potentially concurrent threads of execution on the FX/80 and one additional buffer for events that occur during sequential execution. Each invocation of the tracing routine records a 32-bit timestamp and 32-bit event identifier in the appropriate trace buffer.

Trace instrumentation can be inserted at either source or assembly code levels. Although easier to automate, source instrumentation can create register allocation and access problems.[12] For example, the procedure calling convention on the Alliant FX/80 has the caller save registers. As a consequence, the invocation overhead for a source code trace event depends on the number

[12]These register management problems are in addition to those discussed in §3.2.1.

of registers whose values must be retained across the instrumentation code boundaries. Because the caller does not know what registers are used by the tracing library, it must save all active registers. The magnitude of this register management overhead depends on the current register state. The alternative, saving registers in the callee, fixes the overhead — only the registers used by the trace library need be saved. The disadvantage is that registers may be saved unnecessarily.

Not only does source code instrumentation require register management, it may inhibit or force different code optimizations. The latter depend on both the language and the code generator. On the Alliant FX/80, the C compiler does not restructure the source code prior to code generation and optimization, nor does it generate vector or parallel code. Inspection of the generated assembly code after source instrumentation shows only simple optimizations; source statement boundaries were clearly delimited. As an artifact of the C compiler's limited optimization, the overhead for source instrumentation was nearly invariant across instrumentation points.

Unlike source instrumentation in C, the overhead for a Fortran source instrumentation point is context dependent. The preprocessor of Alliant Fortran compiler generates restructured Fortran source based on detected data dependencies. Although the resulting Fortran is functionally equivalent, statements can be reordered, loops can be fused or distributed, and new variables can be created [197]. Inserting trace instrumentation can inhibit a subset of these transformations.

As an example of the interaction of instrumentation and source restructuring, Table 3.1 shows the generated code for the following code drawn from the Livermore loops [131], both with and without source code instrumentation.

```
      DO 10 k=1, n
C         instrumentation point
          X(k) = Q + Y(k)*(R*ZX(k+10) + T*ZX(k+11))
C         instrumentation point
10    CONTINUE
```

When no source code instrumentation is included, the Alliant Fortran compiler generates code to pre-load the scalar operands (Q, R, and T) in floating point registers (fp7, fp5, and fp6,

Expression	Normal	Instrumented
a = R*ZX(k+10)	fmuls zx+10[d7],fp5,fp0	fmoves zx+10[d7],fp1 fmuls r,fp1,fp0
b = T*ZX(k+11)	fmuls zx+11[d7],fp7,fp1	fmoves zx+11[d7],fp3 fmuls t,fp3,fp2
c = a + b	fadds fp1,fp0,fp0	fadds fp2,fp0,fp0
d = c*Y(k)	fmuls y[d7],fp0,fp2	fmuls y[d7],fp0,fp4
X(k) = Q + d	fadds fp6,fp2,fp2 fmoves fp2,x[d7]	fadds q,fp4,fp5 fmoves fp5,x[d7]

Table 3.1: Fortran Perturbations with Source Code Instrumentation

respectively) before the beginning of loop execution; these instructions are not shown in Table 3.1. When instrumentation is added to the source code, the Fortran optimizer recognized that loading the registers during each loop iteration would eliminate the register save and restore overhead for the instrumentation call. Thus, the instrumented loop iterations fetch the scalar operands from memory. As a consequence, the first two floating point multiplications in the code without instrumentation become a sequence of two floating point register loads followed by two floating point multiplications in the instrumented code.

Given the perturbation variance for Fortran source instrumentation and the desire to post-process the trace data using models of performance instrumentation with known costs, we opted to instrument all applications at the assembly code level. In this mode, the compiler generates code without knowledge of the instrumentation, the fixed instrumentation is inserted in the resulting code, and the code perturbation is context independent.[13] With this approach, an *a priori* measurement of instrumentation overhead is possible and, by hypothesis, *a posteriori* removal of perturbations via performance perturbation models.

To determine instrumentation overheads, we conducted a series of preliminary experiments that measured the *in vitro* costs of trace instrumentation. These costs provide the basis for the *in vivo* tests of our primary hypothesis, that instrumentation overheads can be removed from detailed performance traces. Among the possible perturbations discussed in §3.2, the most significant are increased execution time (i.e., time dilation) and increased memory bandwidth.

[13]We avoided the problems of variable costs for register save and restore by implementing an assembly code version of our trace library with the callee saves convention.

Execution Mode	Level	Mean Time	Standard Deviation
Sequential	C Source	10.26	±1.75
Sequential	Assembly	11.01	±2.52
Concurrent	Assembly	12.25	±2.60

Table 3.2: Instrumentation Time Dilation (Microseconds)

We began by measuring the cost to record an instrumentation trace event for both sequential and concurrent execution on the Alliant FX/80; see Table 3.2. A test program performed one thousand calls to the tracing routine, and call overhead statistics were calculated from the trace data.

Table 3.2 shows instrumentation statistics for three forms of instrumentation. The smaller time for sequential source instrumentation is directly attributable to the C compiler's optimizations for register save and restore. Because the assembly code instrumentation always saves all registers, it has a slightly higher cost. With concurrent tracing, all 8 CE's must write to different trace buffers. In this case, the total cache and memory traffic is greater and is distributed across a larger fraction of the address space. Because there are more cache misses and memory contention, the time dilation is greater.

In our implementation, each trace instrumentation point generates 48 bytes of memory traffic. Using the execution time measurements above, sequential and concurrent tracing can generate at most 4.36 megabytes/second and 31.35 megabytes/second of memory traffic, respectively. Although these are well below the peak bandwidth of the Alliant FX/80 memory bus, the additional memory traffic is substantial, particularly for concurrent tracing. If the application program is memory intensive, the potential memory perturbations of §3.2.1 become real.

3.5 Execution Time Experiments

Our instrumentation hypothesis suggests that the perturbations attributable to detailed performance instrumentation can be minimized in sequential, vector, and parallel execution modes. To test this hypothesis, we conducted a series of instrumentation experiments to determine

Loop	Description	Loop	Description
1	Hydrodynamics Fragment	13	2-D Particle In Cell
2	Incomplete Cholesky - Conjugate Gradient	14	1-D Particle In Cell
3	Inner Product	15	Casual Fortran
4	Banded Linear Equations	16	Monte Carlo Search Loop
5	Tri-diagonal Elimination, Below Diagonal	17	Implicit, Conditional Computation
6	General Linear Recurrence Equations	18	2-D Explicit Hydrodynamics Fragment
7	Equation of State Fragment	19	General Linear Recurrence Equations
8	A.D.I. Integration	20	Discrete Ordinates Transport
9	Integrate Predictors	21	Matrix-Matrix Product
10	Difference Predictors	22	Planckian Distribution
11	First Sum, Partial Sums	23	2-D Implicit Hydrodynamics Fragment
12	First Difference	24	Find Locations of First Minimum in Array

Table 3.3: Lawrence Livermore Loops

the magnitude of performance perturbations as a function of instrumentation frequency and execution mode (i.e., sequential, vector, and parallel on the Alliant FX/80).

All experiments used C and Fortran versions of the Lawrence Livermore loops (LLL) [131], a set of 24 loops often used to benchmark high-performance computer systems; see Table 3.3 for a brief description of each loop. These loops contain a diversity of language constructs, yet remain small enough to permit detailed analysis of performance perturbations. Table 3.4 summarizes the combinations of language and instrumentation used in our experiments. We emphasize that the purpose of our analysis was *not* to characterize the performance of the Livermore loops. Instead, the Livermore loops constitute a set of test cases for our performance perturbation models. Successful prediction of loop execution times and recreation of event times would validate our models.

For each Livermore loop, two experiments were conducted in each category; see Figure 3.3. In the first experiment, trace instrumentation was placed at the beginning and the end of the loop to determine total loop execution time, the so-called *raw* instrumentation. The second

Language	Mode	Type	Description
C	Sequential	Source	sequential C loops, instrumentation at source level
C	Sequential	NOP	sequential C loops, NOP instrumentation using asm() construct
Fortran	Sequential	Source	sequential Fortran loops, instrumentation at source level
Fortran	Sequential	Null	sequential Fortran loops, instrumentation at source level but with instrumentation removed
Fortran	Sequential	Assembly	sequential Fortran loops, assembly level instrumentation
Fortran	Sequential	NOP	sequential Fortran loops, NOP instrumentation of assembly code
Fortran	Vector	Assembly	vector Fortran loops, assembly level instrumentation
Fortran	Vector	NOP	vector Fortran loops, NOP instrumentation of assembly code
Fortran	Concurrent	Assembly	concurrent Fortran loops, assembly level instrumentation
Fortran	Vector Concurrent	Assembly	vector-concurrent Fortran loops, assembly level instrumentation

Table 3.4: Experiment Categories for the Lawrence Livermore Loops

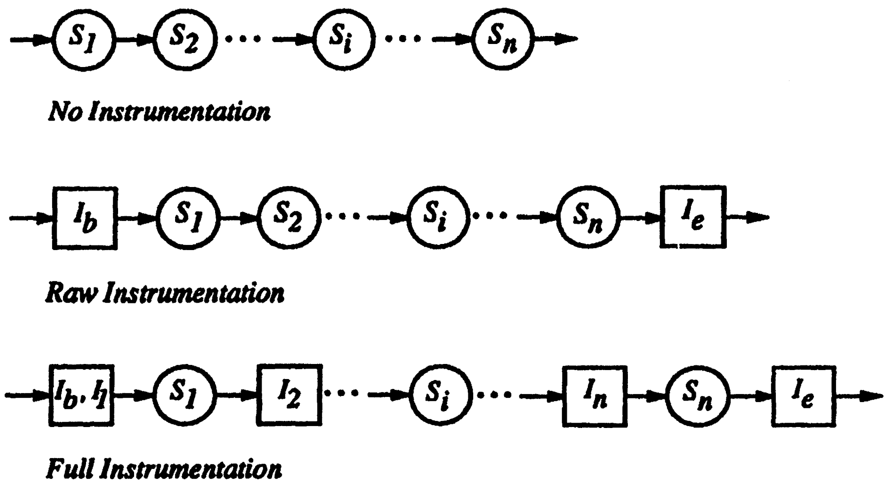


Figure 3.3: Instrumentation Alternatives

experiment produced traces from a *full* instrumentation with trace events for each source statement. For a typical loop, this instrumentation generated over 2000 trace events, with a total instrumented execution time of less than 0.1 seconds. Despite this density of trace instrumentation, our perturbation models recover actual execution time with less than ten percent error, confirming our hypothesis that detailed performance data need not be incompatible with accurate measurement.

In the remainder of this section, we describe the result of an execution time analysis of the experimental results for C, Fortran, vector Fortran, concurrent Fortran, and vector-concurrent Fortran.[14] In each case, we first analyze the the direct perturbations caused by the performance instrumentation using the simple models of §3.3. We then explain any deviations from the models by investigating sources of indirect perturbation (e.g., changes in the generated code).

3.5.1 Sequential C Experiments

The goal of our sequential C experiments was to compare source level instrumentation across languages. As mentioned earlier, the Alliant C compiler's optimizations are but a subset of those performed by the Fortran compiler, and we conjectured that source instrumentation in C was less susceptible to indirect performance perturbations. Figure 3.4 shows the results of

[14]The cross product of Livermore loops, instrumentation types, languages, and parallelism represent an enormous volume of experimental data.

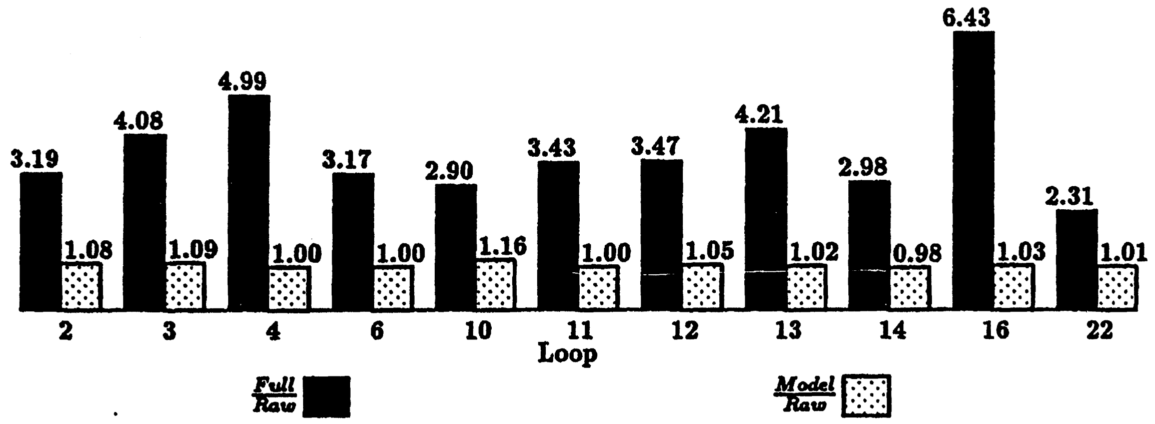


Figure 3.4: Source Instrumentation of C Loops

our experiments with complete source instrumentation for some of the loops. In the figure, the black bars represent the ratio of full instrumentation execution time to the raw instrumentation execution time; note that only a subset of the LLL loops are shown. The dotted bars represent the ratio of the predicted execution time, using the model of §3.3.2.1, to the raw execution time for the same loop. Clearly, the simple perturbation model predicts the major sources of perturbation and accurately predicts actual execution time. This result is true for all of the loops.

Although the simple perturbation model bounds the possible indirect perturbations of source code instrumentation, it cannot quantify these effects. However, replacing the C instrumentation statements with NOPs, retains the perturbations of code generation but removes all ancillary perturbations of instrumentation (e.g., memory referencing). The direct effects of instrumentation, assumed to be constant in our model, are emulated by the fixed execution time of the NOPs. Thus, the difference between the real source instrumentation approximation and the approximation from the NOP instrumentation are largely attributable to the ancillary perturbations. Figure 3.5 shows the result of predicting performance with NOP instrumentation at every source statement. All predictions are within three percent. This small error is attributable solely to the indirect perturbations of source code instrumentation. This supports our claim that the perturbations of code generation are small; a study of the assembly code produced from the C source instrumentation also shows that the changes are minimal. As an example, the model's approximation for loop 10 improves from 1.16 to 1.02. This suggests that there are perturbations different from instrumentation execution time, which the C-Nop

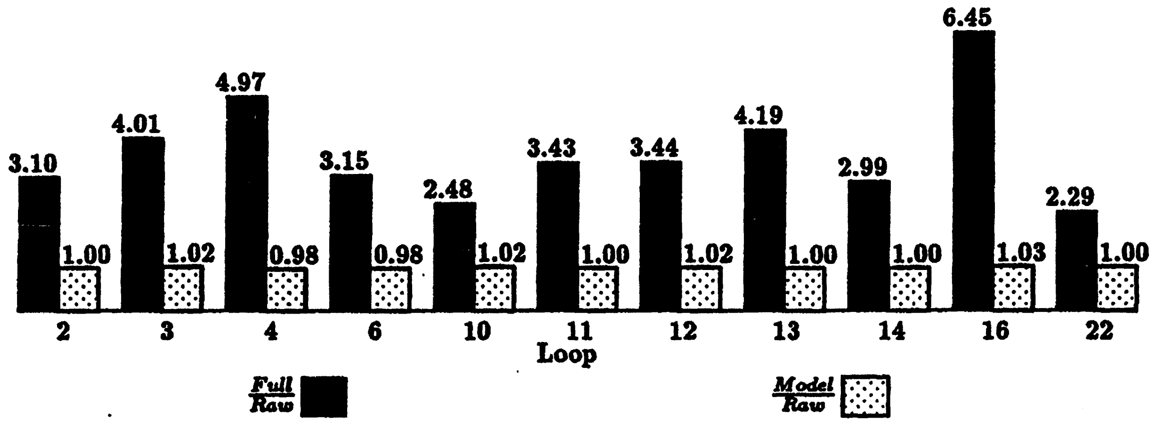


Figure 3.5: NOP Source Instrumentation of C Loops

experiments model, accounting for the slowdown of the trace-instrumented run. Upon inspection of the generated assembly code one sees that a register dependency is introduced by the tracing instrumentation that is not present in the code without instrumentation and the nop instrumented testcases. This dependency stalls the instruction following the instrumentation by 3 cycles on the Alliant FX/8 or 510 nanosecods. If we modify the simple sequential model to include this stalling, the approximation of the instrumented execution improves to 1.09 (within 9 percent of real execution time).

3.5.2 Sequential Fortran Experiments

Unlike source level instrumentation in C, the breadth of the Alliant Fortran compiler's optimizations creates large perturbations with source instrumentation. To quantify these perturbations and to determine an acceptable instrumentation methodology, we conducted experiments using four combinations of source and assembly code instrumentation; see Table 3.4.

Figure 3.6 shows the simplest test, complete instrumentation at the source code level. Unlike the comparable C experiment in Figure 3.4, the Fortran perturbations cannot be explained by our simple perturbation model, The model's approximations differ as much as 80 percent from the actual execution times. Clearly, some perturbations are indirect.

To determine the indirect perturbations, including changes to the generated code, we compiled each loop with source instrumentation inserted. We then removed the generated instrumentation from the assembly code. This *null instrumentation* retains all perturbations of code

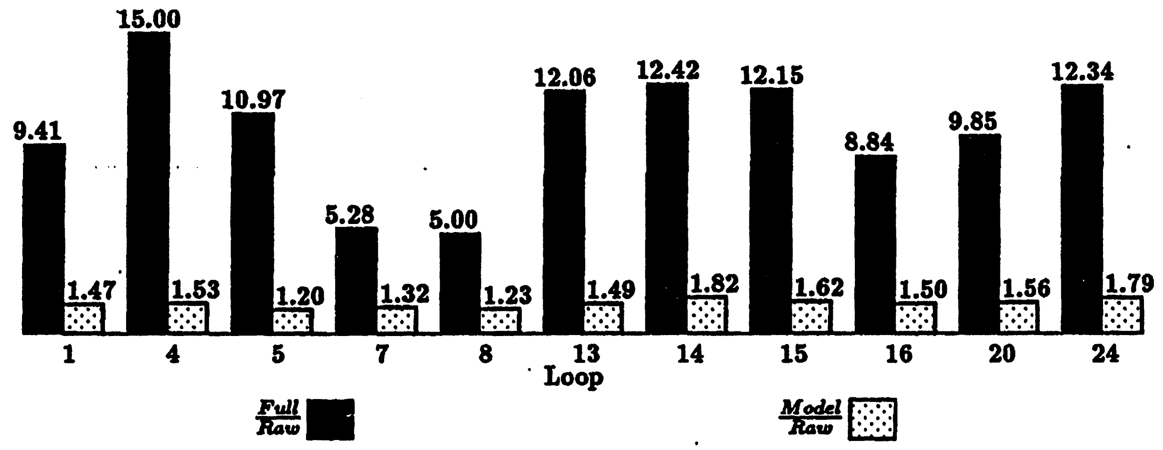


Figure 3.6: Source Instrumentation of Fortran Loops

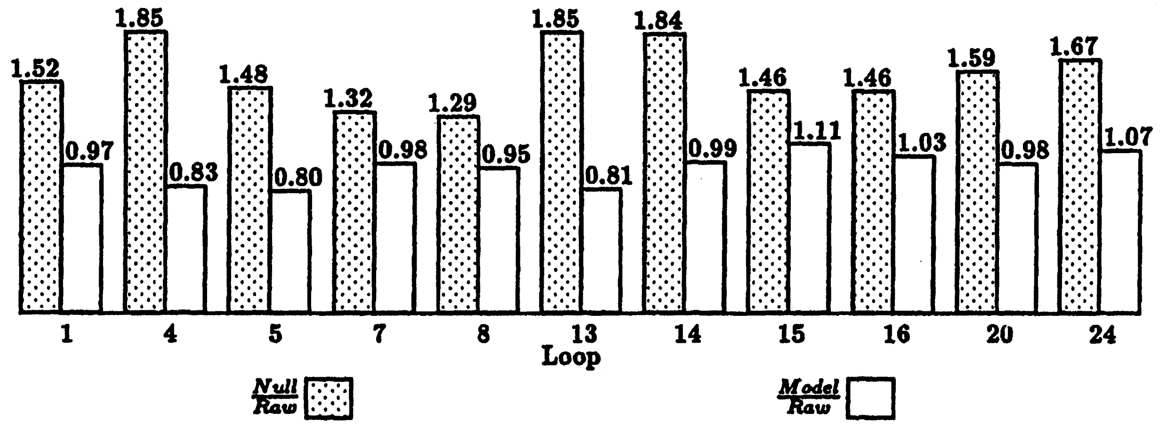


Figure 3.7: Source Instrumentation of Fortran Loops with Instrumentation Removed

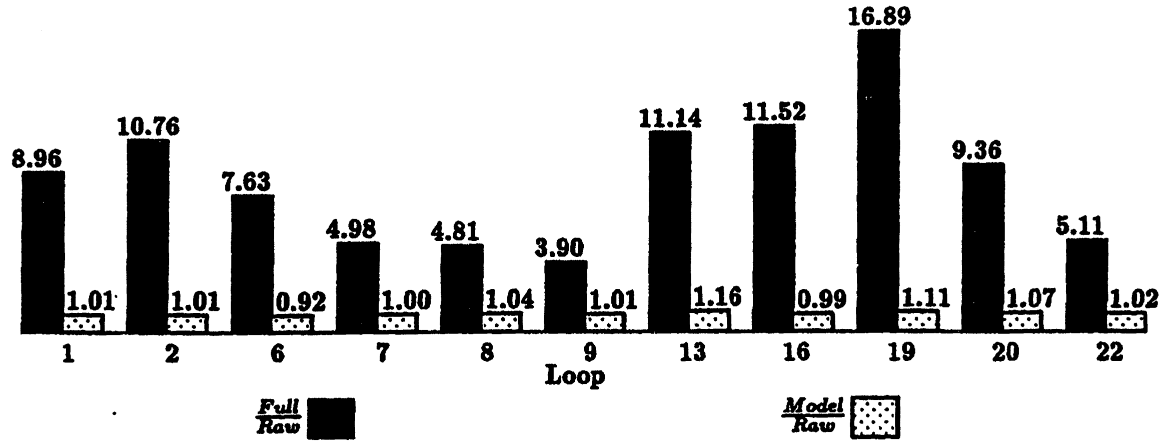


Figure 3.8: Assembly Code Instrumentation of Fortran Loops

generation without the direct perturbations that accrue from execution of instrumentation. As Figure 3.7 shows, these indirect perturbations are substantial; inserting source instrumentation inhibits many code optimizations. Do the perturbations of code generation account for a significant fraction of all perturbations? The white bars in Figure 3.7 show the approximations of the model of §3.3.2.1 when equation (3.6) is applied; that is, the code generation perturbations have been removed from the original approximations. As can be seen, the model predicts well for some loops. However, significant perturbations remain, and one must consider additional perturbations to fully explain the instrumentation's effect on loop performance in all cases.

The complex interactions of source instrumentation and Fortran compiler make isolation of instrumentation perturbations difficult, if not impossible. Because our goal is the systematic application of a standard perturbation model, the remainder of our experiments were conducted by instrumenting the generated assembly code of each loop.[15]

Figure 3.8 shows the merit of Fortran assembly instrumentation, and indirectly, the need for compiler supported performance instrumentation. If code generation is not perturbed, mechanical application of our simple perturbation model permits recovery of total execution time with small error, typically less than five percent. For those Livermore loops where this approach fails, and the error never exceeds 40 percent, detailed analysis reveals an extraordinary ratio of instrumentation to application code (e.g., instrumentation of a single machine instruction).

[15] With suitable modifications, a compiler could generate this instrumentation after all optimization, eliminating indirect perturbations during code generation.

Livermore Loop	Scalar Events	Vector Events	Livermore Loop	Scalar Events	Vector Events
1	2006	68	9	206	12
3	2007	70	12	2004	68
4	1820	68	13	1092	43
6	4162	381	14	17024	3424
7	1994	68	15	10648	1040
8	1395	75	18	4498	203

Table 3.5: Fortran Trace Event Counts

Often, accounting for a single, 170 nanosecond cycle register stall in the instrumented code reduces the error to less than five percent. It is also possible that the instrumentation causes a register dependency, present in the code without instrumentation, to be eliminated. Single cycle instrumentation typically requires hardware support; even in these stress tests, the accuracy of the software model is surprising.

3.5.3 Vectorized Fortran Experiments

Instrumenting the iterations of a sequential loop produces a sequence of trace events for each loop iteration. If the same loop is vectorized and the loop bound is less than the vector register length, a single machine instruction may represent the entire computation. Thus, a vectorized loop contains fewer potential instrumentation points. For example, if the vector loop bound exceeds the vector register length, the loop can be instrumented only at those points were register reloads occur.

Table 3.5 shows those Livermore loops that the Alliant Fortran compiler vectorized and the number of associated instrumentation trace events. The number of vector events depends on the vector length and loop complexity.

Because a vector instruction performs many identical operations, it represents a larger computational granule than a simple scalar instruction. Consequently, the relative perturbation for instrumented vector instructions is less than that for scalar instructions, and perturbation models should more accurately predict execution time. Figure 3.9 shows precisely this result.

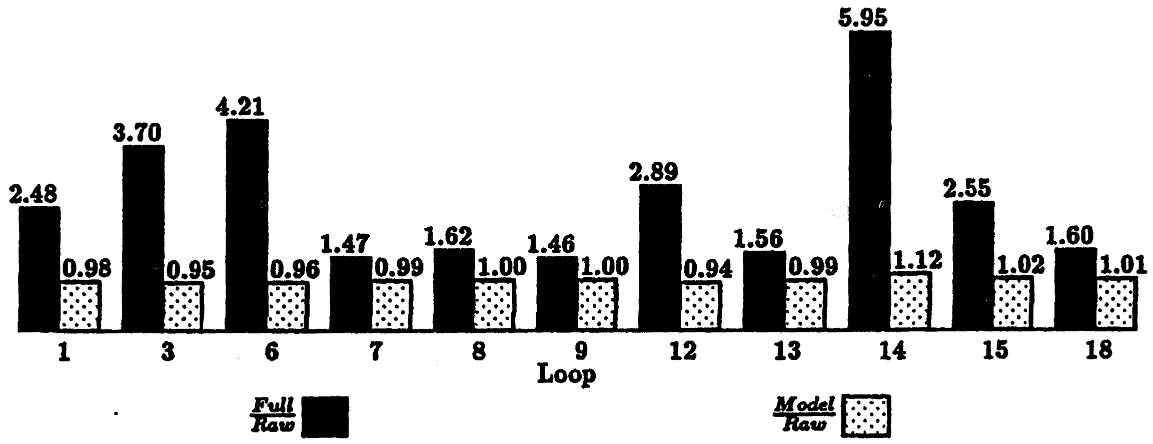


Figure 3.9: Assembly Code Instrumentation of Vectorized Fortran Loops

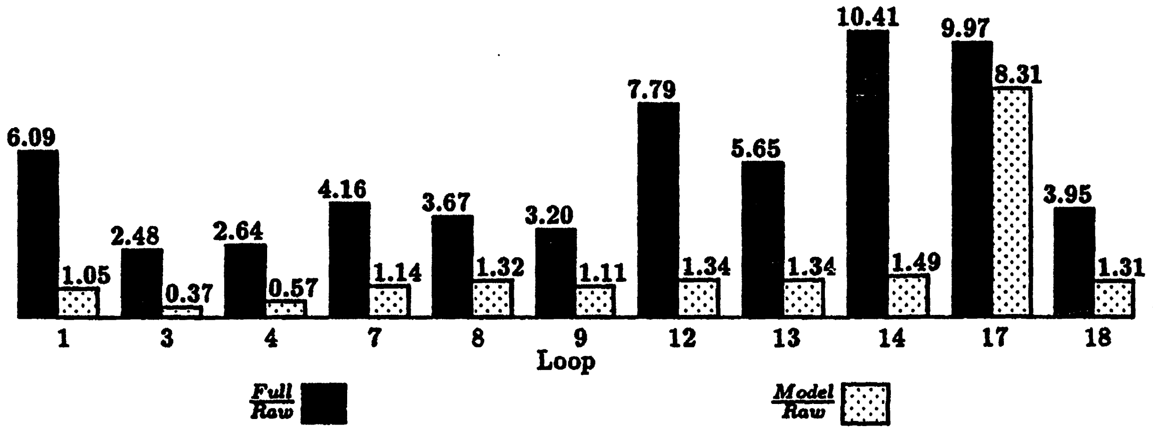


Figure 3.10: Assembly Code Instrumentation of Concurrent Fortran Loops

3.5.4 Concurrent Fortran Experiments

Unlike vectorization, compiling for concurrency does not condense multiple statements; all observable events in a sequential trace remain present, but a subset of the events execute concurrently. Moreover, tracing instrumentation may dominate concurrent execution of some loops (i.e., a significant fraction of concurrent time may be spent in instrumentation code). In these cases, critical paths must be identified with care.[16]

Figure 3.10 shows the result of our concurrent perturbation model for those loops concurrentized by the Alliant Fortran compiler. Clearly, the concurrent perturbation model yields approximations that are both high and low. For most loops, the errors can be explained via a

[16]Recall that the concurrent model of §3.3.2.2 makes restrictive assumptions about execution behavior.

combination of register dependencies and increased memory traffic. For instance, by applying concurrent trace instrumentation overheads in the timing analysis for those loops where tracing dominates execution (8, 12, 13, 14, and 18) we were able to reduce errors to within 10 percent. For other codes, in particular loops 3, 4, and 17, the concurrent execution model assumed by the timing model is violated due to data dependent behavior. Because the timing model does not include critical path analysis, substantial approximation errors occur in these codes. We consider loops 3 and 17 in more detail to explain why these errors occur.

For Livermore loop 3 (Inner Product), our execution time model overestimates the instrumentation perturbation (i.e., the model's estimate of total execution time is too low). Because the perturbation model charges only for direct perturbations, one infers that the instrumentation *reduces* the execution time of the application code. For loop 3, the Alliant Fortran compiler creates a critical section around the update of the inner product sum. Without instrumentation, most processors are blocked on entry to the loop's critical section. Adding instrumentation increases the total computation in each concurrent iteration and reduces the probability of blocking at the critical section. In consequence, the processors spend less time waiting when the code is instrumented; subtracting a fixed overhead underestimates the total execution time. Interesting, this analysis also applies in the case of loop 4 (Banded Linear Equations).

For loop 17 (Implicit, Conditional Computation), our model underestimates the instrumentation perturbation. Surprisingly, the reason is the same as that for loop 3 — the loop contains a critical section. Unlike loop 3, however, the trace instrumentation lies inside the critical section. This increases the probability of contention, and the critical section becomes a larger fraction of the total execution time in the instrumented code. Subtracting a fixed overhead overestimates the total execution time.

The concurrency experiments reveal both the importance of instrumentation placement and the need for accurate critical path analysis. The latter permits identification and removal of indirect perturbations (e.g., synchronization stalls). This is considered in detail in Chapters 4 and 5.

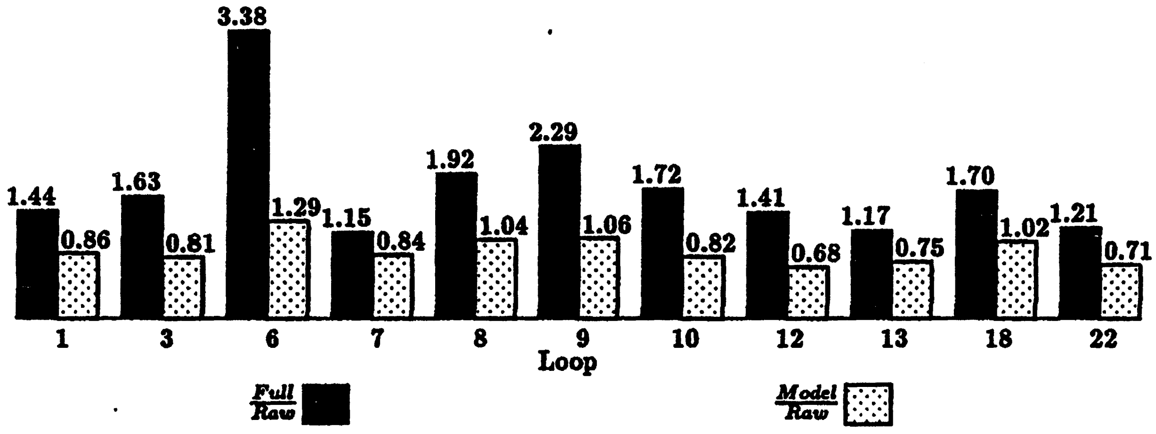


Figure 3.11: Assembly Code Instrumentation of Vector Concurrent Fortran Loops

3.5.5 Vector Concurrent Fortran Experiments

Vector concurrent execution mode combines the perturbations of both vectorization and concurrentization. Fortuitously, the respective perturbations are not additive. Vectorization damps perturbation by increasing computation granules while reducing the number of instrumentation events. Concurrency, however, reduces the execution time and increases the stress on the memory hierarchy.

Figure 3.11 shows the result of our experiments for those loops that executed in vector concurrent mode. In contrast to Figure 3.8 or even Figures 3.9 or 3.10, the ratio of instrumented to actual execution time is small; each processor records a smaller number of events. With one exception, our perturbation model consistently underestimates total execution time. As discussed below, we conjecture that the underlying reason is the same for each loop.

Most loops contain sequences of vector memory operations, separated by instrumentation. Moreover, earlier studies have shown that the Alliant FX/80 is susceptible to cache bank contention in vector concurrent mode [3, 64]. Although the trace instrumentation also references the cache, the request rate is modest compared to that for vector instruction sequences. We conjecture that tracing changes the temporal distribution of memory references, reducing the number of cache and memory bank conflicts. This, in turn, reduces the total execution time.

With rare exception, our performance instrumentation models accurately predict total execution time for the Alliant FX/80's sequential, vector, concurrent, and vector-concurrent modes. In those instances where the models are inaccurate, most inaccuracies can be explained by simple analysis of the instrumented code. This suggests that simple perturbation timing models, coupled with event trace models, would permit detailed, but accurate, performance tracing.

3.6 Event Trace Analysis

The experiments described in §3.5 evaluated the feasibility of predicting total execution time in the presence of massive instrumentation. Experimental results show that global performance measures, such as total execution time, are still computable to a acceptable accuracy via application of performance perturbation models. Clearly, one need not instrument every source language statement to determine total execution time; simpler, less intrusive methods exist. The motivation of this approach is to obtain additional performance data. However, the benefit of this trace data depends on its accuracy (i.e., how well it reflects actual event times and orders).

To determine the accuracy of trace data, one needs a standard of reference. Without a passive hardware monitor to capture and record events, no such standard exists. Instead, one must compare a sequence of event traces, each produced with successively smaller subsets of the complete trace instrumentation. As the number of trace events decreases, the presumed accuracy of the event times and orders increases.

From the 24 Livermore loops, we selected two loops for detailed study. The first, loop two, executes in sequential mode; the second, loop eight, executes in sequential, vector, concurrent, and vector-concurrent modes. For each loop, we created a trace by instrumenting each source language statement, the *full* trace. In addition, we generated traces using two partial instrumentations, each a successive subset of the complete instrumentation, traces *partial-1* and *partial-2*, respectively. After applying the trace perturbation model to each trace, we used Gantt charts [69] to verify qualitative agreement; comparison of the mean percent difference between event times confirmed quantitative agreement.

3.6.1 Loop Two – Incomplete Cholesky Conjugate Gradient

Livermore loop two, shown in Figure 3.12, is an excerpt from an incomplete Cholesky conjugate gradient code that executes in sequential mode. Figure 3.13 shows the Gantt charts of traces

```
            call trace_event(0)
        II = 101
            call trace_event(1)
        IPNTP = 0
            call trace_event(2)
222     CONTINUE
            call trace_event(3)
        IPNT = IPNTP
            call trace_event(4)
        IPNTP = IPNTP + II
            call trace_event(5)
        II = II/2
            call trace_event(6)
        I = IPNTP
            call trace_event(7)
        DO 2 K = IPNT+2, IPNTP ,2
              call trace_event(8)
           I = I+1
              call trace_event(9)
           X(I) = X(K) - V(K)*X(K-1) - V(K+1)*X(K+1)
              call trace_event(10)
2       CONTINUE
              call trace_event(11)
        IF(II.GT.1) GO TO 222
              call trace_event(12)
```

Figure 3.12: Instrumented Livermore Loop Two

Reference			Analyzed			Total	Mean	Percent
Trace	Events	Time μsecs	Trace	Events	Time μsecs	Delta μsecs	Delta μsecs	Delta
Full	333	383.4	Partial-1	236	378.7	933.4	3.96	1.04
Full	333	383.4	Partial-2	139	380.6	402.8	2.90	0.76
Partial-1	236	378.7	Partial-2	139	380.6	429.3	3.09	0.81

Table 3.6: Event Time Differences for Loop Two

from three levels of instrumentation.[17] Events 3 and 11 mark the beginning and end, respectively of each outer loop iteration. Traces *full* and *partial-1* also show the inner loop iterations, marked by events 8 and 10, respectively.

Although the three traces agree qualitatively, there are small quantitative differences. The total execution times, as predicted by the trace perturbation models, do differ by a small amount, but not greater than 1.25 percent. To verify event times, we correlated trace events and compared their times for three combinations of a *reference* trace, which contains the larger number of events, and an *analyzed* trace, which contains a subset of the events in the reference trace. The sum of the absolute differences between matched event times is the total event delta. Dividing this number by the number of events in the analyzed trace yields the mean event delta. Finally, dividing the mean event delta by the number of events yields the percent event delta. Table 3.6 shows the result of this analysis for loop two. The mean difference between corresponding event times is less than 2 percent of the total execution time, or 8 microseconds. This is an excellent match, given the Alliant FX/80 timer resolution of 10 microseconds.

3.6.2 Loop Eight – ADI Integration

Loop eight, an ADI integration, can be both vectorized and parallelized. This permits evaluation of trace perturbation for all the Alliant's execution modes. Figures 3.14 and 3.15 show the Gantt charts for sequential and vector mode, respectively. Although the number of sequential events makes visual comparison difficult, the temporal event correlation for vector mode is striking. This is a direct consequent of the high accuracy of the total execution time approximations.

[17] In Figures 3.13 and 3.15, individual events are marked by the symbol +; the display scale for other Gantt charts does not permit display of individual events. In the figures, event simultaneity is a consequence of limited instrumentation clock resolution; see §3.4.

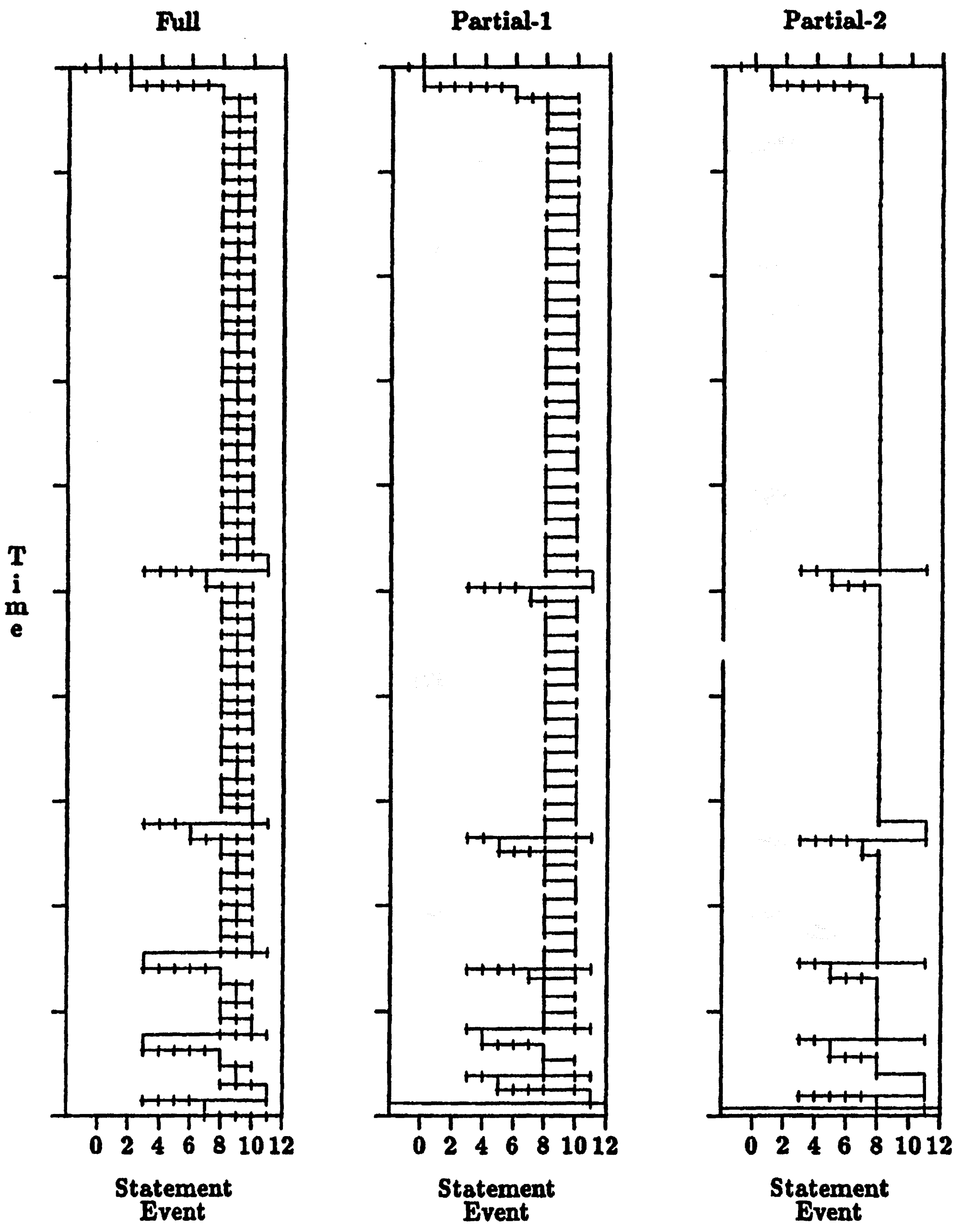


Figure 3.13: Sequential Execution of Loop Two

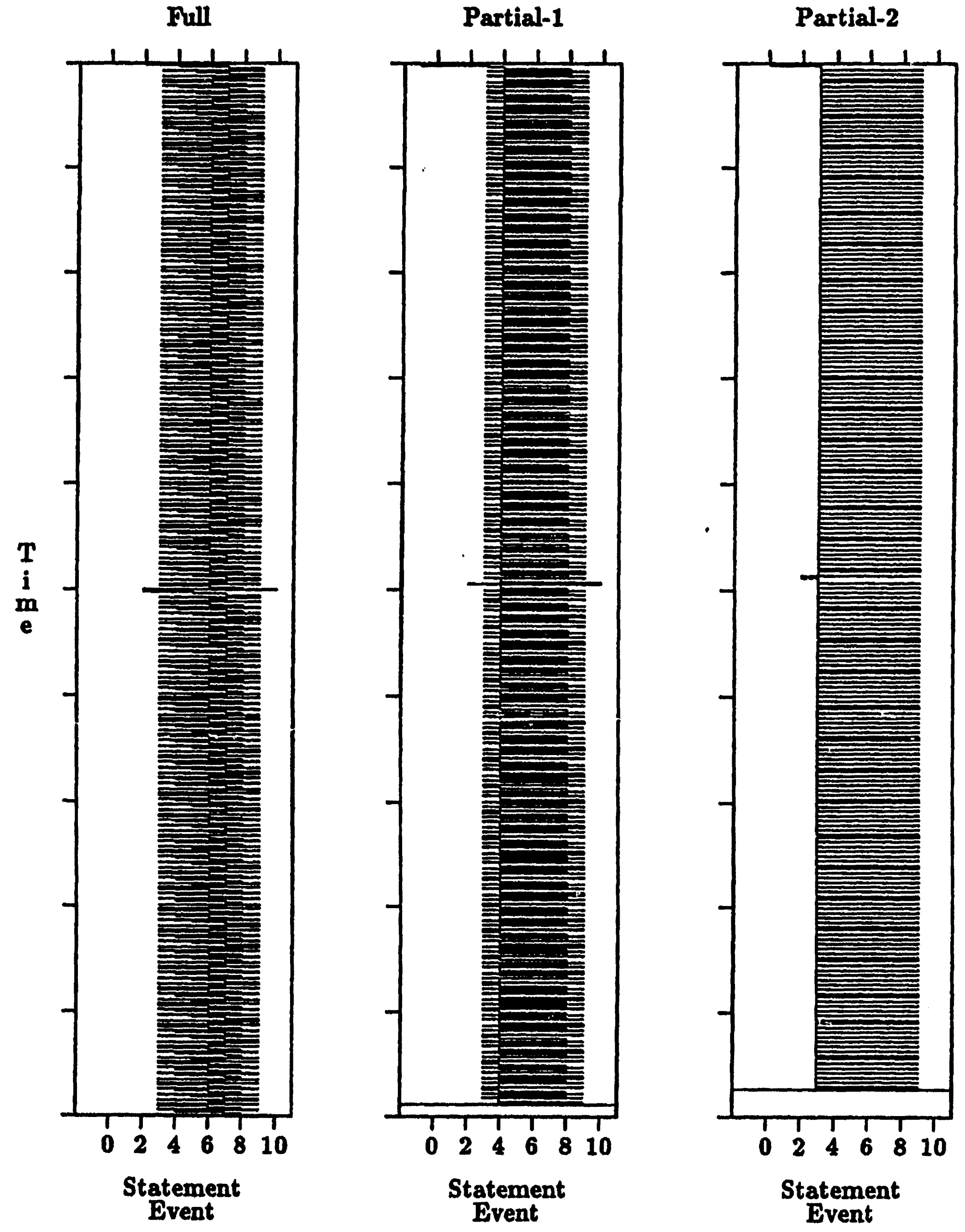


Figure 3.14: Sequential Execution of Loop Eight

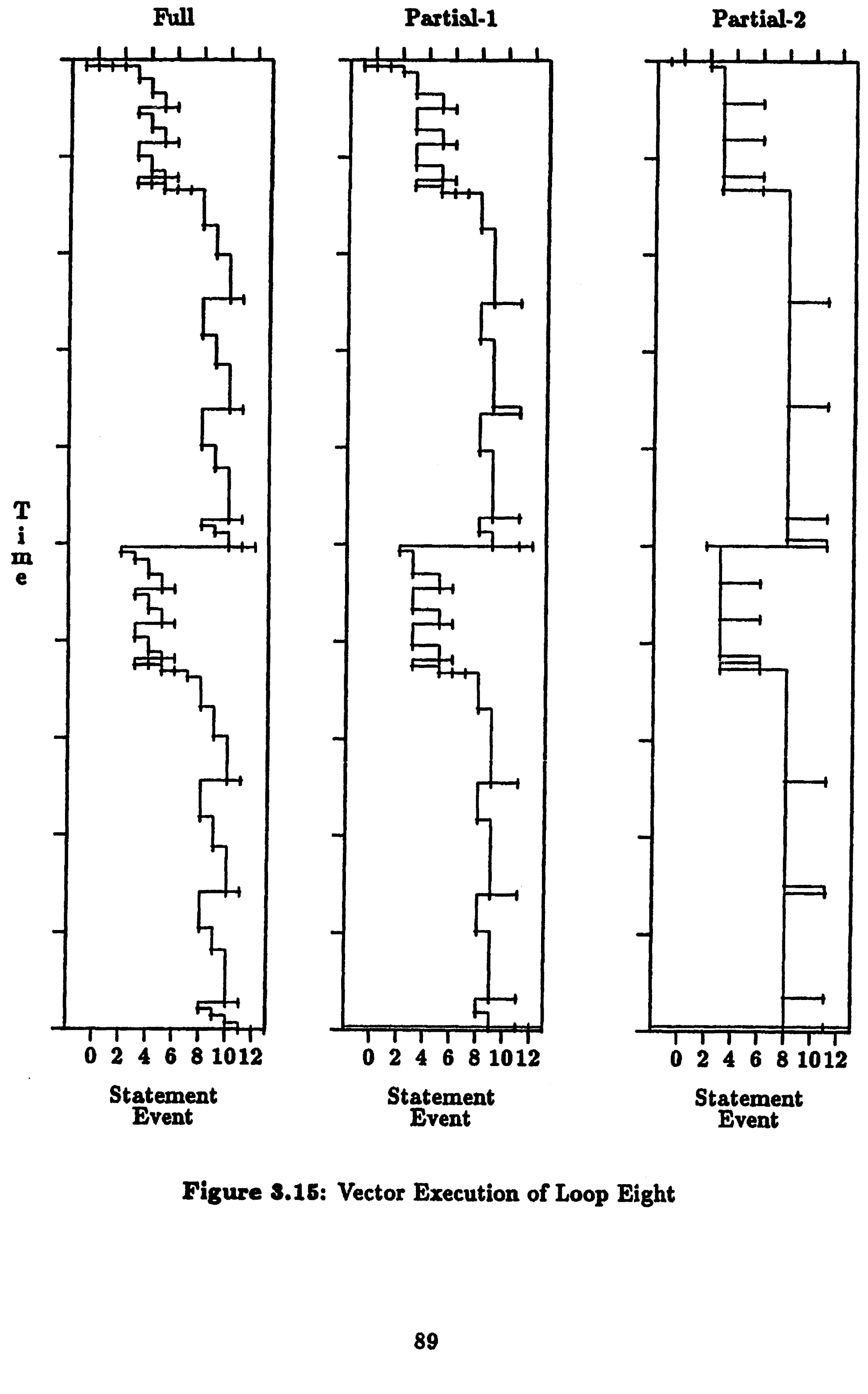


Figure 3.15: Vector Execution of Loop Eight

Reference			Analyzed			Total	Mean	Percent
Trace	Events	Time μsecs	Trace	Events	Time μsecs	Delta μsecs	Delta μsecs	Delta
Full	1395	4100.0	Partial-1	801	4051.8	22415.1	27.98	0.69
Full	1395	4100.0	Partial-2	402	3990.0	24038.8	59.80	1.50
Partial-1	801	4051.8	Partial-2	402	3990.0	12992.1	32.32	0.81

Table 3.7: Event Time Differences for Loop Eight, Sequential Execution

Reference			Analyzed			Total	Mean	Percent
Trace	Events	Time μsecs	Trace	Events	Time μsecs	Delta μsecs	Delta μsecs	Delta
Full	75	1285.2	Partial-1	59	1280.5	174.9	2.96	0.23
Full	75	1285.2	Partial-2	38	1278.5	238.3	6.27	0.49
Partial-1	59	1280.5	Partial-2	38	1278.5	226.9	5.97	0.47

Table 3.8: Event Time Differences for Loop Eight, Vector Execution

The accuracy of the event times, as predicted by the trace perturbation model for sequential and vector execution, is confirmed by Tables 3.7 and 3.8. In sequential execution mode, event timing differences, although as much as 60 microseconds on average, are less than 1.5 percent of the total execution time. The discrepancies for vector mode are even smaller — the mean difference is less than the timing resolution, and the percent difference is less than 0.5 percent.

Figure 3.16 shows the Gantt charts for the sequential thread and all concurrent threads from the concurrent execution of Livermore loop eight on six processors.[18] In this code, there are two concurrent loops of equal complexity, separated by a sequential operation. During concurrent computation, each thread receives roughly the same amount of work. These execution behavior characteristics are similar for each set of traces.

Although there are qualitative similarities between the traces of Figure 3.16, the differences in predicted total execution time suggest substantial error in individual event times. Indeed, the execution time difference between the *full* and *partial-2* trace is 18 percent; see Table 3.9. Surprisingly, however, the percent event delta between the *full* and *partial-1* traces is less than 6 percent; the same is true for the comparison of traces *partial-1* and *partial-2*. As the difference

[18] We used only six Alliant CE's in these experiments to reduce the visual complexity of the figures.

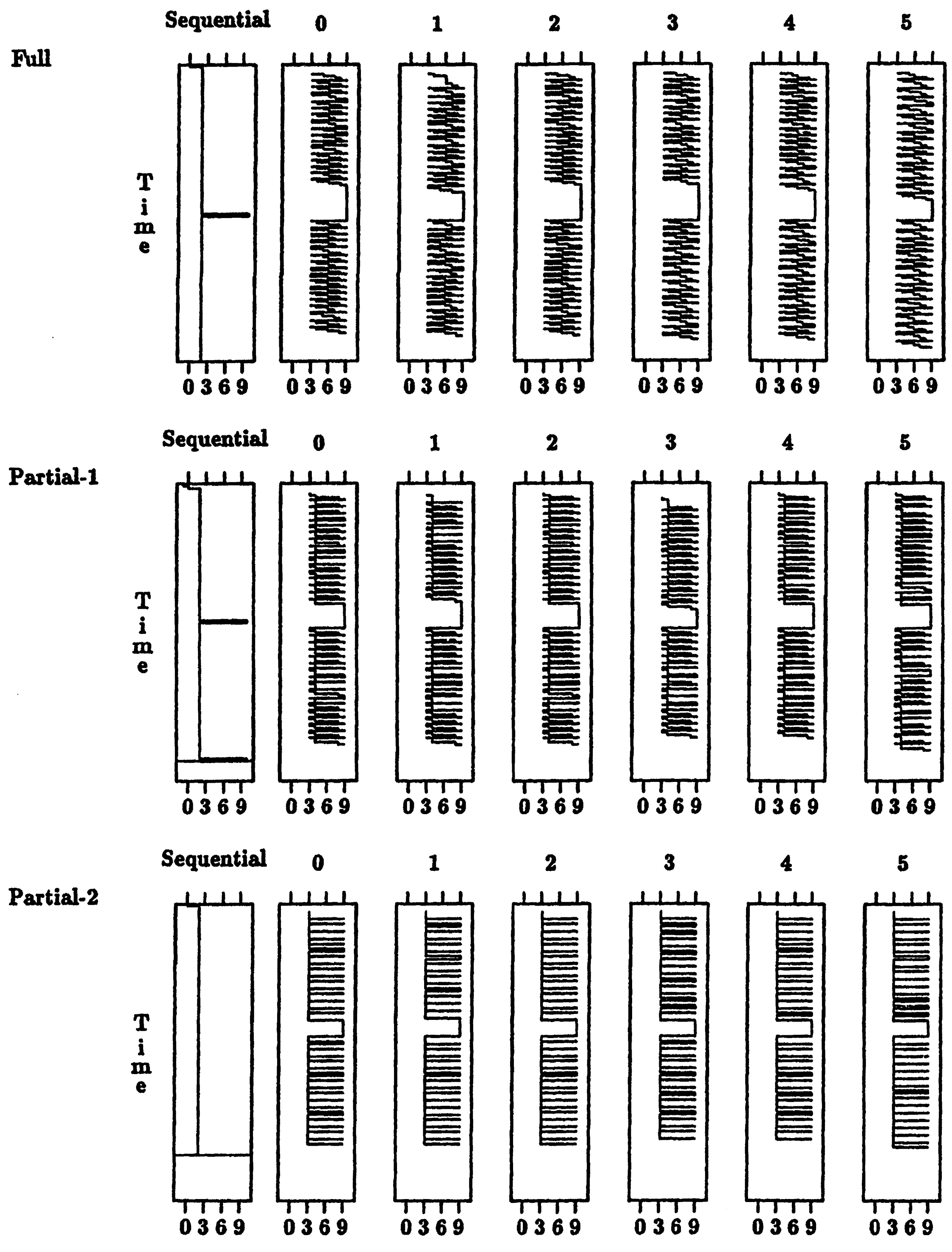


Figure 3.16: Concurrent Execution of Loop Eight

Reference Trace			Analysed Trace			Total	Mean	Percent
Thread	Events	Time	Thread	Events	Time	Delta	Delta	Delta
		μsecs			μsecs	μsecs	μsecs	
Full			Partial-1					
Seq	9	1126.6	Seq	9	1052.1	355.3	39.48	3.75
0	238	1020.3	0	136	989.4	2926.1	21.51	2.17
1	231	1030.3	1	132	989.4	4930.2	37.35	3.78
2	238	1030.3	2	136	989.4	3272.4	24.06	2.43
3	224	1046.6	3	128	963.0	6294.3	49.17	5.11
4	231	1046.6	4	132	963.0	4890.5	37.05	3.85
5	224	1077.5	5	128	1013.0	5631.7	44.00	4.34
Full			Partial-2					
Seq	9	1126.6	Seq	6	952.4	444.6	74.10	7.78
0	238	1020.3	0	68	910.6	4045.4	59.49	6.53
1	231	1030.3	1	66	910.6	5304.8	80.38	8.8
2	238	1030.3	2	68	910.6	4240.9	62.37	6.85
3	224	1046.6	3	64	892.4	4453.3	69.58	7.80
4	231	1046.6	4	66	892.4	4937.9	74.82	8.38
5	224	1077.5	5	64	922.4	5590.1	87.35	9.47
Partial-1			Partial-2					
Seq	9	1052.1	Seq	6	952.4	267.9	44.65	4.69
0	136	989.4	0	68	910.6	2664.8	39.19	4.30
1	132	989.4	1	66	910.6	2859.6	43.33	4.76
2	136	989.4	2	68	910.6	2710.3	39.86	4.38
3	128	963.0	3	64	892.4	3400.2	53.13	5.95
4	132	963.0	4	66	892.4	2535.4	38.42	4.30
5	128	1013.0	5	64	922.4	2762.1	43.16	4.68

Table 3.9: Event Time Differences for Loop Eight, Concurrent Execution

between the number of events in the reference trace and those in the analyzed trace increases (e.g., in the comparison of *full* and *partial-2*), the percent event delta rises. However, even in the worst case it does not exceed 10 percent of the total execution time. Stated another way, the average uncertainty between two matched events in the *full* and *partial-2* traces is less than 10 percent.

Finally, Figure 3.17 shows the Gantt charts for vector-concurrent execution mode. Vectorization reduces the number of possible instrumentation points and the total number of trace events, and the predicted execution times differ by less than 3 percent. Because the two loops in the code are statically scheduled in vector-concurrent mode, unlike the dynamic schedule in concurrent mode, the execution behavior across processors should not differ. In Figure 3.17, the execution signatures are indistinguishable. Moreover, Table 3.10 shows that the percent delta in event times is less that 5 percent. More importantly, the mean differences are at the limits of the timer resolution.

3.7 Comments

The experiments discussed in §3.5 and §3.6 are stress tests for the time-based performance perturbation models. The ability to approximate actual code execution times to within 15 percent from full trace instrumentations, with execution time perturbations exceeding four orders of magnitude, is remarkable, especially for such relatively simple models. Even for certain Livermore Loops that are not well approximated, often minor adjustments in the models to account for register interlock stalls or increased memory reference density due to tracing operations can account for the error. Not only do the models perform well when approximating total execution time of the fully-instrumented Livermore loops, but the accuracy of individual event timings are equally impressive.

The time-based perturbation models accurately capture the effects of instrumentation perturbation when the time and order events occur is execution independent. This is true for sequential execution because the execution states of sequential programs form a total order, and event times are affected only by instrumentation overhead. Our timing model approximations for the Livermore loops in sequential and vector modes were extremely accurate. Even for

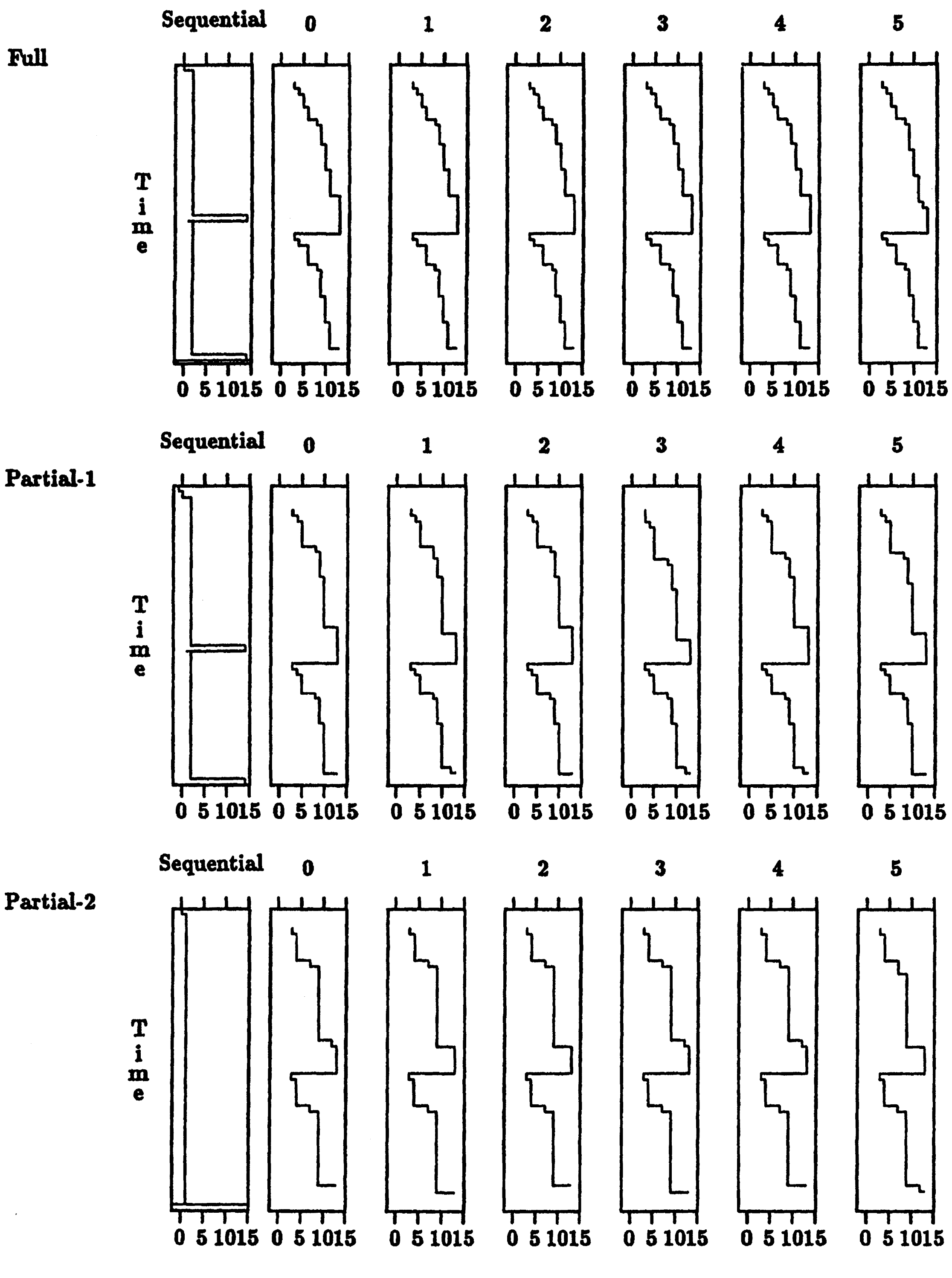


Figure 3.17: Vector-Concurrent Execution of Loop Eight

Reference Trace			Analysed Trace			Total	Mean	Percent
Thread	Events	Time	Thread	Events	Time	Delta	Delta	Delta
		μsecs			μsecs	μsecs	μsecs	
Full			Partial-1					
Seq	10	465.4	Seq	10	470.2	79.0	7.90	1.68
0	22	446.4	0	18	452.0	170.3	9.46	2.09
1	22	446.4	1	18	451.1	198.3	11.01	2.44
2	22	446.4	2	18	452.0	170.3	9.46	2.01
3	22	446.4	3	18	451.1	308.2	17.12	3.80
4	22	446.4	4	18	451.1	198.2	11.01	2.44
5	22	446.4	5	18	452.0	189.5	10.53	2.33
Full			Partial-2					
Seq	10	465.5	Seq	6	460.2	23.4	3.91	0.85
0	22	446.4	0	12	429.7	101.0	8.42	8.42
1	22	446.4	1	12	440.2	84.1	7.01	1.59
2	22	446.4	2	12	429.7	105.1	8.76	2.04
3	22	446.4	3	12	440.2	80.1	6.67	1.52
4	22	446.4	4	12	429.7	101.0	8.42	1.96
5	22	446.4	5	12	439.2	96.8	8.07	1.84
Partial-1			Partial-2					
Seq	10	470.2	Seq	6	460.2	63.5	10.58	2.30
0	18	452.0	0	68	429.7	195.7	16.31	3.79
1	18	451.1	1	66	440.2	182.4	15.20	3.45
2	18	452.0	2	68	429.7	184.3	15.35	3.57
3	18	451.1	3	64	440.2	253.8	21.15	4.81
4	18	451.1	4	66	429.7	204.8	17.06	3.97
5	18	452.0	5	64	439.2	204.2	17.02	3.87

Table 3.10: Event Time Differences for Loop Eight, Vector-Concurrent Execution

some concurrent execution scenarios, typically those with fork-join behavior and no inter-thread dependencies, the time-based perturbation models are good.

In general, concurrent execution involves data dependent behavior. The states of parallel programs inherently form a partial order that must be followed during execution. If dependency control is spread across threads of execution, instrumentation can perturb the timing relationships of events. Direct applications of time-based perturbation models will fail because they *do not* capture these inter-thread event dependencies. Under the timing model assumptions of event independence, approximated event timings for concurrent execution can violate the required partial order. Furthermore, critical performance phenomena such as synchronization behavior cannot be accurately modeled using timing information alone.

Clearly, concurrent perturbation analysis necessitates a model of event dependencies and instrumentation. The next two chapters explore these issues.

Chapter 4

Event-Based Performance Perturbation: Synchronization

"Curiouser and curiouser."

— Alice, "Alice In Wonderland"

The performance of parallel computations often depends on the relative ordering of dependent events across multiple threads of execution. Each parallel thread can be viewed as making transitions between phases of independent and dependent execution. That is, either the thread can proceed with a computation independent of the activities in the other threads, or the thread is dependent on some conditions that must be satisfied before proceeding. In cases where a thread is waiting for a synchronization action, overall performance is reduced. However, parallel performance is also dependent on the efficient allocation and utilization of resources, mainly processor, memory, and network resources. Scheduling, load balancing, data partitioning, and communication overhead are among the many performance issues that must be addressed.

The fundamental problem with making detailed measurements of parallel computations is not performance degradation, as it with sequential computations, but rather the perturbation of the set of "likely" event orderings, resulting in the re-mapping of event occurrence to threads of execution, the re-assignment of computational resources [113], and changes in the behavior of resource use. Unlike parallel debugging approaches that attempt to detect data races in parallel programs by applying an event-based, partial order theory of "feasible" program execution

[48, 53, 145, 146, 156], perturbation analysis must recover the actual run-time performance behavior from a perturbed performance measurement.

If performance instrumentation is designed correctly [86], an un-instrumented parallel execution that satisfies Lamport's *sequential consistency* criterion[1] [105, 116] implies that the performance measurement will be *non-interfering* and *safe* [86]. If the performance measurements involve only the detection and recording of event occurrence (i.e. tracing), the partial order relationships will be unaffected and the set of feasible executions[2] will remain unchanged [133, 146]. This consequence will also be true in the less restricted condition of *weak ordering* [4]. Thus, perturbation analysis begins with a total ordering of measured events consistent with the *happened before* relation [104] defined by the original partial order execution. To this total order, we can apply time-based perturbation analysis to thread events that occurred during independent execution to remove the instrumentation overhead. Similarly, event-based perturbation analysis (to be discussed below) can be applied to the synchronization operations (e.g., barriers, semaphores, advance/await) that implement the dependency relationships. As long as the total ordering of dependent events present in the measured execution is maintained during the analysis, the approximated execution also will be a feasible execution. We will call such an approximated execution a *conservative approximation.*

However, the important question is not whether the conservative approximation is a feasible execution, but whether it is a likely execution. The set of likely executions is the subset of the feasible executions that are most probable. Computing the likelihood distribution of feasible executions is a extremely difficult problem, requiring a model of time and concurrent execution. Analytical queueing models have been used to predict the performance of parallel computations [84, 85], including general, parallel execution structures with precedence constraints [117, 142, 190, 191] and synchronization [2]. However, these approaches typically model execution time behavior stochastically, limiting their use in practice. Other approaches attempt to model time dependent behavior in concurrent software. Lane [106] proposes the *event dependency tree* model that includes time semantics but lacks a broad set of synchronization operations to make

[1]A parallel execution is sequentially consistent (or equivalently has interleaving semantics [173]) if the result is the same as if the operations were executed in some sequential order obtained by arbitrarily interleaving the thread execution streams.

[2]Helmbold and Bryan refer to the set of feasible executions defined by the partial order of program events as the *partially ordered set* [86].

it a practical approach. Haase [82] and Shaw [174] on the other hand, each define a *time logic* based on program statements for reasoning about timing properties in programs. Shaw's work is more robust, considers a larger class of program constructs and uses interval arithmetic in the logic specification, but it only superficially treats timing issues in dependent concurrent execution. Shaw comments that an approach to concurrent timing analysis must consider the specific context within which synchronization statements are used, including their overheads and interactions.

The inability to predict likely executions makes it difficult to bound the error of conservative approximations. Simple examples can demonstrate significant errors for conservative approximations (i.e., the execution order is not representative of a likely execution without instrumentation). Furthermore, no intrusive performance measurements can possibly allow event-based perturbation analysis to determine the proper assignment and use of resources in the approximated execution. To improve the "accuracy" of the conservative approximation, additional information must be provided to the perturbation analysis process that describes certain behavioral properties of the computation (e.g., data dependency information and loop scheduling algorithms). The perturbation analysis can use this information to make more "liberal" approximations. Although the liberal approximations might be more accurate than conservative ones, in the sense that they are closer to likely executions, it is still difficult to show error bounds without a more formal timing model.

In this chapter, we describe methods for conservative perturbation analysis of synchronization operations found in concurrent programs. The approaches described are constructive; they can be applied to a trace of a measured execution to eliminate instrumentation overhead while maintaining observed dependency relationships. The main issues addressed concern what data must be measured to resolve dependent relationships between the threads and what assumptions are made about synchronization operation. In §4.2 we consider a general case of dependent execution between two threads to emphasize the importance of measuring synchronization operations for correct perturbation analysis. In §4.3, §4.4, and §4.5 we present the perturbation analysis models of barrier, semaphore, and advance/await synchronization, respectively. In Chapter 5, we expand the perturbation analysis to larger program segments, mainly loops. Initially, however, we provide a brief background discussion on dependencies.

4.1 Dependence Preliminaries

A *data dependence* [97] is defined from the order of read and write accesses to a data variable in a sequential execution. During parallel execution, the sequential order of accesses must be maintained to insure correct data values. A data dependence is either a *flow dependence* (a variable is read after it has been written), an *anti-dependence* (a variable is written after it has been read), or an *output dependence* (a variable is written after previously written) [12, 197]. A *forward dependence* means that a statement S_j dependent on a statement S_i is syntactically after S_i in the program; the dependency is from S_i to S_j. A *backward dependence* means that a statement S_i dependent on a statement S_j is syntactically before S_j in the program; the dependency is from S_j to S_i. If a statement S_i is dependent on a statement S_j, we use the notation $S_i \rightarrow S_j$ to denote the dependency.

A *control dependence* [97] arises from the flow of control between statements. If the execution of S_j is affected by the execution of statement S_i, then a control dependence exists between statement S_i and statement S_j. We use the term *execution dependence* [197] to describe an execution-time ordering relationship between two events occurring on the same thread. If the event e_i occurs before the event e_j, then e_j is execution dependent on e_i, but only in the context of that particular execution.

4.2 Event Perturbation Modeling

In the previous chapter, the notion of a "time basis" was used to compute the approximate time of event occurrence. If an event x occurs before an event z on the same thread of execution, z is execution order dependent on x. For sequential execution, the approximated time that event x occurs, $t_a(x)$, could be used as the time basis for computing $t_a(z)$. However, during concurrent execution, the occurrence of an event z may depend on both intra-thread execution order dependencies *AND* inter-thread control and data dependencies. When intra- and inter-thread dependencies exist, choosing a time basis for resolving the approximated execution time of a dependent event is non-trivial.

Below, we reason about the constructive application of event-based perturbation analysis by first considering a simple set of general cases involving dependent execution behavior. In subsequent sections, we analyze more specific types of dependency synchronization. Throughout the discussion, event perturbation analysis should be viewed as a "process" that uses measured event information, plus a knowledge of event semantics, to derive an approximation to event occurrence times. In this respect, the accuracy of event perturbation analysis is limited not only by what can be measured, but also what can be inferred about execution behavior.

Consider the problem of approximating the time for an event z, $t_a(z)$, from two other events x and y. Events x and z occur on the same thread, T_2, and z is execution order dependent on x. The event y occurs on another thread, T_1. By assumption, z is also dependent on y. This dependency of z on y is input data for the event perturbation analysis, perhaps obtained from a data dependence analysis [12]. The measured execution times for x, y, and z are $t_m(x)$, $t_m(y)$, and $t_m(z)$, respectively. By assumption, the approximated execution times for x and y, $t_a(x)$ and $t_a(y)$, respectively, have been resolved previously. For convenience, we define the measured time between two events i and j, Δ_m^{i-j}, as

$$\Delta_m^{i-j} = |t_m(j) - t_m(i)|.$$

Similarly, we define the approximated time between two events i and j, Δ_a^{i-j}, as

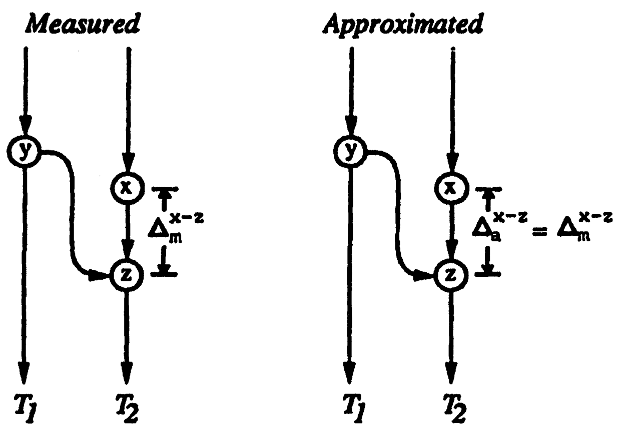
$$\Delta_a^{i-j} = |t_a(j) - t_a(i)|.$$

Figures 4.1, 4.4, 4.6, and 4.8 show the four cases that can occur in the perturbation analysis of events x, y, and z. Each case shows the measured and approximated timelines for threads T_1 and T_2. Time increases from top to bottom in each diagram.[3] The measured timelines reflect the relative execution ordering and timing of the events for the threads with instrumentation included. The approximated timelines reflect the execution time approximations as generated by the perturbation analysis. Because $t_a(x)$ and $t_a(y)$ are, by assumption, known, the problem is to determine $t_a(z)$. The four cases represent the four possible event orders in the measured and approximated executions.

4.2.1 Case One

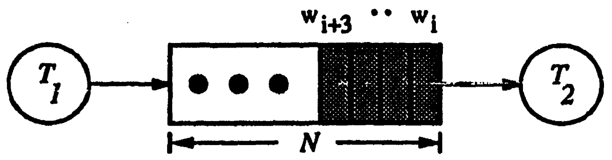
In case one, shown in Figure 4.1, the relative ordering of x and y is the same for both the measured and approximated timelines. Because y occurs before x, the time basis for z is

[3]The timelines are used only to show the ordering relationship of the three events; the measured and approximated time scales need not be the same.



$$t_a(z) = t_a(x) + \Delta_m^{x-z} - \alpha$$

Figure 4.1: Measured and Approximated Event Ordering: Case One



```
       program producer                program consumer
       i = 0                           i = 0
loop:  call produce(w)          loop:  call wait(x)
       call wait(y)                    call output(w)
       call input(w)                   call post(y)
       call post(x)                    call consume(w)
       call event(i)                   call event(i)
       i = i + 1                       i = i + 1
       goto loop                       goto loop
       stop                            stop
       end                             end
```

Figure 4.2: Producer-Consumer Example

independent of y. Thus, $t_a(z)$ must be computed using $t_a(x)$ as the time basis. Clearly, all computation that occurred between x and z in the measured execution must also exist in the approximated execution. The execution time of this computation, denoted by Δ_m^{x-z}, is used to approximate the time that event z occurred. That is,

$$t_a(z) = t_a(x) + \Delta_m^{x-z} - \alpha,$$

where α is the instrumentation overhead.

4.2.2 Case Two

By assumption, the events y and z reflect a dependent execution relationship between the two threads. However, they may or may not explicitly indicate the points of synchronization where thread T_2 tests the dependency and thread T_1 satisfies the dependency. A simple example will make this clear. Suppose our measurement captures the behavior of a producer-consumer computation [34]; see Figure 4.2. Thread T_1 produces items of work, w, which are placed in a buffer of length N. Thread T_2 consumes the items of work in the order of their placement. A counter x indicates the number of currently used positions in the buffer (initially 0) and a counter y the number of currently unused positions (initially N). The routines post and wait are used to control access to the buffer:

```
subroutine post(c)               subroutine wait(c)
integer c                        integer c
c = c + 1                 loop:  if (c.LE.0) return
end                              c = c - 1
                                 goto loop
                                 end
```

The producer generates events indicating that it has placed an item of work in the buffer; the event is designated by an item number, i. The consumer generates events indicating that it has consumed the work item; again, the event designates the specific item. The producer and consumer code are shown in Figure 4.2.[4]

Figure 4.3 shows a portion of a measured execution for the producer-consumer example for T_1 and T_2. The white nodes represent the captured events; e_i is the event representing the

[4]The input routine puts items in the buffer, the output routine removes items from the buffer, and the event routine logs the event to the trace for the respective thread.

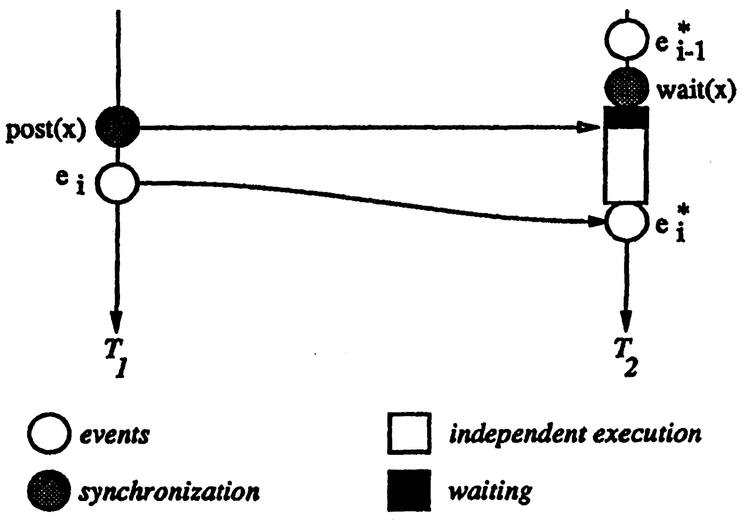


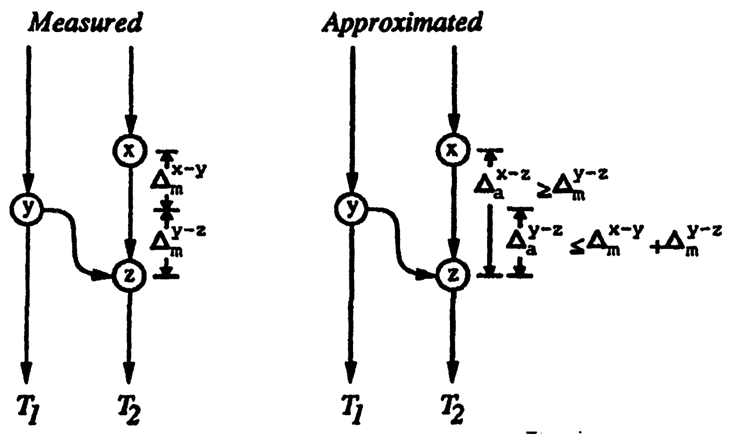
Figure 4.3: Producer-Consumer Execution

ith work item put in the buffer and e_i^* is its dependent event representing the ith work item consumed. The shaded nodes represent the synchronization operations for which no events are captured. The black region represents waiting time encountered in the measured execution. Unless the synchronization operations are also monitored, the perturbation analysis can only use measured information about the events e_i, e_{i-1}^*, and e_i^* in approximating the actual execution.[5]

We refer to un-monitored synchronization operations as *hidden dependencies*. For the following general cases, it is important to remember that the event perturbation analysis has no knowledge of the hidden dependencies and, thus, incomplete knowledge of the actual operational characteristics of dependency synchronization. Our goal is to understand what the perturbation analysis can conclude in these cases regarding the relative occurrence of event z in the approximated execution and what bounds can be placed on the accuracy of $t_a(z)$. We will use the producer-consumer example to illustrate our analysis.

In case two, shown in Figure 4.4, x precedes y in both the measured and approximated execution. Without knowing where the hidden dependence is tested on thread T_2 relative to where it is satisfied on T_1, the perturbation analysis must pessimistically assume that the dependence represented by y and z is satisfied by T_1 and tested by T_2 exactly at time $t_m(x)$. Why? Because

[5]The event e_i corresponds to the ith work item produced and the event e_i^* corresponds to the ith work item consumed.


Measured
Approximated
Δ_m^{x-y}
Δ_m^{y-z}
$\Delta_a^{x-z} \geq \Delta_m^{y-z}$
$\Delta_a^{y-z} \leq \Delta_m^{x-y} + \Delta_m^{y-z}$
x
y
z
T_1
T_2

$$\begin{aligned} t_a(z) &= t_a(y) + \Delta_a^{y-z} - \alpha \\ &= t_a(y) + \Delta_m^{x-y} + \Delta_m^{y-z} - \alpha \end{aligned}$$

Figure 4.4: Measured and Approximated Event Ordering: Case Two, Analysis

all computation represented by $\Delta_m^{x-y} + \Delta_m^{y-z}$ could be entirely independent, except for the synchronization test that would immediately succeed. This assumption is pessimistic because all this time must be accounted for in the approximation. However, in the approximated execution, it is possible that the dependency between y and z is not satisfied until time $t_a(y)$,[6] implying thread T_2 would wait from $t_a(x)$ to $t_a(y)$. Hence, in the pessimistic approximation, event y must be regarded as the time basis for z and $\Delta_m^{x-y} + \Delta_m^{y-z}$ work must be performed by thread T_2 after $t_a(y)$. Thus,

$$\begin{aligned} t_a(z) &= t_a(y) + \Delta_a^{y-z} - \alpha \\ &= t_a(y) + \Delta_m^{x-y} + \Delta_m^{y-z} - \alpha. \end{aligned}$$

Case two is demonstrated in the producer-consumer example shown in Figure 4.5. The synchronization for the hidden dependency shows how the pessimistic approximation can be in error. As required, the order of events e_{i-1}^* and e_i is the same in the measured and approximated executions. However, because

$$t_m(e_i) - t_m(e_{i-1}^*) < t_a(e_i) - t_a(e_{i-1}^*),$$

[6]Note, there is no information to indicate otherwise.

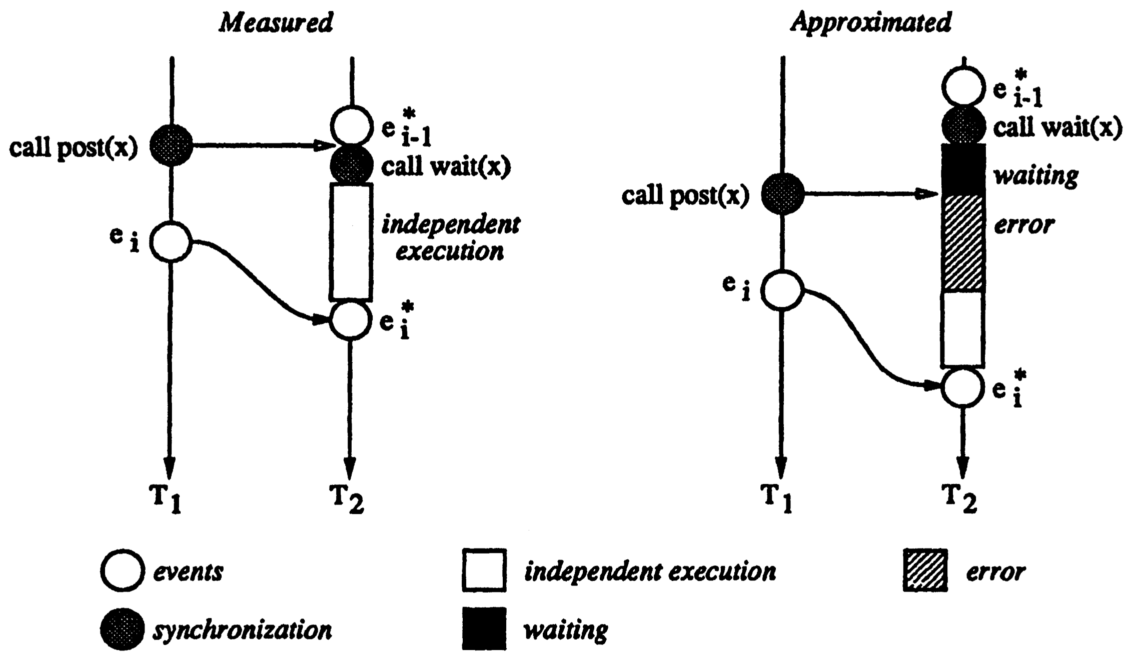


Figure 4.5: Measured and Approximated Event Ordering: Case Two, Example

the hidden dependency might have been satisfied (indicated by the post(x) synchronization) at the time shown in the approximated timeline for T_1. If events representing the hidden dependency had been captured, the waiting time indicated could be approximated. However, since these events are not captured, the pessimistic analysis must assume the post action does not occur until $t_a(e_i)$. In this example, $t_a(e_i^*)$ is over-approximated by the amount of the error shown.

As implied in the example, the actual time event z occurs relative to x and y can differ significantly from the pessimistic approximation. For instance, if $t_a(y) = t_a(x)$, the approximation could over-estimate the correct time by as much as Δ_m^{x-y}, if Δ_m^{x-y} was, in fact, waiting time in the measured execution. The bounds on the time approximations between the events are

$$\Delta_a^{x-z} \geq \Delta_m^{y-z}$$

and

$$\Delta_a^{y-z} \leq \Delta_m^{x-y} + \Delta_m^{y-z}.$$

4.2.3 Case Three

In cases three and four, the order of events x and y changes from the measured execution to the approximated execution. In case three, shown in Figure 4.6, event z's dependency on y is satisfied by time $t_m(x)$ in the measured execution, because the hidden dependency is satisfied before event y on thread T_1. However, the approximated times of x and y suggest that the approximated time of z should be based on $t_a(y)$. It is easy to imagine how the order of x and y can change from the measured to the approximated execution. If there is more instrumentation overhead prior to x on thread T_2 than before y on thread T_1, removing this overhead could result in the order of x and y being reversed in the approximation.

Should the time basis for $t_a(z)$ be $t_a(y)$? This depends on what the measured data indicates regarding where the hidden dependency test on T_2 actually occurs. If it can be determined that T_2 explicitly waits on T_1, $t_a(y)$ will be chosen as the time basis for $t_a(z)$.

Figure 4.6 shows two approximated timelines for case three; one for each of the two possible sub-cases. Notice that all computation performed by T_2 between x and z in the measured execution, as represented by Δ_m^{z-x}, must occur in the approximated execution; Δ_a^{z-x} in the approximated execution is at least Δ_m^{z-x}.

In sub-case A, the approximated time difference between x and y, Δ_a^{x-y}, is greater than the measured time between x and z, Δ_m^{z-x}. Waiting has to occur on T_2 between $t_a(x)$ and $t_a(z)$ to account for the extra time. Thus, event y is selected as the time basis for z. If the cost of testing a satisfied dependency is Δ_t, the approximated time of z's occurrence after y is given by the inequality

$$\Delta_t \;\leq\; \Delta_a^{y-z} \;\leq\; \Delta_m^{z-x}.$$

The inequality bounds Δ_a^{y-z} because no measured information is available to the perturbation analysis to indicate where the hidden dependency actually is satisfied. Thus, in sub-case A,

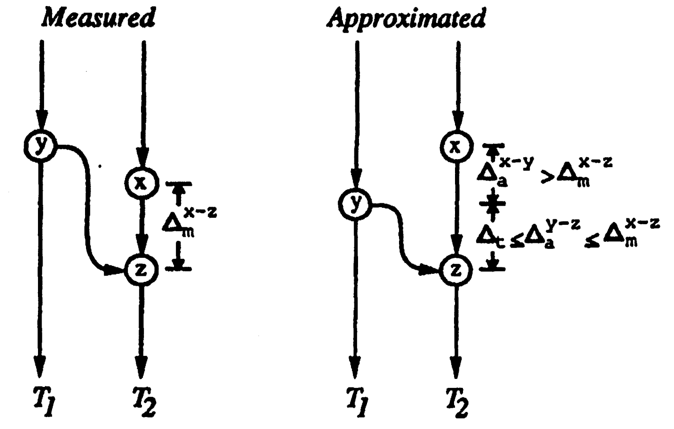
$$t_a(y) + \Delta_t \;\leq\; t_a(z) \;\leq\; t_a(y) + \Delta_m^{z-x} - \alpha.$$

Sub-case A is demonstrated in Figure 4.7 for the producer-consumer example. Because

$$|t_m(e_i) - t_m(e_{i-1}^*)| < |t_a(e_i) - t_a(e_{i-1}^*)|,$$

we use $t_a(e_i)$ as the time basis for event e_i^*. Unknowing when the post action occurs, it must be assumed to occur (under a pessimistic approximation) at $t_a(e_i)$. The example shows when

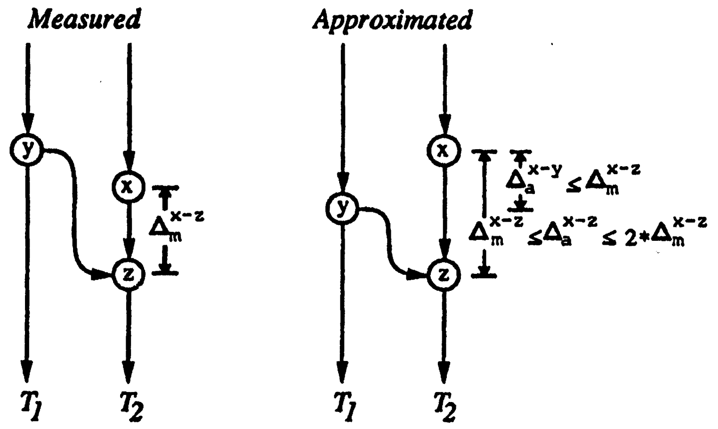
Sub-Case A



Measured
Approximated
y
x
z
Δ_m^{x-z}
$\Delta_a^{x-y} > \Delta_m^{x-z}$
$\Delta_t \le \Delta_a^{y-z} \le \Delta_m^{x-z}$
T_1
T_2

$$t_a(y) + \Delta_t \;\le\; t_a(z) \;\le\; t_a(y) + \Delta_m^{x-z} - \alpha$$

Sub-Case B



Measured
Approximated
y
x
z
Δ_m^{x-z}
$\Delta_a^{x-y} \le \Delta_m^{x-z}$
$\Delta_m^{x-z} \le \Delta_a^{x-z} \le 2*\Delta_m^{x-z}$
T_1
T_2

$$t_a(x) + \Delta_m^{x-z} - \alpha \;\le\; t_a(z) \;\le\; t_a(x) + 2\,\Delta_m^{x-z} - \alpha$$

Figure 4.6: Measured and Approximated Event Ordering: Case Three, Analysis

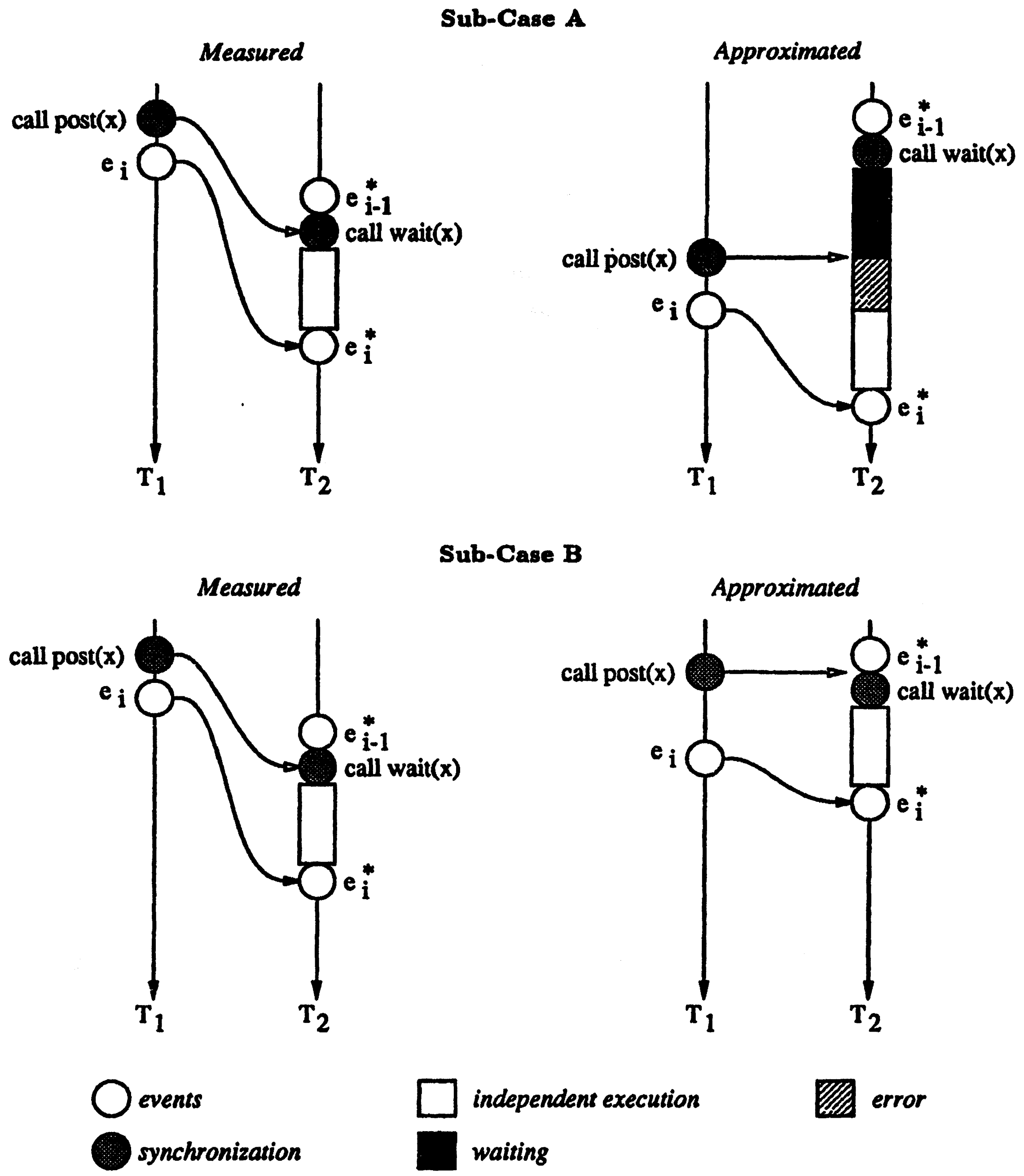


Figure 4.7: Measured and Approximated Event Ordering: Case Three, Example

the post might occur in the approximation and the error introduced in the approximation for $t_a(e_i^*)$.

In Figure 4.6, sub-case B selects event x as the time basis for z if the approximated time difference between x and y, Δ_a^{x-y}, is less than the measured time between x and z, Δ_m^{x-z}. Here, the perturbation analysis cannot conclude whether waiting occurs. If no waiting occurs, Δ_a^{x-z} is correctly estimated by Δ_m^{x-z}, the independent execution time in the measured execution. However, if waiting does occur, the approximated time difference between x and z can be as much as twice the measured execution time, $2\Delta_m^{x-z}$. How? Even though $\Delta_a^{x-y} < \Delta_m^{x-z}$, the hidden dependency might not be satisfied until immediately before $t_a(y)$ in the approximated execution. If the hidden dependency is tested immediately after event x, thread T_2 must wait in the approximation until after the hidden dependency is satisfied. Thus, the approximation of $t_a(z)$ in sub-case B is bounded by the inequality

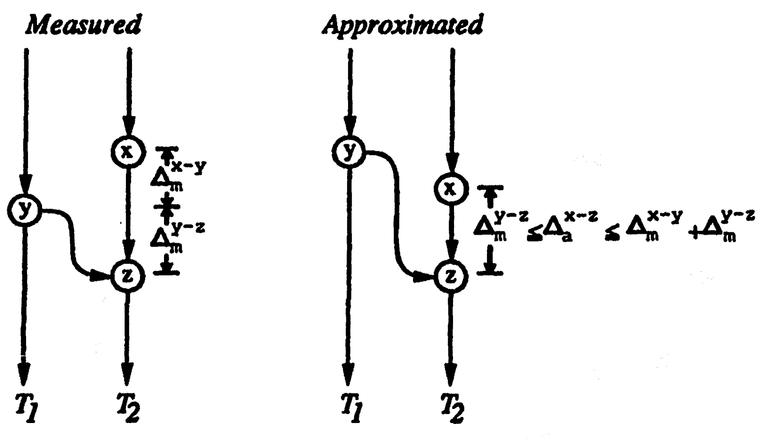
$$t_a(x) + \Delta_m^{x-z} - \alpha \;\leq\; t_a(z) \;\leq\; t_a(x) + 2\,\Delta_m^{x-z} - \alpha.$$

Sub-case B of case three is demonstrated in Figure 4.7. Although we have been describing pessimistic approximations, the perturbation analysis can also make optimistic approximations in the cases where $t_a(z)$ is bounded by an inequality. In this case, we have shown an optimistic approximation of sub-case B (i.e., it is assumed no waiting occurs).

4.2.4 Case Four

Finally, case four results when event y occurs after x in the measured execution but before x in the approximated execution; see Figure 4.8. The event x serves as the time basis in the approximation, but the estimate of Δ_a^{x-z} depends on when the hidden dependency between y and z is tested by T_2 (i.e., how much T_2 waits in the measured execution). Because Δ_m^{x-y} already includes any waiting time, Δ_a^{x-z} cannot be more than $\Delta_m^{x-y} + \Delta_m^{y-z}$. Similarly, because the hidden dependency between y and z is known to be satisfied by $t_m(y)$, Δ_a^{x-z} cannot be less than Δ_m^{y-z}.

In the approximated execution, no waiting occurs. If the hidden dependency represented by events y and z is tested at $t_m(x)$ and satisfied at $t_m(y)$, $\Delta_a^{x-z} = \Delta_m^{y-z}$; Δ_m^{x-y} would be waiting time in the measured execution. Conversely, if the hidden dependency is tested by T_2 at $t_m(z)$



Measured
Approximated
Δ_m^{x-y}
Δ_m^{y-z}
$\Delta_m^{y-z} \le \Delta_a^{x-z} \le \Delta_m^{x-y} + \Delta_m^{y-z}$
x
y
z
T_1
T_2

$$t_a(x) + \Delta_m^{y-z} - \alpha \;\le\; t_a(z) \;\le\; t_a(x) + \Delta_m^{x-y} + \Delta_m^{y-z} - \alpha$$

Figure 4.8: Measured and Approximated Event Ordering: Case Four, Analysis

and satisfied by T_1 at $t_m(x)$, no waiting occurs and

$$\Delta_a^{x-z} = \Delta_m^{x-y} + \Delta_m^{y-z}.$$

Thus, the approximation $t_a(z)$ is bounded by the inequality

$$t_a(x) + \Delta_m^{y-z} - \alpha \;\le\; t_a(z) \;\le\; t_a(x) + \Delta_m^{x-y} + \Delta_m^{y-z} - \alpha.$$

Figure 4.9 demonstrates case four in the producer-consumer example. Because the analysis assumes that no waiting occurs in the measured execution, all of the time $t_m(e_{i-1}^s) - t_m(e_i^s)$ must accounted for in a pessimistic approximation. However, as shown, undetected waiting does occur, introducing an error in the approximation.

4.2.5 General Synchronization Events

The uncertainty in the approximations of the time that event z occurs in cases two, three, and four above reflects the lack of "measured" synchronization events that resolve the assumed dependency between y and z. Thread T_2 must test this dependency somewhere between events x and z, but the event perturbation analysis has no additional information to determine the test location. Given the boundaries of the dependency test on thread T_2 (i.e., when the dependency is satisfied), more accurate time approximations might be possible. However, obtaining this information requires additional instrumentation.

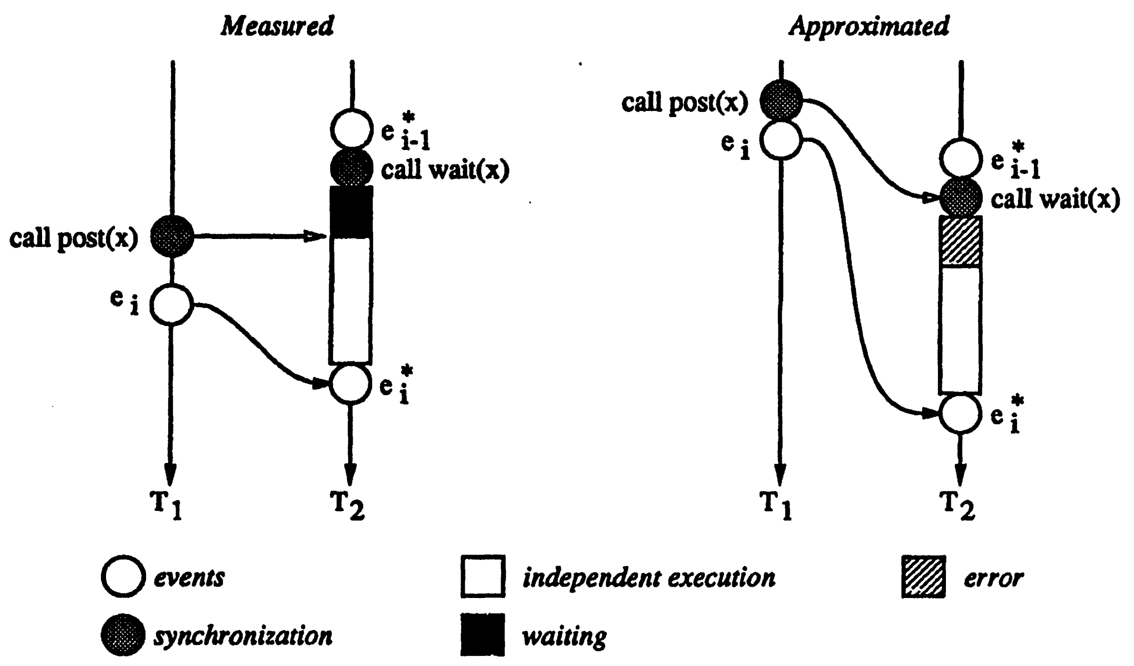


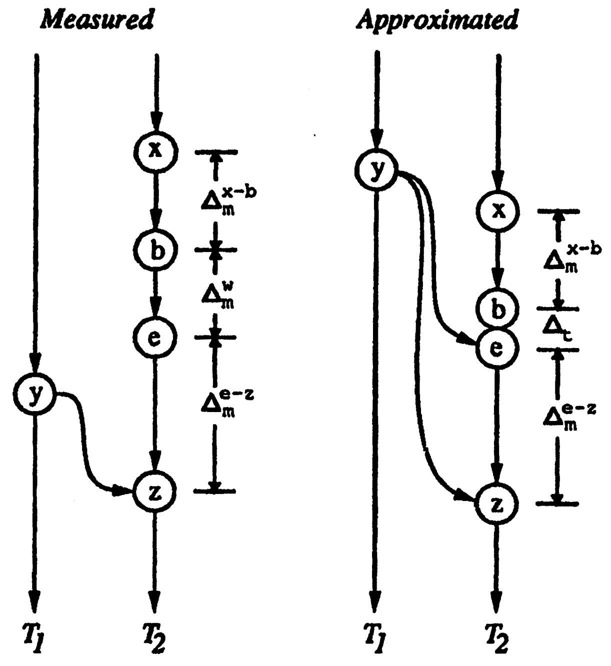
Figure 4.9: Measured and Approximated Event Ordering: Case Four, Example

In the framework of the general cases presented above, assume that there are two additional events that bound the dependency testing between y and z (i.e., events b and e). Figure 4.10 shows general case four with these events included in the measurement. The measured amount of synchronization waiting is denoted by Δ_m^w.

The added waiting events allow the time approximation of z, $t_a(z)$, to be quantified by a single value. Because the waiting time is known, the perturbation analysis can determine the amount of "independent" work on thread T_2 between events x and z that must be completed in the approximated execution. The occurrence of x after y in the approximated execution implies that the dependency test will immediately succeed. Thus, the time Δ_m^w can be removed when approximating $t_a(z)$. That is,

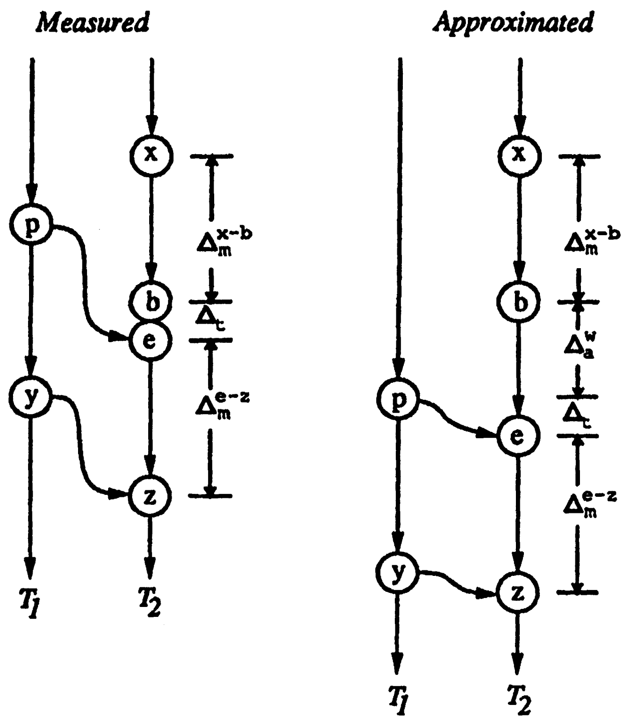
$$t_a(z) = t_a(x) + \Delta_m^{x-b} + \Delta_m^{e-z} + \Delta_t - \alpha.$$

Instrumenting only for dependency waiting is not sufficient to remove all inaccuracies in event-based perturbation analysis approximations. Even with events b and e measured in the general case two, $t_a(z)$ would still be bounded by an inequality. The problem is clearly seen in the example in Figure 4.4; unless the perturbation analysis can determine where the



$$t_a(z) = t_a(x) + \Delta_m^{x-b} + \Delta_m^{e-z} + \Delta_t - \alpha$$

Figure 4.10: Case Four with Dependency Testing Events



$$t_a(z) = t_a(p) + \Delta_t + \Delta_m^{e-z} - \alpha$$

Figure 4.11: Case Two with Dependency Post and Wait Events

post operation occurs in the approximated execution, it must assume T_2 waits until $t_a(e_i)$. Thus, in addition to waiting events, there must be measurements of when dependencies have been satisfied. We let event p designate the synchronization "posting" action that indicates a dependency has been satisfied. Figure 4.11 shows the general case two with both post and wait instrumentation added.

In the measured execution, the dependency is satisfied on thread T_1 before it is tested on T_2, and no waiting occurs. In the approximated execution, the post event occurs after the dependency testing begins. Waiting occurs on thread T_2 until the dependency is satisfied. Although the wait instrumentation can be used to determine the amount of computation before and after the synchronization and, therefore, the relative offset of event z from event e, it cannot fix $t_a(z)$ at a single point in time. However, the resolution of the post event p in approximated

time does provide the time basis for the resolution of e and, hence, z. That is,

$$t_a(z) = t_a(p) + \Delta_t + \Delta_m^{e-z}.$$

The use of post events to get tighter estimates of $t_a(z)$ in case two applies directly to case three.

Intuitively, using more instrumentation to increase the accuracy of event-based perturbation analysis conflicts with the general notions of the instrumentation uncertainty principle; mainly, that measurement volume and accuracy are contrary. Although the additional instrumentation used to capture dependency synchronization actions will perturb measured, and therefore approximated behavior, it makes it possible to resolve certain characteristics of dependent execution behavior, such as the amount of time spent waiting. This was demonstrated in the general cases above. In Chapter 5, we give experimental results that validate this claim.

4.3 Barrier Perturbation Analysis

Based on the general cases of dependent events, to derive accurate approximations of a concurrent execution from its measured performance, the event-based perturbation analysis requires measurements of synchronization actions. Furthermore, the perturbation analysis must understand the semantics of the synchronization operations so that the measured data can be applied correctly in the approximated time domain. There are many possible forms of synchronization operations that could be analyzed [6]; we consider three: barrier, semaphore, and advance/await synchronization. These were chosen because they represent a cross-section of synchronization alternatives. Our goals in analyzing these three forms of synchronization are, first, to determine what measurements must be made to apply perturbation analysis, and, second, to understand the approximation capabilities of the perturbation analysis techniques. We begin with barrier synchronization.

4.3.1 Barrier Operation

Simply, a *barrier* is a piece of code used to synchronize multiple threads of execution at a single point in time. Each thread participating in a barrier synchronization will first enter the barrier, wait for all the other threads to arrive, and then exit the barrier with all other threads. The unique feature of the barrier is that all threads will block until the last thread

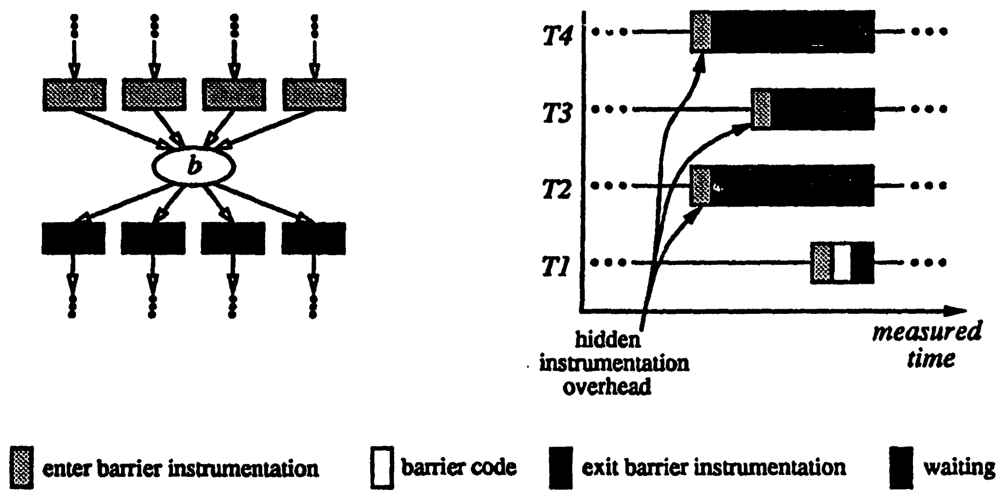

b
T4
T3
T2
T1
hidden instrumentation overhead
measured time
enter barrier instrumentation
barrier code
exit barrier instrumentation
waiting

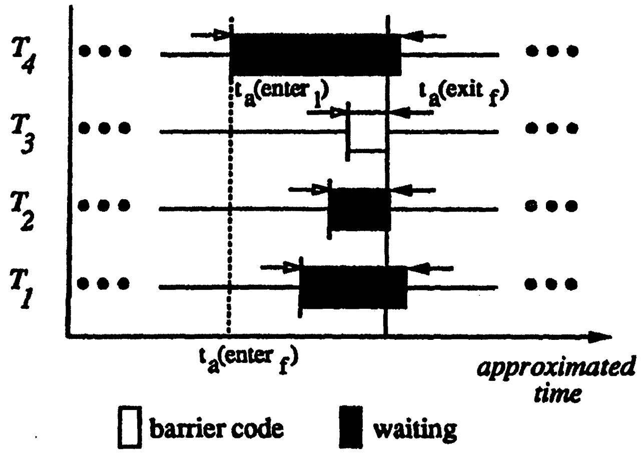
Figure 4.12: Barrier Synchronization of Four Threads with Instrumentation

enters, establishing a point in the computation where the states of all threads participating in the barrier synchronization are known. This point occurs when the barrier synchronization has been satisfied and all threads are allowed to proceed.

4.3.2 Barrier Instrumentation

The performance instrumentation of a barrier should allow one to determine the sequence of thread arrivals at the barrier, the waiting time of each thread, and the time the threads exit the barrier [9, 11]. This analysis requires capture of two barrier events for each thread: *enter* and *exit*. The *enter* event is recorded immediately after a thread enters the barrier and the *exit* event is recorded immediately before the thread exits the barrier. From the standpoint of perturbation analysis, we are primarily interested in the *exit* events — these establish a time basis for all following events on each thread; see below. However, the *enter* events are also important for barrier performance analysis.

Consider Figure 4.12 which shows four threads synchronizing at a barrier *b*. When a thread reaches the barrier, it executes instrumentation corresponding to barrier entry (indicated by light shading) before executing the barrier code. After the last thread reaches the barrier and the threads have synchronized, they all execute instrumentation code corresponding to barrier exit (indicated by darker shading) before continuing along their separate execution paths.



$$
\begin{aligned}
t_a(exit_f) &= t_a(enter_l) + t_m(exit_f) - t_m(enter_l) - \alpha \\
t_a(exit_j) &= t_a(exit_f) + t_m(exit_j) - t_m(exit_f) - \alpha
\end{aligned}
$$

Figure 4.13: Barrier Performance Approximation

The barrier events are recorded in a trace for each thread. To uniquely identify a particular barrier, we assume the *entry* and *exit* events recorded are additionally typed with this information. The ability to distinguish different barriers is required by the perturbation analysis.

The measured timeline shows an example of how the threads might have executed at the barrier. Diagrammatically, we represent the barrier's synchronization code executed after the last thread reaches the barrier by □. The enter and exit barrier instrumentation and the waiting time on each thread are shown. The timeline reflects only the measured execution behavior. It is clearly possible that instrumentation on each thread prior to the barrier can affect the order that threads arrive. Thus, the timing relationships shown between the *entry* and *exit* events may not be representative of actual barrier behavior.

4.3.3 Barrier Approximation

The perturbation analysis of the instrumented barrier is based solely on the *exit* events. The *enter* events are used to approximate certain performance characteristics of the barrier, but it is the *exit* events that are used to resolve the time approximations of future events. Figure 4.13 shows a possible timeline of the approximated barrier execution. We assume the *enter* events for each thread, whose approximated time of occurrence is represented by the white arrows, have been previously resolved by perturbation analysis. The approximated time the first thread enters the barrier is denoted by $t_a(enter_f)$ and the approximated time the last thread enters the barrier is denoted by $t_a(enter_l)$.

The perturbation analysis problem is to approximate the barrier exit behavior. By assumption, all threads are known to be present at the barrier at approximated time $t_a(enter_l)$. The approximated execution time of the barrier code after this point is given by

$$t_m(exit_f) - t_m(enter_l) - \alpha,$$

where $t_m(exit_f)$ is the measured time the first thread exits the barrier, and $t_m(enter_l)$ is the measured time the last thread enters the barrier. The approximated time the first thread exits the barrier is

$$t_a(exit_f) = t_a(enter_l) + t_m(exit_f) - t_m(enter_l) - \alpha.$$

Because we use $t_m(exit_f)$ in computing $t_a(exit_f)$, the thread first to exit the barrier in the measured execution is the same one that exits first in the approximated execution. The approximated time the jth thread exits the barrier after the first is given by

$$t_a(exit_j) = t_a(exit_f) + t_m(exit_j) - t_m(exit_f) - \alpha,$$

where the event $exit_j$ is obviously the *exit* event of the jth thread. The ordering relationships of barrier exit in the measured execution are maintained in the approximated execution with $t_a(exit_f)$ serving as the time basis for all other *exit* event approximations.

The barrier perturbation analysis assumes that the overhead for executing the barrier code remains unchanged from the measured to the approximated execution. Furthermore, the analysis assumes that approximated barrier performance depends solely on the approximated entry times of the threads and the measured barrier code overhead. In general, the execution of the barrier code can take variable time, depending on the barrier's implementation [9]. For instance, if a single lock is used to control access to a counter indicating the number of threads which have entered the barrier, the lock becomes an access "hot spot" [151] in the computation. The performance of barrier code implemented in this manner will depend on the degree of lock contention; in general, the higher the contention, the poorer the performance. Because the measured thread arrival behavior can be different from the actual behavior, there can be errors in the perturbation analysis because of differences in barrier code performance. These errors can be both positive (over-approximations due to poorer barrier code performance in the measured execution) and negative (under-approximations due to poorer barrier code performance during the actual execution). Ideally, the barrier performance would be modeled by the perturbation analysis based on known performance behavior of the barrier implementation. The barrier entry event times for each thread would provide barrier entry timings to the model.

4.4 Semaphore Perturbation Analysis

The perturbation analysis of barrier synchronization could be easily understood because the barrier creates a global synchronization point. If all threads meet at a barrier synchronization, the perturbation analysis of events after the barrier will not be affected by the errors resulting from the perturbation analysis of events prior to the barrier. In effect, the barrier provides event-based perturbation analysis with a means of partitioning the computation between barrier synchronizations and isolating perturbation analysis errors to individual partitions.

Not all synchronization operations are as well behaved. In particular, the semaphore represents one of the most primitive forms of synchronization, and hence one of the least restrictive. The semaphore establishes a synchronization relationship between only two events and, thus, does not have the global synchronization ramifications of the barrier. Furthermore, the pairing of semaphore synchronization events can change from actual to measured execution.

4.4.1 Semaphore Operation

The basic *semaphore* embodies the most recent history of two operations: **P** and **V**. The **P** operation signals that some execution dependency has been satisfied. The **V** operation checks the state of the semaphore to determine whether a dependency has been satisfied. The semaphore's state indicates only whether the last operation on the semaphore was at **P** or a **V**. If it was a **P**, the next **V** operation will not wait. Otherwise, the next **V** operation will wait. If S is a semaphore whose value can be P or V, the semantics of the **P** and **V** synchronization operations are shown below:

```
P(S):  $S = P$
V(S):  if ($S$ equals $P$)
           $S = V$
       else
           wait until $S$ equals $P$
           $S = V$.
```

More complex semaphores, such as counting semaphores, maintain additional history of P and V operations as well as allow multiple threads to wait. From a perturbation analysis perspective, we will consider only the basic semaphore case. All the problems we encounter with this case also appear in the more complex semaphore types.

4.4.2 Semaphore Instrumentation

There are five important events that must be monitored in the semaphore's operation: the P event, the V_s event, the V_w event, the V_p event, and the V_e event. The P event corresponds directly to the P operation on the semaphore and is recorded immediately after the P operation is performed. The V_s event is recorded immediately before V code is executed and signifies that the semaphore state is about to be tested. If the V operation blocks, the V_w events is recorded before the thread begins to wait; otherwise, the V_p event is recorded before the semaphore state is set to V. Finally, the V_e event is recorded immediately before a thread performing a V operation leaves the semaphore.

Because multiple semaphores may be active in a program, to correctly identify the P and V operations on a particular semaphore during perturbation analysis, information uniquely identifying the semaphore must must be recorded with the P and V events.

4.4.3 Semaphore Approximation

The perturbation analysis of a semaphore's instrumentation is concerned not so much with the removal of instrumentation overhead that comes with recording the P and V events, although this is important to achieve timing accuracy, but rather with identifying anomalous semaphore operation that might occur in the approximated execution as a result of perturbation analysis errors. In fact, this is the main reason for studying the semaphore form of synchronization.

Under a limited set of assumptions regarding how a semaphore is used during execution, we describe several cases where questions arise concerning how event-based perturbation analysis should resolve approximated semaphore behavior. To simplify the discussion, we assume in the following cases that an equal number of **P** and **V** operations are performed on each semaphore.

4.4.3.1 Single-P, Single-V

If a semaphore is used by only two threads, every **V** operation will be "satisfied" by a unique **P** operation; success of the *i*th **V** operation will depend only on the *i*th **P** operation. We refer to such a semaphore as a ***single-P, single-V*** semaphore. The P and V events for this semaphore can be matched explicitly by the perturbation analysis based on their order of occurrence. For each *i*th P-V pair, the time-order relationship of the P and V events must be maintained in the approximated execution.

Figure 4.14 shows three different approximated executions that can result from the perturbation analysis of a single-P, single-V semaphore. The left graph in cases A and B shows the ordering relationship of the P and V events in the measured execution. The right graph shows the events in the approximated execution. In case A, no waiting is encountered at the semaphore in the measured execution because $t_m(P) < t_m(V_s)$. However, in the approximated timeline, $t_a(V_s) < t_a(P)$ and tread T_2 must wait. We assume events V_s and P have been resolved in the approximated execution and that $t_a(V_p)$ and $t_a(V_e)$ must be determined. There is no information from the measured execution to indicate how long after event P the event V_p occurs in the approximated execution, so we must assume it is immediate.[7] This establishes $t_a(V_p)$ as the time basis for approximating $t_a(V_e)$. Thus,

$$t_a(V_e) = t_a(V_p) + t_m(V_e) - t_m(V_p).$$

The amount of waiting time can be calculated as

$$t_a(V_p) - t_a(V_s) + (t_m(V_p) - t_m(V_s)).$$

In case B, the opposite conditions exist. That is, waiting occurs in the measured execution but not in the approximated execution. Instead of the event V_p indicating the **V** operation

[7]Performance measurements of semaphore operation could be applied here to establish a minimum separation between P and V_p.

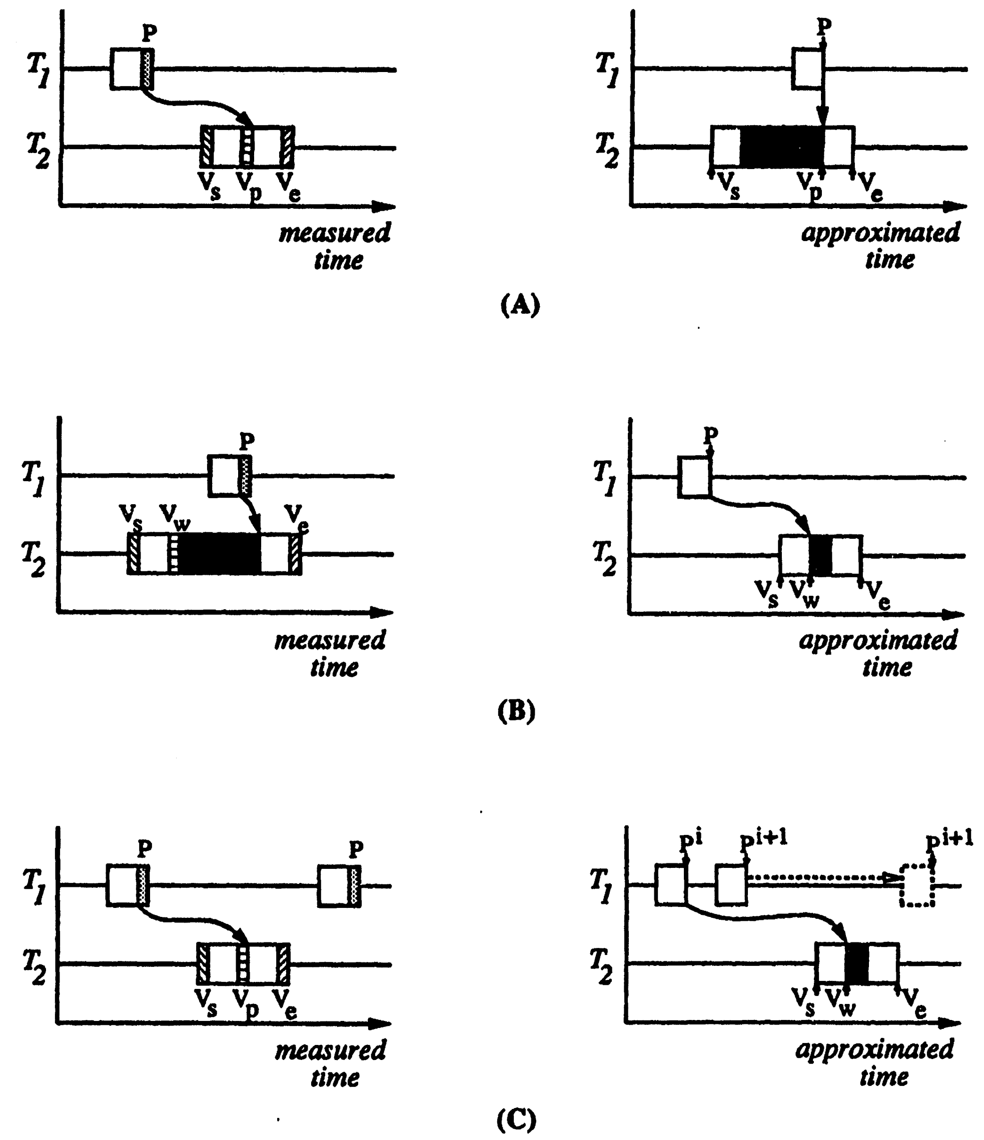


Figure 4.14: Single-P, Single-V Perturbation Analysis

succeeded, we are interested in when waiting begins. The V_w event provides a time basis for $t_a(V_e)$ in the approximation. We can calculate from the measured execution how long after event P the event V_e occurs in the approximated execution. From this value, the approximated time of V_e is given by

$$t_a(V_e) = t_a(V_w) + t_m(V_e) - t_m(P).$$

Because the approximated time of the P is not affected by semaphore waiting, it is possible that the situation in case C may be encountered due to perturbation analysis errors. As shown, two successive P events, P^i and P^{i+1}, are approximated to occur prior to the ith V events. This violates the assumed operation of the semaphore. There are two recourses if this situation occurs. The perturbation analysis could halt, indicating that a violation of semaphore execution semantics has occurred. A less abrupt action logs the violation, adjusts the approximated time of P^{i+1} to be immediately after $t_a(V_e)$ (see figure), and continues the perturbation analysis.

Noticc that the situation presented in case C cannot be true of the approximated time of the V events. That is, given two successive V_e events in the trace, V_e^i and V_e^{i+1}, and the ith P event, P^i,

$$t_a(P^i) < t_a(V_e^{i+1}).$$

The approximated time $t_a(V_e^{i+1})$ will always be adjusted by the perturbation analysis to be greater than P^i.

4.4.3.2 Multiple-P, Single-V

During execution, a V operation will always be satisfied by one particular P operation — either the one that occurred most recently (if S equals P), or the next one to occur. During perturbation analysis for any of the following semaphore cases, once the P event has been identified for a particular V event, the approximation techniques discussed for the single-P, single-V semaphore case can be applied. Hence, we will not reiterate these details but instead will focus on those situations where it is unclear how the perturbation analysis should proceed.

Figure 4.15 shows the two different approximations that can result from the perturbation analysis of a *multiple-P, single-V* semaphore. This semaphore type allows multiple threads to perform P operations but only one thread can perform V operations. As before, it is assumed

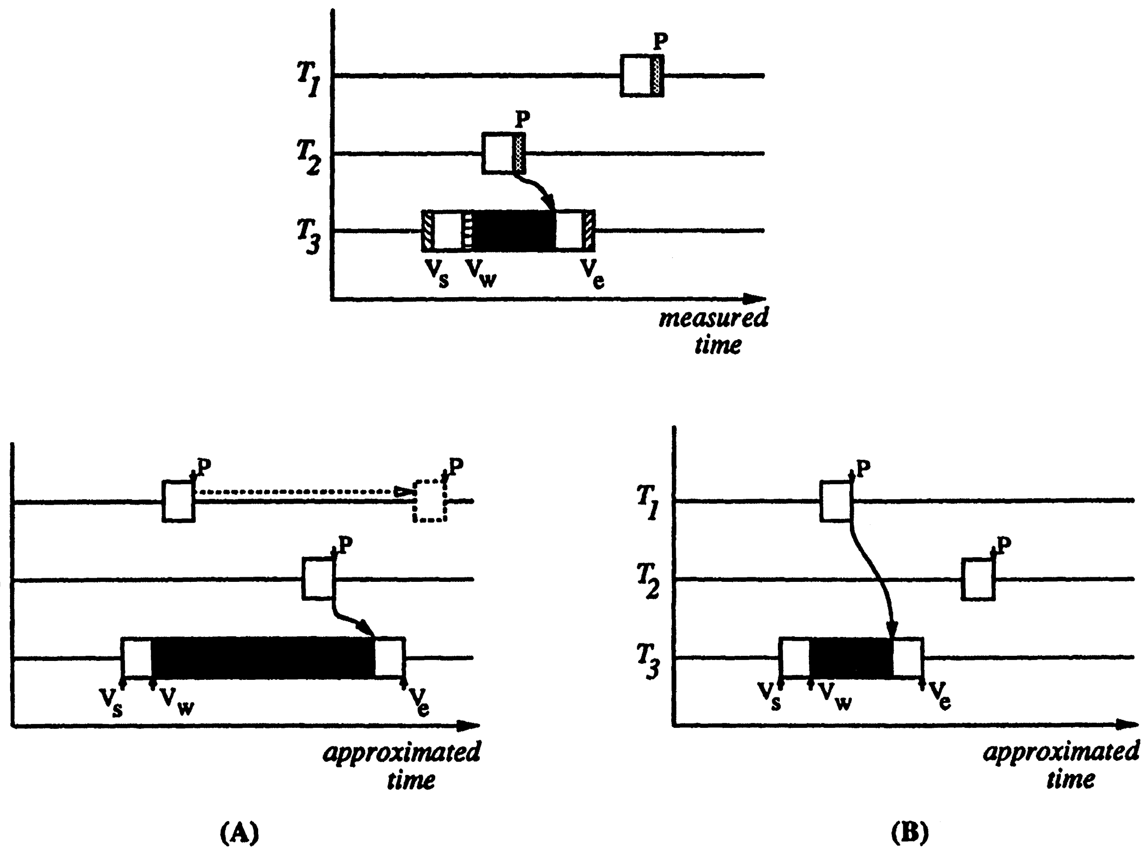


Figure 4.15: Multiple-P, Single-V Perturbation Analysis

that the same number of **P** and **V** operations are performed. Therefore, the ith **V** operation is satisfied by the ith **P**.

In the measured execution timeline, we see the **V** operation on thread T_3 satisfied by the **P** operation on thread T_2 as indicated by the arrow. If the **P** operation on thread T_2 precedes the **P** operation on thread T_1 in the approximated execution (as determined by the P and V events), thread T_2's **P** operation continues to satisfy T_3's **V**. However, if the order of the two **P** operations is reversed in the approximated execution, there is a question about which of the **P** operations should satisfy the **V**.

In the approximated execution for case A, the pairing of **P** and **V** operations determined from the measured execution has been maintained by the perturbation analysis; the **P** operation on T_1 is matched with a future **V** operation on T_3. Because the perturbation analysis can measure the time between T_2's P event and $t_m(V_e)$, it can apply this in the estimation of $t_a(V_e)$. However, the existence of two **P** operations before one **V** operation completes in the approximated execution violates the assumptions of semaphore operation. Perturbation analysis can be discontinued when this violation is detected, as in the single-P, single-V case. In order to continue perturbation analysis, the time of occurrence of P on T_1 must be adjusted to be after $t_a(V_e)$, as shown.

Alternatively, the perturbation analysis could emulate the semaphore's operation and match **V** operation with the earliest **P** operation on any thread that satisfies it. In case B of Figure 4.15, the P event on thread T_1 is used as the time basis to calculate $t_a(V_e)$. Because the **P** operation on thread T_1 is not matched with the **V** operation in the measured execution, the time $t_a(V_e) - t_a(P^i)$ must be estimated by $t_a(V_e) - t_a(P^j)$, where P^i is the P event on T_1 and P^j is the P event on T_2.

Although the approach followed in case B avoids violations in semaphore operation, it implicitly assumes that the computation following V_e on T_3 is independent of which threads' **P** operation satisfied the **V** operation. In general, no additional information is known from the measurement that indicates that this assumption is valid. Devoid of any additional knowledge from the user, the perturbation analysis cannot assume that the measured computation after V_e is independent of the measured P-V pairing without running the risk of introducing significant errors in the approximation. However, it is not unreasonable to think that additional knowledge might be provided to the event perturbation analysis indicating that this approach is valid.

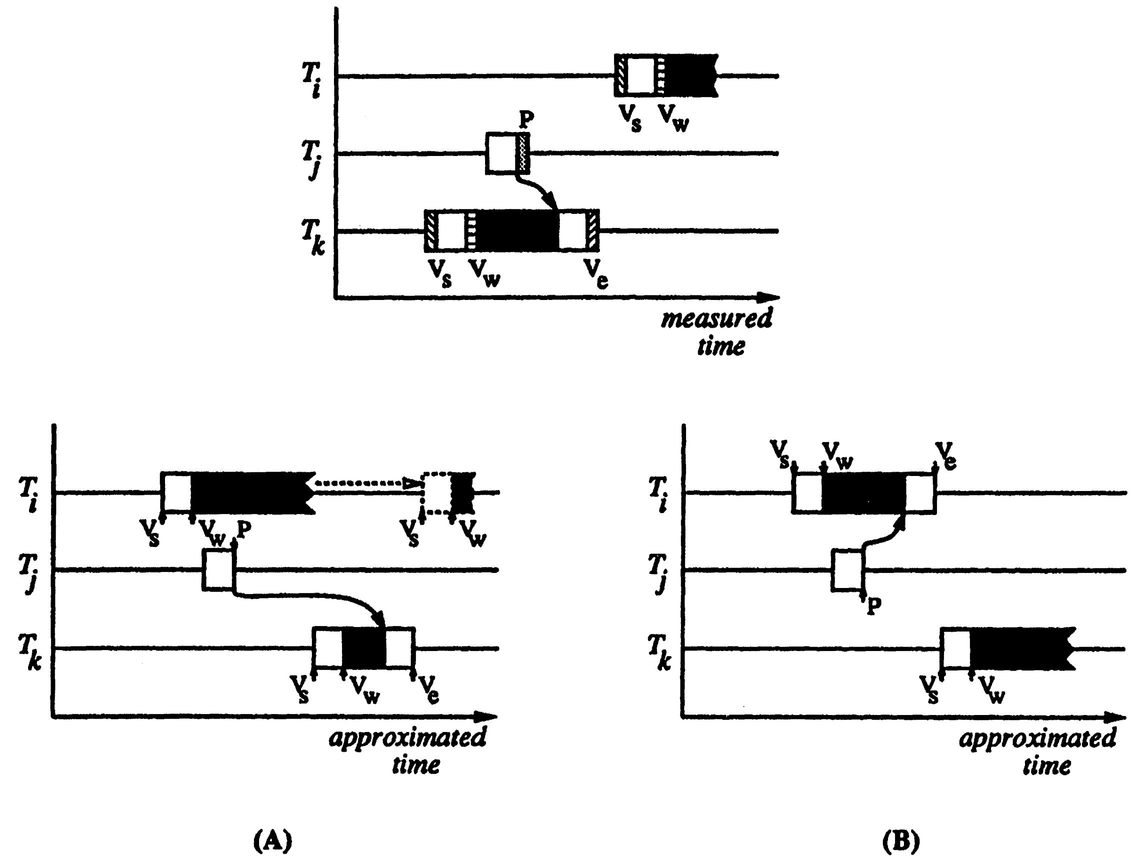


Figure 4.16: Single-P, Multiple-V Perturbation Analysis

4.4.3.3 Single-P, Multiple-V

If multiple threads perform V operations on a single semaphore with only one thread performing the P operation, we have the *single-P, multiple-V* semaphore case. Figure 4.16 shows two examples of an approximated execution involving a single-P, multiple-V semaphore. As in the case of the P operations in the multiple-P, single-V semaphore case above, if the ordering relationships of the V operations are maintained in the approximated execution of the single-P, multiple-V semaphore, the perturbation analysis of the single-P, single-V semaphore will apply. However, if the order has changed, as in the examples shown, the perturbation analysis must resolve the *P-V* pairing in the approximated execution.

Case A shows the measured P-V pairing maintained in the approximation. However, we again have the problem of a perceived violation of the semaphore's operation. In this case, the semaphore does not function as a first-come, first-serve system; the V operation on T_1 waits until after the V operation on T_3 has been satisfied, even though it appears first in approximated time. To correct this violation, the perturbation analysis must adjust the time the event V_e occurs on T_1 to be after $t_a(V_e)$ on T_3, as shown in the figure.

Case B adheres to the first-come, first-serve semaphore operation by matching the P event on T_2 with the V events on T_1. As in case B of the multiple-P, single-V semaphore, the perturbation analysis makes the implicit assumption that the computation on T_1 after event V_e is independent of the P-V pairing relationship in the measured execution. This assumption could cause the approximation execution to misrepresent actual execution behavior. Only in the case additional knowledge about the execution behavior is provided should the perturbation analysis proceed on this assumption.

4.4.3.4 Multiple-P, Multiple-V

Finally, the *multiple-P, multiple-V* semaphore case shown in Figure 4.17 demonstrates the variations in approximated execution that can occur under different assumptions of P-V reordering. The *multiple-P, multiple-V* semaphore allows several threads to perform P and V operations. The possible approximated execution characteristics of the multiple-P, single-V and single-P, multiple-V semaphores are combined in the multiple-P, multiple-V semaphore.

In case A, the measured P-V matching is maintained by perturbation analysis, thereby estimating a significant waiting period on thread T_4 until its V_e event occurs. The violation of the operational semantics of the semaphore by the P event on T_2 is clearly apparent, unless the approximated time of occurrence of P on T_2 is adjusted as shown. In case B the perturbation analysis matches the P events with the V events in the order of approximated execution occurrence. Although this makes it possible to reduce the waiting time on T_4 by approximating an earlier execution time for event V_e, the perturbation analysis again proceeds on the assumption that this re-assignment of P and V events is acceptable execution behavior. Again, the user must provide explicit directives to the perturbation analysis that such techniques can be safely applied.

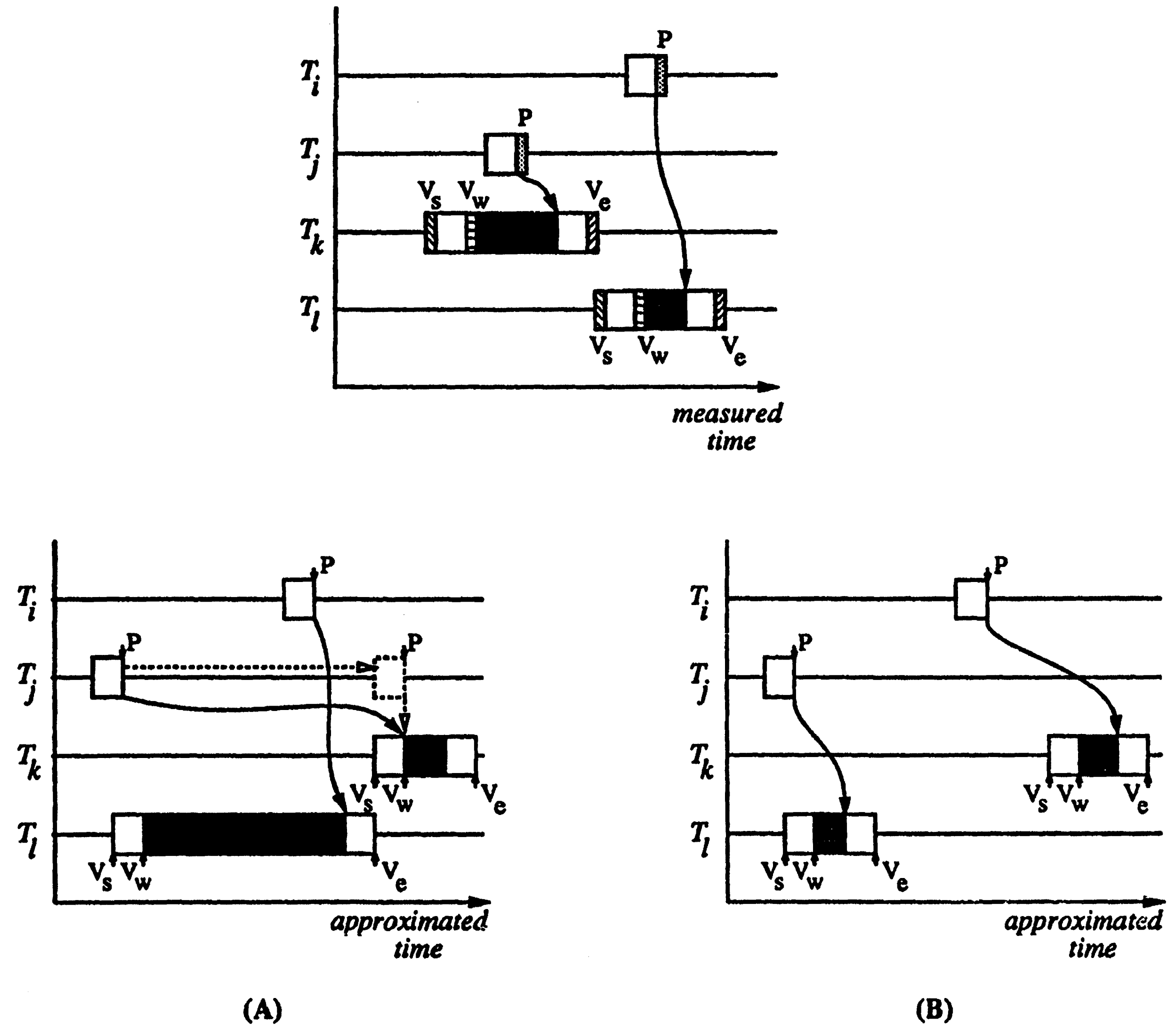


Figure 4.17: Multiple-P, Multiple-V Perturbation Analysis

4.5 Advance/Await Perturbation Analysis

As observed in the perturbation analysis of semaphores involving multiple P and/or V operations, reordering measured *P-V* event pairs in the approximated execution fundamentally depends on what is known about the execution behavior between concurrent threads. This knowledge cannot be measured from the execution but must be supplied externally during the perturbation analysis process. The *advance/await* form of synchronization makes explicit the "post" and "wait" actions involved in the synchronization. Because the synchronization is explicit, the perturbation analysis cannot reassign *advance* and *await* events in the approximation. However, the concurrent work constrained by these operations might be scheduled differently in the actual execution than what is observed from the measured events. The perturbation analysis does not know *a priori* that work reassignment to threads is allowed for the same reasons governing *P-V* event reassignment. However, the use of external execution information to reassign the work bounded by *advance* and *await* events during perturbation analysis can lead to significant differences in approximated execution behavior.

4.5.1 Advance/Await Operation

The advance/await form of synchronization is a special case of the general single P, single V semaphore. Each await operation synchronizes with a unique advance operation. Each advance/await operation pair can be thought of as operating on a unique semaphore. A general advance/await synchronization variable, A, stores the history of advance operations. The semantics of the advance and await operations are shown below:[8]

```
advance(A, i):  mark in A that i was advanced
  await(A, i):  if (i has not been advanced in A)
                   wait until i has been advanced
```

4.5.2 Advance/Await Instrumentation

To correctly identify which advance and await operations should be paired during perturbation analysis, *advance* and *await* events must be recorded with a unique value identifying the pair.

[8]Our definition of the advance and await operations is more general than what is often described. However, the perturbation analysis discussion still applies in the more restrictive case.

Typically, advance/await synchronization is used to control the execution of loops with iteration dependencies [38, 39]. The unique identifier in this case might be the loop iteration index. In general, the instrumentation for capturing the *advance* and *await* events must generate the unique identifier itself. From the semantics of the **advance** and **await** primitives above, this unique identifier could be the argument i.

4.5.3 Advance/Await Approximation

One interesting aspect of advance/await synchronization is that the pair of threads synchronizing can change dynamically during execution. Although the perturbation analysis of the single P, single-V semaphore applies directly to each pair of *advance* and *await* events, on which thread(s) the events occur is an artifact of the measured execution; the same thread could even perform both the **advance** and the **await** operations.

Because of the strict synchronization enforced by the **advance** and **await** operations, a partial ordering of these actions can be determined directly from the measured *advance* and *await* events. Independent of how **advance** and **await** operations are assigned to threads, this partial ordering must be maintained in the approximated execution. Figure 4.18 shows part of a measured computation involving advance/await synchronization. The arrows represent the ordering relationships between different pieces of work imposed by the **advance** operations, shown by light shading in the figure, and the **await** operations, shown by darker shading.

Without knowledge of the scheduling policy for assigning work to threads, the perturbation analysis must maintain not only the partial order of the *advance* and *await* events but also the event-to-thread assignment. This is shown in the approximated execution graph A. Obviously, due to prior perturbation removal, the relative ordering of the pieces of work, as well as the waiting delays, may change from the measured execution. In case A, because of the partial ordering of work and the fixed scheduling assumption, thread T_3 must wait until work u completes on T_1 before executing work x. Likewise, thread T_2 must wait for T_3 to complete work x.

If additional information is made available to the perturbation analysis indicating that the work bounded by advance/await synchronization can be assigned to any available thread as long as the partial order is maintained, alternative scheduling strategies could be applied in the approximation to reduce waiting delays. Case B is an approximation generated when the

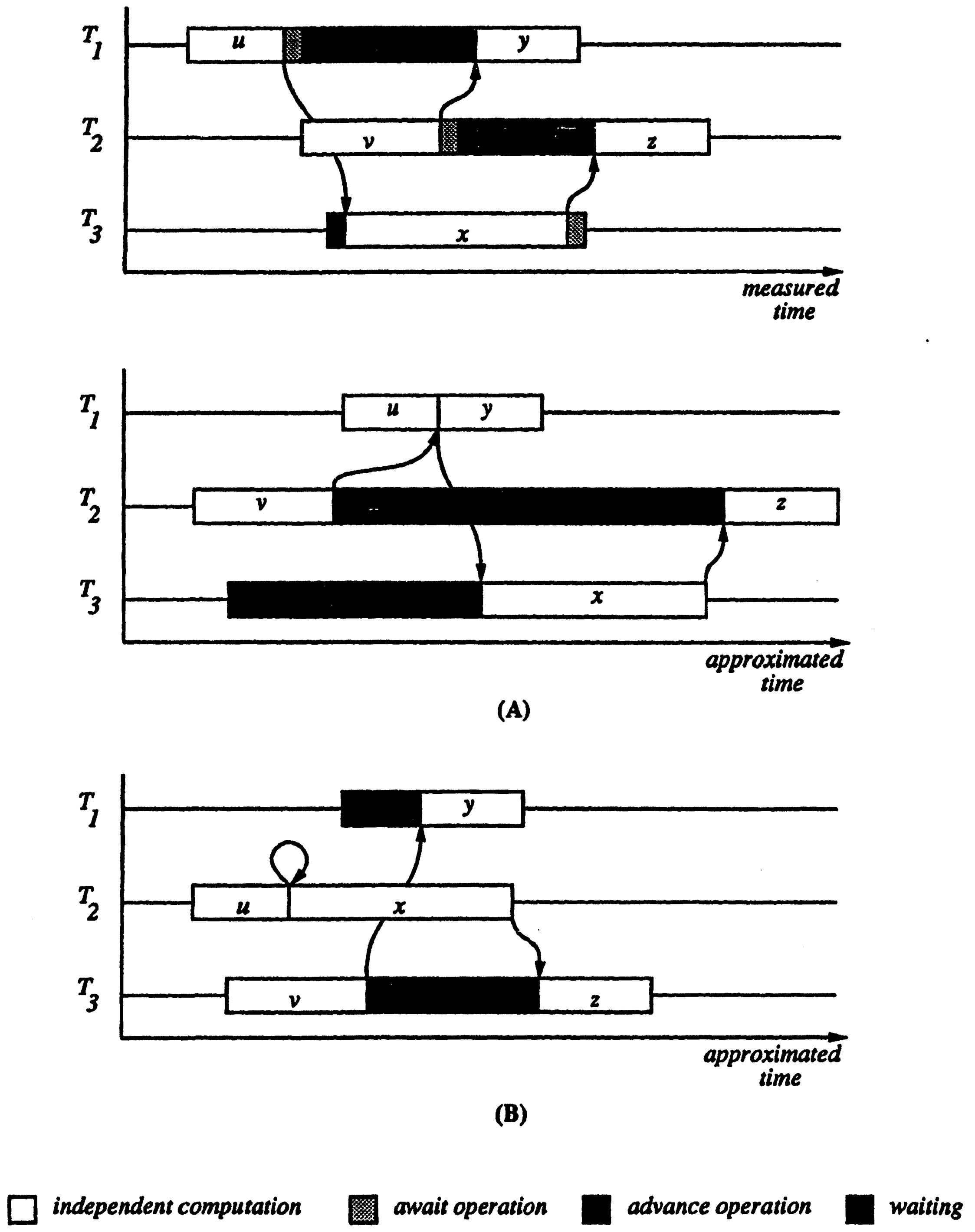


Figure 4.18: Measured and Approximated Execution Involving Advance and Await

perturbation analysis simulated a dynamic work scheduling policy. The result is a smaller total execution time with significant reductions in waiting delay. In theory, the perturbation analysis could compute a minimum critical path execution time for the partial order computation by applying a generalized bin packing algorithm that ignores thread scheduling issues.

The same basic question arises in the case of advance/await perturbation analysis as with semaphore synchronization. What is reasonable for the perturbation analysis to be able to assume about how the work between synchronization points is executed? The conclusion is that without additional knowledge from the user regarding work independence and reordering, no assumptions can be made without risking approximation errors or causing execution violations.

4.6 Comments

Given the operational definition of synchronization constructs used in a program and a trace of the program's execution containing enough event data to identify individual instances of synchronization use, the event-based perturbation analysis methods discussed above can be applied to remove instrumentation overhead while maintaining sequential consistency. However, the "conservative" approximation that results from the application of these techniques leaves the assignment of events to threads the same and the matching of dependent events unchanged. In cases where the approximated execution behavior violates the semantics of synchronization operation (as discussed in the semaphore analysis), the conservative perturbation analysis either will return an error, or will flag the error, adjust the event approximations to remove the violation, and continue.

Perturbation due to instrumentation has two effects on the events occurring during concurrent execution: temporal effects and resource assignment effects. In addition to the slowdown caused by instrumentation overhead, temporal effects include possible event re-orderings as the measurement alters the set of likely partial order executions. Resource assignment effects occur because the instrumentation changes the dynamic resource demands. In instances where the computation dynamically adapts to resource availability, instrumentation can perturb resource allocation and utilization. This is particularly important to understand with respect to processor assignment. Performance approximations can differ significantly from actual execution (as observed in the advance/await analysis), unless resource assignment effects are taken into account.

Unfortunately, the conservative approximation does not attempt to quantify performance effects dues to dependent event re-ordering or resource use. Any perturbation analysis approximation must be safe [86] (i.e., must not violate the partial ordering relationships) and, thus, must be provided sufficient measurements that capture the ordering dependencies during execution. However, the accuracy of perturbation analysis depends not only on more precise synchronization measurements (see §4.2), but also on additional knowledge of actual (likely) execution behavior, unattainable from measurements alone. This can include the scheduling policies used by a program, more detailed data dependency information, and even the performance characterization of certain synchronization operations (e.g. barrier performance). This information can be used together with the performance measurements obtained from the event data to drive (in a sense) a simulation of the execution. Although the approximations resulting from this more "liberal" perturbation analysis approach are potentially closer to the set of likely executions, such a result is difficult to analytically verify [174].

Chapter 5

Event-Based Performance Perturbation: Loop Analysis

Persecution is used in theology, not in arithmetic, because in arithmetic there is knowledge, but in theology there is only opinion. So whenever you find yourself getting angry about a difference of opinion, be on your guard; you will probably find, on examination, that your belief is getting beyond what the evidence warrants.

— Bertrand Russell, *"Unpopular Essays"*

Devoid of any user-supplied information, event-based perturbation analysis must assume that all events occurring on a thread of execution (and the work bounded by these events) during the performance measurement remain present on the same thread in the approximated execution. If the partial order and thread assignment of events in the actual execution remain unchanged in the approximated execution, this would confirm that the perturbation analysis preserved execution order and, therefore, that any perturbation error was due directly to errors in the timing measurements. Of course, it is exactly the ordering and timing relationships of the actual execution that is being observed. Furthermore, the actual execution might have inherent nondeterministic behavior, represented by a set of possible execution orderings. The perturbation analysis of a single performance measurement, while potentially resulting in an actual execution ordering, may reflect only one possible execution.

Clearly, to further understand and quantify the perturbations caused by the instrumentation of concurrent computations, we must consider an execution context larger than the localized analysis of individual synchronization operations treated in Chapter 4. To be complete, we should consider the combined effects of multiple forms of synchronization in different execution environments and for a variety of concurrent computations. In this analysis, we would study the localization of perturbation influence, the propagation of perturbation analysis errors, and the confidence of analysis approximations, especially as it relates to the nondeterministic properties of the computation. Given the large scope of such a project (due mainly to the diversity of concurrent execution environments) a comprehensive treatment of all factors represents a major undertaking. At a minimum, the effort presumes knowledge of concurrent execution behavior with which the perturbation analysis approximations can be compared. Unfortunately, the formulation and understanding of standard concurrent execution models is still an open research area [91].

Instead, we focus in this chapter on the influences of instrumentation on the execution of parallel loops. Parallel loops form an execution context large enough to observe interesting perturbation effects, but small enough to make perturbation analysis possible. Furthermore, there has been substantial work done in the area of parallel loop classification and execution [12, 38, 40, 97, 197] that can provide a basis for a perturbation study

We begin in §5.1 with a formal treatment of loops with no execution dependencies between iterations. Although such loops can be executed entirely in parallel given as many processors as loop iterations, the practical limitation of a finite number of processors implies an assignment of iterations to processors via some scheduling algorithm. Instrumentation can change the scheduling behavior, particularly the iteration assignment properties. Moreover, the instrumentation overhead can influence certain dynamics of the loop execution, such as causing more or less scheduling contention, or changing the amount of waiting at critical regions; this was observed in §3.5. Our goal is to understand the limits of perturbation analysis for these loops under a restrictive set of execution assumptions.

In §5.2, we consider loops with dependencies between iterations. We use an empirical approach to study the performance of event-based perturbation analysis in this context. We compare the accuracy of perturbation analysis given different levels of event instrumentation on a set of synthetic loops. The goal is to provide empirical evidence that more accurate perturbation analysis sometimes requires more detailed instrumentation of synchronization behavior, as discussed in Chapter 4.

In §5.3, we take the Lawrence Livermore loops that could not be accurately approximated using time-based perturbation analysis and re-instrument them for event-based analysis. Since these loops contain iteration execution dependencies, they do not conform to the restrictive parallel model supported by timing analysis. Instead, an event-based perturbation analysis is required. We present results showing the improvements achieved in the execution time approximations and discuss other performance characterizations that can be obtained.

5.1 DOALL Loop Perturbation Analysis

In most scientific applications, loops contain a significant portion of the computation and are a key source of parallelism [98]. Because of this, the execution of loops on parallel processors has been studied extensively [12, 38, 40, 97, 197]. The important performance issues concerning parallel loop execution deal with iteration scheduling [96, 153, 189] and iteration synchronization [134, 188, 189] — the obvious goal being to execute the loop as fast as possible. A performance characterization of loop execution might include a trace of the time each iteration was scheduled and on what processor, the time each iteration completed, and the synchronization events. The iteration begin and end times can be used to quantify scheduling overheads and to determine load balance, while the synchronization events capture processor waiting information.

However, the measurement of parallel loop execution can have several effects on performance behavior, as observed in Chapter 3. An event-based perturbation analysis of parallel loop measurements attempts to quantify these effects. Although a taxonomy of parallel loops [40] and the formulation of their parallel execution are well defined, there is no pre-existing theoretical or empirical foundation for understanding the performance effects due to execution-time perturbations caused by instrumentation.

In this section, we consider the constrained problem of executing a parallel loop on a fixed number of processors where there are no execution dependencies between the loop iterations. Such a loop is commonly referred to as a DOALL loop [40]. In principle, the loop iterations could be executed in any order on any thread (processor). In practice, iterations typically are scheduled in some execution-independent fixed order. The most restrictive scheduling mechanism is *static scheduling* [96]. Here iterations are statically assigned to processors. The most general case is *dynamic self-scheduling* [50], where each iteration could potentially be executed on any of the processors. There are a variety of other scheduling algorithms between these two extremes, such as *dynamic block scheduling* [96] and *guided self-scheduling* [154], but for our analysis we will consider only static scheduling and dynamic self-scheduling.

Our goal is to determine, under the assumptions of a fixed number of processors, the effects of instrumentation on the performance of a DOALL loop and what is possible to conclude about actual execution from perturbation analysis. We will first consider the case where the the DOALL loop is instrumented to record the time each iteration begins executing and the time it ends. In addition to the execution timestamp and the processor identifier, the iteration index is recorded. Because we assume no dependencies between the iterations, no synchronization events need be recorded.

5.1.1 Static Scheduling

With static scheduling, the assignment of iterations to processors is deterministic. Hence the execution ordering of iterations on a single processor will be the same in both the actual and the approximated execution. The only "nondeterminism" in the execution will be the variations in iteration times the instrumentation might cause. These effects can be quantified by an additional experiment that captures the beginning time of the first iteration and the ending time of the last iteration on each processor. The approximated loop execution time in this case can be compared with the approximated execution from the per iteration instrumentation to determine the timing effects of more intrusive instrumentation. To the extent that the time values differ indicates possible influences of the instrumentation on the iteration times.

Because there is a fixed assignment of iterations to processors, time-based perturbation analysis is sufficient in the case of statically scheduled DOALL loops. In fact, each processor participating in the DOALL loop can be analyzed separately. If processor P_i is assigned iterations $(r, s, \ldots, t)$, the execution time accumulated by P_i in the DOALL loop is given by

$$\sum_{j=r,s,\ldots,t} t(j)$$

where $t(j)$ is the execution time of iteration $j \in (r, s, \ldots, t)$. The execution time of P_i in the approximated execution is given by

$$\sum_{j=r,s,\ldots,t} (t_m(j) - \beta - \gamma) \quad = \quad \sum_{j=r,s,\ldots,t} t_m(j) - n_i(\beta + \gamma)$$

where $t_m(j)$ is the measured execution time of iteration $j \in (r, s, \ldots, t)$, β is the measured fixed overhead of the instrumentation at the begin of the iteration, γ is the overhead of the instrumentation at the end of the iteration, and n_i is the number of iterations assigned to P_i. The total time of the DOALL loop will be the longest approximated execution time of any processor participating in the loop.

5.1.2 Dynamic Self-Scheduling

If every processor competes for each iteration, the assignment of iterations to processors is, in general, nondeterministic. In practice, under certain known conditions and depending on the characteristics of the iteration times, some assignments of iterations to processors are more likely than others. Adding instrumentation overhead to every iteration skews the set of likely iteration-to-processor assignments. The amount of perturbation in the assignment depends directly on the relative sizes of the instrumentation overhead and the iteration times.

Because DOALL loops contain no synchronization operations (save for the implicit scheduler synchronization at iteration assignment), one might initially consider a time-based perturbation analysis approach for dynamically self-scheduled DOALL loops. In general, it is impossible for the perturbation analysis (time-based or event-based) to conclude that an approximated DOALL execution is representative of or "close to" one of the likely actual executions. We could empirically validate the approximated total execution time by comparing it against the measured execution time of the DOALL loop, but even an empirically accurate time approximation does not necessarily imply a representative iteration assignment.

However, an event-based perturbation analysis must default to a time-based approach when there are no synchronization events in the measurement that capture dependent operation and when there is no additional information supplied regarding how iterations are assigned to processors and if the perturbation analysis is allowed to re-assign iterations. In the case of the DOALL loop there is no dependent operation and, hence, no synchronization events will be recorded in a performance measurement. On the other hand, knowledge that the DOALL loop is dynamically self-scheduled might allow an event-based perturbation analysis to be improved. Below we consider the effectiveness of event-based approximations for different classes of DOALL loops and different instrumentation perturbation scenarios. Our goal is to start with a default time-based approach and to consider successively more complex cases until the approach breaks down. We then consider how knowledge of scheduling behavior can be used to regain accuracy in the approximations.

For this study, we simplify the DOALL execution model as follows. The loop startup time and the iteration assignment time are assumed to be zero, and all processors are initially assigned an iteration. The iterations are numbered from one to N and are assigned to processors in increasing order and on a first-come, first-serve basis; effectively, a processor will never wait for an iteration.

5.1.2.1 Uniform Iteration Time, Constant Instrumentation Overhead

With this simplified model, we first consider the case where all iterations have the same execution time λ and the iteration instrumentation overhead (begin and end) is a constant α; i.e., $\beta = \gamma = \alpha$. The perturbation analysis independently applies a timing analysis to each thread of execution. The following lemma holds.

Lemma 1 *For DOALL loops with a constant iteration execution time λ, the measured difference in execution time between the beginning of iteration i and any iteration $j < i + P$ cannot be greater than $\lambda + 2\alpha$, where P is the number of processors participating in the DOALL loop and α is the instrumentation overhead. That is,*

$$t_m(j) - t_m(i) \leq \lambda + 2\alpha.$$

Note, the measured iteration execution time includes the iteration instrumentation. Furthermore, for any iteration $k > i + P$,

$$t_m(k) - t_m(i) > \lambda + 2\alpha.$$

Proof: Suppose there exists an iteration $i < j < i + P$ such that

$$t_m(j) - t_m(i) > \lambda + 2\alpha;$$

i.e.,

$$t_m(i) + \lambda + 2\alpha < t_m(j).$$

Let iteration i be assigned to processor p. At time $t_m(i)$, a processor $s \neq p$ is either executing an iteration or has just finished an iteration and will be assigned an iteration after i on a first-come, first-serve basis. If the former is true, s will be assigned a new iteration k at some time

$t_m(k)$,

$$t_m(i) < t_m(k) \leq t_m(i) + \lambda + 2\alpha.$$

If the latter is true, s will be assigned a new iteration k at the same time p is assigned i; $t_m(k) = t_m(i)$. In either case, each of the processors other than p will be assigned an iteration greater than i at a time less than or equal to $t_a(i) + \lambda + 2\alpha$. Processor p will be assigned a new iteration at time $t_a(i) + \lambda + 2\alpha$. Thus, for any iteration k, $i \leq k \leq i + P$,

$$t_a(i) \leq t_a(k) \leq t_a(i) + \lambda + 2\alpha.$$

A contradiction results. The second part of the lemma is proved in a similar manner. ■

Corollary 1 *For DOALL loops with a constant iteration execution time λ, the approximated execution time difference between the beginning of iteration i and any iteration $j < i + P$ cannot be greater than λ, where P is the number of processors participating in the DOALL. That is,*

$$t_a(j) - t_a(i) \leq \lambda.$$

Furthermore, for any iteration $k > i + P$,

$$t_a(k) - t_a(i) > \lambda.$$

Proof: The proof follows the same approach as in Lemma 1 except that the instrumentation overhead for iteration begin and end, 2α, is removed. ■

The simple conclusion from Lemma 1 is that the perturbation effects of instrumentation with constant overhead on the measured execution time ordering of DOALL iterations with a constant execution time are bounded. Furthermore, Corollary 1 implies that the approximated execution will be within λ of the true result.

5.1.2.2 Non-Uniform Iteration Time, Proportional Instrumentation Overhead

We now consider the case where the iterations can have different execution times, but the instrumentation overhead (iteration begin and end instrumentation) will be proportional to the iteration execution time.[1] That is, if $T(i)$ is the execution time of iteration i, $o(i) = \beta T(i)$, where $o(i)$ is the total instrumentation overhead for iteration i and β is the constant of proportionality to be applied to all iterations. The following lemma holds in this case.

[1]This condition might occur, for instance, if a different number of instructions are executed in each iteration and every instruction was instrumented.

Lemma 2 *Let $s(i)$ be the amount of time after the start of the DOALL loop until iteration i begins in the actual execution; i.e. $s(i) = t(i) - t(DOALL)$. Let $s_m(i)$ be defined similarly for the measured execution. For DOALL loops with unequal iteration execution times, if the instrumentation overhead for each iteration is proportional to the iteration's execution time, $o(i) = \beta T(i)$, then*

$$s_m(i) = \beta s(i).$$

Proof: Proof by Induction.

Basis: The DOALL execution model assumes every processor receives an iteration at the beginning of the loop. Thus, the first P iterations $1, \ldots, P$ are assigned to the P processors when the loop begins. For iteration i, $1 \leq i \leq P$, $s(i) = 0$, and $s_m(i) = \beta * 0 = 0$.

Induction Hypothesis: Assume $s_m(i) = \beta s(i)$ is true for all iterations $\leq i$. After i iterations have been assigned, let $R = r_1, \ldots, r_P$ denote the current set of iterations assigned to processors $1 \ldots P$, respectively, at time $s_m(i)$

Induction Step: In the measured execution, the next iteration $i+1$ will be assigned to the processor p when $s_m(r_p) + o(r_p) \leq s_m(r_k) + o(r_k), 1 \leq r_k \neq r_p \leq P$. This will occur when

$$\begin{aligned} s_m(i+1) &= s_m(r_p) + o(r_p) \\ &= \beta s(r_p) + \beta T(r_p) \\ &= \beta(s(r_p) + T(r_p)). \end{aligned}$$

From the induction hypothesis, we know $s(r) = \frac{s_m(r)}{\beta}$, $\forall r \in R$. In the actual execution, iteration $i+1$ would have been scheduled processor j where

$$s(r_j) + T(r_j) \leq s_m(r_k) + T(r_k),$$

$1 \leq r_j \neq r_k \leq P$. Substituting for $s(r)$ above, we find that $r_j = r_p$. Thus, in the actual execution, iteration $i+1$ would have been scheduled when

$$s(i+1) = s(r_p) + T(r_p)$$

From the equation $s_m(i+1) = \beta(s(r_p) + T(r_p))$ above, we have

$$s_m(i+1) = \beta(s(r_p) + T(r_p)) = \beta s(i+1). \qquad \blacksquare$$

Lemma 2 indicates that, under the model assumptions, the iteration assignment behavior (when iterations are scheduled) remains unchanged, in a relative sense, between the actual and measured DOALL execution. Because the perturbation analysis can determine $s_m(i)$ from the instrumentation measurements of each iteration i, it can derive an accurate approximation of the entire DOALL loop execution given only knowledge of β.

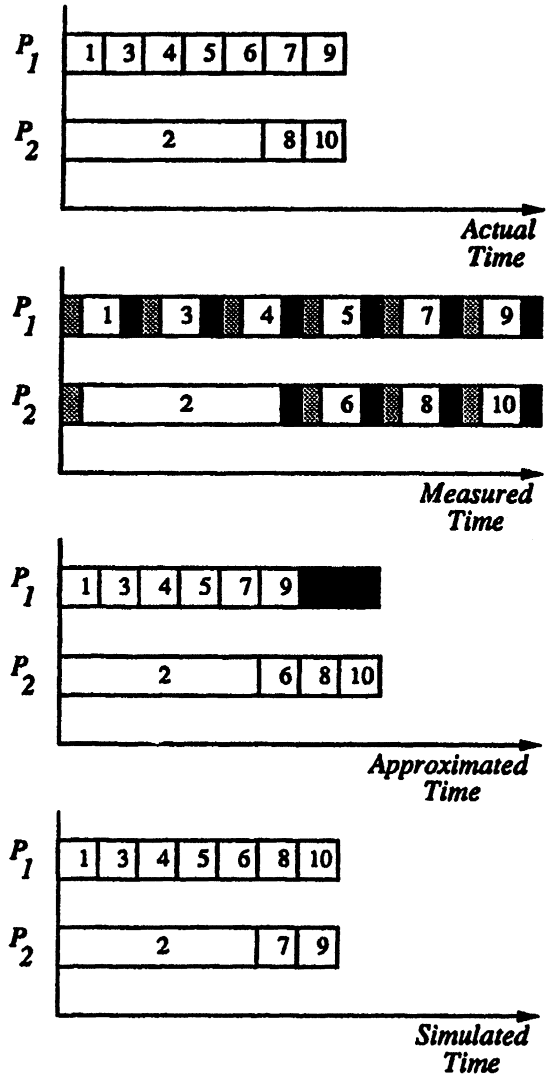
Corollary 2 *Let $s_a(i)$ be the amount time after the start of the DOALL loop until iteration i begins in the approximated execution (assuming β is known); i.e., $s_a(i) = t_a(i) - t_a(DOALL)$. For DOALL loops with unequal iteration execution times, if the instrumentation overhead for each iteration is proportional to the iteration's execution time, $o(i) = \beta T(i)$, then $s_a(i) = s(i)$.*

Proof: From the measured execution, $s_m(i)$ is known for each iteration i. If β is known by the perturbation analysis, $s(i)$ can be approximated exactly by s_m/β from Lemma 2. Thus, $s_a(i) = s_m/\beta = s(i)$. ∎

The perturbation analysis used in this case is still basically a time-based approach because only the overhead of instrumentation, $o(i)$, is being removed.

5.1.2.3 Non-Uniform Iteration Time, Constant Instrumentation Overhead

If we consider again the case of a constant instrumentation overhead per iteration but allow iteration times to be non-uniform, the iteration assignment behavior between an actual and a measured execution can be quite different. Figure 5.1 shows a simple example. Because of the unequal distribution of instrumentation overhead across the two processors, the timing relationships between the iterations changes. In this case, iterations 5 and 6 are scheduled at the same time in the measured execution whereas they were consecutively executed on processor P_1 in the actual execution. When the instrumentation is removed by perturbation analysis to produce the approximated execution (using a timing analysis approach for each processor), significant timing deviations from the actual DOALL execution result. As seen, it is even possible to have approximated iteration orderings that violate the assumed increasing order of iteration assignment. In this case, iteration 7 appears before 6, and 9 appears before iteration 8. This is an immediate indication that an approximation error has occurred. Another indication of an approximation error is when a processor waits for remaining iterations to be assigned, as in the case of processor P_1 and iteration 10.



▨ iteration begin instrumentation

■ iteration end instrumentation

■ waiting

Figure 5.1: DOALL Loop Example with Non-Uniform Iteration Times

Without knowledge of the iteration scheduling algorithm, the perturbation analysis cannot estimate the error in the event approximations using only time-based approaches. With knowledge of the iteration scheduling algorithm, the perturbation analysis can use the approximated execution times of the iterations to drive a simulation of the DOALL execution. However, the simulated schedule will be computed under the assumptions of the scheduling algorithm. Any errors in the simulated execution will be attributed to these assumptions.

Figure 5.1 shows a simulated execution using iteration execution times approximated from the measured execution under the assumption of dynamic self-scheduling with round-robin resolution in the case of scheduling ties. Since the original execution was self-scheduled, the simulated execution is in closer agreement with the actual execution. The differences in iteration assignments to processors could be due to errors in the iteration execution time approximations or to the assumptions made by the simulation regarding tie resolution.

In addition to improving the accuracy of execution approximations, a simulation-based perturbation analysis approach can be applied in a predictive manner to estimate DOALL execution in simulated environments other than that which produced the measured execution. For instance, the user could request the perturbation analysis to predict an optimal iteration scheduling, or a scheduling for a different number of processors.

5.2 A DOACROSS Perturbation Analysis Study

Often, the iterations of a parallel loop have data dependencies on other iterations; either the result produced in one iteration is used in a later iteration, or data fetched in one iteration is updated in a later iteration [97]. Unlike DOALL loops, the execution of parallel loops with iteration dependencies is not entirely nondeterministic — the execution is constrained by the synchronization points within each iteration and by their partially ordered execution. Although the scheduling aberrations in the measured execution resulting from instrumentation overhead can still occur (see Figure 5.1), requiring some form of scheduling simulation in the perturbation analysis to improve approximation accuracy, the dependent execution can restrict the possible run-time behaviors [2].

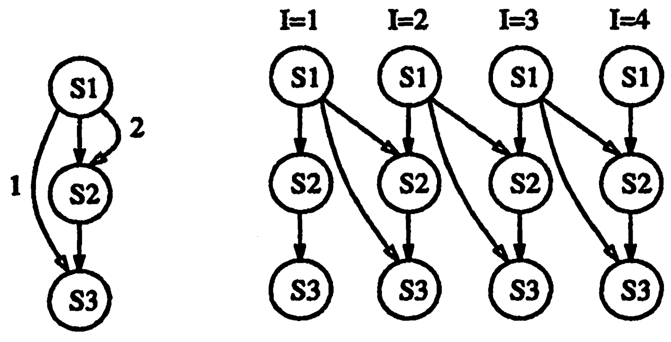
As observed in Chapter 3, time-based perturbation analysis of parallel loops containing dependencies can produce significant approximation errors. Event-based approaches that apply the synchronization perturbation models discussed in Chapter 4 must instead be used. To evaluate the practical capabilities of event-based perturbation analysis of parallel loops with iteration dependencies, we implemented a trace analyzer that understands the advance and await synchronization constructs. With this tool we studied the effects of software instrumentation on a series of synthetic parallel loops with cross-iteration dependencies. We varied the placement and degree of instrumentation to determine the effects monitoring (or not monitoring) synchronization points has on perturbation analysis accuracy. In the study, the perturbation analysis did not simulate any scheduling algorithm but instead relied solely on the synchronization ordering data found in the trace. The perturbation analysis techniques for advance/await synchronization discussed in Chapter 4 were used.

To validate the performance of the event-based perturbation analyzer, we applied it to the Lawrence Livermore loops [128, 131] for which time-based analysis failed. These loops contain execution ordering dependencies not accounted for by the timing models. We re-instrumented the loops to capture the advance and await synchronization points. The results of the perturbation analysis are given in §5.3.

5.2.1 The DOACROSS Loop Model

A DOACROSS loop [39, 38] contains data dependencies (see Chapter 4) between iterations that that must be enforced to maintain correct execution order. The notion of a *data dependence distance* [197], d, is used to quantify the dependencies within an iteration of a DOACROSS loop. If iteration $i + d$ is dependent on iteration i, the distance of the data dependence is d. A *data dependence graph* [100] is useful for viewing the program dependencies. It is drawn as a directed graph with each statement represented by a node and the directed arcs showing forward and backward dependencies. The data dependence distance between statements is often included in data dependence graphs. An *expanded data dependence graph* [100] (also called an *iteration space graph* [197]) for a DOACROSS loop shows the loop statements for each iteration and the dependencies for each particular statement execution. Figure 5.2 shows an example DOACROSS loop drawn as a data dependence graph and an iteration space graph. In Figure 5.2, the distance of the $S1 \rightarrow S2$ dependency is 2, and the distance of the $S1 \rightarrow S3$ dependency is 1. For our purposes, we will focus on *constant-distance* dependencies [197]; i.e., dependencies

```
DOACROSS I=1,N

   S1:  A[I+3] = . . .
   S2:  . . . = A[I+1]
   S3:  . . . = A[I+2]

END DOACROSS
```



DOACROSS
Loop Example

Data Dependence
Graph

Iteration Space
Graph

Figure 5.2: DOACROSS Loop Example

with constant data dependence distances. Constant-distance dependencies occur frequently in numerical programs [38].

Some form of synchronization is required in DOACROSS loops to correctly execute statements with data dependencies during parallel execution. There are several data synchronization schemes that can be applied to handle constant-distance dependencies [188]. We concentrate on advance/await synchronization [5, 134] because the semantics of the operation fit well with the meaning of constant-distance dependencies. To simplify the discussion, we consider only one-level DOACROSS loops; i.e., no loop nesting.

Figure 5.3 shows the DOACROSS loop structures used for our synthetic experiments. Ignoring the synchronization operations for the moment, the basic loop structure has three code bodies. The loop is entered at the top and executes these three code bodies in sequence within each iteration. Before loop entry, we assume the execution is sequential. After the loop is entered, the iterations are dynamically self-scheduled across the processors. An implicit barrier is assumed at the end of loop. After the last iterations have executed on each processor, all processors will synchronize before exiting the loop. The computation after the loop is assumed to be sequential.

Each body reflects a block of computation. However, in the DOACROSS loop labeled *Forward* in Figure 5.3, we assume a forward dependency [38] exists between body 1 (B1) and body 3 (B3); $B1 \rightarrow B3$. To satisfy this dependency during parallel execution, an advance

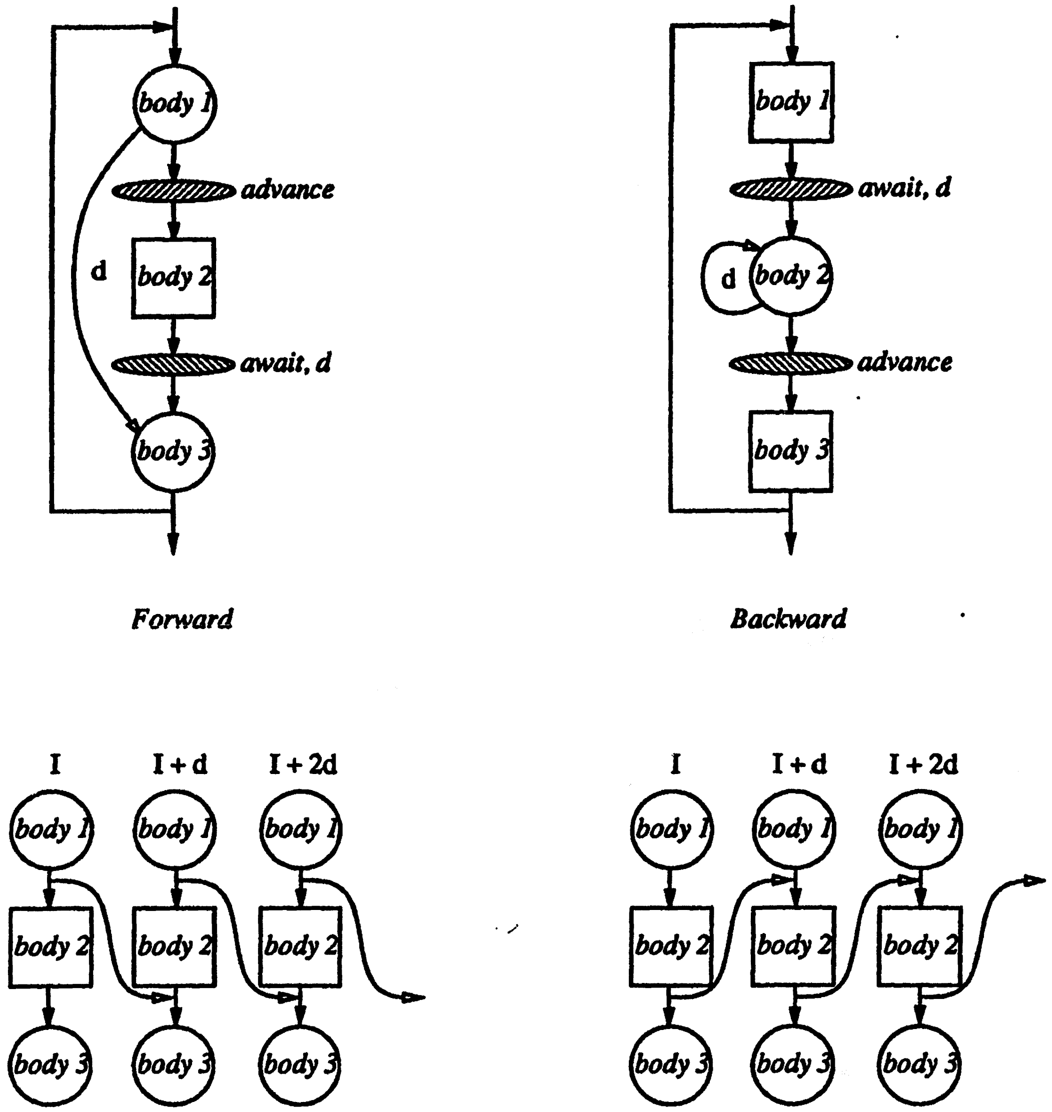


Figure 5.3: Synthetic DOACROSS Loop Structures

operation is placed between B1 and body 2 (B2) and an await operation is placed between B2 and B3. In the DOACROSS loop labeled *Backward* in Figure 5.3, we assume a backward dependency [38] exists between statements within B2. In this case, an await operation is placed between B1 and B2, and an advance operation is placed between B2 and B3. The dependence distance, *d*, is a parameter of the synthetic experiment for both the forward and backward loop structures; see §5.2.3. The iteration space graphs for both the forward and backward loops are also shown in Figure 5.3.

5.2.2 DOACROSS Instrumentation

The synthetic DOACROSS loops were instrumented in four ways. Figure 5.4 shows these four instrumentations for the forward DOACROSS loop; the same instrumentations were applied to the backward case. The *raw* instrumentation is used to measure the total execution time of the synthetic loop. The beginning of the DOACROSS loop is designated by event 0 in the instrumentation and the end of the loop by event -1. The *iteration* instrumentation records the beginning (event 1) and end (event 4) of each iteration in addition to the raw instrumentation events.

The *no-synchronization* instrumentation places instrumentation around the body code involved in the dependency but does not explicitly instrument the synchronization points themselves. In the case of the forward DOACROSS loop shown, events 2 and 3 are added to provide instrumentation for B1 and B3. In the case of the backward DOACROSS loop (not shown), events 2 and 3 are placed around B2. The *synchronization* instrumentation provides instrumentation for loop begin and end, iteration begin and end, all body code, *AND* the synchronization operations. Event 5 is recorded to indicate the completion of the advance operation (await in the backward case). Event 6 is recorded to indicate the completion of the await operation (advance in the backward case).

5.2.3 Synthetic DOACROSS Experiments

The set of synthetic experiments were defined by the relative execution times of the loop body code and the data dependence distance *d*. We made the body execution time dependent on the loop iteration being executed to test different body execution time distributions. Different distributions of execution times could be chosen for the three respective body components of

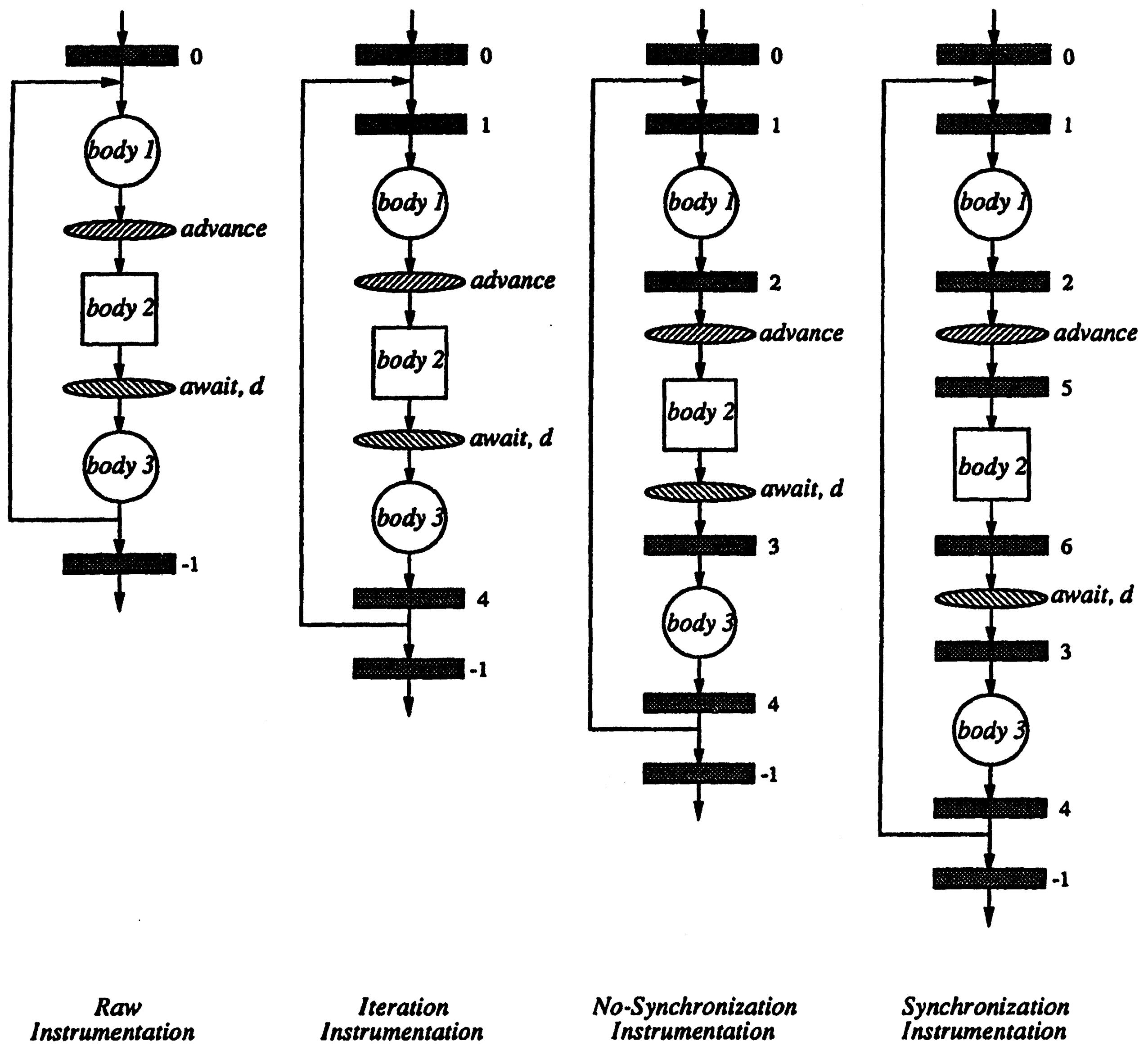


Figure 5.4: Instrumentation of Synthetic Forward DOACROSS Loop

Distribution	Description
Fixed	The number of null loop iterations is fixed at a constant.
Increasing	The number of null loop iterations increases from a starting value with each successive DOACROSS iteration.
Decreasing	The number of null loop iterations decreases from a starting value with each successive DOACROSS iteration.
Triangular	The number of null loop iterations first increases from a starting value with each successive DOACROSS iteration, but then begins decreasing at the same rate after half the DOACROSS have been assigned.
Inverted	The number of null loop iterations first decreases from a starting value with each successive DOACROSS iteration, but then begins increasing after half the DOACROSS have been assigned.
Random	The number of null loop iterations is drawn from a random sample for each DOACROSS iteration.
Normal	The number of null loop iterations is drawn from a normal distribution for each DOACROSS iteration.

Table 5.1: Execution Time Distributions for DOACROSS Experiments

the loop. The different execution time distributions we used in the construction of the synthetic DOACROSS experiments are described in Table 5.1.

Generating a synthetic DOACROSS experiment allows us to easily test the accuracy of event-based perturbation analysis for different combinations of body execution times and dependence distance. Although a large number of experiments were performed, we report in §5.2.5 the results for the experiments shown in Table 5.2. Each row of Table 5.2 actually designates a set of experiments as indicated in the description. Each set of experiments was run for both forward and backward loop configurations and with dependence distances of 1, 2, 4, and 8.

5.2.4 Target Environment

The Alliant FX/80 was chosen as the target machine because of its instruction set supports advance/await-style synchronization. The details of the FX/80 environment are discussed in §3.1. All the experiments were performed with all eight FX/80 processors executing in parallel. The loops were dynamically self-scheduled on the processors. The instrumentation support was

Experiment	**Description**
FIXED-1	**Both body 1 and body 3 are fixed at 10 iterations of the null loop. Body 2 is fixed and the number of iterations of the null loop is drawn from the set {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}.**
FIXED-2	**Both body 1 and body 3 are fixed and the number of iterations of the null loop is drawn from the set {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}. Body 2 is fixed at 10 iterations of the null loop.**
INCREASE-1	**Both body 1 and body 3 are increasing starting with 1 iteration of the null loop and incrementing by 1 with each successive DOACROSS iteration. Body 2 is fixed and the number of iterations of the null loop is drawn from the set {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}.**
INCREASE-2	**Both body 1 and body 3 are fixed and the number of iterations of the null loop is drawn from the set {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}. Body 2 is increasing starting with 1 iteration of the null loop and incrementing by 1 with each successive DOACROSS iteration.**
RANDOM-1	**Both body 1 and body 3 are random with iterations of the null loop falling in the range of 0 to 100. Body 2 is fixed and the number of iterations of the null loop is drawn from the set {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}.**
RANDOM-2	**Both body 1 and body 3 are fixed and the number of iterations of the null loop is drawn from the set {10, 20, 30, 40, 50, 60, 70, 80, 90, 100}. Body 2 is random with iterations of the null loop falling in the range of 0 to 100.**

Table 5.2: Synthetic DOACROSS Experiments

altered from the timing experiments (see §3.4) to additionally store the loop iteration number with every event. This was necessary to support the advance/await synchronization analysis.

5.2.5 Synthetic DOACROSS Experiment Results

The goal of the synthetic DOACROSS study was to test event-based perturbation analysis across a variety of loop executions. Although the set of experiments should not be regarded as comprehensive, the results do provide some insight into the robustness of the event perturbation models. We are primarily interested in the model's performance on the experiments with synchronization instrumentation; the traces from these experiments contain all synchronization information needed to approximate the actual execution. The tests with the other forms of instrumentation are important for observing the approximation error that results when synchronization information is not recorded.

We look first at the complete set of results for the test cases with synchronization instrumentation. Tables 5.3 and 5.4 give the approximation errors of the event-based perturbation analysis of the synchronization instrumentation for the forward and backward experiments, respectively. All the errors are calculated using the estimated execution time of the DOACROSS loop from the raw instrumentation. The errors are reported as ranges that bound the errors within the corresponding set of experiments. Results from the four dependence distances tested are given.

In all cases, the approximation errors are significantly smaller, in absolute terms, than the measured errors. The measured execution time includes the instrumentation overhead and can be as much as 102% (greater than a factor of two slowdown) in error of the actual execution time. In contrast, the approximated execution errors overall are in the range -2.7% to 2.1% for the forward experiments, and -3.5% to 4.2% for the backward experiments.[2] The greatest improvements are in the FIXED-1 and FIXED-2 sets of experiments. Curiously, no differences are seen in measured errors across dependence distance for the forward experiments. We suspect this to be an artifact of the experiments. In general, if the degree of waiting is unaffected by the instrumentation, the measured error and the dependence distance will not be correlated;

[2]Negative percent errors indicate approximated execution times less than actual.

Experiment	Dependence Distance			
	1	*2*	*4*	*8*
FIXED-1				
measured percent error	27.6 to 96.9	27.3 to 95.9	27.6 to 95.9	27.4 to 97.8
approximated percent error	-0.1 to 0.2	-0.3 to -0.3	0.0 to 0.2	0.3 to 2.1
FIXED-2				
measured percent error	15.9 to 96.4	16.2 to 96.5	16.1 to 96.1	15.9 to 97.2
approximated percent error	-0.4 to 0.3	0.2 to 0.7	0.0 to 0.3	-0.2 to -0.1
INCREASING-1				
measured percent error	16.7 to 27.0	16.6 to 27.0	16.4 to 27.0	16.4 to 27.3
approximated percent error	-1.9 to -0.6	-1.9 to -0.8	-1.9 to -0.9	-1.1 to -0.9
INCREASING-2				
measured percent error	13.1 to 42.4	12.9 to 41.8	12.7 to 41.8	12.7 to 42.4
approximated percent error	-1.7 to -0.3	-2.5 to -0.4	-2.5 to -0.7	-1.7 to -0.7
RANDOM-3				
measured percent error	15.8 to 25.5	15.8 to 25.5	16.2 to 26.2	16.2 to 26.2
approximated percent error	-2.0 to -1.1	-2.7 to -1.1	-1.9 to -0.8	-1.1 to -0.8
RANDOM-2				
measured percent error	12.8 to 47.3	12.9 to 46.3	12.9 to 47.3	12.9 to 46.1
approximated percent error	-2.3 to -1.2	-1.7 to -1.0	-1.0 to -0.5	-1.0 to 0.4

Table 5.3: Forward DOACROSS Measured and Approximated Errors

Experiment	Dependence Distance			
	1	*2*	*4*	*8*
FIXED-1				
measured percent error	**11.2 to 99.6**	**11.8 to 100.6**	**12.3 to 98.8**	**25.2 to 96.0**
approximated percent error	**-0.7 to 0.2**	**-0.1 to 2.7**	**0.8 to 1.3**	**-0.2 to 0.8**
FIXED-2				
measured percent error	**15.8 to 100.4**	**15.9 to 102.0**	**16.1 to 97.8**	**16.2 to 97.8**
approximated percent error	**-3.4 to 1.2**	**-0.6 to 4.2**	**0.0 to 0.6**	**0.0 to 1.6**
INCREASING-1				
measured percent error	**11.0 to 44.4**	**12.5 to 28.0**	**13.3 to 27.8**	**16.6 to 29.7**
approximated percent error	**-3.4 to -0.5**	**-3.2 to -0.1**	**-1.5 to -0.9**	**-1.8 to -0.9**
INCREASING-2				
measured percent error	**17.1 to 21.8**	**13.4 to 21.5**	**12.7 to 23.2**	**12.7 to 42.9**
approximated percent error	**-3.5 to -1.2**	**-0.7 to -0.6**	**-0.9 to -0.8**	**-1.6 to -0.8**
RANDOM-1				
measured percent error	**11.2 to 49.4**	**11.6 to 27.9**	**13.9 to 25.4**	**16.0 to 25.2**
approximated percent error	**-3.4 to -2.4**	**-2.1 to 0.3**	**-1.7 to -0.2**	**-3.4 to -1.3**
RANDOM-2				
measured percent error	**22.8 to 25.6**	**13.1 to 19.7**	**12.9 to 18.5**	**12.8 to 34.8**
approximated percent error	**-3.1 to -2.0**	**-3.2 to -0.1**	**-3.5 to 0.3**	**-2.9 to -0.4**

Table 5.4: Backward DOACROSS Measured and Approximated Errors

this is the case for our forward experiments where little or no waiting occurs in the measured execution.

In general, the dependence distance could affect the execution behavior and the degree of waiting, possibly altering total execution time. This is true in the case of the backward experiments. As a rule of thumb, increasing dependence distance for backward synchronization reduces the likelihood of synchronization waiting. However, if the execution time of the critical section bounded by the **await** and **advance** operations (i.e., body 2 in the backward experiments) is non-uniformly distributed, the instrumentation overhead could have different relative influences on waiting at different times during the execution. Increasing the dependence distance in this case can both positively and negatively affect the measured error. We see both of these effects in the backward experiments.

Figures 5.5 through 5.10 show the measured versus approximated errors for the different sets of forward experiments. Figures 5.11 through 5.16 do the same for the backward experiments. These curves again demonstrate the good accuracy of the approximations and show the effects of varying body execution times within each experiment set.

The question we addressed with the experiments using the iteration and no-synchronization instrumentation methods was how approximation accuracy is affected when synchronization events are not recorded. The perturbation analysis of these experiments applied the default timing model to removing the instrumentation overhead from the trace of each processor's events. Because less monitoring is done in these experiments, we would expect to see smaller measured errors than in the experiments with synchronization instrumentation. However, the approximation errors in the iteration and no synchronization instrumentation experiments will depend on how much synchronization waiting occurs that the timing analysis does cannot take into account.

Figures 5.17 and 5.18 show the measured and approximated errors for the different instrumentation test cases for the set of forward FIXED-1 experiments. The relationship of the measured errors is as expected. Surprisingly, however, the timing-based analysis delivers almost as good an error range as the event-based analysis. The principal reason for this is that only a small amount of waiting occurs in these forward experiments. In fact, from the event-based analysis, we calculated the percentage waiting time for each processor and found it to be zero. Figures 5.19 and 5.20 show similar results for the set of experiments INCREASE-1. We sur-

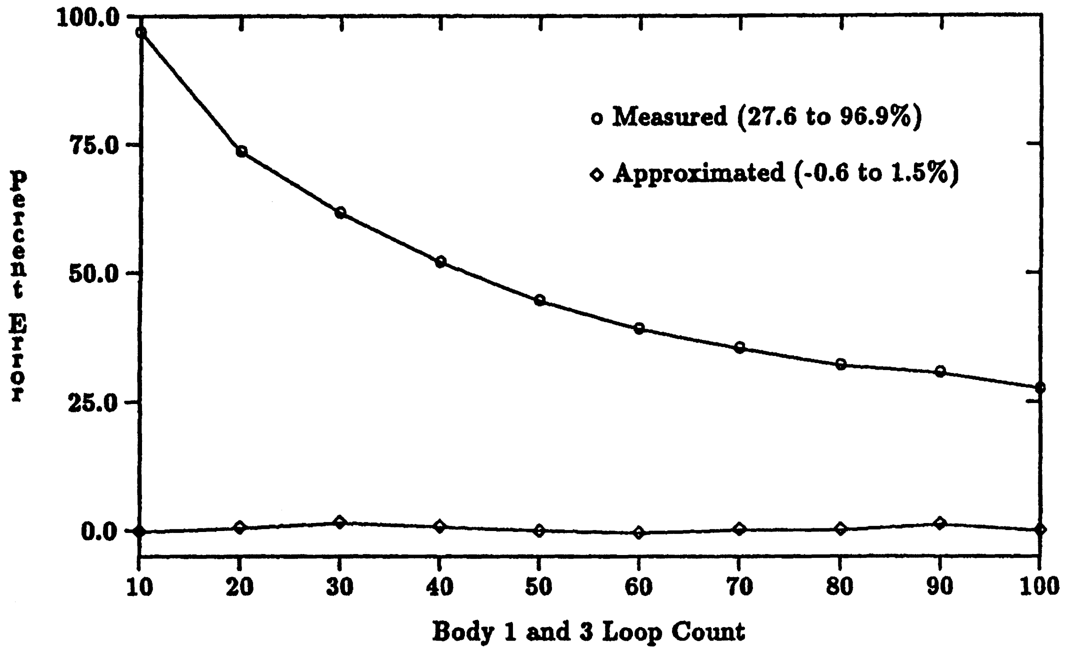


Figure 5.5: FIXED-1, Forward, Synchronization, $d = 1$

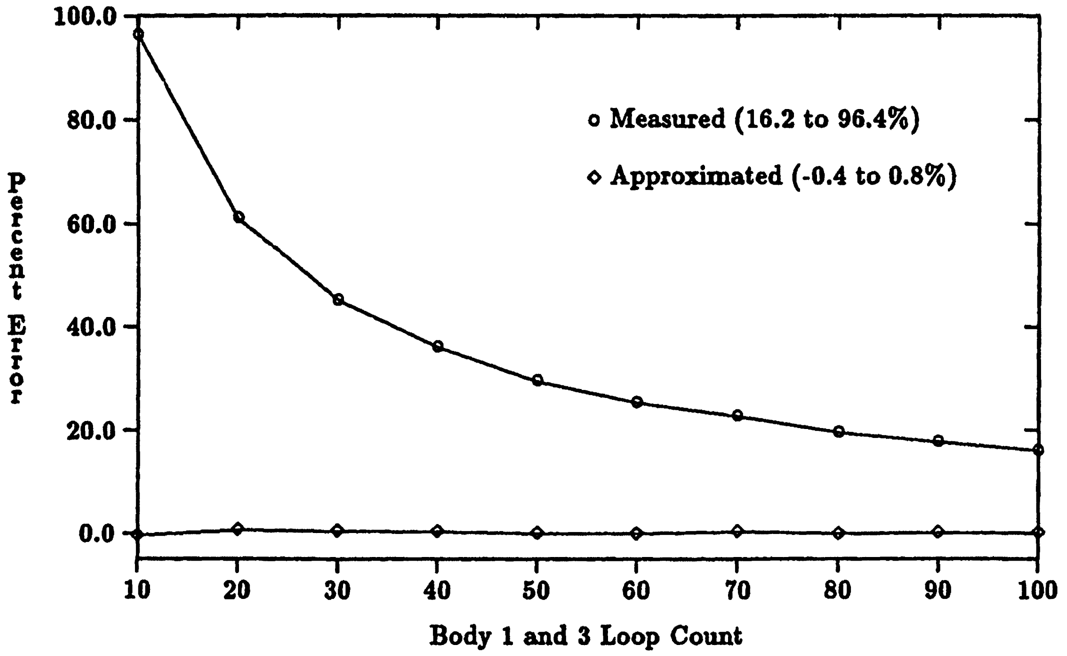


Figure 5.6: FIXED-2, Forward, Synchronization, $d = 1$

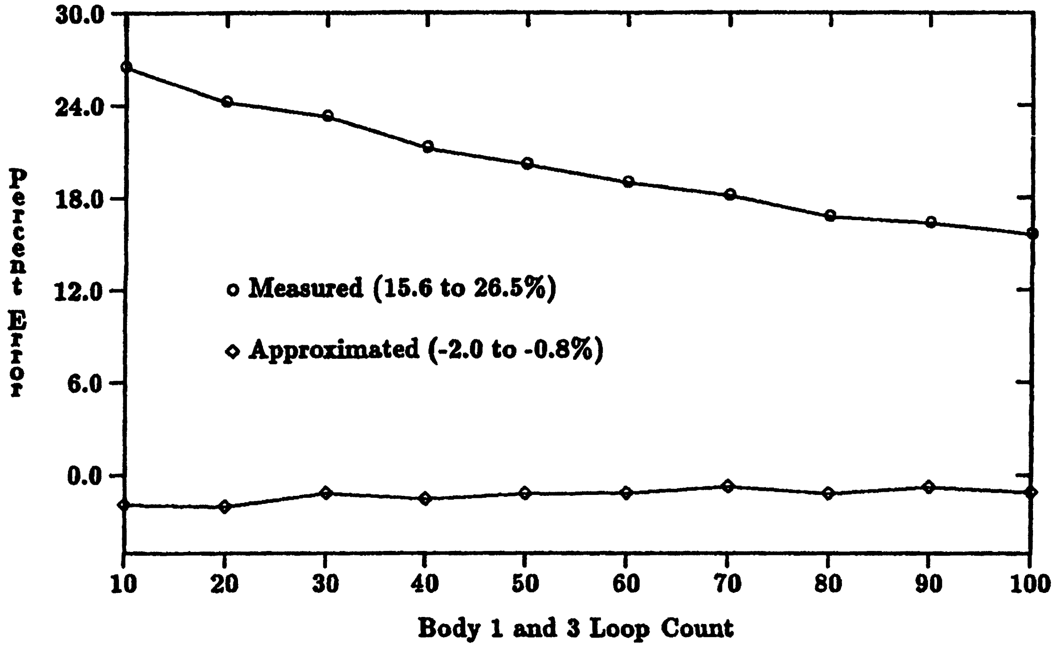


Figure 5.7: INCREASING-1, Forward, Synchronization, $d = 1$

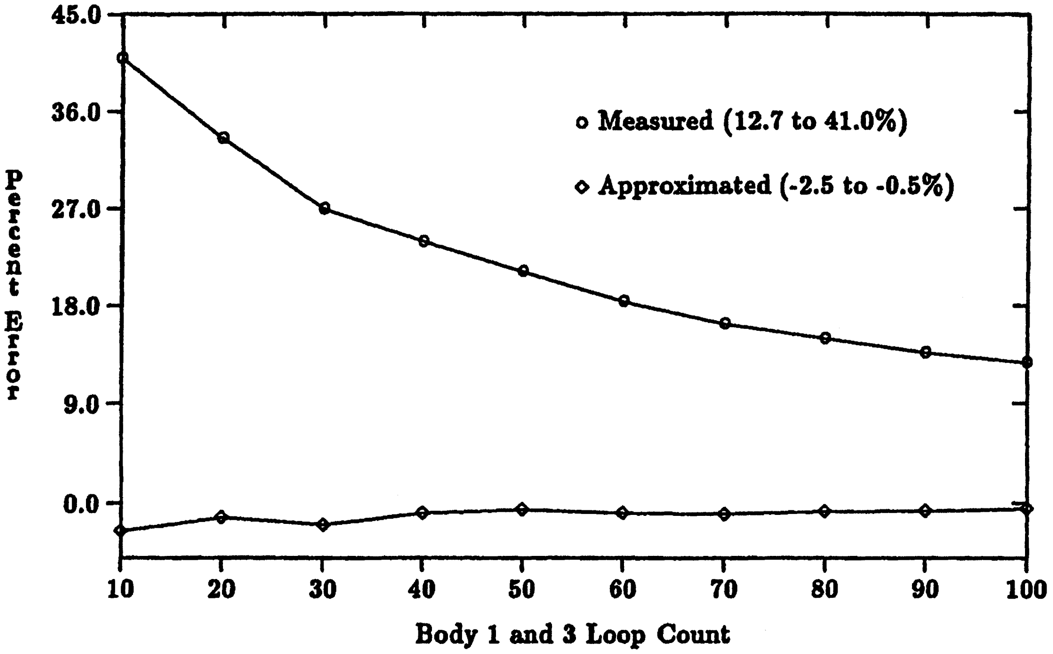


Figure 5.8: INCREASING-2, Forward, Synchronization, $d = 1$

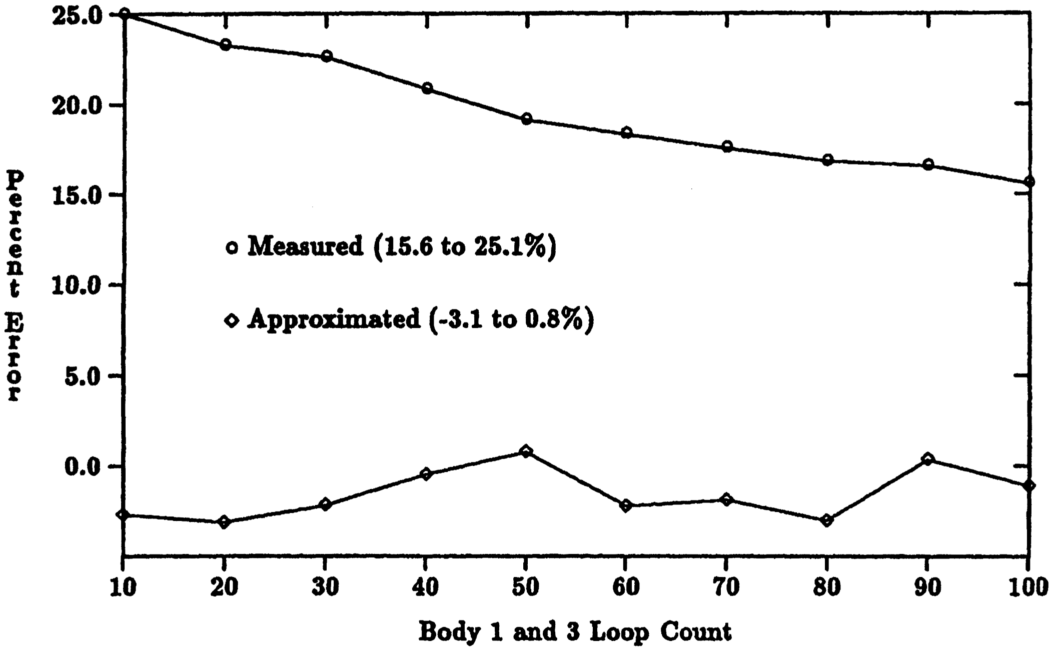


Figure 5.9: RANDOM-1, Forward, Synchronization, $d = 1$

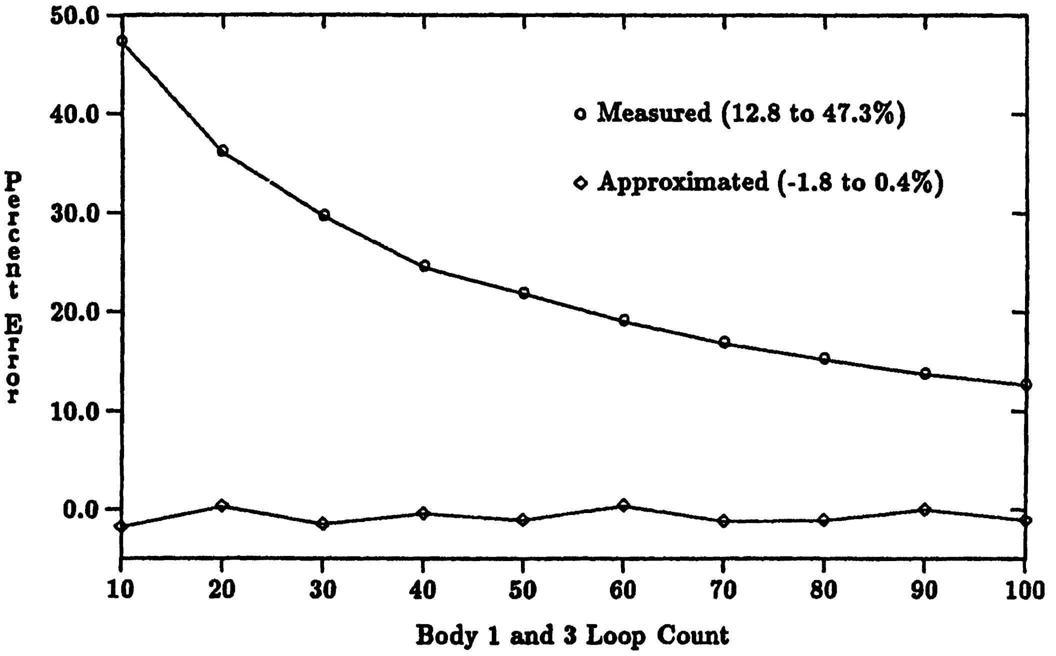


Figure 5.10: RANDOM-2, Forward, Synchronization, $d = 1$

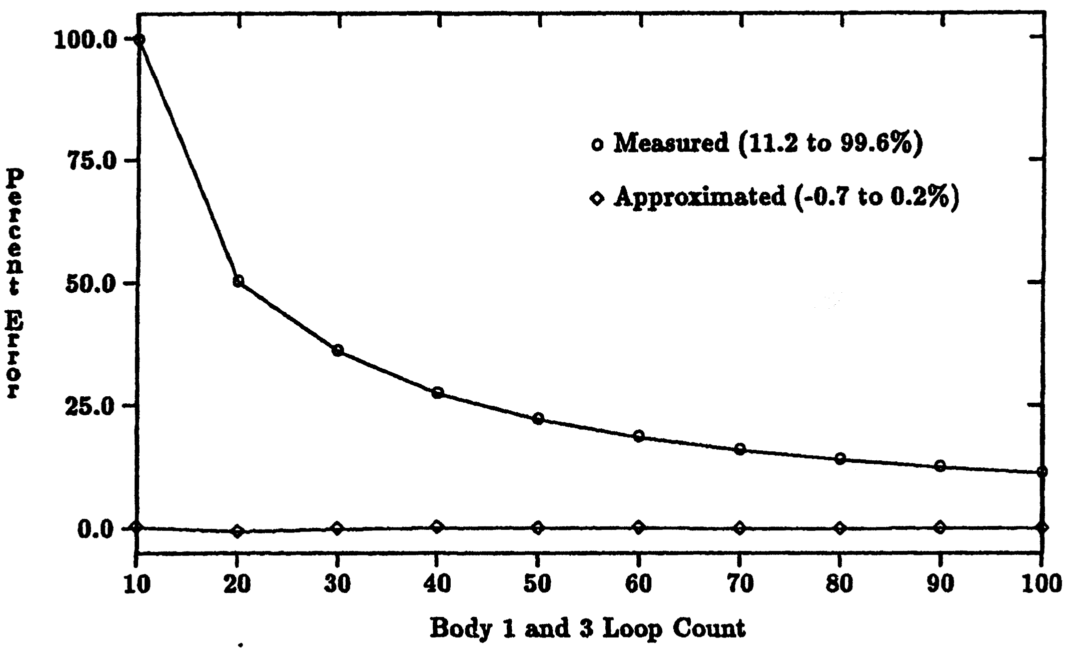


Figure 5.11: FIXED-1, Backward, Synchronization, $d = 1$

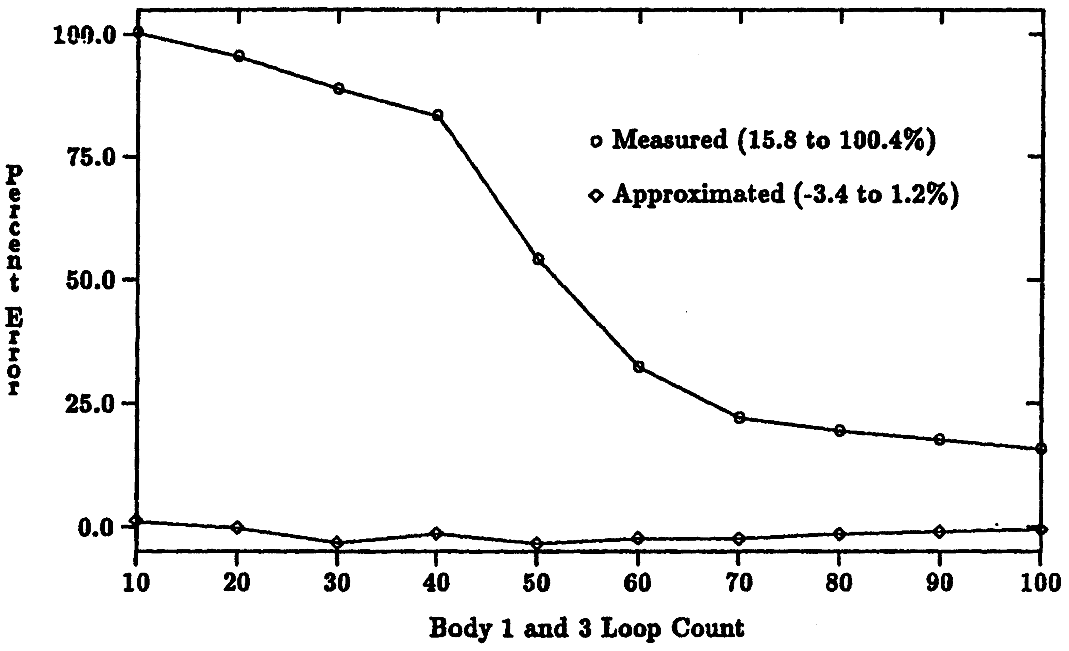


Figure 5.12: FIXED-2, Backward, Synchronization, $d = 1$

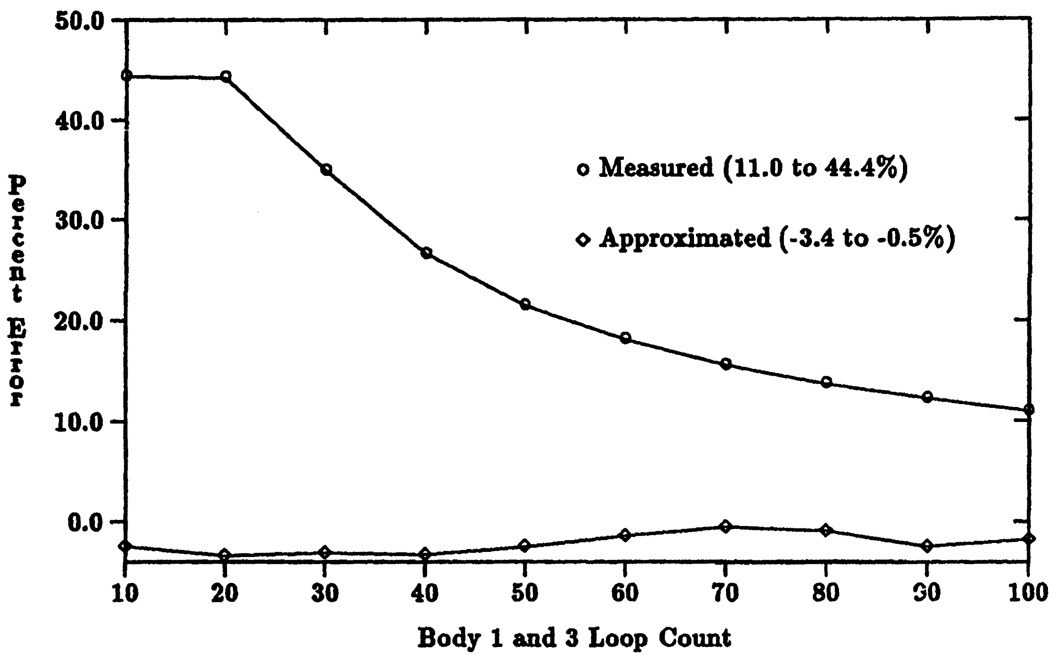


Figure 5.13: INCREASING-1, Backward, Synchronization, $d = 1$

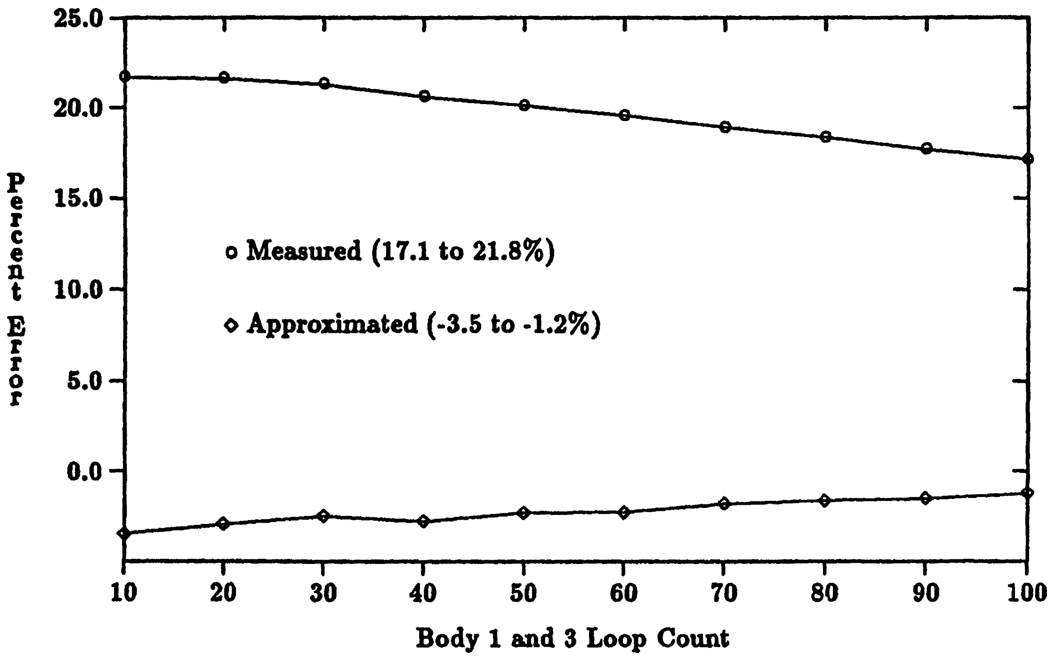


Figure 5.14: INCREASING-2, Backward, Synchronization, $d = 1$

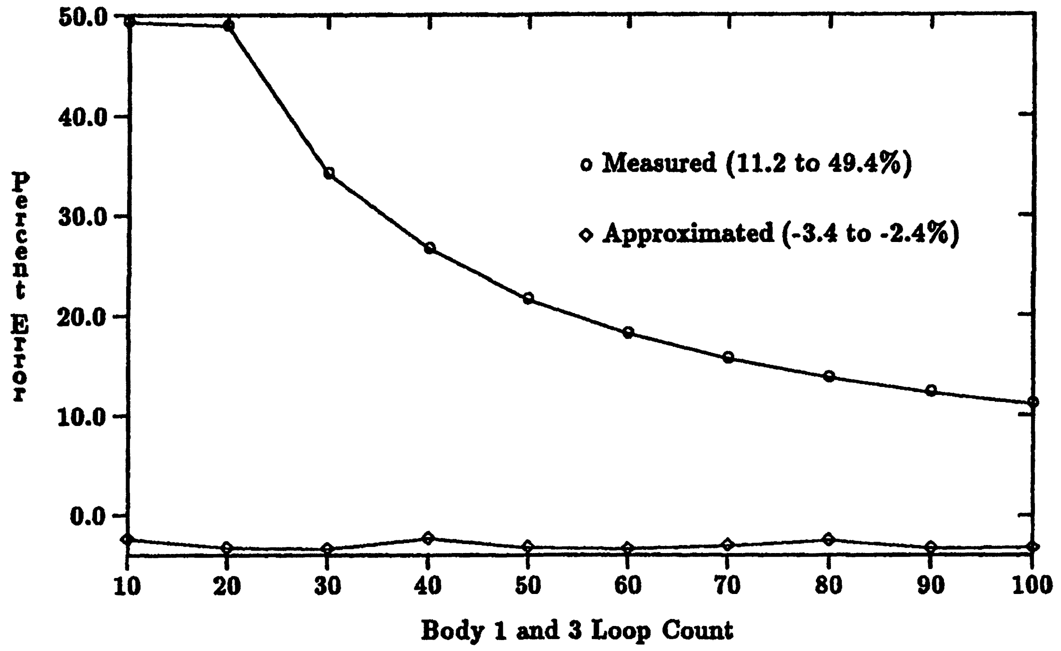


Figure 5.15: RANDOM-1, Backward, Synchronization, $d = 1$

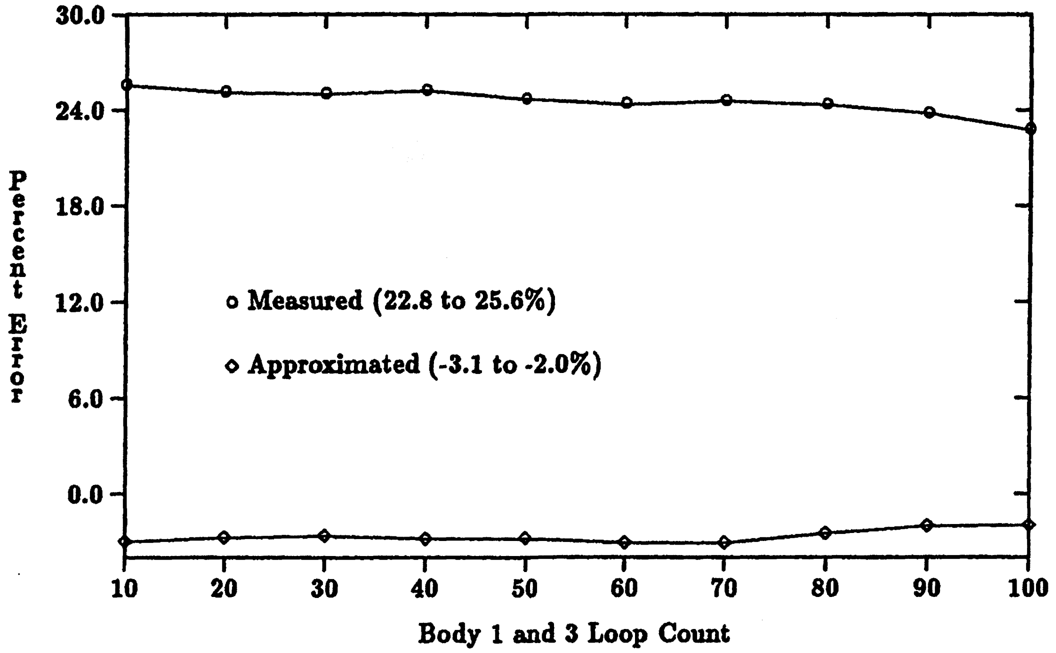


Figure 5.16: RANDOM-2, Backward, Synchronization, $d = 1$

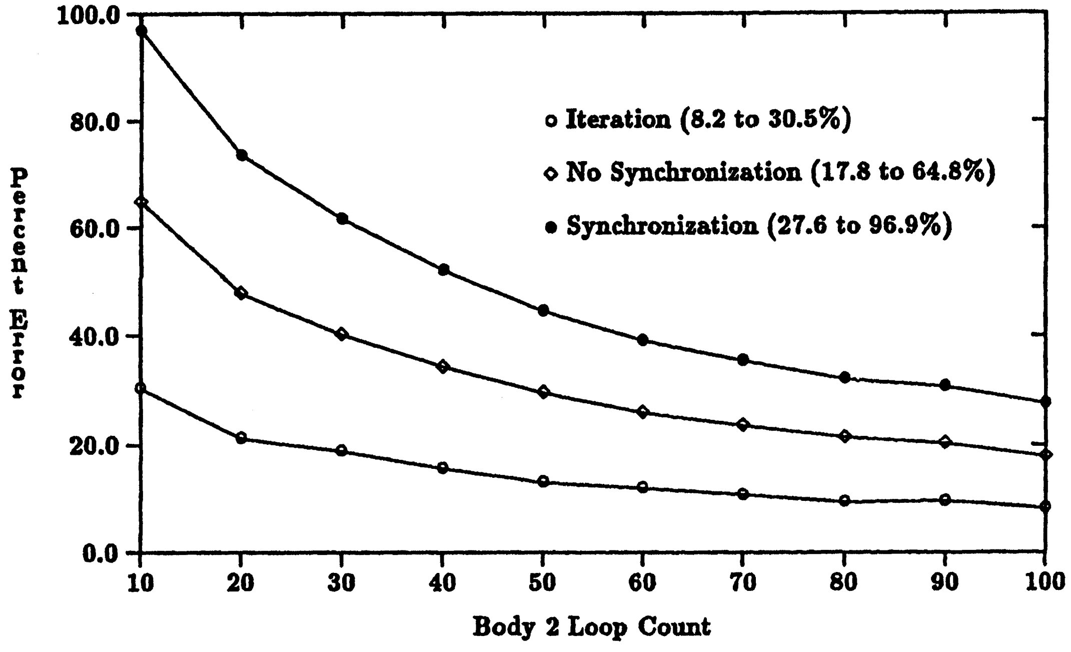


Figure 5.17: Measured Execution Error – FIXED-1, Forward, $d = 1$

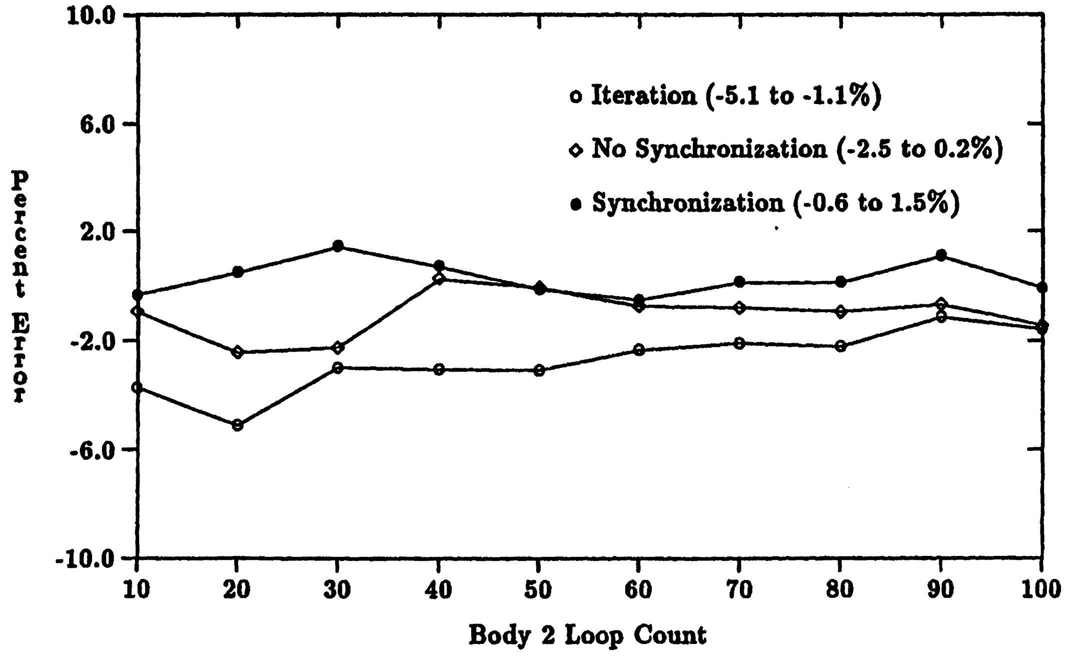


Figure 5.18: Approximated Execution Error – FIXED-1, Forward, $d = 1$

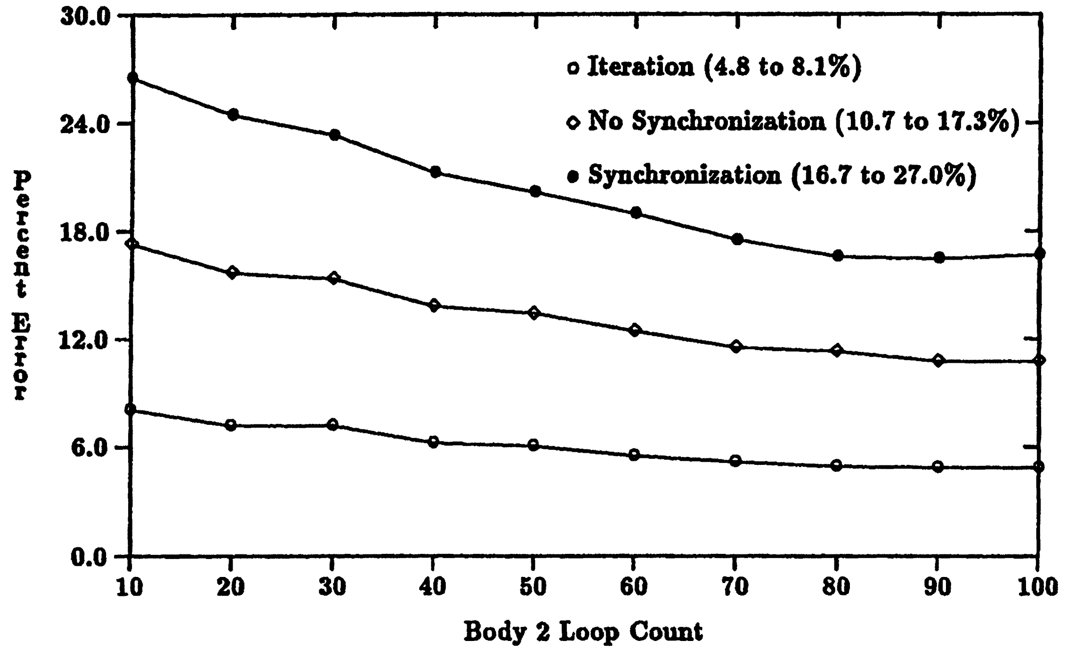


Figure 5.19: Measured Execution Error – INCREASING-1, Forward, $d = 1$

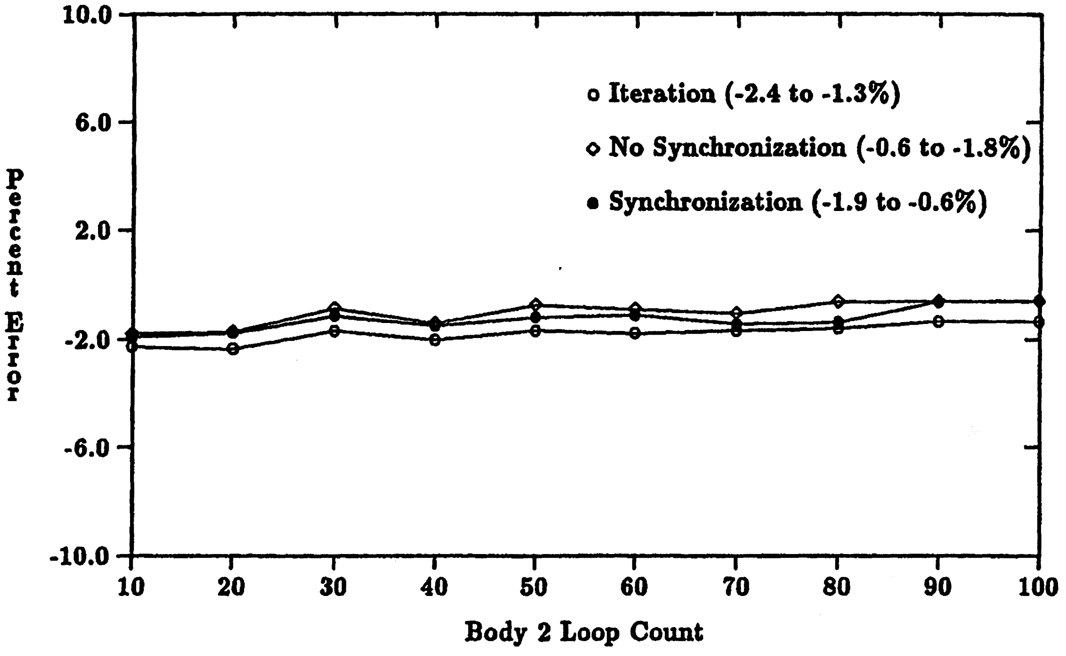


Figure 5.20: Approximated Execution Error – INCREASING-1, Forward, $d = 1$

mise from these and the other forward experiments that when an execution of a DOACROSS loop encounters little or no waiting, a timing-based approximation will be almost as accurate as an event-based approximation.

In the case of the backward experiments, waiting occurs during the DOACROSS execution. Figures 5.21 and 5.22 show the execution history trace for each processor for the measured and approximated execution of the FIXED-1 experiment with body 2 fixed at ten iterations using synchronization instrumentation. The execution time accumulated while in event 2 on the graphs is directly representative of synchronization waiting time. The approximated execution graphs, drawn to the same scale as the measured graphs to emphasize the degree of time adjustment, shows the approximate waiting behavior of the actual execution.

Figures 5.23 and 5.24 show the measured and approximated errors for the different instrumentation test cases for the set of backward FIXED-1 experiments. The measured error for the test cases with no-synchronization instrumentation is less than in the corresponding backward experiments because only body 2 is being instrumented. However, some of the instrumentation overhead is also hidden by processor waiting, keeping the measured error low. In other words, the instrumentation less severely impacts the critical path as in the forward cases. This reasoning also applies to the measured errors for iteration instrumentation. The measured errors for the test cases with synchronization instrumentation remain high because the instrumentation does affect the time between synchronization operations.

The approximated errors show the effects of waiting on the default timing-based analysis. Whereas the event-based perturbation analysis can use the synchronization information found in the traces of the test cases with synchronization instrumentation to recognize waiting situations during execution, the timing-based approach removes instrumentation overhead, disregarding execution overheads caused by synchronization for data dependencies. In fact, the timing approach can produce an approximated execution that not only has high error but that potentially violates the DOACROSS data dependencies. The large errors in the test cases with iteration and no-synchronization instrumentation are representative of the inability of the timing-based analysis to correctly resolve waiting behavior.

Figures 5.25 and 5.26 show similar results for the set of backward FIXED-2 experiments. In addition, we see the effects increasing the execution time of the body components outside the critical section has on the errors. There are large approximation errors for the test cases with

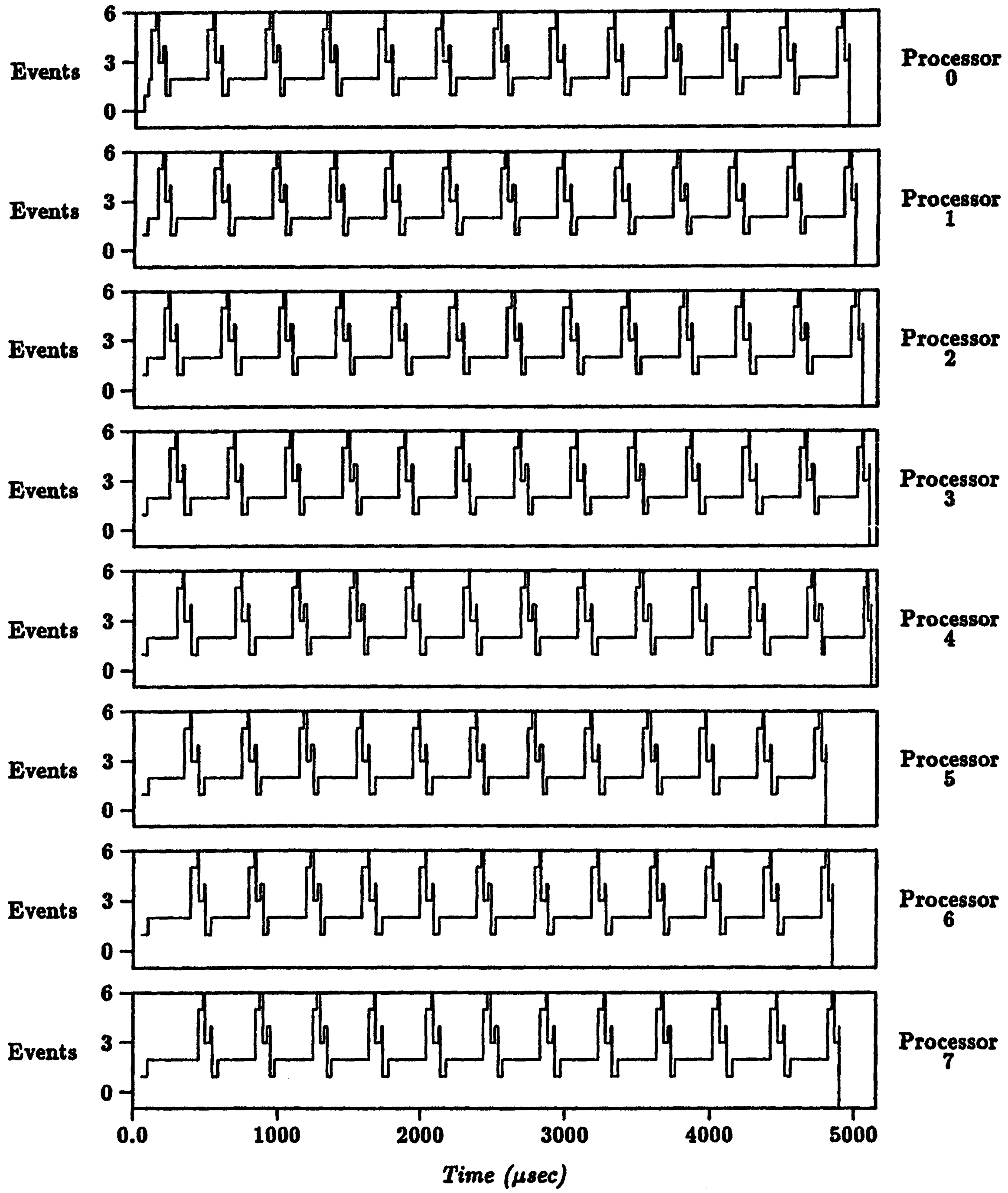


Figure 5.21: Measured Execution History – FIXED-1, Backward, Synchronization, $d = 1$

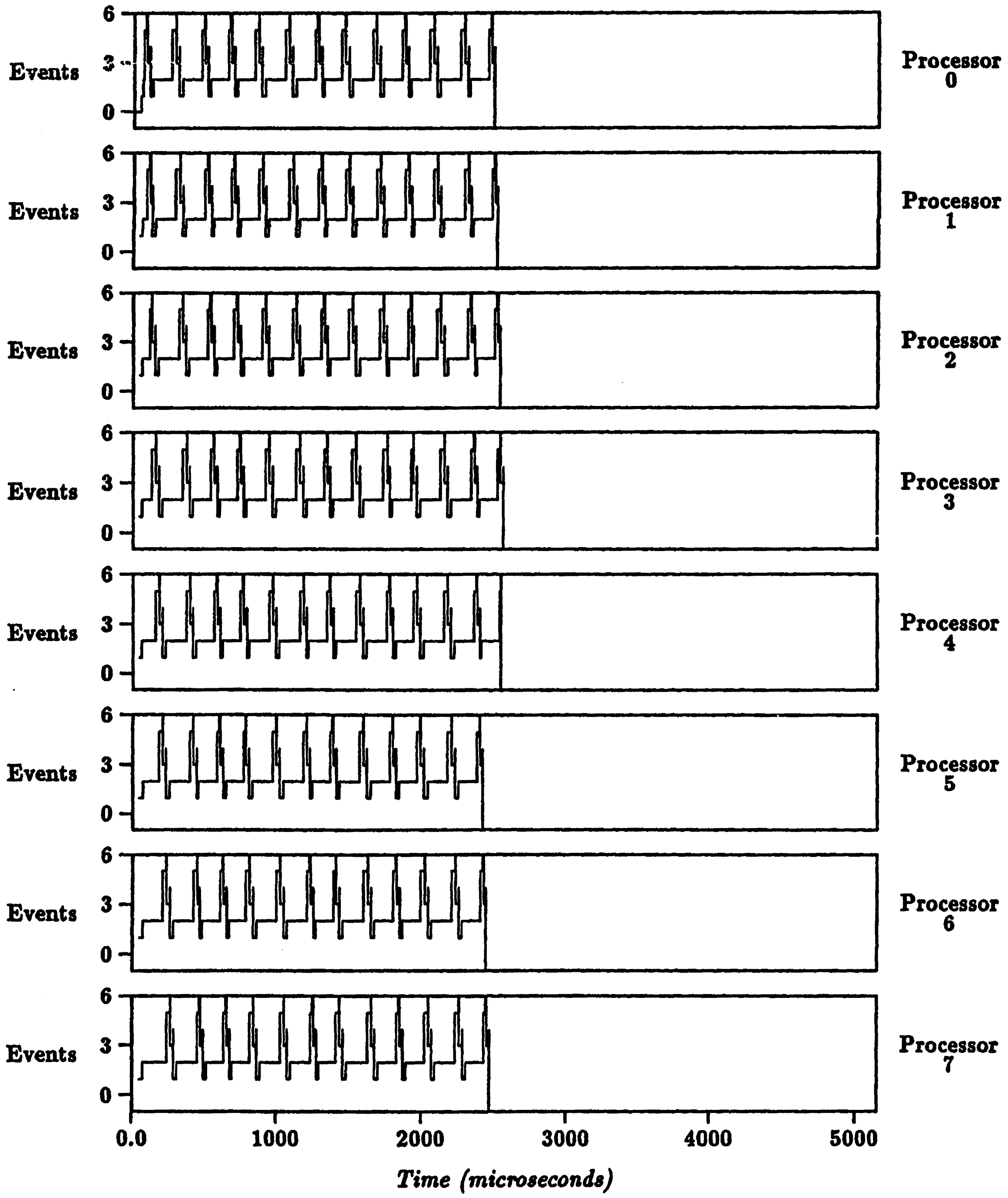


Figure 5.22: Approximated Execution History – FIXED-1, Backward, Synchronization, $d = 1$

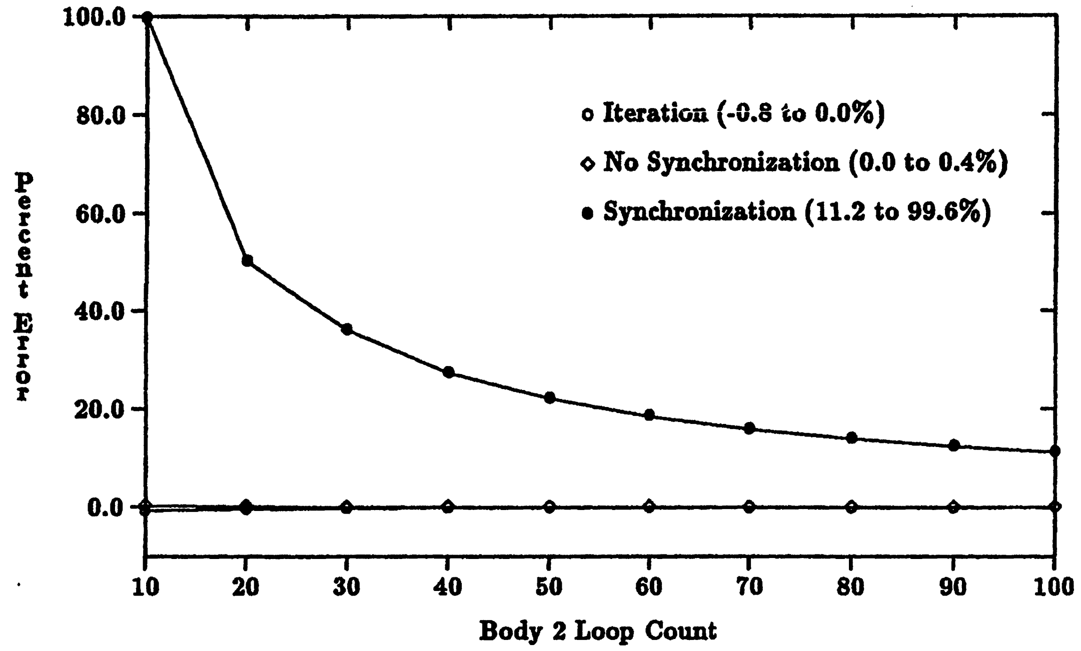


Figure 5.23: Measured Execution Error – FIXED-1, Backward, $d = 1$

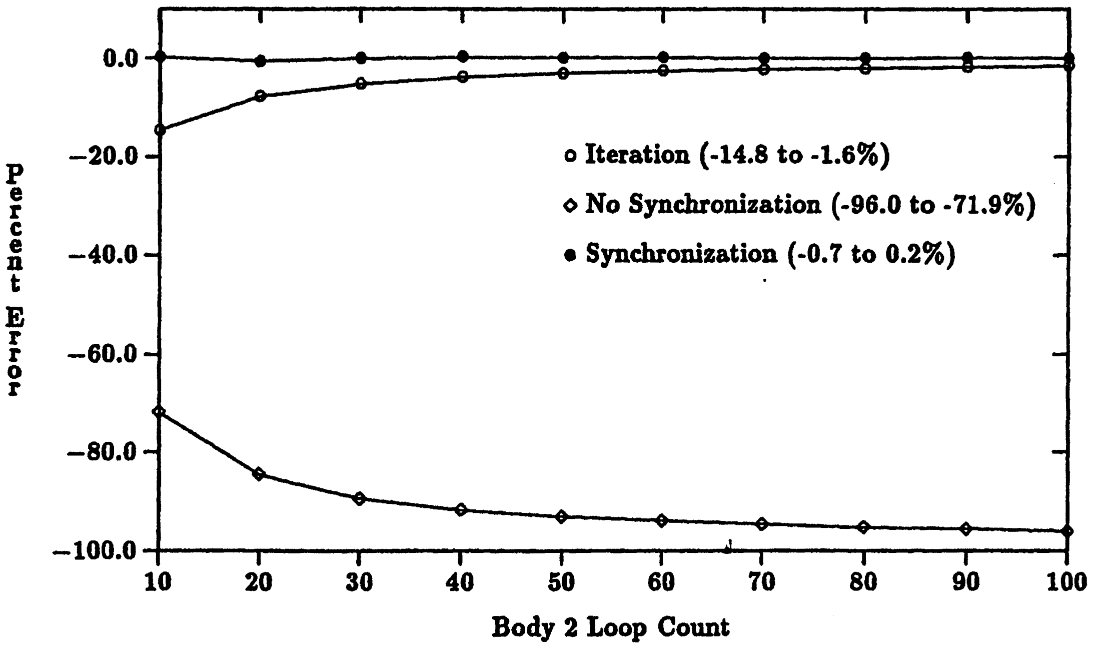


Figure 5.24: Approximated Execution Error – FIXED-1, Backward, $d = 1$

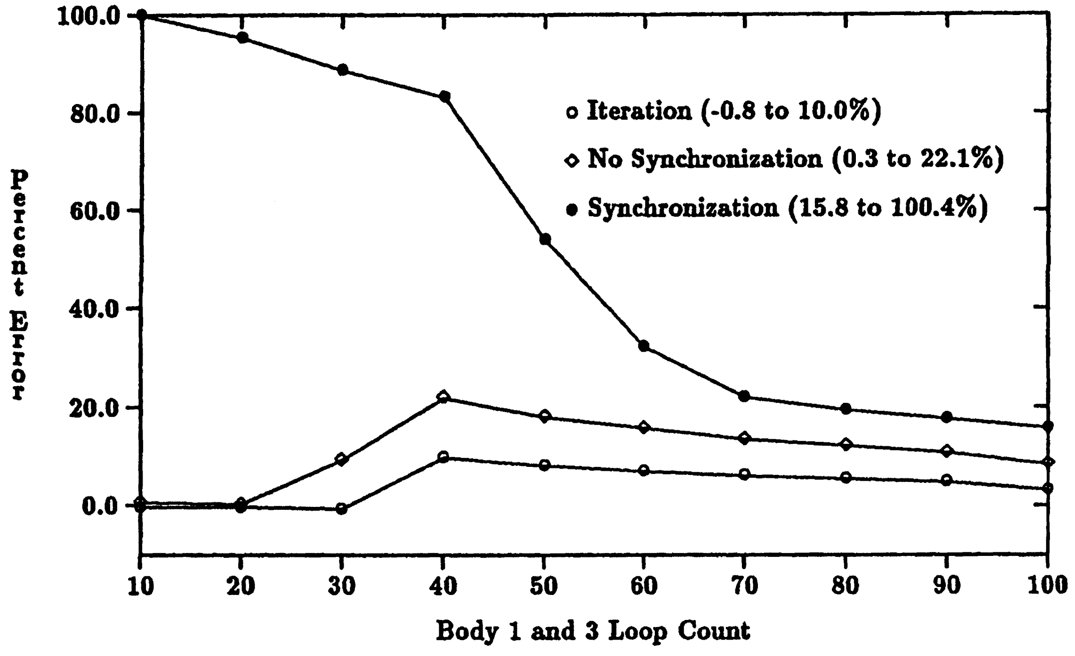


Figure 5.25: Measured Execution Error - FIXED-2, Backward, $d = 1$

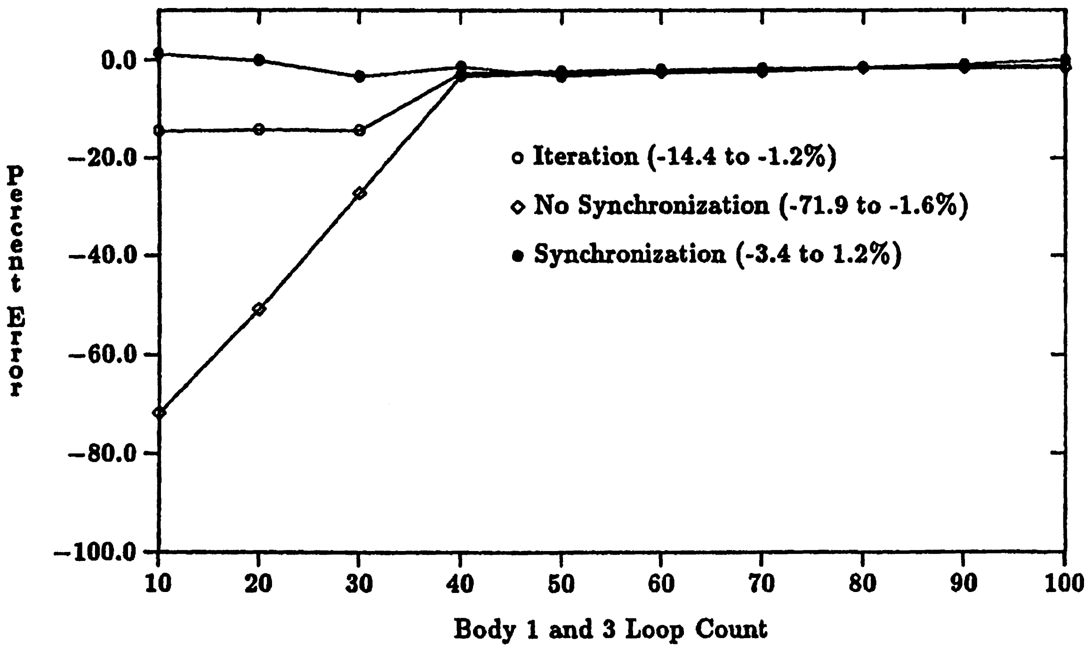


Figure 5.26: Approximated Execution Error - FIXED-2, Backward, $d = 1$

iteration and no-synchronization instrumentation until a body 1 and body 3 size of 40 iterations of the null loop is reached. It is at this point that we suspect the percentage of waiting time in the execution has been reduced effectively zero and the timing-based analysis could deliver accurate approximations. We also see support for this hypothesis in the measured errors — as the relative execution time of body 2 (and the associated synchronization waiting) decreases, the total execution time will be more affected by the instrumentation of body 1 and body 3. The instrumentation overhead will no longer be hidden and should, therefore, be reflected in the measured time. Event-based analysis revealed that waiting time decreased and became zero at 40 null loop iterations.

5.3 Event-Based Perturbation Analysis of the Livermore Loops

In §3.5, we saw that not all of the Livermore loops were amenable to timing-based perturbation analysis of concurrent execution. Loops 3, 4, and 17 in particular, show significant errors in the timing model approximations on the Alliant FX/80. All three loops execute as DOACROSS loops with advance/await synchronization to prevent concurrent access to a critical section. The concurrent execution behavior of the loops violates the assumptions of the time-based perturbation analysis. An event-based approach must be applied instead. To evaluate the effectiveness of our event-based perturbation analysis, in particular the models for advance/await synchronization in DOACROSS loops, we applied the event trace analysis tool to these three Livermore loops.

Figure 5.27 shows the structure of the three loops, including the placement of DOACROSS loop begin and end, and the advance and await synchronization operations. Every statement in the loop was instrumented — each node in the graph corresponds to a statement in the loop and, hence, an event in the trace. The statement dependencies are shown by the white arrows. The events following the advance and await operations are identified as special synchronization events by the analysis tool so that the advance/await semantics can be correctly enforced.[3] The end of the DOACROSS loops are handled as barriers.[4]

[3]As discussed in 5.2, the instrumentation for event perturbation analysis stores additional information with synchronization events to correctly determine their relationship. In the case of the loops here, we store the iteration number with every event.

[4]The Alliant FX/80 uses a hardware-based barrier synchronization mechanism to terminate parallel loops.

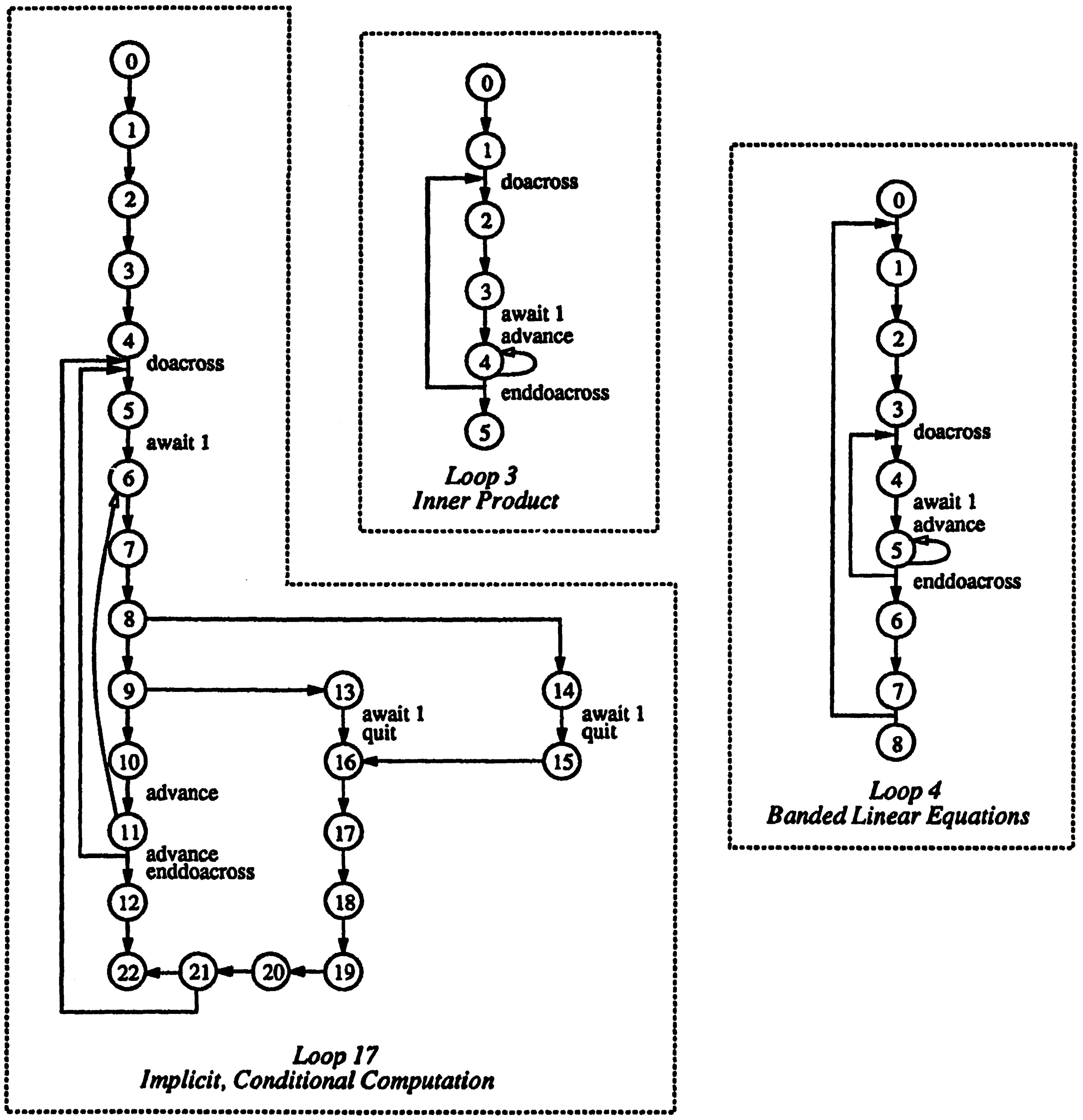


Figure 5.27: Lawrence Livermore Loops 3, 4, and 17

Livermore Loop	Execution Ratio			
	$\frac{\textit{Time Measured}}{\textit{Raw}}$	$\frac{\textit{Event Measured}}{\textit{Raw}}$	$\frac{\textit{Time Approximated}}{\textit{Raw}}$	$\frac{\textit{Event Approximated}}{\textit{Raw}}$
3	2.48	4.56	0.37	0.96
4	2.64	3.38	0.57	1.06
17	9.97	14.08	8.31	0.97

Table 5.5: Loop Execution Time Ratios per Instrumentation

Table 5.5 gives the ratio of different execution time approximations to the raw execution time of the loops. The $\frac{\textit{Time Measured}}{\textit{Raw}}$ ratio compares the measured execution time of the loops from a full instrumentation for timing analysis to the raw time. The $\frac{\textit{Event Measured}}{\textit{Raw}}$ ratio compares the measured execution time from a full instrumentation for event analysis to the raw time. The $\frac{\textit{Time Approximated}}{\textit{Raw}}$ ratio shows the accuracy of the approximation from timing-based perturbation analysis. For loops 3 and 4, the timing model approximates a smaller execution time (indicated by the ratio being less than one), than that measured with the raw instrumentation. As discussed in §3.5, the timing model does not treat the *advance* and *await* events as special, and waiting times that should result because of the synchronization are not maintained in the approximation. In contrast, the timing-based model approximates an execution time slower than actual for loop 17. Here the instrumentation in the critical section of the loop results in greater waiting during measured execution which cannot be removed by the timing analysis.

Applying event-based perturbation analysis to ensure the partial ordering of advance and await operations in the approximated execution significantly improves the accuracy of the execution time estimation, as seen in the $\frac{\textit{Event Approximated}}{\textit{Raw}}$ column of Table 5.5. In the case of loop 3, the actual execution time is 2.7 times that of the timing-based approximation (a -63 percent error). However, event perturbation modeling improves the approximation to be a factor of 0.96 of the actual time (a -4 percent error). The improvements in the loop 4 approximation are similar. The event model achieves a 6 percent error in this case. The most dramatic improvement in accuracy is found in the case of loop 17. Even with a 14 times slowdown in the measured execution, event-based perturbation modeling accurately resolves the timing of *advance* and *await* synchronization events to produce an approximation with only a -3 percent error. The advantage over the timing-based model is also apparent — a factor of over 8 in improved accuracy is achieved.

Processor							
0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
4.05 %	8.09 %	4.05 %	2.70 %	4.05 %	5.40 %	2.70 %	4.05 %

Table 5.6: DOACROSS Waiting Time in Loop 17

In addition to producing total execution time approximations, the event analysis can also generate statistics about loop execution such as the amount of waiting on each processor and the degree of parallelism across processors. As an example, we computed the percentage of total execution time spent waiting on each processor in loop 17. These results are shown in Table 5.6. A execution time history graph of waiting time for each processor is shown in Figure 5.28. The sequential portions before and after the parallel DOACROSS loop are shown as processor zero active.

From the waiting information, we can compute the level of parallelism in the computation. The average level of parallelism of loop 17, excluding the sequential portions, is 7.5. More insight into parallelism behavior can be gained from a graph of parallelism over time. This graph is shown in Figure 5.29. All the waiting and parallelism curves presented were generated from the execution approximations of the event-based perturbation model.

5.4 Comments

The event-based perturbation analysis of DOALL and DOACROSS loops has demonstrated several important points. Although DOALL loops are nondeterministic in their execution, default time-based analysis can result in significant approximation errors. The reason is that performance instrumentation can perturb how resources (particularly processing resources) are assigned during execution. This is also the case with DOACROSS loops. No performance data can be captured that will indicate the resource allocation strategy in the actual execution. This information must be obtained from outside the performance measurement. The event-based perturbation analysis uses this information to construct an event-driven simulation of the execution environment from which an approximated execution is produced.

The DOACROSS loop represents a constrained, partial-order execution. Although the problems with DOALL loops still exist, the perturbation effects of performance measurement is more

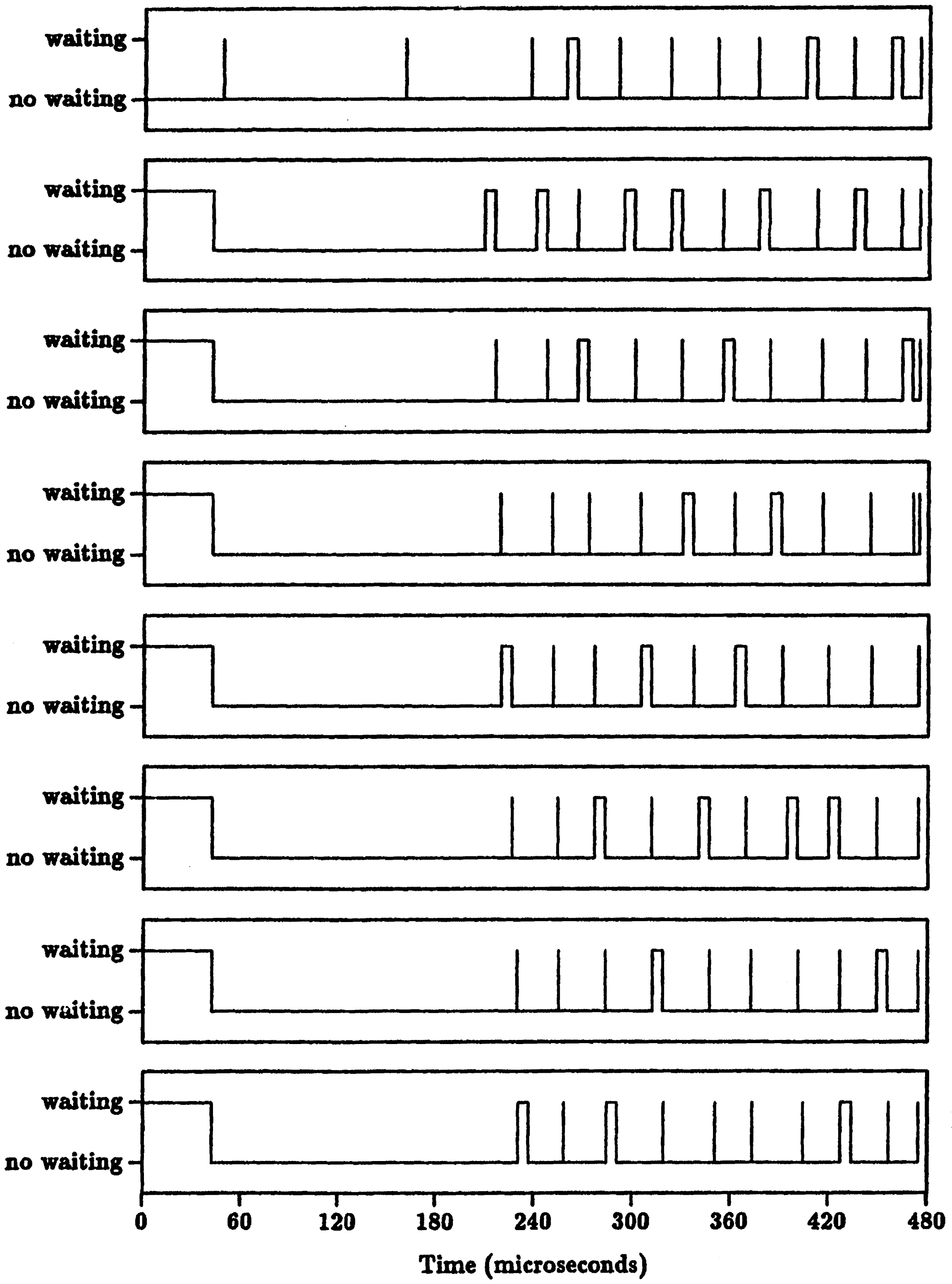


Figure 5.28: Approximated Waiting Behavior in Livermore Loop 17

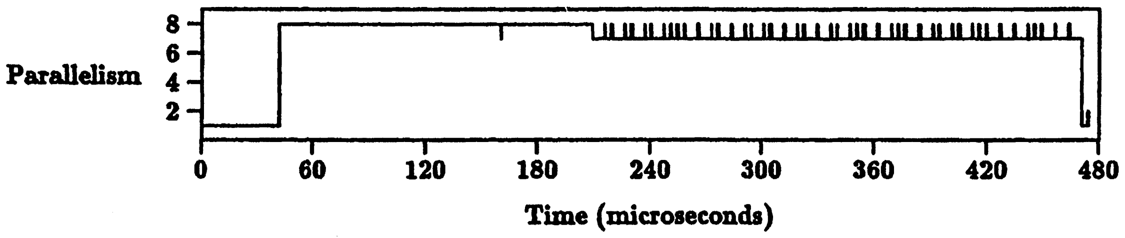


Figure 5.29: Approximated Parallelism Behavior in Livermore Loop 17

restricted. It is even possible for the performance intrusion to be hidden by other aspects of the computation. All synchronization operations must be instrumented in DOACROSS loops to correctly account for waiting behavior. This demonstrates the need in some performance measurement scenarios to risk more performance perturbation due to extra instrumentation in order to achieve more accurate performance approximations.

Chapter 6

Performance Visualization

What is to be sought in designs for the display of quantitative information is the clear portrayal of complexity. Not the complication of the simple; rather the task of the designer is to give visual access to the subtle and the difficult — that is, the revelation of the complex.

Graphics *reveal* data.

— Edward Tufte, "The Visual Display of Quantitative Information"

Performance visualization is simply the application of visual display techniques to the analysis of performance data. The main goal of performance visualization is to improve our ability to understand performance data. This can occur in two ways. First, there is the potential insight gained from a graphical representation of performance data that would be difficult or impossible to obtain otherwise. The challenge here is in determining the types of visual displays that are most useful for presenting performance data. Clearly, the attributes of the data will be important in evaluating the appropriateness of different display alternatives. Even without the "gestalt" insight, performance visualization serves as a tool for data exploration. It can be used to show all or part of the performance data at several levels of detail. Performance visualization environments allow the performance analyst to "investigate" the performance data by providing facilities to filter the data set, to define functional combinations or reductions of data elements, to select among various display choices, to compare different data values, or to inquire for performance data details. Performance visualization, in this respect, also encompasses the display of real-time performance data.

Although there is a well-founded discipline for the graphical presentation of statistical data [31, 32] and significant research advances have been made in visualizing scientific data [58], the visualization of computer system performance data has remained *ad hoc*, a practice engaged in only by a few. Certainly, there are past examples where graphics have been used to display performance data [49, 95], but it has not been until recently that tools which incorporate performance visualization as a main component of their design have begun to emerge. Below, we give a brief review of the developments in this area.

There has been a growing use of performance visualization ideas in integrated programming, debugging, and performance measurement environments, particularly for parallel systems [36, 45, 57, 59, 74, 115, 137, 170, 182]. The performance tools for the Erlangen processor array [59] are an early example of how graphical displays of processor execution events are used to observe parallelism. The PIE project [74, 170, 171] treated performance visualization as an integral component in the programming and instrumentation environment design. Although the extent of the performance views supported by the PIE system was somewhat limited, it is a good example of how visualization can be used for performance debugging [115]. Miller's IPS-2 system [137], the work by Fowler, LeBlanc, and Mellor-Crummey [57, 111], and Couch and Krumme's Triplex tools [36] all emphasize the use of performance visualization in the context of a parallel program measurement system. The Poker programming environment of Snyder [182] together with the Voyeur visualization tool [184], and the Schedule tool by Dongarra and Sorenson [45] demonstrate how the visualization of parallel programs is supported in a general programming system.

Other tools have focused on specific applications of performance visualization. Seager developed the GMAT tool [169] to graphically observe the operation of multitasking programs run on the Cray X-MP. Stone [186] introduced the notion of a *concurrency map* as a graphical representation of potential concurrency behavior of a program. The Seecube system of Couch [35] offers a variety of topological display choices for graphically representing program performance on a hypercube message passing system. Moviola [56] is an interactive execution history browser used to show the synchronization activity of concurrent programs. PIE has been used to visualize context switch behavior of parallel programs [114] for use in the analysis of scheduling algorithms. Dongarra has developed the MAP [23] and SHMAP [43] tools to visualize shared memory access patterns of parallel matrix algorithms.

Lastly, there is an increasing interest in the real-time display of performance information. Several of the tools described above support a replay mode for viewing execution histories, but few provide the ability to display performance data in real-time. The Triplex toolset [36] is an exception, allowing the user to view state variables and summary statistics from multiple processors as a program's execution proceeds. Katseff [93], with his Software Oscilloscope, investigated tradeoffs in the frequency and amount of data recording versus the rate of display update. Recently, Trammel [192] has studied the visualization of system behavior in real-time, focusing on efficient system performance measurement to reduce the observer (probe) effect and different tradeoffs in data presentation.

In this chapter, we develop three general themes of performance visualization. The first is that there is a need for performance visualization; that is, performance data analysis benefits from the use of graphics. We discuss the characteristics of performance data that make it interesting to apply graphics. Second, performance visualization must be interactive and automatic. If the performance visualization process is awkward or time-consuming, the performance analyst will seek simpler tools. It must be easy to use and it must provide the performance analyst an automatic way of displaying and interacting with the performance data. Lastly, performance visualization is an experimental process, dependent on the need to gain different perspectives on the performance data. Performance visualization environments must provide the performance analyst the flexibility to explore the performance data and make display choices based on current observations.[1]

We will describe, by way of example and discussion of the above three themes, several prototype performance visualization tools we have developed; see §6.2. Our main interest in describing these tools is to demonstrate the utility of visualization for performance observation and to motivate the design of a general performance visualization environment architecture. The ideal performance visualization environment would support the definition and creation of performance displays that can be easily matched to different performance data values. We define a general performance visualization architecture in §6.3 that reflects these goals. A prototype performance visualization environment developed following the general architecture guidelines is discussed in §6.4.

[1]This theme has many corollaries with Tukey's concept of exploratory data analysis [195].

6.1 Performance Data and Displays

If the goal of performance visualization is to aid in the understanding of performance data, we need to have a better characterization of the data itself and of the performance phenomena we want to observe. As the performance of a computer system is inextricably tied to the combined performance of its constituent hardware and software levels, for any performance experiment, we may collect performance data about the hardware, the system software, the run-time environment, and the application software. We must be able to examine this multi-level data both separately and collectively. Separately, the performance data of each level reflects both the use of components at that level and their dynamic interaction. Collectively, the data provide a global performance view and allow one to correlate inter-level behavior.

Performance data can be generally characterized according to the type of performance analysis that it supports: *static* or *dynamic*. Static performance analysis reports performance summaries or instantaneous performance state. The performance data often are in the form of usage statistics (counts and times), such as in the case of program profiling, but also can be samples of state variables or performance metrics at particular times. Dynamic performance analysis, on the other hand, captures changes in performance behavior, allowing the performance analyst to observe time-based performance phenomena. The performance data in this case must explicitly or implicitly encode timing information; typically the performance data is in the form of a time-stamped trace.

The type of performance data collected depends on the requirements of the performance experiment being conducted. In general, the complexity of performance data measured (types and amount) is directly related to the complexity of the performance phenomena being observed. To understand a complex performance phenomenon, we must be able to analyze a potentially large volume of information that describes complex relationships within and between performance levels. In addition, the visualization of the data will depend on a knowledge of the performance behavior the data represents.

6.1.1 Performance Data Views

We adopt a framework proposed by LeBlanc, Mellor-Crummey, and Fowler [111] for classifying performance data as different "views" of a program's execution. They draw an important

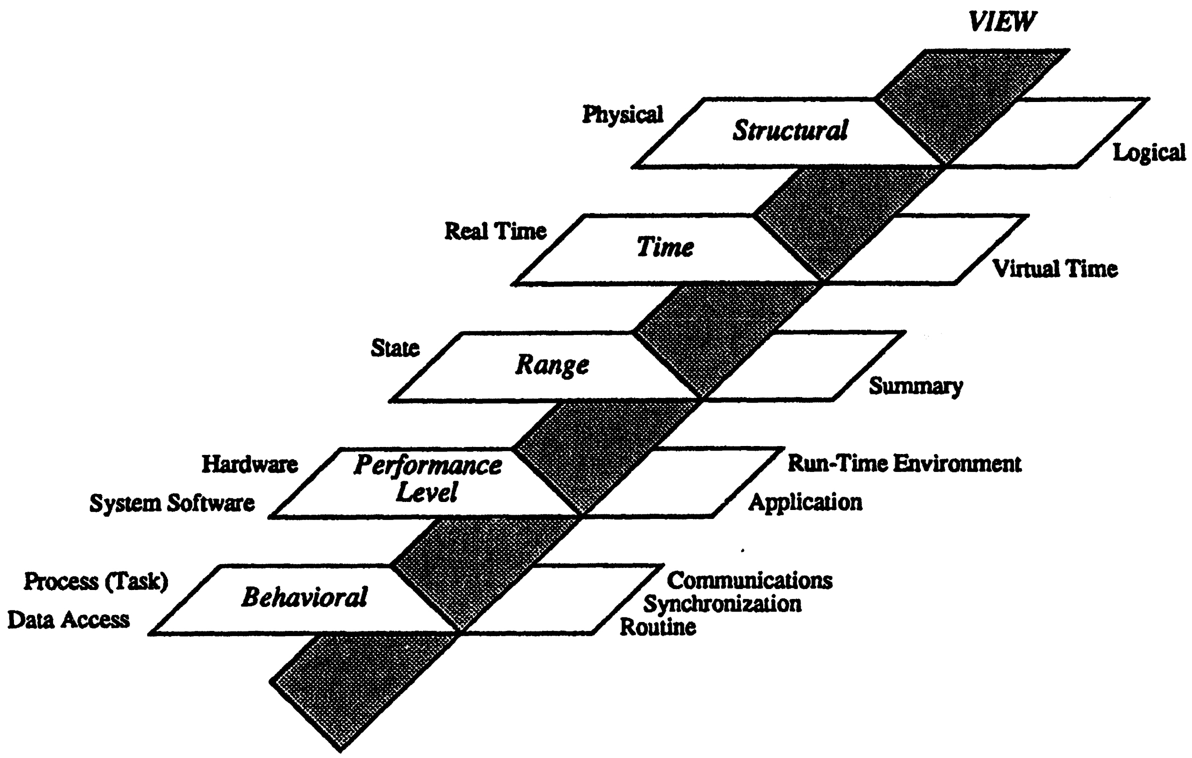


Figure 6.1: Performance View Dimensions

distinction between a view and a visualization — a *view* generally defines *what* performance information is to be presented; a *visualization* describes *how* the information is displayed. Although our discussion of fundamental view dimensions (see below) and types is different than [111], the basic approach is similar. We give a more general definition of view dimensions which includes the performance data levels.

Performance data views are described across several dimensions and any particular view can be a composition of more basic views; see Figure 6.1. One fundamental dimension (or view attribute) contrasts a *physical view* with a *logical view*; this might be regarded as a *structural* dimension. A physical view provides information on the operational use of physical resources during execution. This could be the allocation of processors, the functioning of a memory system, or the operation of communication resources. A logical view relates the performance data to some conceptual execution framework. An example can be taken from message-passing systems. The physical view of one node exchanging a message with another might indicate the identities of each node, the size of the message, and the communications path in terms of intermediate nodes and links. A logical view, on the other hand, might instead identify the message type, and the sending and receiving tasks, but not the processing nodes.

The notion of *time* need not be included in the definition of physical and logical views. More appropriately, time reflects another fundamental view dimension, ranging from a *real time view* to a *virtual time view*. Essentially, a point on the time dimension defines ordering relationships between execution events. A real time view bases this ordering on some derivative of a real-time clock. A virtual time view defines the observable ordering of events based on causal relationships; for instance, Lamport's notion of *logical time* as defined by the *happened before* relation.[2] A performance data view including the time dimension implies the existence of the information needed to derive timing relationships, real or virtual. We can add the time attribute to the above example by recording either the actual time the message was sent and received (a real time view) or information concerning the relative order of message communication (a virtual time view). Notice that both time views are valid with either the physical or logical views.

Another fundamental view dimension contrasts performance data about the state of an execution with data that summarizes execution performance; a *range* dimension. We define a *state view* as embodying performance information relative to an execution state. A *summary view*, however, reports performance data accumulated over multiple execution states. Although there is an intuitive continuum of state and summary views from an execution state view at the smallest observable instance of time (real or virtual) to a performance summary view of the entire execution, the difference between a state view and a summary view depends on the definition of execution state. An example of a state view might be the number of floating point operations produced by a subroutine for a particular call instance; the state here is defined by a subroutine invocation. An associated summary view would be the total number of floating point operations produced by that subroutine for the complete program execution.

The three fundamental performance data view dimensions discussed above define certain properties of the data but not what the data represents. A fourth view dimension can be defined based on performance data level. A performance level view associates the performance data with the analysis of a particular component of the execution. Views of this type include the

[2]We use the term virtual time instead of logical time so as not to confuse it with a logical view.

hardware view, *system software view*, *run-time environment view*, and *application view*. This deviates from [111] which treats views of different performance levels in the context of *process interaction views* and *process state views*. Our position is that performance levels define a more basic dimension for performance analysis since any performance experiment will necessarily involve all performance levels even if data is captured for only a subset. A performance view at both the hardware and system software levels can provide important insight for system design and analysis. And because system performance is manifest in the application software, system level performance visualization indirectly provides application performance insight. However, a system performance view must be directly coupled with an application performance view to understand the interactions of different performance levels.

One final fundamental dimension is that of performance behavior. A *performance behavior view* relates the performance data to a particular performance behavior. It places the interpretation of the performance data in the context of some behavioral abstraction [14] of interest. Some the more important behavior views might be the *process (or task) behavior view*, the *data access behavior view*, the *communications behavior view*, the *synchronization behavior view*, and the *routine behavior view*. More than any of the other dimensions, the performance behavior dimension embodies performance data meaning.

Any performance analysis will necessarily be based on a multiplicity of execution views. No single view will be complete — any particular view, while providing insight into some performance phenomenon, will insufficiently characterize some other aspect of the execution. The fundamental view types are too general to be applied directly. Rather they reflect the basic view attributes from which more refined views will be composed in the context of a specific performance analysis. Additionally, they define the basic view properties that will be important to consider when developing appropriate performance display approaches.

6.1.2 Performance Data Displays

The performance data from any experiment can potentially be large and can represent multiple views of an execution that describe complex relationships across several performance dimensions. For each performance view, there are many different ways to present (visualize) the information. Tufte's definition of a graphic visualization is one that attempts to efficiently communicate a complex set of quantitative ideas [194]. For performance visualization, we use graphic displays to express the characteristics of the different view dimensions. Although graphics can display a large amount of quantitative information, interesting phenomena can be lost in the sheer volume of information. Performance data must be presented in ways that emphasize important events while eliding irrelevant details, but that also support performance data investigation, including selecting performance views, changing display types and their attributes, and querying for performance data details.

The main question for performance visualization concerns what types of displays are appropriate for the performance behavior one wants to observe. Because the most useful set of displays are likely to be determined only from experience and will depend on the personal preference and needs of the performance analyst, we approach the question by listing several display types and commenting on their use in performance visualization.

The following display types have been or can be used in performance visualization:

- two-dimensional (2-D) line and scatter plots
- 2-D surface and contour plots
- three-dimensional (3-D) plots
- analog and digital meters
- histograms and LEDs (discrete histograms)
- PIE charts
- Chernoff faces and Kiviat diagrams
- strip charts and Gantt charts
- activity profiles and timeline displays
- event graphs
- time-process diagrams, synchronization graphs, and concurrency maps
- execution graphs, task graphs, and callgraphs
- activity matrix displays
- topological displays

Two-dimensional and three-dimensional plots have long been used for the display of general statistical data [195, 31], including multi-dimensional performance statistics. Parallel execution speedup, floating point performance, and memory reference performance, among other performance metrics, have been presented using line plots. Scatter plots can effectively display density distributions and have been used to show summary views of memory referencing behavior [43] and the distribution of floating point performance across application program routines [122]. Surface and contour plots highlight regional activity within a data set. We have used these displays, plus 3-D perspective plots, to show both physical and logical views of message communication behavior [124].

Unlike 2-D and 3-D plots, analog and digital meters, histograms, LEDs, PIE charts, Chernoff faces, and Kiviat diagrams only show one or a few performance values. They can be replicated or used in combination to show more values but they are particularly suited to displaying performance state values and reflecting changes in that state, not in the display of large performance data sets. Meters, histograms, and LEDs can be used to show any single value performance metrics, including processor utilization [21], process state times [79], and certain virtual memory statistics [192]. In addition, histograms and LEDs are applicable to the display of physical hardware state (e.g., showing buffer queue lengths in a switching network [147]) or the display of physical state system views (e.g., message buffering in a message passing system [35]). The graphics are simple enough in these displays that the performance value is easily conveyed and changes in the value easily observed. Chernoff faces [30] and Kiviat diagrams [143] are presentation devices that take advantage of our visual system's sensitivity to subtle changes in human facial expression and shape, and to detect small deviations from perfect symmetry in star designs, respectively. Possibly because of their cartoon-like appearance, Chernoff faces have not been applied in a serious manner to performance visualization. Kiviat diagrams, however, have been extensively used to show physical hardware and system performance views [95, 52].

Strip charts and Gantt charts [69] are general purpose 2-D plotting displays but with time (real or virtual) as either the abscissa or the ordinate. These displays have been used in many ways, including to show CPU utilization [137], speedup [18], page faults [192], memory access rates [183], and levels of parallelism [111, 103, 115]. Just as strip charts and Gantt charts show how a performance value changes over time, activity profiles [74] and timeline displays [35] show when different components of an execution are active over a time interval. A 2-D graph format is still used except the execution components, rather than the performance metric, are placed on the non-time axis and lines are used to represent regions of time the component is active. The activity profiles in [74] and [114] are used to show system and run-time behavior through process and task activity. Couch [35] uses timeline displays to shown hardware and system level message communication activity.

Event graphs are a more sophisticated form of activity profiles and timeline displays because time regions are graphically-coded to reflect different types of activity (or events) in the same display. Seager [169] used event graphs effectively to show multi/micro tasking behavior on

Cray multiprocessors. In his displays, the activities on the tasks are shown as different event icons representing multitasking operations. More recently, Lehr [115, 114] has used event graphs to show time-based process and CPU views of application performance, where the event regions are colored to indicate the execution of a particular routine or tasking operation. System views also have been displayed in this way, with regions indicating different process states and context switch events [114, 161].

An enhanced form of the event graph is found in time-process diagrams [81], concurrency maps [186], and synchronization graphs [56]. Time is still the fundamental axis with the other axis most often describing application processes or tasks. The unique feature of these displays is the time-based representation of process interaction showing both potential and observed temporal relationships among the processes. Processes interact in two ways: through communication (the exchange of data information) and through synchronization (the exchange of control information). The IDD tool [81] shows message transmission and receipt between processes in a message passing system. Moviola [56, 111] displays synchronization actions between processes, helping to visualize process-local and program-global dependency behavior. Stone [186, 185] uses concurrency maps as a graphical way of displaying potential concurrent combinations of independent and dependent process events. This is an interesting example of showing a logical and virtual time view in a single display.

Execution graphs, task graphs, and call graphs all show a state view of program execution performance via a program structure display. The components of the program are shown as nodes in some logical relationship, usually a hierarchical task invocation graph or routine calling graph, and are graphically animated as the program state changes during execution. The PARET tool [147] uses execution graphs in a general manner to characterize the interactions of tasks, modules, and objects in a parallel computer. The Schedule tool [45, 44] uses an acyclic directed graph to show the structure of a parallel application's execution. In both tools, during execution replay, the nodes of the graph are highlighted to indicate the execution state of the components. A task graph, in the form of a task genealogy tree, is used in GMAT [169] to show task creation history, task state (running or waiting), task operation (multitasking and microtasking events), and the assignment of tasks to processors. PIE [74] has promoted the idea of a general purpose call graph display that allows any program execution object with parent/child relations to be represented in the form of a tree. The visualization system of

Zimmermann [199, 200] provides a call graph display of execution flow where modules, routines, and other program components, such as synchronization objects, are included. A multi-threaded call graph display has been used in the Faust environment [76] to show the routine calling state of parallel execution threads, and Miller [137, 198] has used a program call graph to show critical paths in a program.

Execution graphs, task graphs, and call graphs reflect application state views; activity matrix displays and topological displays have been used to show both state and summary views but with a hardware and system configuration representation. Activity matrix displays associate each axis with some set of physical or logical resources of the system. Each matrix cell graphically reflects the activity occurring when the resources represented by the corresponding coordinates are taken in combination. Voyeur [184] humorously uses activity matrix displays to show system load balancing behavior between processors as animated sharks and fishes. Couch [35] has made use of activity matrix displays, as well as topological displays, to show hardware and system performance of hypercube message passing multiprocessors. His topological displays include several representations of the logical hypercube interconnection of processors: binary Pascal triangle, Gray code, multi-dimensional hypercube, FFT butterfly, and toroidal grid. During execution replay, the nodes and links in the displays are colored to signify the current state of various performance statistics, such as processor utilization, and change color to indicate different activity within the system, such as the use of a link for message communication.

Performance data can not only characterize instantaneous or cumulative performance on different levels, but also can embody time-varying performance dynamics. Phases and transitions in performance behavior can be seen only by examining time-relative sequences of performance data instances. Although some of the performance displays above are specifically designed to show changing performance values and state over time, any of the displays through animation or multiple windows can be used to show time-based performance behavior.

The time dimension can be presented either spatially or temporally, corresponding loosely to a state-based or summary-based (historical) view, respectively [111]. The Gantt charts, activity profiles, and timeline displays use space to present time, translating the temporal dimension of performance data into a spatial dimension, usually in the form of an axis on a graph. Temporally proximate performance data are shown as spatially proximate in the display. In addition, real-time views can be shown with the true timing intervals maintained in the display, or using "time warping" to graphically compress time regions with little or no performance data as in the case of GMAT [169].

Temporal presentation of time-based performance data uses animation [25] to divide an execution into time frames. Typically, the execution state is represented through a combination of statistical displays (e.g., plots and meters) and structure displays (e.g., task graphs and topological displays), onto which performance data is superimposed over time. The changing graphical attributes of the displays reflect the performance dynamics.

In addition to the basic display types and the methods for displaying time, many graphical techniques applied in scientific visualization to present complex quantitative data [58] could be used in performance visualization graphics. There techniques take advantage of the abilities of the human visual system to interpret and identify anomalies in false color data [46], and to extract features and recognize patterns in graphic images. Performance visualization of complex parallel systems, in particular, pose a vexing performance interpretation problem of displaying potentially large amounts of performance state data in ways that emphasize important behavior while eliding irrelevant data. The techniques of scientific visualization for visual data compression, visual cuing, and sophisticated graphics make it natural for this problem.

However, depending on the monetary constraints for performance measurement and visualization, there may be practical limits to the range of dispay techniques a performance analyst can apply. For instance, scientific visualization often requires expensive, high-performance computer equipment for graphics algorithm processing and rendering. If the techniques and tools for performance visualization are to find pervasive application among performance analysts, the system used to display performance data should be only a small relative fraction of the cost of the system being measured. For most performance analysts, high-performance graphics systems are still a luxury, but this does not mean we should not continue to investigate approaches.

In another example, near real-time processing and display of detailed performance data (e.g., memory reference patterns) implies prodigious, often unrealistic, computing, storage, and display requirements. *A posteriori* examination of performance data means that *all* data of potential interest must be captured *a priori*. Despite the consequent increase in storage requirements, this is often desirable — it permits performance data browsing across the entire range of execution, and it permits data capture at a level of detail incompatible with near real-time processing. However, *a posteriori* examination also precludes dynamic system or application reconfiguration based on observed performance. Real-time display of even a portion of the captured data would permit the performance analyst to selectively enable and disable performance instrumentation based on observed behavior, reducing the storage requirements.

Despite certain advantages of interactive, real-time performance display, for most performance visualization systems, *a posteriori* display data is unavoidable because there is insufficient communication bandwidth to transmit performance data to an external host without distorting the performance being measured; see §2.5. Furthermore, the analysis and visualization hardware must be able to process the performance data at the rate it is received. The physical operation of graphics monitors place an additional restriction on the ability to visualize dynamic changes in performance data in real time. If the frequency of change in a performance metric or state is faster than the monitor's refresh rate, normally once every $\frac{1}{60}$ of a second, only part of the performance data will be shown. The human eye also limits the maximum rate that changes in visual context can be perceived. Finally, the real-time graphics processing required to render a performance image can further restrict the rate performance data is presented.

For the application of advanced visualization techniques, including real-time display, to performance data presentation, a balance must be reached between the need to visually observe performance state and dynamics, and the capabilities of the available measurement and display environment. To raise the level of performance observability, we must enhance performance analysis and display techniques, but we must do so aware of the fact that to receive broad acceptance, the resulting tools must be easy to use and cost-effective to apply.

6.2 Case Studies in Performance Visualization

The purpose of performance visualization is to lend insight into the performance data while providing an environment for performance data investigation. The insight gained from presenting performance data views via different performance displays can only be assessed in the context of existing visualization tools and their use in real performance analysis scenarios. Below we describe several case studies where tools have been developed for specific performance visualization purposes. Our goal is to demonstrate the utility of the performance visualization techniques in the context of these tools both in terms of performance data insight and support for performance investigation. We use the performance level addressed by the tools to organize the discussion.

As discussed in the previous section, performance data can be presented in a variety of ways. The tools below embody certain choices for display alternatives. We assess the tools not on the choice of display but rather on the extent that performance visualization improves performance observability in the tool's target environment. In general, an integrated performance visualization system should permit the performance analyst to select those display alternatives that best match his or her conceptual image of the data, preferences, and needs. An architecture for such a *performance visualization environment* is presented in §6.3 and a prototype environment is discussed in §6.4.

6.2.1 Hardware Performance Visualization

A common perception of hardware performance visualization might be waveforms on a logic analyzer. Although the idea of viewing a trace of hardware operation is a good one, the use of logic analyzers for general hardware performance observation is not. Typically, hardware performance data is collected in the form of counter values captured by a hardware monitor. These can include cache misses or hits, memory references, and floating point operations. Individual counts can be presented using a variety of single value displays discussed in the previous section. Often, however, the performance analyst wants to observe the execution history of multiple performance metrics and be able to correlate these metrics at different points in time. Although this necessitates a hardware monitoring system capable of capturing hardware performance traces, the interesting visualization challenges come in determining how dynamic, rather than static, hardware performance should be observed.

The case studies below feature visualization approaches for multiple samples of hardware data. The first is a post-mortem visualization of a trace of hardware performance data from the Cray X-MP supercomputer [29, 107, 122]. The next two are dynamic, animated visualization tools for looking at hardware data captured from the Cedar multiprocessor system [99].

Although not shown in these examples, the visualization of hardware performance sometimes can be enhanced by using a graphical representation of the physical system as a visual cue. This might be applied in the case of visualizing processor utilization, memory referencing, or interconnection network operation.

6.2.1.1 Cray Performance Visualization

The Cray X-MP [29] supercomputer architecture supports up to four vector processors, each with multiple scalar and vector functional units, hardware for vector chaining, and two load ports and one store port to memory. Furthermore, the X-MP provides a hardware performance monitor [107] capable of recording hardware level statistics for up to 32 different performance metrics of machine operation. Best floating point operation performance is achieved on the X-MP during vector operation when vector chaining and memory port use are maximized. Unlike the average floating point performance of an application, a dynamic view of floating point operation reveals significant performance variability. A similar view of memory operations would allow one to correlate floating point operation performance with memory system activity, possibly providing insight into the efficiency of the vector computation.

We conducted a Cray X-MP performance observability study [122], to be discussed more fully in Chapter 7, that used the hardware performance monitor to collect trace histories of machine performance for an application code execution. We use data from this study to demonstrate the utility of a tool to visualize the hardware performance data and to provide insight into the vector processing and memory referencing operation. Although the architecture supports multiprocessor configurations, we concentrate here on single processor performance.

The tool, Xgantt, is based on the Gantt chart and shows time-based performance data; see Figure 6.2. The performance analyst specifies each Gantt display through a set of commands that define the range and labeling of the abscissa (time) and ordinate (performance metric) axes, and indicates the file where the sequence of performance values is stored. Xgantt controls positioning and zooming on the displays through the use of a scrollbar; the left mouse button sets the left edge of the visible region, the right mouse button sets the right edge, and the middle button allows the region to be repositioned.[3] The performance analyst also can select a point in a chart using the mouse. The closest time with a performance value is selected and that value is reported through a textual window. The zooming and selection functions facilitate performance investigation.

Figure 6.2 shows seven performance metrics from the Cray X-MP hardware performance monitor drawn in the range of zero to seven seconds.[4] The ability to view several performance

[3] We see the entire region in Figure 6.2.

[4] All performance metrics are given as rates in terms of millions of units per second.

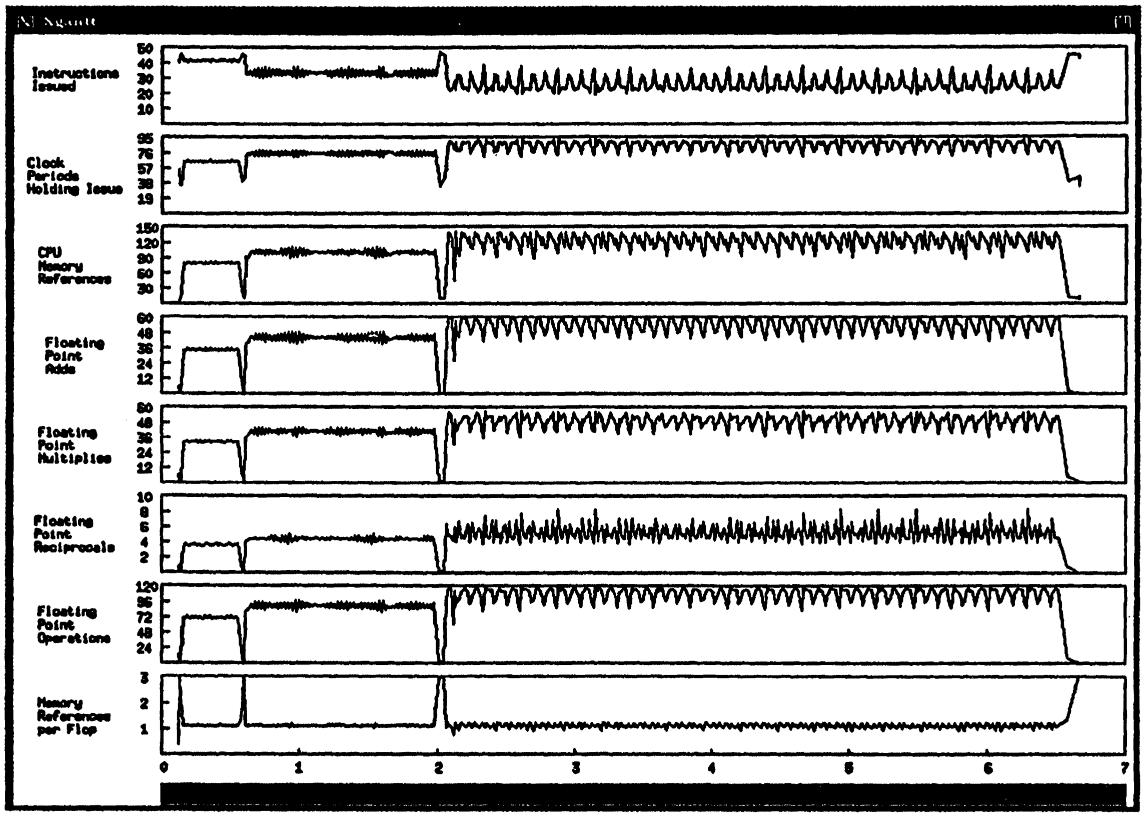


Figure 6.2: Xgantt Display of Cray X-MP Hardware Performance Data

metrics simultaneously over time allows one to visually correlate the behavior of different machine functions, and to associate that behavior with regions in the execution history. Here the curves clearly reflect three major phases of the computation. In addition to showing the raw performance values, the floating point and memory reference curves show increasing rates in successive phases indicating either that a different, higher performing set of vector operations are being performed, or that the vector operations are becoming more efficient. The `Memory References per Flop` curve indicates that the basic performance properties of the vector operations are unchanged (each phase shows the same number of memory reference per floating operation), leading one to believe the vector execution efficiency is improving. In fact, knowledge of the application code substantiates this position (the executed code is the same in the three phases) as does another performance metric (not shown) indicating an increase in vector length in successive phases — increased vector lengths improve the efficiency of vector operations by reducing vector startup overheads.

The Xgantt tool provides a basic example of a performance visualization tool that can provide significant performance insight (in this case into the hardware performance of the Cray X-MP) and interactive support for performance investigation. Although we used the Xgantt tool to show hardware performance data, visualization of time-based data for all performance levels could use the Xgantt Gantt-style displays and query control.

6.2.1.2 Cedar Performance Visualization

The Cedar machine [99] has a multi-cluster, multi-vector processor architecture with a hierarchical memory system and a global two-stage, Omega network connecting processors to a shared global memory. Cedar is a particularly interesting machine on which to conduct performance studies because of its multiple levels of parallelism (vector-level within a processor, loop-level between cluster processors, and task-level between clusters) and the complex interactions of the multiple levels of memory. Currently, the Cedar system consists of four clusters each with four processors. The final target configuration is a 32-processor system — four clusters with eight processors per cluster.

The Cedar hardware has been built with visibility connectors for monitoring the operation of hardware elements. Among the signals that can be observed are *CPU activity*, indicating whether a processor is active or idle, *network busy*, indicating the presence of network contention

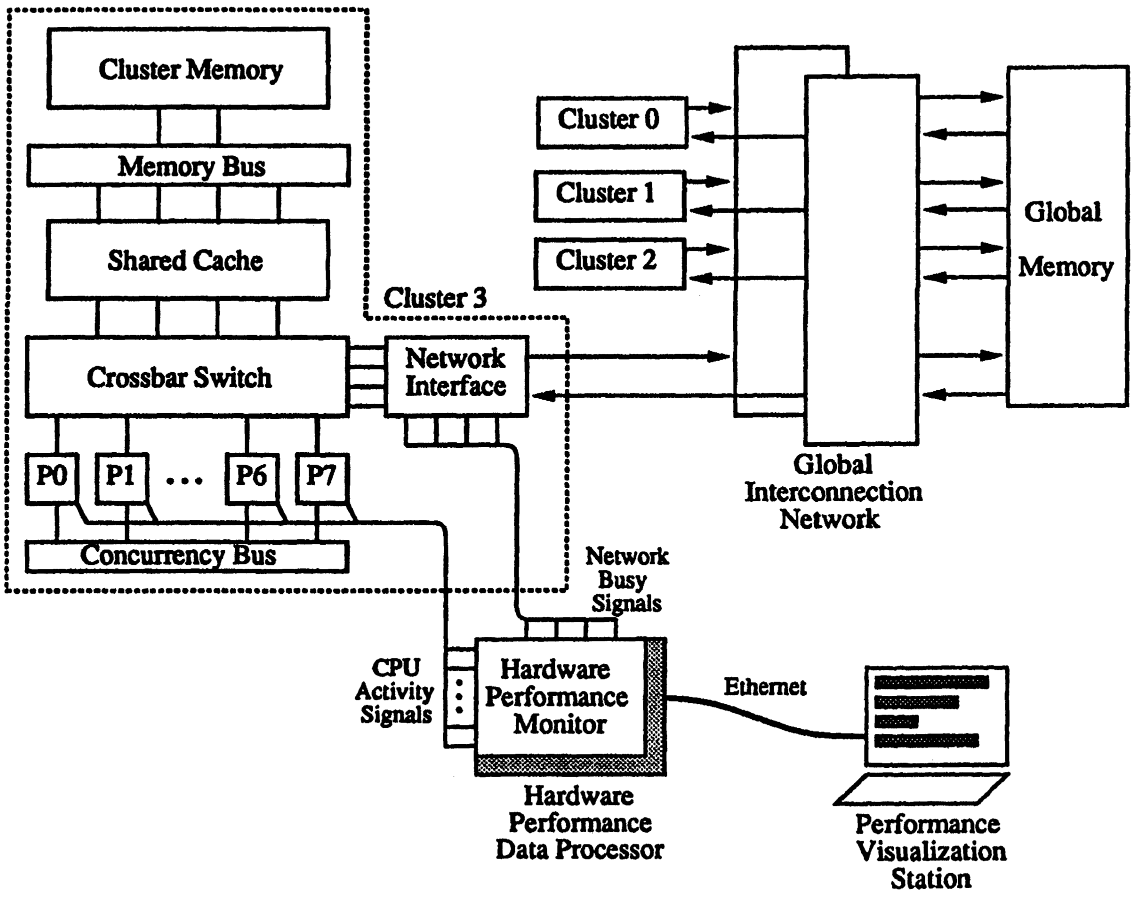


Figure 6.3: Cedar Machine Architecture and Hardware Monitor

and thus a stalling of a processor's access to global memory, and *memory operation*, classifying the type of memory request to a memory module. In addition to hardware signal visibility, hardware performance monitoring equipment also has been developed [109, 7, 16]. Figure 6.3 shows the Cedar architecture together with the configuration of the hardware performance monitor for the two experiments discussed below.

As with any multiprocessor system, the degree of real-time parallelism achieved during execution is a measure of machine efficiency. Observing processor activity at the hardware level provides a raw performance characterization of parallel operation. In the first hardware performance visualization experiment for Cedar, we constructed an analysis and display tool to show CPU activity in real-time across the 16-processor Cedar machine. The hardware performance monitor was configured to capture the state of every processor activity signal for each 170 nanosecond clock cycle. The 16-bit vector describing Cedar-global CPU activity for a particular clock cycle was used as an address into a 64K array of counters and the counter representing that combination of CPU activity was incremented. The counter values were sampled by a processor connected to the hardware monitor approximately every 1.8 seconds. After each sample was read, the counters were immediately cleared to capture the next CPU activity sample. Meanwhile, the processor computed two sets of statistics: the number of cycles each individual CPU was active during the sample interval, and the concurrency distribution in the interval (i.e., the number of cycles i CPUs were active, $0 \leq i \leq 16$). After being computed, these 32 values are sent to the performance visualization station for display.

A snapshot of the CPU activity displays is presented in Figure 6.4; the hardware performance data comes from a study of a conjugate gradient algorithm written for the Cedar system [132]. The top sixteen bargraphs shown the relative distribution of CPU activity, with each bargraph indicating the percentage of the sample interval that that CPU was active. These displays are useful for observing a balance or imbalance of activity within the system. The next set of sixteen vertical bargraphs show the relative distribution of concurrency during the interval. The bargraph for each concurrency level reflects the percentage of the sample interval Cedar was running with that level of concurrency. The sum of the bargraph values totals 100 percent. A shifting of the concurrency distribution to the higher concurrency levels is an indication of greater parallelism being achieved in the interval. The bottom display is a matrix of historical CPU concurrency. The last sixteen states of CPU concurrency are shown, from the most recent on the left to the oldest on the right, where within each column the sixteen processor concurrency level is at the top and the one processor level at the bottom. The concurrency values are mapped to colors based on heat intensity with dark blue signifying low values (cool), yellow/orange medium range values (warm), and red high values (hot). This historical CPU concurrency display allows one to see the variability in CPU concurrency across a range of samples rather than just the instantaneous distribution from the CPU concurrency display.

As new CPU statistics are received, the CPU activity and CPU concurrency bargraphs are updated, and the historical CPU concurrency matrix is shifted right with the last CPU

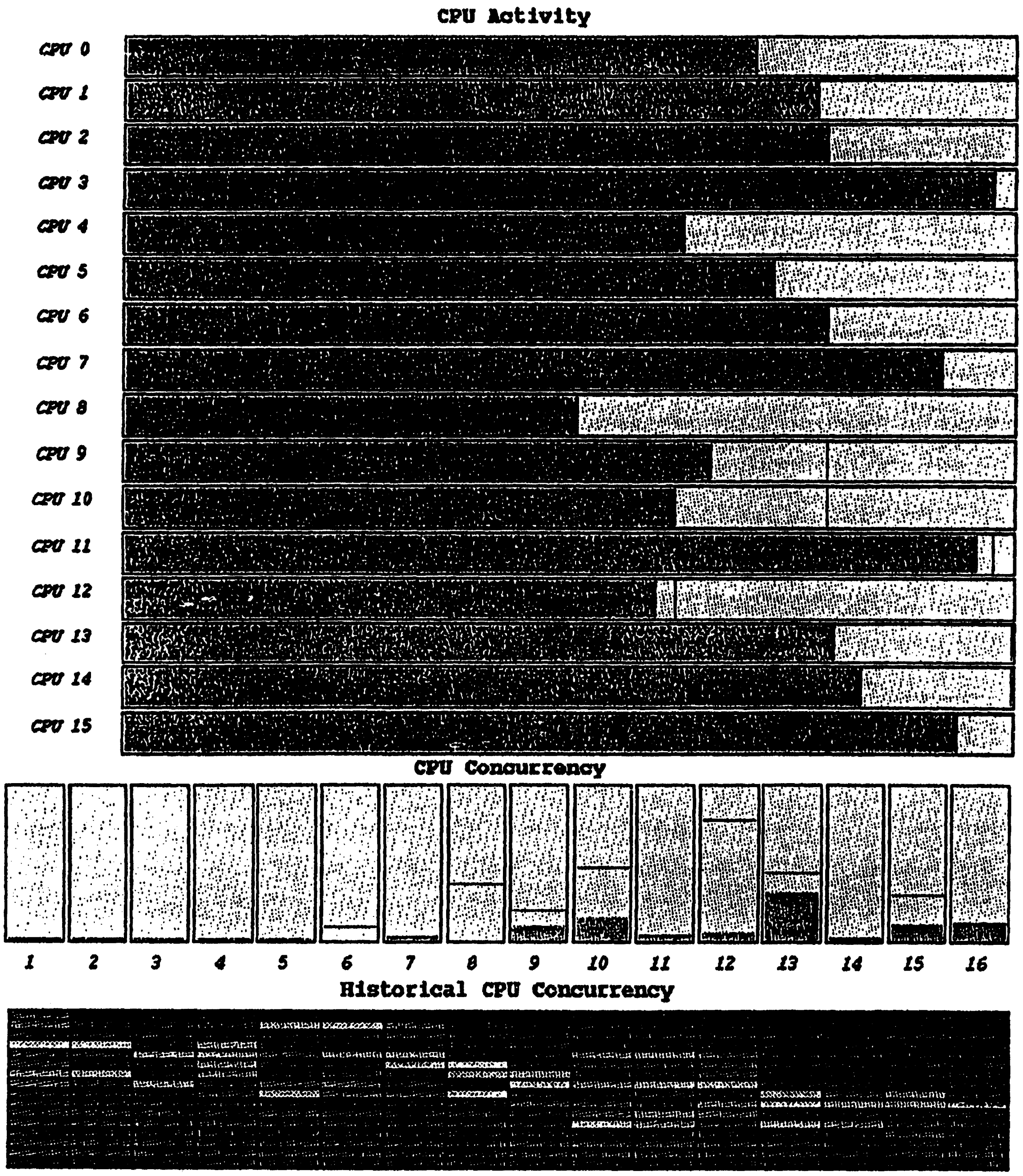


Figure 6.4: Cedar Processor Activity Performance Visualization

concurrency vector added on the left.[5] Control of the animation rate is important if one wishes to gain performance perspectives at different levels of granularity. It would be easy to increase the real-time update interval, but the 1.8 second update interval is currently the minimum update period. The hardware data processor is unable to read the 64K counter values, reduce them to the 32 CPU activity statistics, and then transmit the statistics to the remote visualization station in less time. Because only a few data values are being displayed, sustained update rates could be greater if the CPU statistics were received that fast. We have measured display update rates of eight updates per second in a system where pre-recorded, reduced statistics were read from a file and transmitted to the visualization station for display. Thus, the reading of the counters and the data reduction are the bottlenecks. The counter access performance is limited by the bandwidth of the hardware monitor interface. The measured overhead to read the 64K counters is roughly 0.5 seconds. Thus, slightly less than 1.3 seconds is needed to generate the CPU activity statistics. Only a faster processor will reduce this time further.[6]

The second Cedar hardware visualization experiment focused on observing Cedar network performance behavior. Because the overall performance of a multiprocessor system with a hierarchical memory organization, such as Cedar, critically depends on minimizing contention to components (buses, networks, memory modules) within the memory system [68, 64, 67], understanding where contention occurs during an execution will provide insight into how memory referencing behavior affects memory system operation. Via the Cedar visibility connectors, we could monitor network busy signals generated at each processor-network interface. These reflect contention on network access paths to the shared global memory. We configured the hardware monitor to count the number of network busy signals generated at each *global interface board* (*Gib*) — the hardware realization of the processor-network interface. Five hundred, one millisecond samples were collected for each processor, representing network behavior during a half second interval.[7] Again, the samples were read by the hardware data processor but this time all data reduction was done at the remote visualization station.

[5]Unfortunately, much of the visualization dynamics cannot be seen in Figure 6.4. This is also true of Figure 6.5.

[6]We currently are using Motorola 68020-based Sun 3/160 as the hardware data processor.

[7]Because of hardware monitor constraints, only four processors in one cluster could be monitored.

The visualization display in Figure 6.5 shows network busy samples over a half second interval for four processors in a single Cedar cluster; the data are taken from a study of vector prefetch performance [63].[8] The Maximum Network Busy display shows the maximum percentage of time, in a one millisecond interval, that a processor found the network busy; one hundred percent busy is at the top. This metric is easily computed, and the display provides a quick indication of any serious contention. The Average Network Busy display shows the average level of network busy for a processor during the half second interval. This metric is important for observing the balance of network contention, or more appropriately its inverse, network access. In both the Maximum and Average Network Busy displays, the increase in the metric from Gib 0 to Gib 3 reflects the fixed network contention resolution scheme used in Cedar. Gib 0 of a cluster has highest priority (less likely to observe contention), and Gib 3 has the lowest priority (most like to observe contention). Whereas the Maximum and Average Network Busy displays show summary views, the Network Busy graphs detail every network busy state for each processor in the interval. The first millisecond period is shown on the left and the last on the right; the entire range represents the half second time interval. Time-based network busy behavior can be observed clearly from these graphs and can be compared across processors.

Finally, the Network Busy Intervals Kiviat display shows four ranges of network busy activity for each processor. Each quadrant of the display represents a processor, and each line within a quadrant represents a different range of relative network activity: 0 to 25 percent busy, 26 to 50 percent busy, 51 to 75 percent busy, and 76 to 100 percent busy. The number of samples in each range determines a point on each axis. The points are connected by line segments, forming a sixteen-sided polygon. Within each quadrant, the shape can provide a quick visual interpretation of the distribution of network busy activity. Symmetry in the Kiviat diagram signifies a balanced and equal distribution of network busy activity across processors. Moreover, as the display updates occur, rotational changes in the diagram reflect increases (clockwise rotation) or decreases (counter clockwise rotation) in network activity.

As with the CPU activity displays, the network busy displays are animated, changing approximately once a second. With successive animation frames, one can quickly observe quantitative differences between samples through qualitative changes in the Network Busy waveforms.

[8]The Cedar global network interface hardware supports vector prefetch operations from the cluster processors and maintains buffers for prefetched vector data.

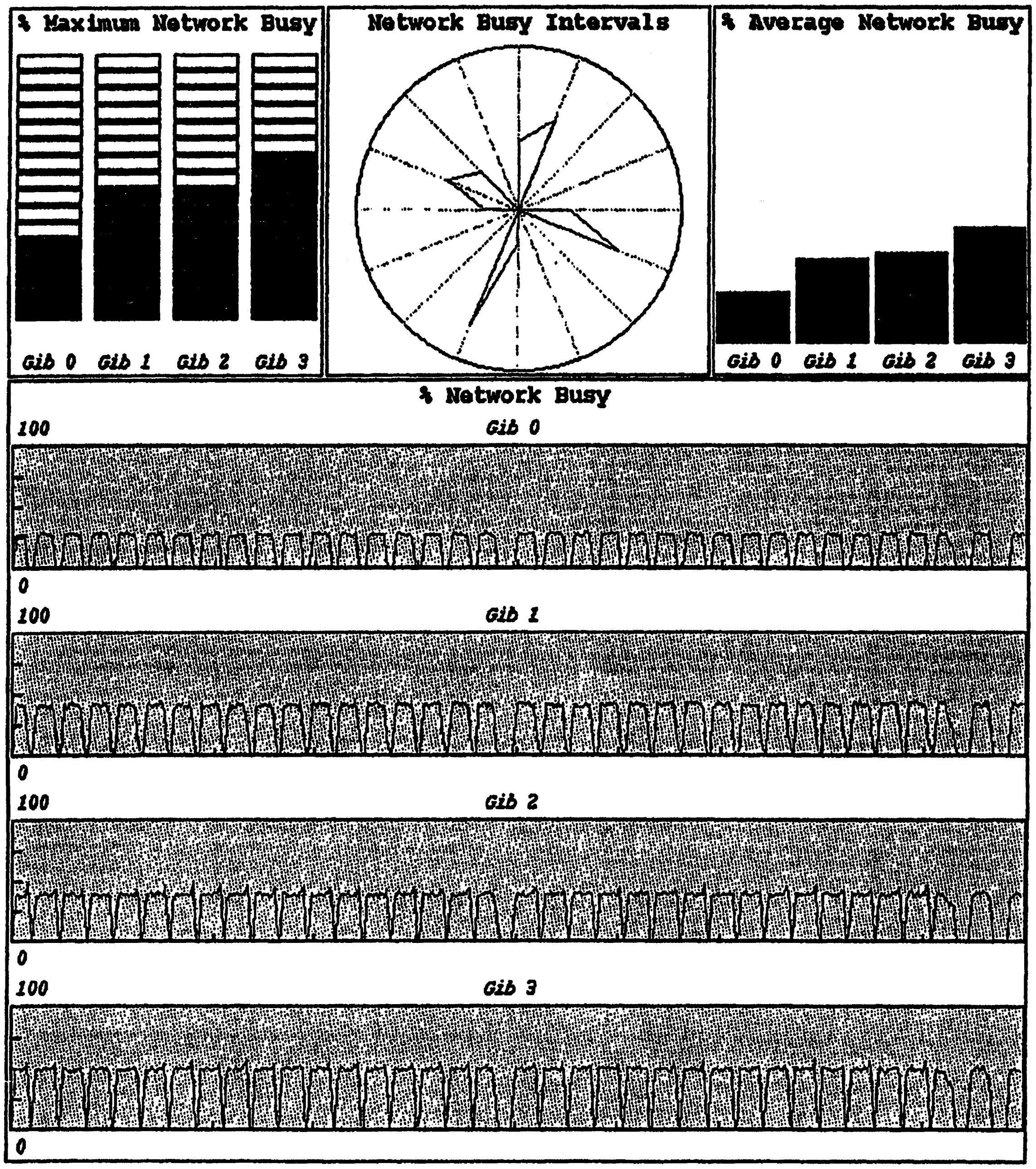


Figure 6.5: Cedar Network Busy Performance Visualization

In the vector prefetch study, we observed differences in vector block size and network access intensity between display updates. However, one second is the fastest rate the sample values can be read, transmitted to the visualization station, processed to produce the maximum, interval, and average metrics, and displayed. Because the samples reflect only a half second interval, some data is lost between successive display updates. Perhaps a more appropriate time interval is one second, but a larger number of samples (1000 per processor) would have to be captured to maintain the same level of detail, further increasing the overhead. Although the total number of samples is less than in the CPU activity case, a slower interface is used to acquire the network busy signals plus a larger amount data is communicated to the remote visualization station. Thus, the bottlenecks to faster animation rates in the network busy case are the network busy data access and the remote communication; the performance statistics processing was done on significantly less data (2000 versus 64K) and by a ten times faster processor.[9]

Both Cedar performance visualization tools demonstrate real-time visualization environments. Although the display types were moderately simple, they effectively convey the hardware-level performance data. A textual presentation of the data certainly cannot be interpreted as quickly. More importantly, the tools emphasize the need to consider all aspects of the monitoring, analysis, and display system used for real-time visualization. As our need for more detail performance data and faster animation increases, we must identify and remove bottlenecks present along the entire monitor-analyze-display path.

6.2.2 System Performance Visualization

In general, system behavior is transitory, depending on the current workload and state of the system resources. Although state-based views are needed to understand how resources are presently assigned and being used, summary views that embody historical resource behavior also are important for understanding the system dynamics. A static or average performance state analysis may mask transients. Thus, time-varying measurements, at different levels of detail, are required to determine both system performance trends and transitions. The display of historical data must simultaneously include gross behavior and permit detailed operation to be investigated.

[9]We used a Sun SparcStation for the performance visualisation station.

Color/Graphic	Description
blue	user computation
gray	idle
red	OS activity for message processing
green	system code
horizontal lines	message transmission

Table 6.1: Encoding of Color Graphics for iPSC/2 Event Graph Display

An event graph display has been developed by Rudolph [165] for studying operating system events relating to message communication in the Intel iPSC/2 multiprocessor [10, 161].[10] We discuss it here as an excellent example of a performance visualization environment that can effectively display a large volume of performance data but that also provides support for investigations of detailed behavior.

The interactive event graph tool simultaneously shows the system state of each iPSC/2 processing node over a user-controlled time period. Each state is graphically encoded, revealing patterns of system activity on each node. The activities displayed include application execution, operating system actions, message related actions, and idle time. The encoding of color graphics is given in Table 6.1. Further graphic highlighting allows individual communication events (send and receive) to be observed and matched between nodes.

Figure 6.6 shows the highest level view the tool provides of system activity during an application's execution on the iPSC/2.[11] The display shows a window on the two-dimensional space of processors and time — here the entire trace of system events is seen. Because the number of events can be larger than the display resolution, the tool divides the trace into equal-sized blocks of time, and for each block, calculates the system activity that dominates the corresponding time interval. The extent to which an activity dominates its block is indicated by the intensity of the graphic pattern. Lighter shades of blue indicate smaller percentages of time devoted to application code. Similarly, lighter shades of red and green indicate smaller ratios of message or system code activity. However, for idle time, darker shades of gray indicate smaller ratios of idle time. This is because idle time is a negative quantity — less idle time

[10]See Chapter 2 for a description of the iPSC/2 and the operating system instrumentation.

[11]The application is a Simplex linear optimisation algorithm [187, 55]. The event displays shown have been reproduced from [161].

means more activity. At this time scale, we see that application processing dominates the execution, but there are areas of other system activity that we might want to observe in more detail.

The `Up` and `Down` buttons control scrolling across processors — in this case, system activity from all eight processors participating in the computation is shown. Similarly, the `Left` and `Right` buttons control scrolling forward and backward in time. The `Expand` and `Zoom` menus at the upper left of the display window control the fraction of the trace that will fill the display. Figure 6.7 is the result of a display expansion, showing roughly 0.5 seconds of the event trace. In this figure, the dark bands of processor activity are clearly delimited by intervals of communication and idle time. We have chosen the `All Events` selection (not shown) from the `Show Events` menu to display message communication events. Message sends are marked by an `s` and message receives by an `r`. Clicking the mouse on any event marker displays additional data about that event. In this case, we clicked on a send marker in the node 7 graph and were presented a textual window showing the corresponding message size, receiving node, and message transmission times.

Because idle time spent waiting for messages has an obvious negative impact on parallel program performance, we can expand the trace display further to focus on system behavior in regions where high idle times occur. In Figure 6.8, we have expanded the display by a factor of 100 to show only five milliseconds of the program's total execution. The horizontal lines denote intervals when the iPSC/2's autonomous routing hardware was transmitting a message. The selected message communication pair delineates one such region and clearly shows the receiver on node 5 waiting for the message transmission by node 4.

The event graph tool used to show iPSC/2 system behavior is a simple but powerful display mechanism for performance visualization of large system event traces. The ability to focus on regions of interest to see more detail permits performance investigation, whereas the global, summary views convey aggregate system activity. As with the Xgantt tool, however, only a single display type is used by the tool. Ideally, the performance analyst should be allowed to graphically present system performance data in a variety of ways, depending on preference and need. In §6.3 we return to this issue.

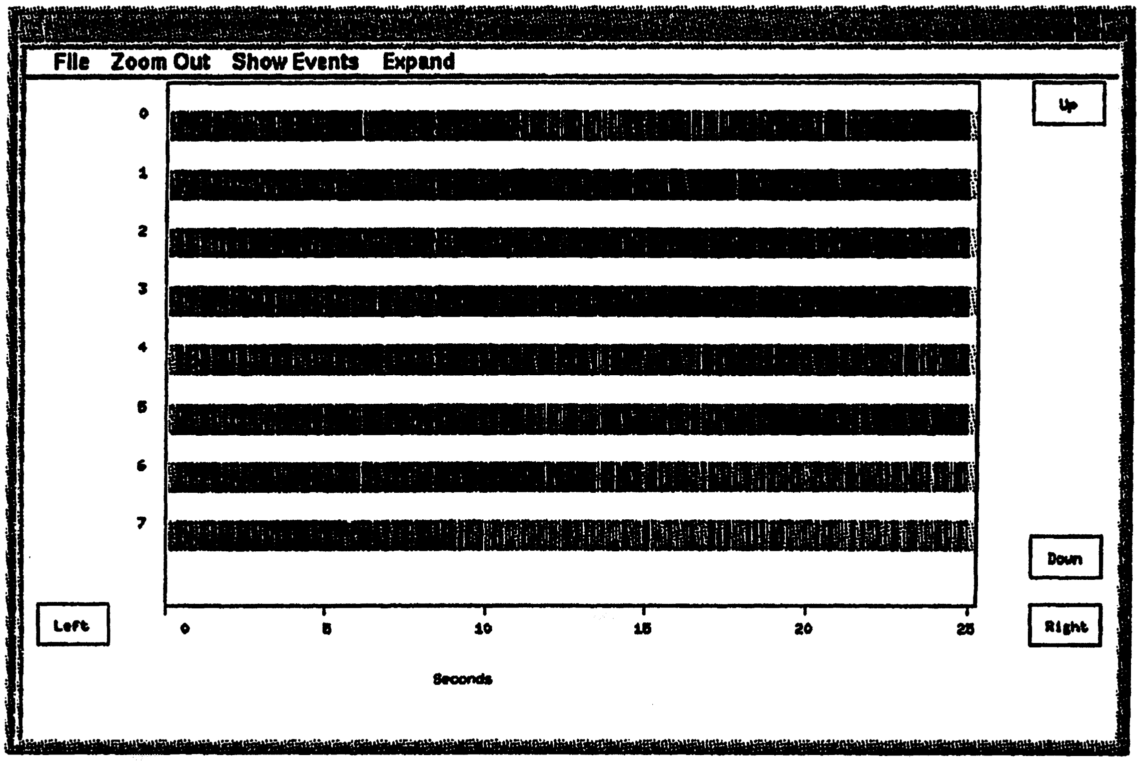
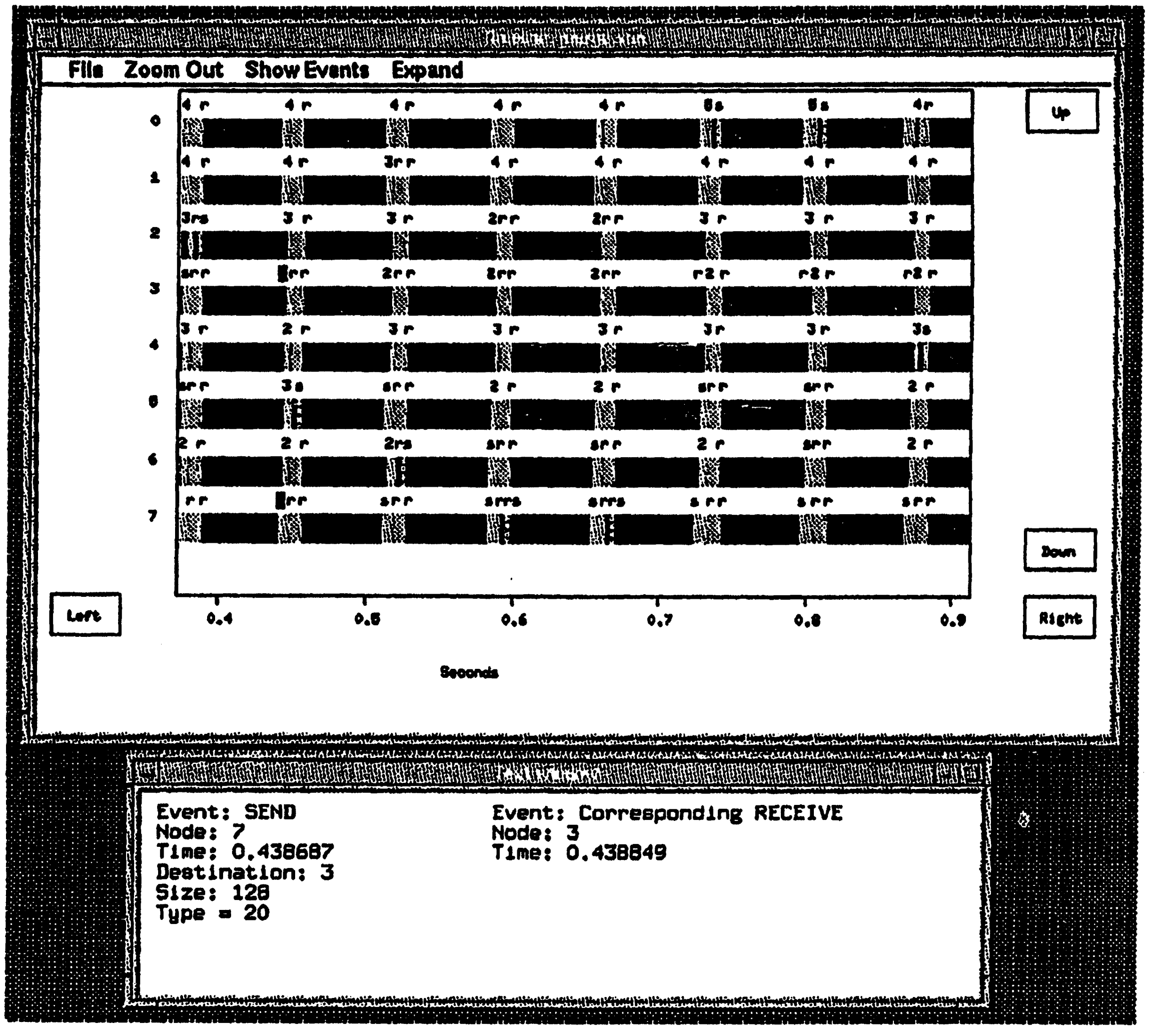


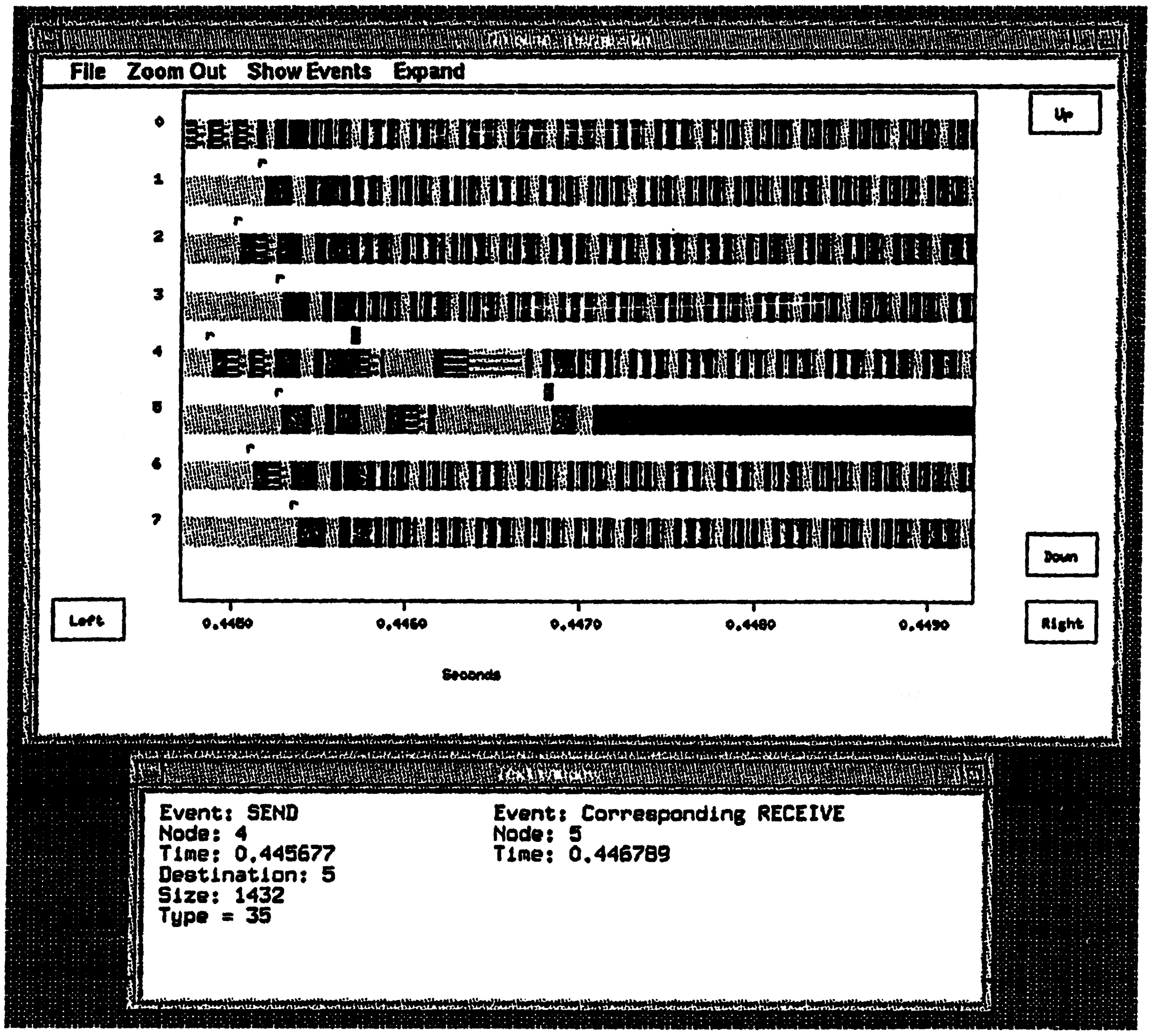
Figure 6.6: System Event Trace Timeline (Entire Trace)



Event: SEND
Node: 7
Time: 0.438687
Destination: 3
Size: 128
Type = 20

Event: Corresponding RECEIVE
Node: 3
Time: 0.438849

Figure 6.7: System Event Trace Timeline (Detail)



```
Event: SEND                     Event: Corresponding RECEIVE
Node: 4                         Node: 5
Time: 0.445677                  Time: 0.446789
Destination: 5
Size: 1432
Type = 35
```

Figure 6.8: System Event Trace Timeline (Fine Detail)

6.2.3 Program Event Visualization

Views of application performance data might be divided between *program event views* and *algorithm views*. In a program event view, the performance data represent the occurrence of events associated with the execution of program statements, such as routine entry and exit events. On the other hand, an algorithm view reflects the operation and performance of the algorithms used in a program. Although the display of algorithm views may provide a more logical and meaningful representation of the performance of an application, program event views are necessary to associate performance behavior with the actual code executed. This section describes a tool for program event visualization of multitasking applications run on the Cedar system. The section following gives an example of algorithm visualization.

Program event views are represented either as summary statistics, as produced by profiling tools [28, 72, 73] or event traces. Summary statistics, while reporting the distribution of program execution time, do not permit a dynamic characterization of program flow or the interaction of program components. Event traces capture this behavior, and a tool to visualize program events should provide a time-based display of event occurrence. However, tools to study an event trace typically are specialized to particular types of event data. Usually, trace analyses and displays are developed around some event interpretation model [180]. Although this approach will give specific information about particular events and their occurrence, it is not particularly easy to extend; new events often require new analysis and display techniques.

The program event visualization tool discussed below emphasizes a major theme in performance visualization implementation — advances in the development of performance visualization environments demand advances in the technology to build performance visualization environments. To rapidly prototype performance visualization tools, we must first design and build reusable components that provide standard operation for common functions [123]. In §6.3 we argue for an extensible, general purpose performance visualization architecture that can be applied in many performance data scenarios. Here, we give an example using trace-based program event data.

In the first phases of application performance measurement, a user often is interested in such performance information as the time a certain event occurs in the computation, the relative sequence of events on different execution threads, or the state of each task. In this case, the user

Trace Control	This component is responsible for reading the trace, positioning within the trace, and searching for particular events.
Event Control	This component provides event definition services. It allows the user to associate names and graphic icons with events. The mapping of events to graphic icons can be controlled interactively, as can the visibility of events in a task trace display.
Task Display	This component opens viewports onto the trace and allows events for tasks assigned to the viewports to be displayed in a Gantt chart-style form.
Event Display	This component controls how events are shown when "clicked" on in the task display. A standard event display is provided but the user can override this default by specifying event display modules that will be dynamically linked with JED at run-time.

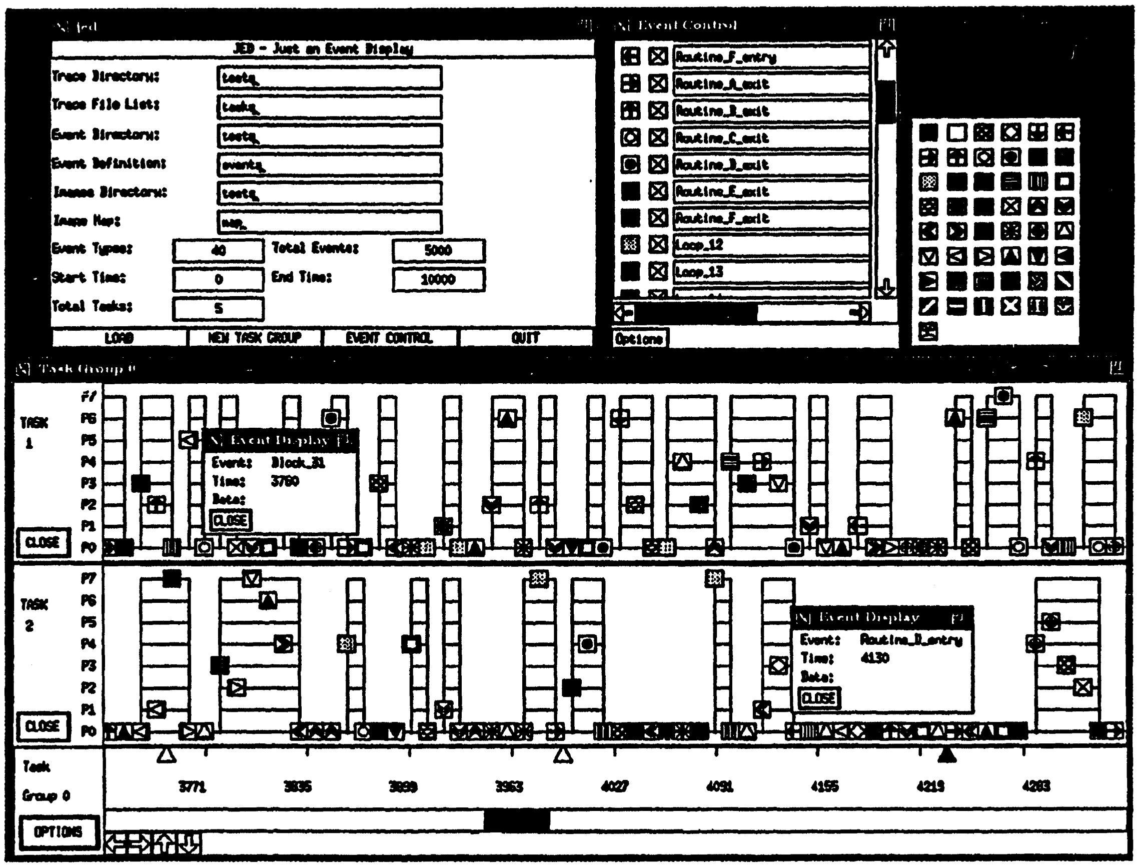
Table 6.2: JED Functions

might only desire to observe the sequence of trace events, each event's type, its time and place of occurrence, and other event-specific data. A simple presentation of individual events, together with basic support for event query, constitutes a rudimentary program event visualization tool with a reusable foundation for further extension and customization.

In this framework, we developed a generic program event visualization tool call JED [121] for displaying program events from multitasked Cedar application programs.[12] JED provides support for event trace management, event trace display, and event query, but allows user-definable event specification and user-customizable graphical event presentation. Furthermore, JED allows the user to extend analysis and display functionality. JED contains four separate functions as described in Table 6.2.

Figure 6.9 shows a sample JED session. The top-level interface to JED (the `jed` window in Figure 6.9) allows the user to specify the trace file locations and the events that appear in the task traces. The event control component of JED maintains information about the events in the traces and how they are to be shown in the displays. A definition file is provided at initialization for labeling events and indicating how data for events should be displayed. Additionally, a user-supplied image map associates events with graphic event icons. Interactive control of event displays includes changing event graphics and visibility. The `Event Control` window shows

[12]JED stands for (J)ust an (E)vent (D)isplay tool.



JED - Just an Event Display

Trace Directory: tests_
Trace File List: tasks_
Event Directory: tests_
Event Definition: events_
Images Directory: tests_
Image Map: map_
Event Types: 40 Total Events: 5000
Start Time: 0 End Time: 10000
Total Tasks: 5

Routine_F_entry
Routine_A_exit
Routine_B_exit
Routine_C_exit
Routine_D_exit
Routine_E_exit
Routine_F_exit
Loop_12
Loop_13

Event: Block_31
Time: 3760
Data:

Event: Routine_D_entry
Time: 4130
Data:

Figure 6.9: JED Visualization Session

the current event specifications, including icon assignment and visibility control. Clicking on an event icon in the `Event Control` window raises the `Image Map` window (unlabeled in Figure 6.9), allowing the user to change event graphics. Doing so will have an immediate effect in the event trace displays.

To see the trace for a task in JED, the user must first open a `Task Group` window; a *task group* is essentially a viewport (time window) into the trace files for all tasks assigned to the task group. A *task display* is a region of the task group window created when a task is assigned to a task group. The `Task Group 0` window in Figure 6.9 shows task displays for tasks 1 and 2. The graphics part of the displays shows events for the each task that occur in the time region defined by the task group. Events are shown by their graphic icons. Clicking on an event icon in a task display opens an `Event Display` window. This provides additional information about the selected event. In the case of our Cedar implementation, the events for each processor of a cluster are shown separately in the task display. Lines are drawn to show sequential and concurrent activity; vertical lines indicates concurrency transitions. The vertical alignment of the task displays allows inter-task events to be correlated in time.

The basic functionality of JED can be statically customized through an event specification file, and dynamically, through the event and task display controls. In this manner, JED provides a reusable platform for program event visualization of different Cedar application programs.[13] However, JED also supports functional extensions by allowing the user to dynamically link modules that will graphically display event data, overriding the default textual representation.

JED has much in common with the BBN Gist tool [15]. However, unlike Gist, JED provides a simple environment foundation for program event visualization that can be customized and extended by the user. Future performance visualization systems will be built from reusable performance data analysis and display components, and will provide facilities for customization and extension. JED is a example of such tool. In §6.4 we describe efforts in the development of a general purpose performance visualization environment.

[13]JED has been used in the study of circuit simulation codes [65] and in tests of the Cedar Fortran run-time library [78, 77, 87].

6.2.3.1 Algorithm Performance Visualization

An application's execution (and indirectly its performance) is a result of the problem being solved and the algorithms used to solve it. Observing the operational behavior of an application in terms of the computational properties of its algorithms provides insight on the efficiency and correctness of the solution approach [25]. Indirectly it can provide insights to improving application performance.

A general tool to visualize application performance is necessarily incomplete, given the infinite number of problem areas and algorithms. However, viewing certain key algorithm components, mostly related to the data structures used, can sometimes provide a clear understanding of algorithm and application operation. The Balsa-II algorithm animation system [26] is an application domain-independent algorithm visualization environment. It has been effectively applied to the display of algorithms such as sorting and searching. In scientific applications, numerical algorithms often use vector and matrix data structures to maintain execution state. Displaying vector and matrix operations can provide insight into the performance of these codes. Below we discuss matrix visualization as a technique for application performance visualization.

Matrix visualization has been applied in two general ways to the study of numerical algorithms. The MAP tool [23] and its successor SHMAP [43] were developed to understand the access patterns of parallel matrix algorithms. These tools display the matrices used in an algorithm and color their elements to represent the history of element access. The display is animated, allowing the user to observe the pattern of access for each matrix as the algorithm proceeds. The tools are useful not only for verifying algorithm operation but in seeing how the temporal locality of matrix references change. They have been applied in the development of block structured numerical algorithms for hierarchical memory systems, where knowledge of the memory address assignment of matrices and the memory management of the different memory levels is crucial [43].

Tuchman and Berry have developed the MatVu tool to look specifically at the values of matrices used in numerical algorithms [193]. This tool has proved useful in understanding the behavior of linear algebra problems, particularly the chaotic behavior of some numerical approaches. The tool has led to new algorithms with substantial performance improvements for certain applications [20].

Because matrix visualization can be applied to other performance levels as well (see §6.4.2), we have developed a general purpose matrix visualization tool for X windows call Xmatrix. Using Xmatrix, the performance analyst can control all aspects of the matrix display by interactively loading data files, choosing display attributes (e.g., color maps, labels), and selecting matrix regions for separate display. Xmatrix also supports the animation of multiple matrix data sets. Figure 6.10 shows Xmatrix being used to view the convergence behavior of a one-sided Jacobi algorithm used in singular value decomposition [193].[14] Four frames from the animation are shown, labeled by the iteration count of the algorithm. The convergence of the matrix to block diagonal form is clearly apparent.

Similar to the Xgantt and JED tools discussed earlier, Xmatrix is a user-customizable performance visualization tool. As with the Xgantt tool, Xmatrix can be used to display a variety of performance data from different performance levels. However, each of these tools limits the number of fundamental display types. Moreover, the format of the performance data is restricted, limiting the ability to combine different performance views in a common visualization framework. It is our belief that general performance visualization tools can be built that are not domain restrictive, but, to the degree possible, separate view semantics from display, letting the user construct views from the available performance data and match them to displays that are most meaningful to the type of performance data being visualized. The following section describes a general performance visualization environment architecture that supports this goal.

6.3 Performance Visualization Environment Architecture

For many performance visualization applications, there are common issues one must address when designing a visualization environment. Problems of filtering and analysis can be found in all performance visualization systems, although the implementations are typically performance data dependent. Also, many of the displays may be similar across systems. Although the display types are common, often the display code is re-implemented with each new visualization tool. The reason is that every visualization environment must solve another shared problem:

[14]The details of the algorithm can be found in [193]. Figure 6.10 has been reproduced with permission from A. Tuchman and M. Berry.

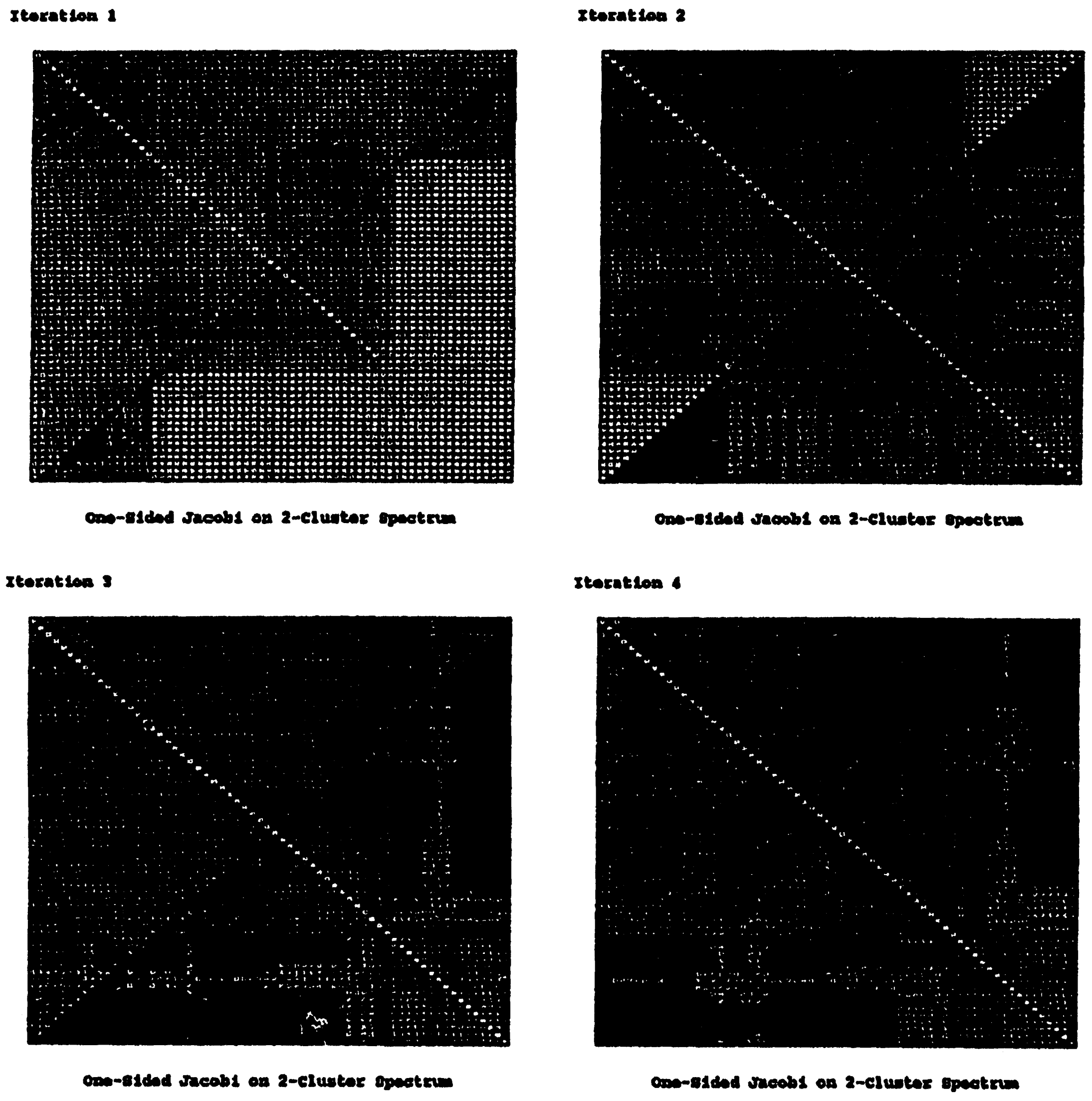


Figure 6.10: Xmatrix Display of Singular Value Decomposition Convergence

binding the performance data semantics, as represented by the filtered and analyzed performance database, to the data display. The tight binding that occurs in the implementation of most performance visualization environments restricts their range of application and limits their reuse.

There is strong motivation for designing any performance visualization system with a general purpose performance visualization environment architecture. Logical interfaces between components of the environment can be defined which, in turn, promotes modular development. Organizational and functional standards can be adopted in the development of component modules to promote module reuse [123]. Finally, existing environments may be more easily adapted to new performance visualization applications.

From another viewpoint, the development of a general purpose performance visualization infrastructure based on the environment architecture would support the rapid prototyping of new visualization tools [159]. Reuse and ease of customization would be strongly supported by the infrastructure. New environment prototypes could use existing filtering, analysis, and display modules, as well as the general organizational framework, in their construction.

A general performance visualization environment architecture is shown in Figure 6.11. The performance data conceptually forms a database from which performance data views are constructed. A performance data view should be thought of as an abstraction of the total performance data.[15] A view is generated through any combination of data filtering, reduction, and analysis. Each view understands the format of the performance database and maintains an internal view state that is accessible through well-defined mechanisms. In a general performance visualization environment, the performance analyst should be able to choose from currently defined views and create new views either by changing the parameters of a current view, combining views, or specifying the functions of a new view.

Display components should be developed in such a way that their implementation is independent of performance data semantics. Because a display might be used for a variety of performance data views, embedding data semantics in the display would restrict its domain of applicability. Rather, a set of *resources* and *methods* should be provided by each display. Display resources are the configuration parameters used to define a display instance; in the performance visualization environment, the performance analyst can customize the display's

[15]This notion is similar to that of a view in database systems [196].

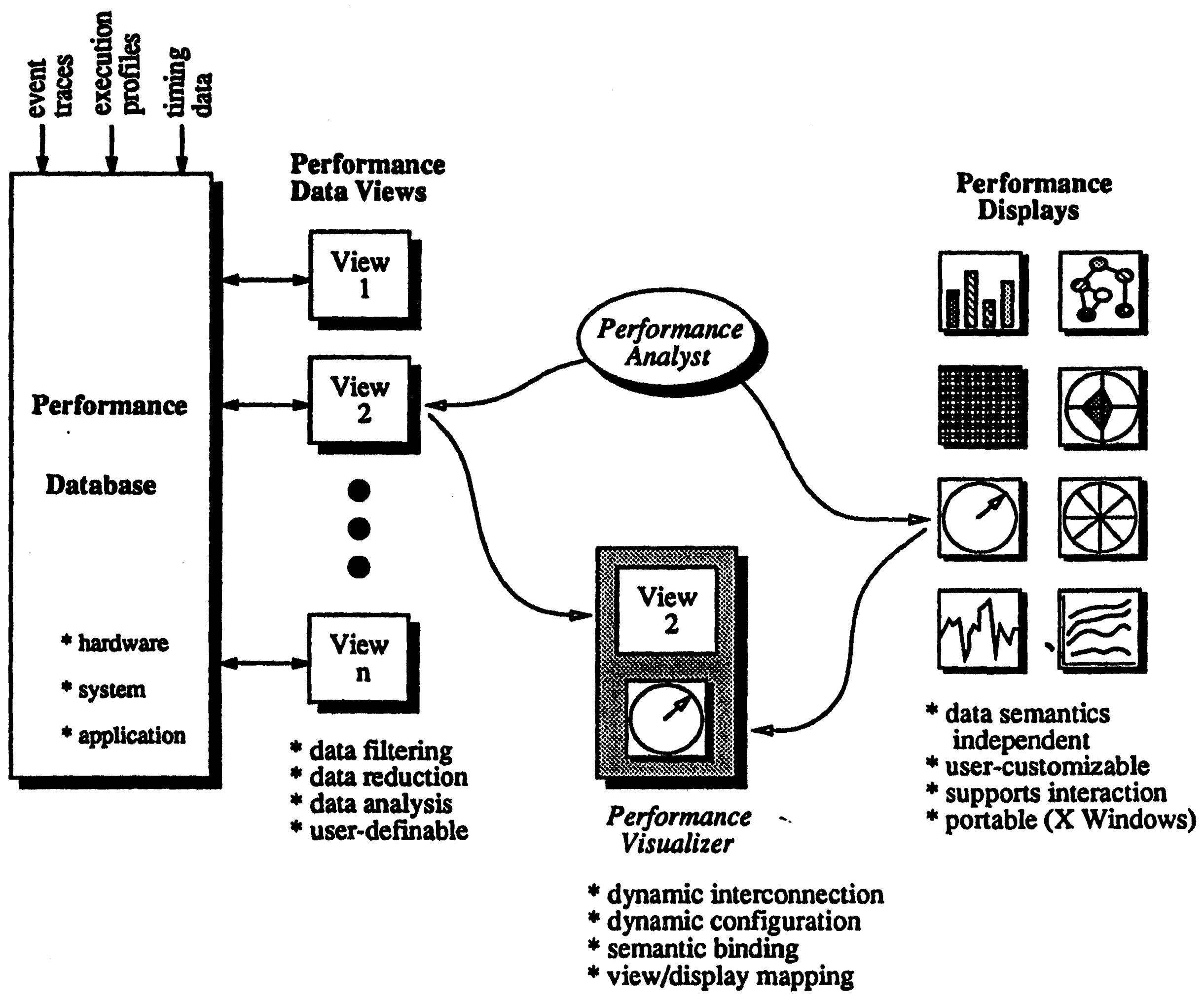


Figure 6.11: Performance Visualization Architecture

resources. The display methods control the display's operation. Each display should also support standard interface conventions, including user interaction; this promotes ease of use and rapid construction of new displays. Basing development of performance displays on a common graphics and windowing system, such as X Windows [168], will enhance portability.

The ability to support multiple performance views and displays in a general performance visualization environment reflects our belief that the performance analyst should be allowed to *explore* interesting performance visualizations; i.e., view/display mappings. The environment architecture should provide support for dynamically selecting a performance data view and connecting it with a display to create what we will term a *performance visualizer.* The performance visualizer, via access to the internal view state, produces a view-specific visualization using the selected performance display; effectively, it provides the semantic binding between view and display. The ability to dynamically construct performance visualizers allows performance visualizations to be reconfigured based on the performance analyst's current interests.

Integration and flexibility are the twin keys to an effective performance visualization environment [124, 127]. If the interactions among the environment components are awkward or inefficient, the performance analyst will seek simpler tools. Similarly, if the environment does not permit diverse approaches to performance data reduction, analysis, and display, including addition of new environment components (e.g., data filters and displays), its functional lifetime will be limited. Given the implications of integration and flexibility, the general performance visualization environment architecture presented above offers a framework for meeting the following specific environment design goals.

- The environment should support **multiple analysis levels**, including hardware, system software, and application.
- It should be possible to **dynamically configure** data analysis and visualization components, allowing the performance analyst to change data perspectives during execution.
- The individual analysis and visualization components should be **easy to build** for many different application types and system software environments.
- An **extension interface** should permit addition of new components or modification of existing components without intimate knowledge of environment infrastructure.

- The environment should be fast, preferably fast enough to process real-time data.
- Finally, the environment should be portable to different systems. Although the mechanisms for performance data capture are inherently system dependent, performance data reduction (e.g., computation of sliding window averages) and visualization are largely system independent.

6.4 Performance Visualization Environments: HyperView

We have designed and implemented a prototype performance visualization environment, HyperView, for distributed memory, message passing multiprocessors that follows the architecture design guidelines discussed above. Our goals for implementing the HyperView environment were to investigate the application of performance visualization ideas in a practical context. HyperView integrates performance data analysis and visualization in an interactive system that allows the performance analyst to browse through a trace of system and application execution.

Two generations of the HyperView environment have been built. The first explored the display of architectural and system activity via a multiplicity of topological displays. Unfortunately, the design of the first HyperView prototype could not easily be extended to the display of application performance. The second generation HyperView system addressed this issue by logically decoupling data analysis from display, permitting the performance analyst to explore interesting performance data at any level by dynamically interconnecting new performance displays and data analysis. Because the contrast in the two generations of HyperView is important, both are discussed below.

6.4.1 First Generation

The first HyperView prototype [124] was inspired by *Seecube* [35], a visualization system for the NCUBE hypercube [83]. Although many of the displays were borrowed from Seecube, the performance data and analysis were different. HyperView's implementation also was based

[16]Portions of §6.4 appear in the papers "Visualising Parallel Computer System Performance" in the book Instrumentation for Future Parallel Computing Systems [124], "An Integrated Performance Data Collection, Analysis, and Visualisation System" in the Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications [126], and "Integrating Performance Data Collection, Analysis, and Visualisation" in the book Parallel Computer Systems: Performance Instrumentation and Visualisation [127].

on the X Window system [168]. This improves the portability of the environment and allows the data analysis and display components of HyperView to be decoupled — both logically and physically. This decoupling not only makes the visualization portions independent of message passing hardware and system software, it also is important for performance — the data analysis and display components can run on physically distinct (and distributed) machines with appropriate processing capabilities— and dynamic system reconfiguration — new analysis actions or displays can be enabled without altering the interface between the components.

HyperView's organization consisted of two main cooperating modules: state analysis and visualization. The HyperView state analysis component accepts event traces generated by the processors of a message passing system. Because the performance data capture is separate from analysis and visualization, the event traces can be generated from actual program execution, or via simulation, permitting study of new message passing architectures. HyperView was initially tested with data obtained from simulation of communication hardware for different message passing paradigms [75], including store-and-forward message switching, circuit switching, staged circuit switching, and wormhole circuit switching. Our experience had shown that visual comparison of system dynamics quickly revealed differences in communication paradigms. Thus, an interactive and dynamic visualization system seemed natural in this context.

In a distributed memory parallel system, such as a hypercube, each node must record events based only on local knowledge; the absence of global memory precludes data sharing with the granularity necessary to dynamically maintain a consistent, global state. Although the simulation event trace contained information to describe complete messaging behavior, the global state of the system had to be constructed during state analysis. The state information maintained by HyperView included individual message tracking in addition to statistics reflecting the performance of each node and link.

The HyperView visualization component permits simultaneous display of the dynamic system state via a variety of differing state views and their associated displays. Each view/display combination emphasizes certain system aspects (e.g., the network topology, the multiplicity of partially overlapping paths from a source to a destination node, or queues of waiting messages).

In addition to graphical displays, HyperView provides statistical displays at both macroscopic levels (e.g., number of messages transmitted) and microscopic levels (e.g., link utiliza-

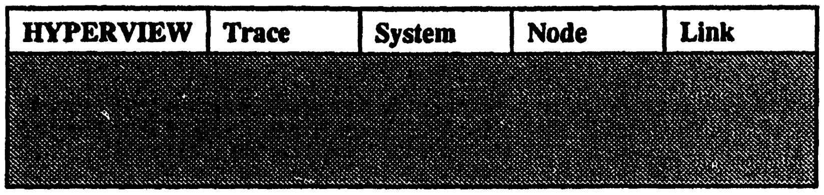
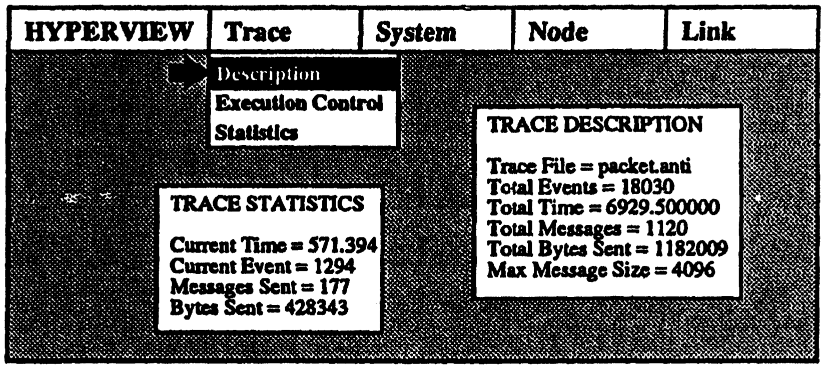


Figure 6.12: Top Level HyperView Window



TRACE DESCRIPTION

Trace File = packet.anti
Total Events = 18030
Total Time = 6929.500000
Total Messages = 1120
Total Bytes Sent = 1182009
Max Message Size = 4096

TRACE STATISTICS

Current Time = 571.394
Current Event = 1294
Messages Sent = 177
Bytes Sent = 428343

Figure 6.13: HyperView Trace Menu and Trace Description Window

tion). HyperView also permits selective display of message traffic and statistics, permitting the performance analyst to isolate anomalous behavior for further study.

Because HyperView is a dynamic performance visualization system, much is lost in description of static, monochrome images. Despite these limitations, we discuss HyperView as a performance analyst might encounter it, beginning with the top-level user interface window shown in Figure 6.12. User menus are shown at the top of the window. Pulling down the **Trace** menu lists the `Description`, `Execution Control`, and `Statistics` items shown in Figure 6.13.

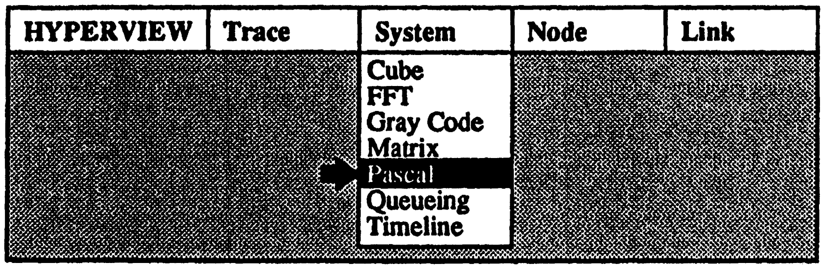


Figure 6.14: HyperView System Menu

6.4.1.1 Trace Description

In the `Trace Description` window (see Figure 6.13), a performance analyst can select, by clicking the mouse on the `Trace File` item, a trace file that contains the event information captured during system execution. A dialogue box allows the user to enter the trace file name. After reading the trace file, HyperView computes the trace summary statistics shown in the `Trace Description` window. During state analysis, HyperView computes the the time varying global state of the message passing system, including number of events, messages and bytes transmitted. Throughout the visualization session, these statistics can be viewed by selecting the `Statistics` item in the `Trace Description` menu; see the `Trace Statistics` window in Figure 6.13.

6.4.1.2 Execution Control

The `Execution Control` window (not shown) controls updates to the graphical and statistical displays. During state analysis, HyperView identifies a series of globally consistent display points. During updates of the trace display, HyperView moves between these display points. The current point can be marked by a position in time and/or location in the event trace. When a new time or event is selected, HyperView moves to the next consistent system state and updates the currently open displays. Because the event trace is processed *a posteriori*, the performance analyst can move both forward and backward in time.

A *frame* in HyperView corresponds to a displayed system state. The user can change three aspects of frame display — mode, rate, and state differential. Frames can be displayed either in single-step mode or continuously. In single-step mode the user must explicitly request display of the next frame. Conversely, continuous mode automatically advances to the next frame specified by the frame rate and differential controls.

Via a frame rate control, the performance analyst can adjust the interval between display of new frames. The third aspect of frame control is the change in system state, in events or time, between successive, displayed frames. This state difference is the minimum of the specified number of events-per-frame and the number of clock-ticks-per-frame. By adjusting the display mode, frame rate, and state differential, the performance analyst can study gross behavior, examining a small subset of all states, or examine the trace event by event.

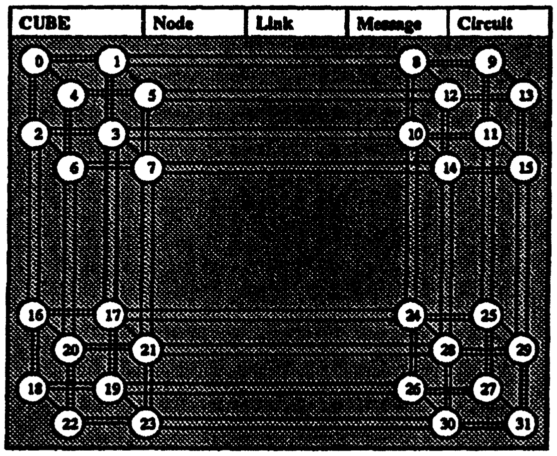
6.4.1.3 Global Statistics

The `Statistics` window shows the global system state, both cumulative message statistics and current node and link activity. Because the performance analyst can browse the trace, cumulative statistics are not monotonic — they reflect performance data relative to the current trace state.

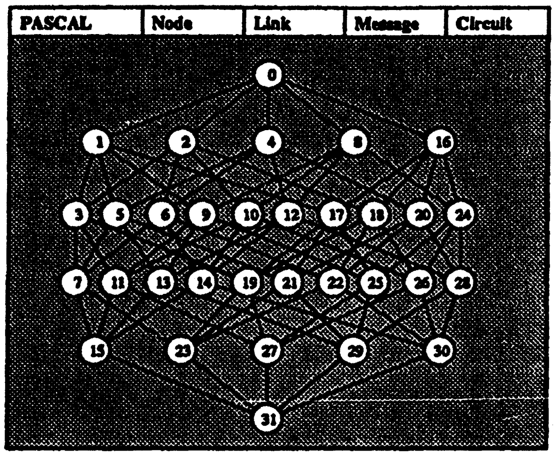
6.4.1.4 System Displays

Figure 6.14 lists the dynamic system displays provided by HyperView. The *CUBE*, *PASCAL*, *FFT*, and *QUEUE* displays are shown in Figure 6.15. Each display gives a different representation of the distributed memory system (in this case the system has a hypercube topology) that shows current system activity as highlighted nodes and links. Each provides a different insight; collectively they convey system dynamics. For example, the *CUBE* display is the "natural" multi-dimensional representation of a hypercube. In contrast, the *FFT* display emphasizes message routing paths. The *GRAY CODE* display (not shown) emphasizes sub-cube communication — communication links connecting the two $D-1$-dimensional subcubes of a D-dimensional hypercube appear as parallel lines [35]. The *PASCAL* display reflects the logarithmic combining (e.g., global minimization) when logical trees are embedded in the hypercube topology. Finally, the *QUEUE* display shows the instantaneous state of the message queues at each node. Each message awaiting transmission is shown as a small box. Communication transients appear as bursts of enqueued messages. Similarly, the effects of differing communication paradigms (e.g., store-and-forward message switching and circuit switching) appear as differences in mean queue size.

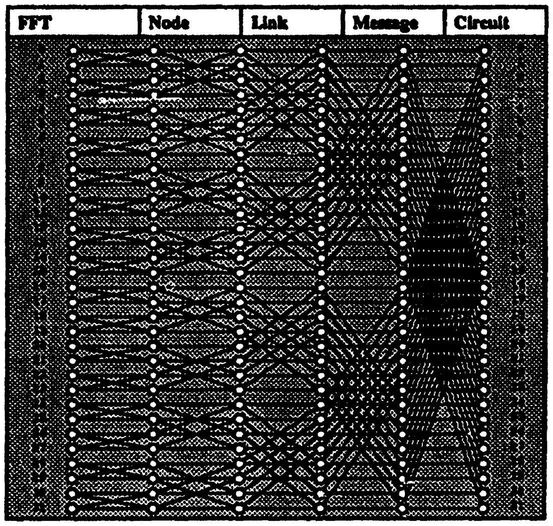
In all displays, colors emphasize activity — links change color when messages are sent, nodes flash when processing messages. Moreover, each topological display window supports pull-down menus for inquiries about nodes, links, messages, and circuits. Unwanted detail can be elided via the `Node` and `Link` menus. For example, display of any combination of transmitting, active, or receiving nodes and links can be disabled. Figure 6.16 shows the `Link` menu; the `Node` menu is similar. `All`, `Active` and `Transmitting` select the displayed link states. In Figure 6.16, only the active links (i.e. links currently held for message communication) are shown.



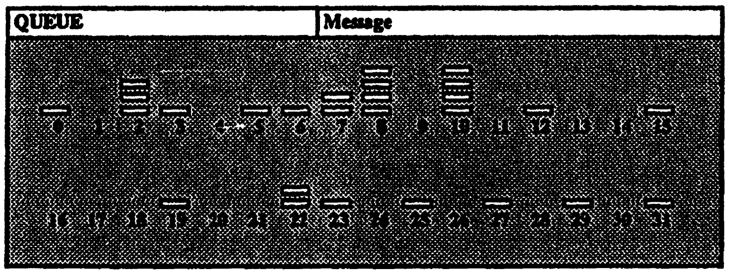
(a) Cube Display



(b) Pascal Display



(c) FFT Display



(d) Queue Display

Figure 6.15: HyperView Topology Displays

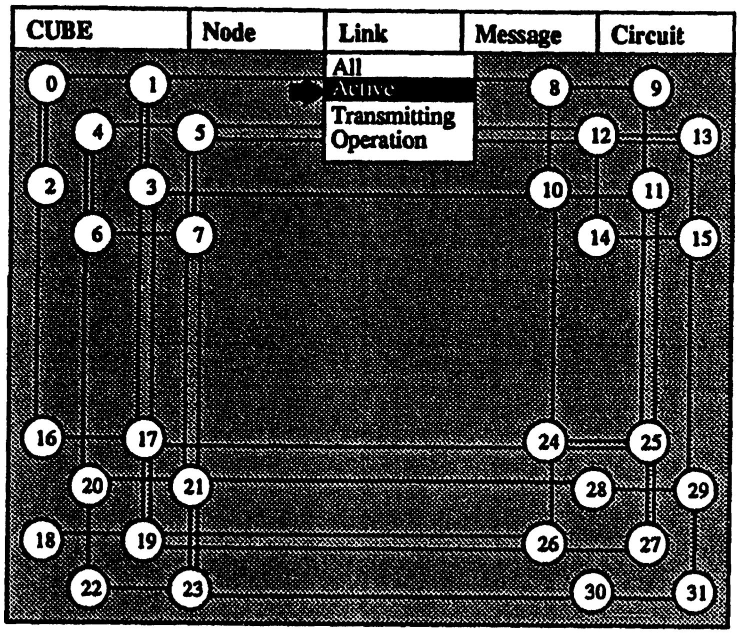


Figure 6.16: HyperView Link Activity Window

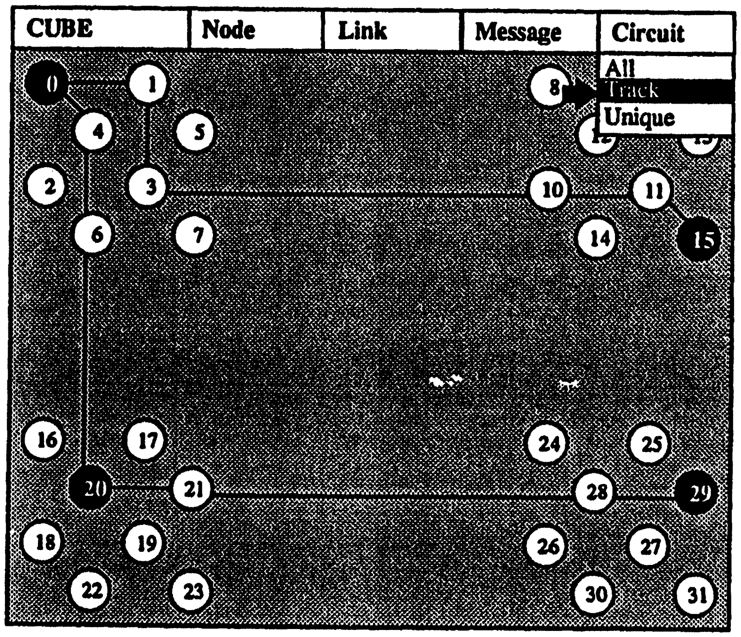


Figure 6.17: HyperView Circuit Activity Window

6.4.1.5 Message and Circuit Tracking

In addition to elision of unwanted node and link details, HyperView supports *message tracking* and *circuit tracking.*[17] After identifying source and destination nodes, *only* those messages in transit between the specified nodes are displayed. Figure 6.17 illustrates message and circuit tracking in a system with circuit switched communication. In the figure, nodes zero and twenty have been selected for circuit tracking and message tracking, respectively. Node zero is transmitting a message to node fifteen along the path shown. The intermediate nodes on the path are not active (they are shown as white) because only circuit connections have been established there. Concurrently, node zero is sending a message to node twenty, and node twenty is sending a message to node twenty-nine.

Message and circuit tracking have proven invaluable when comparing communication paradigms. By eliding extraneous detail, the dynamics of circuit establishment in both fixed and adaptive routing paradigms can be easily compared.

6.4.1.6 Limitations

Although the first generation HyperView system proved insightful for understanding the behavior and performance of different simulated message communication paradigms, its use in studying real application performance on the Intel iPSC/2 hypercube was less rewarding. The design did not allow the easy extension of performance displays and the displays themselves were strictly defined to present a certain type of performance data. We regard the ability to let the performance analyst choose data and its display as of major importance to the general acceptance of a performance visualization system. This capability was lacking in the first generation HyperView design.

6.4.2 Second Generation

A performance visualization environment must be extensible, permitting addition of new display formats and performance metrics, and flexible, permitting use with a variety of systems and applications. The environment must be easy to use and permit diverse approaches to visualizing

[17]The choice and semantics depend on the underlying hardware communication paradigm.

performance data. The second generation HyperView system [127] sought to overcome the problems of extensibility and flexibility in the earlier design.

6.4.2.1 Infrastructure Design

The main principle followed in the second generation HyperView design was to logically separate the performance data semantics from the displays. Given the diversity of performance data and analysis, a variety of performance displays are required to display the dynamics of system performance. We purposely implemented a variety of generic displays without any embedded data interpretation. In addition to these displays, the environment includes a set of data filters that process the event data input; although the semantics of each filter's internal state differ, these semantics need not be known by the display tools. By isolating semantic issues, different filter data can be displayed in multiple ways using different standard views. Via an environment control, the binding of data filter and display can be changed dynamically, allowing the user to select the filters and display formats best suited to his or her needs.

Thus, the HyperView data analysis and visualization system contains a *user interface*, an *event preprocessor*, a set of generic data analysis *filters*, a set of filter-display interfaces called *strainers*, and a set of *displays*. The user interface allows the performance analyst to configure and manage the filters, strainers and displays. Via this interface, the user can change the attributes of performance views and experiment with different display combinations. The event preprocessor converts a trace event stream from a performance experiment into a standard format for use by the event filters. The event filters accept the trace events and maintain an internal performance state reflective of the filter type. The HyperView prototype has been configured to process traces from operating system and application performance instrumentation on the Intel iPSC/2 [165, 166]. As such, the filters currently include

- message counts and message volume, selected by message type, message size, and source and destination node,
- processor state, including utilization and context switches, and
- program state, including current procedure and execution profile.

Each filter has an associated set of event strainers that, via access to the internal filter state, create a view-specific data representation.

The environment defines several interface standards, allowing addition of new filters, strainers and displays. These include standards for module initialization, termination and user customization. The builder of a standard module (i.e., filter, strainer, or display) need only meet these interface standards. The environment intrastructure then provides intermodule communication. This design promotes extensibility and design flexibility.

The following describes the performance displays constructed for the second generation HyperView system. Examples are then given of HyperView application.

6.4.2.2 Performance Displays

The diversity of the performance data demands an equally rich set of performance displays. Although the most useful set of displays can be determined only from experience, we constructed a prototype set of displays using the widgets of the X window system [168]. The displays include

- dials
- bar charts
- LEDs
- Kiviat diagrams
- matrix views
- X-Y plots
- contour plots
- gantt charts
- strip charts, and a
- general purpose graph display.

Each display is configurable, via the environment control and the X window manager, and is capable of displaying data of various types.[18] In addition, the portability provided by X permits the display environment to execute on a wide variety of vendor workstations.

Three examples illustrate our approach to display development. First, we consider an *LED* display that can represent a scalar with a finite range of assumable values; see Figure 6.18a.[19] The display includes an single method, used to set the scalar value, and four display resources, *range*, *levels*, *markers*, and *interval*. The *range* resource defines the range of displayable values. Within this range, the *levels* resource controls the number of display levels and the mapping to

[18]Several of the displays were used in the performance visualisation of Cedar hardware operation in Figures 6.4 and 6.5.

[19]Conceptually, an LED widget provides a bar chart with user definable discretisation of display values.

a level (i.e., what values within the display range are mapped to each level). For the current display value, the display will enable the level indicators, either with bitmaps or colors, from the minimum value up to, and including, the level for current display value. The *markers* resource defines the bitmaps or colors associated with each display level. Finally, the *interval* resources specifies the number of display updates over which a maximum value, a "high water mark," should be shown. Figure 6.18 shows sixteen LED displays, each with thirty levels.

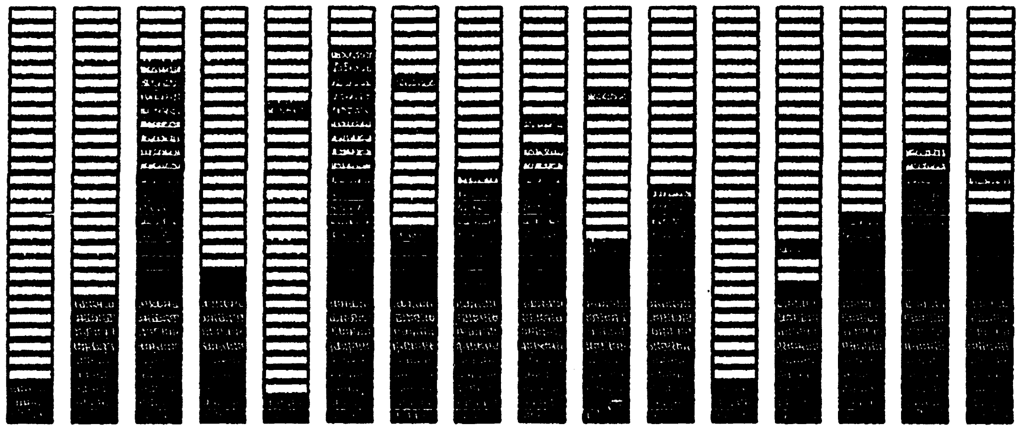
Our second example is a general matrix display that can be used to show one- and two-dimensional array data.[20] Widget resources specify the number of matrix rows and columns, the range of expected values expected, and the mapping of values to bitmaps or colors. The single display update method accepts a pointer to an array of display values. In the display, each matrix cell corresponds to a particular value in the array. Its representation depends on the value's position within the range of values and the bitmap or color corresponding to that position. Figure 6.18b shows a twenty row matrix with twenty columns; see Figure 6.21 for matrix displays of actual performance data.

Finally, the graph display can draw an input graph (representing nodes and arcs in a virtual coordinate space) within the dimensions of the display window. Resources specify node shapes, labels, and connectivity. Via the graph display methods, nodes can be bitmap or color filled, and links can be flashed or color coded. Finally, mouse clicks on nodes or links are reported to the application using the widget.[21] The generality of the graph display permits many uses, including display of procedure call graphs and communication traffic. Figure 6.18c shows a Pascal triangle representation of a 32-node hypercube topology. The colors of nodes and links might be used to display node and link utilization.

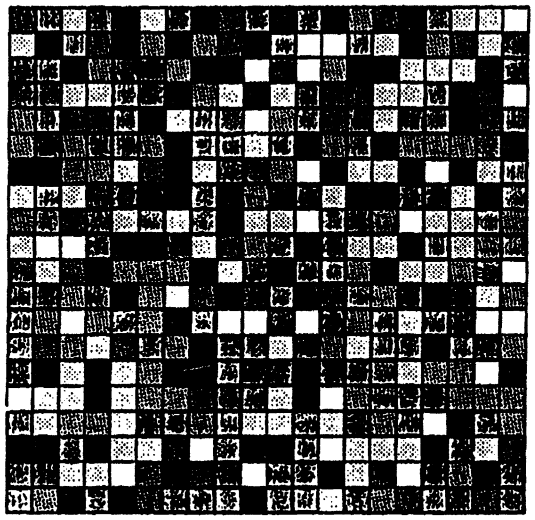
6.4.2.3 HyperView Examples

As example of HyperView's ability to flexibly present performance data using a variety of displays, the following shows its application to performance visualization of trace data from a parallel implementation of a Simplex linear optimization algorithm [187, 55]. Because the HyperView performance visualization environment and its display widgets are designed to show dynamic performance, again much is lost in the description of static, monochrome images.

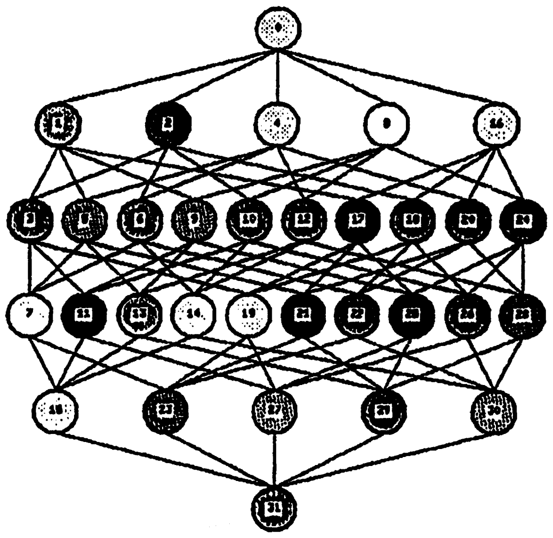
[20]The matrix display widget was the main component in the implementation of Xmatrix.
[21]Most displays provide some form of *callback* functionality.



(a) LED

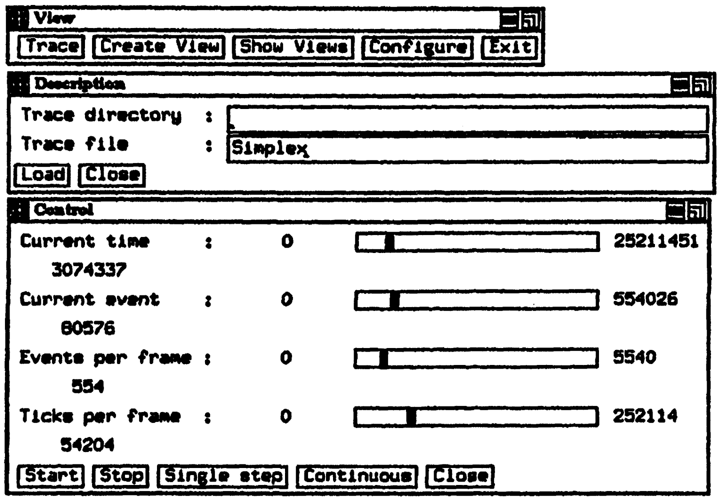


(b) Matrix



(c) Graph

Figure 6.18: Performance Data Displays



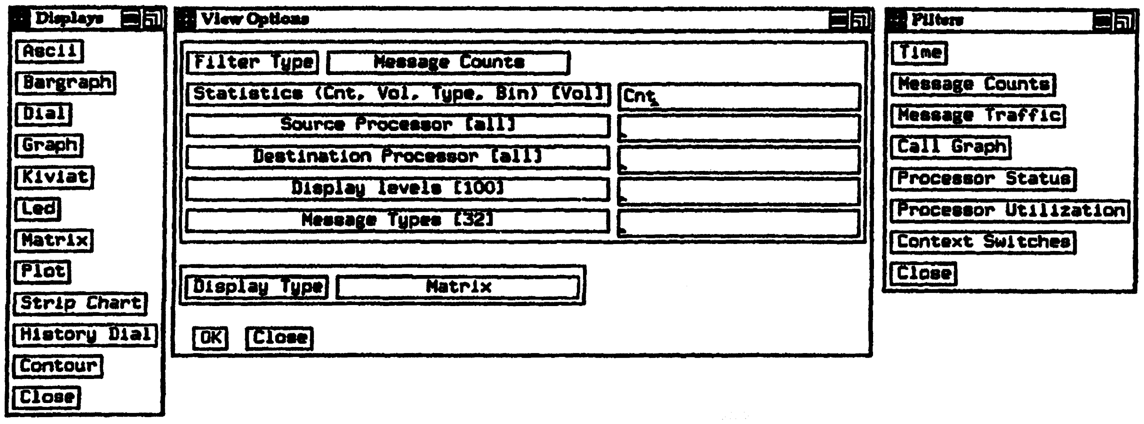
```
Trace directory  :
Trace file       : Simplex

Current time     :   0   25211451
   3074337
Current event    :   0   554026
   80576
Events per frame :   0   5540
   554
Ticks per frame  :   0   252114
   54204
```

Figure 6.19: Visualization Environment Control

The top-level user interface, the `Visual` window, is shown in Figure 6.19. The command buttons in the `Visual` window permit the performance analyst to control the environment and create new display views. The trace description and control windows shown in Figure 6.19 are created via a pulldown menu from the `Trace` button. In the `Description` window, a performance analyst can select a trace file that contains the event information captured by the compiler and operating system instrumentation.

As the name suggests, the `Control` window allows the performance analyst to control the updates to the displays. It has the same basic functionality as in the first generation HyperView implementation. The current time and current event displays show the current trace location and are updated automatically. Again, the user can change the mode and state differential aspects of frame display. Frames can be displayed either in single-step mode or continuous mode as before.

As discussed in §6.4.2.1 and §6.4.2.2, the visualization environment includes a set of data analysis filters and a set of displays. Figure 6.20 shows the configuration interface for connecting filters and displays. The `View Options` window is created via the `Create View` button in Figure 6.19. In turn, the `Display table` and `Filter table` windows are created via the `Filter Type` and `Display Type` buttons. In Figure 6.20, the user has previously selected a filter that counts the volume and number of messages transmitted among processors, and has opted to display



Filter Type: Message Counts
Statistics (Cnt, Vol, Type, Bin) [Vol]: Cnt
Source Processor [all]
Destination Processor [all]
Display levels [100]
Message Types [32]
Display Type: Matrix

Figure 6.20: Visualization Environment Configuration

message counts as a source-destination processor matrix. Finally, the options particular to this display choice are shown in the `View Options` window, including the defaults. Because display views are dynamically bound to data filters, the user can add new display views at any time and in arbitrary combinations.

Figure 6.21 shows some of the currently operational display views. The event trace file used to generate these views was obtained via compiler and operating system instrumentation [166] on an eight node Intel iPSC/2, and reflects the computation and communication behavior of a Simplex linear optimization code [187]. The `Time Stamp` window shows the effects of local processor clocks. The `CntxtSw:All PEs Interval` window displays the number of state changes by each node processor in a sliding window of time. Although not clearly shown in Figure 6.21, the history dial display leaves shadows of the dial pointer, showing the rate of change in the displayed variable. The `Interval User` window shows a plot of the amount of computation time spent in user mode averaged over a sliding time interval. Although the computation is well balanced, the disparity in processor computation times for an interval is evident. Time spent in other system states can also be displayed. The current state of each processor (idle, user computation, system state, or message transmission) is shown in the `PE Status` window using a one-dimensional matrix (vector) display. A Kiviat display is used to show the percentage of time each node processor is busy in a sliding interval.

The displays in the upper left of Figure 6.21 show communication traffic. The message count and volume bar charts show the fraction of message counts and volume sent by each processor.

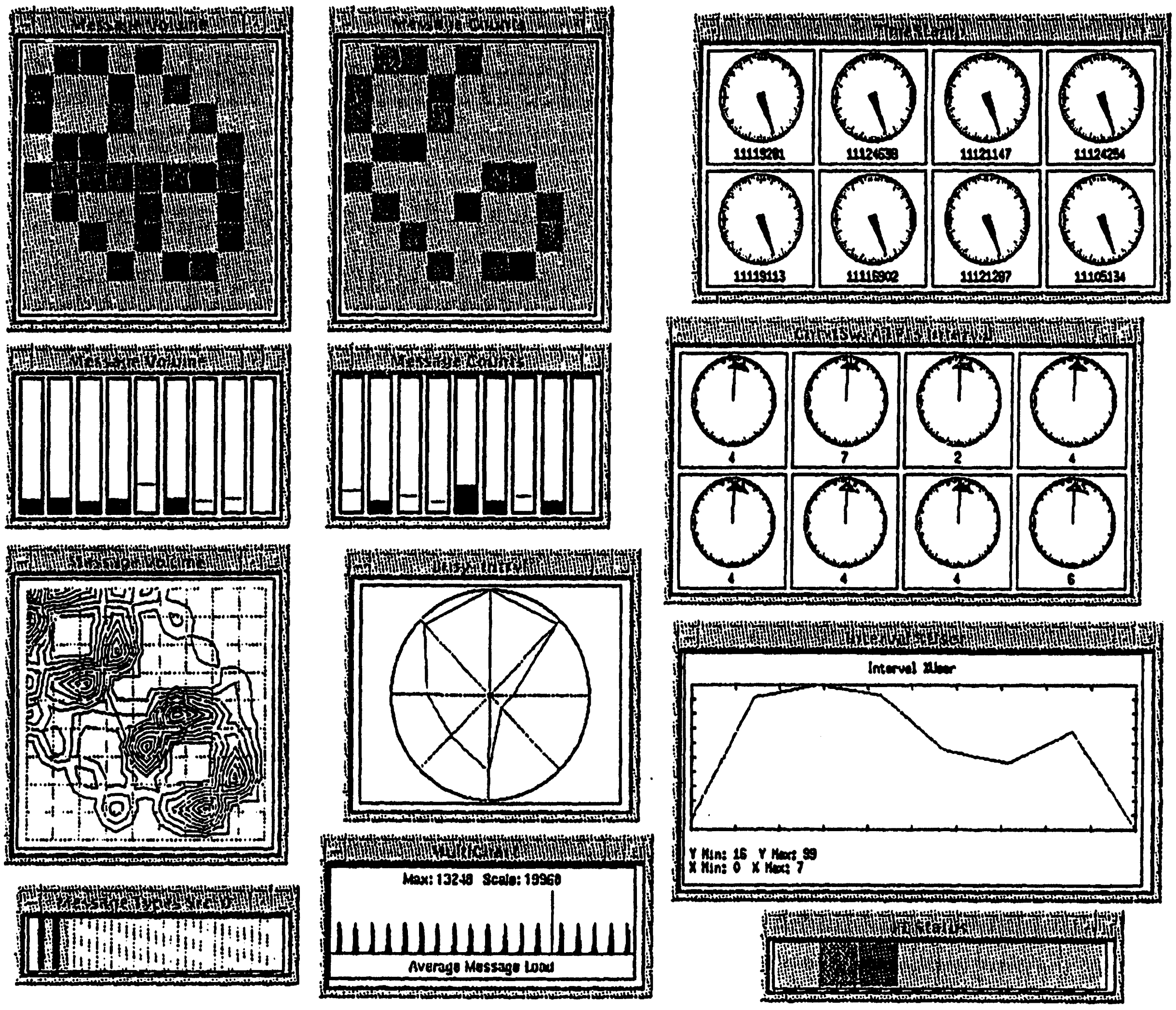


Figure 6.21: Visualization Environment Displays (Simplex Trace)

The two-dimensional matrix displays of message count and volume reflect the total number of messages and volume of data, respectively, sent between each source (row) and destination (column) processor; the ninth row and column denote communication traffic to and from the host processor. Notice that the distribution of message sizes is bimodal; the diagonal pattern in message counts display is much less pronounced in the message volume display. Why? In each cycle of the simplex algorithm, the processor collectively determine a global minimum via logarithmic condensation [187], this generates many small messages — the diagonal pattern seen in the message counts matrix display. This minimization is followed by the broadcast of a row or column of the optimization constraint matrix. These row broadcasts constitute most of the communication data volume but only a small fraction of the total message count. The `Message Bins Src:0` window gives a vector display of message size distribution; as just noted, most messages are small. Finally, the average message load display at the bottom of Figure 6.21 shows the sliding window average of the network communication traffic; in the display, the repeated cycles of communication traffic caused by minimization and broadcast are evident.

The display views and options of Figure 6.21 are but a subset of those possible. Each display has many options and can be bound to many data filters; this permits diverse display idioms.

6.5 Comments

The quintessential theme of this thesis is that the growing complexity of computer systems demands advances in performance observability. In this chapter, we have shown several examples how performance visualization techniques and tools can be applied to improve performance observability. In addition, we have proposed a performance visualization environment architecture for the development of new tools with common design principles and have demonstrated the architecture ideas in two prototype environments. However, equally important as the tools is their pervasive application to performance evaluation problems. We comment on two points in this regard: expense and acceptance.

First, if the system used to visualize performance data is more expensive than the system being measured, the application of performance visualization as a general purpose tool will be lost. In contrast, high-performance graphics equipment is often required for scientific visualization applications where complex graphics rendering and transformation algorithms must be performed. Whereas computational scientists are willing to pay a premium for scientific visualization tools because these tools are so fundamental to their work, the cost-effectiveness of performance visualization tools is less certain. Although the cost advantages of achieving better performance on a computer system are clear, especially for expensive supercomputer systems, and, in some cases, improved performance has a fundamental effect on the ability to do science [157], the practical choice of spending some fraction of the cost of the machine on performance visualization resources versus additional computer system options most often would lean to the latter — increasing capacity, such as adding more processors to a multiprocessor system, has more appeal than merely improving efficiency at the same cost. The fallacy that performance problems can necessarily be solved by increasing computer system resources is becoming apparent [157], particularly for parallel processing systems [80], but the issue of expense for performance visualization tools, and other tools for performance observability for that matter, is still important in their general availability.

The second comment is that the acceptance of performance visualization tools depends on how much they improve a performance analyst's productivity. Not only must the tools be regarded as offering better capabilities for performance investigation, they must be able to be applied in a time-effective manner. We have shown examples of how performance visualization can improve performance insight and can assist the performance analyst in interrogating the performance data. However, performance analysis is an experimental and iterative process that is part of a larger design-implement-evaluate-optimize cycle. Achieving a high-performance system or application in a timely manner is the end goal of this development cycle, and the performance analyst will choose his or her tools accordingly. Performance visualization tools should be easy to learn and to apply or else the user might opt for simpler approaches, even at the loss of reduced capabilities.

Chapter 7

Performance Observability Study

The purpose of computing is insight, not numbers.

— Richard Hamming

The underlying theme of the thesis and the motivation for the development of integrated performance environments is that improvements in performance observability are worthwhile. Intuitively, many would agree that greater insight into performance behavior has a beneficial impact on better performance evaluation and optimization. In practice, however, the lack of available detailed performance measurement systems, especially on high-performance machines, makes it difficult to find actual evidence to this effect. Moreover, this is fundamentally a qualitative judgement that often depends on the user's level of sophistication and perspective.

This chapter describes results from a study of performance observability we performed on the Cray X-MP and Cray 2 systems [122]. This work involved several aspects of performance observability discussed in the previous chapters. The goal of the study was to investigate the question of whether additional performance measurement and analysis tools applied in a real system with full application codes has a significant advantage over the commonly applied techniques. We use as our criteria the ability to gain greater insight into application performance behavior and to deliver greater observable performance detail. We do not argue the issue of whether a quantitative increase in performance observability leads to a qualitative improvement

[1] Part of the results presented in this chapter appears in the paper "Tracing Application Program Execution on the Cray X-MP and Cray 2" to be included in the Proceedings of the 1990 Supercomputing Conference [122].

in the user's ability to optimize a code's performance. Rather we conclude that such result would be a logical consequence of better performance observability methodology and techniques.

7.1 Supercomputer Performance Characterization

The performance of an application is rarely in static equilibrium during execution. For supercomputers this situation is even more acute as, in general, the performance range is greater and performance variations among program segments can be more pronounced. The complexity in the performance space surrounding the use and interactions of advanced architectural, hardware, and software features of supercomputers often implies that minor alterations in execution behavior can create large changes in achieved performance. Simply put, application performance, especially for supercomputers, can be highly variable and depends significantly on the dynamic interaction of the code with the high-performance features of the machine.

The problem facing the performance analyst is how to characterize an application's operation both in terms of its overall performance and its dynamic execution behavior — what code is being executed when and how the machine resources are being used. Profiling tools typically report application code performance in the form of summary statistics showing how execution time is allocated across program code blocks [72, 73]. Whereas summary performance statistics directly identify code segments consuming large fractions of total execution time, they do not provide insight into how the application executes over time nor its dynamic use of machine resources.

In part, the goal of this chapter is to present techniques for capturing and analyzing dynamic program execution on supercomputer systems and to show that performance measurements of execution behavior can provide greater insight than summary statistics. But this conclusion is not unexpected — other research efforts have reported similar findings [59, 71, 133, 141, 170, 176, 177], although few have been in the context of supercomputer systems. There has been a reluctance, however, to generally apply techniques to capture dynamic execution state for fear that the measurement system will corrupt the execution behavior being observed, particularly in the case of high-performance systems. In many cases, this fear is largely unfounded and simple perturbation models, as described in Chapters 3, 4, and 5, can be applied to recover true execution performance [113, 128]. To the extent that it does occur, the increased insight into applications operation offered by the trace data must be weighed against the perturbations, and therefore performance inaccuracies, in observed execution behavior.

In the following sections, we describe a trace-based measurement system implemented for the Cray supercomputers and its use in characterizing the performance dynamics of full application codes. The Cray supercomputers provide a practical environment to test the merits of trace-based performance characterization. The Cray compilers support automatic instrumentation at the routine level to capture entry and exit events [37, 108]. The existence of a fast-access, high-resolution system clock allow fine-grained timing measurements. Furthermore, the Cray X-MP includes a hardware performance monitor which can be sampled to show the run-time behavior of several hardware metrics.

The remainder of the chapter is organized as follows. In §7.2, we briefly review the standard profiling tools available on the Cray systems. In §7.3, we describe the tracing system developed for the Cray X-MP and Cray 2. We introduce the applications codes from the Perfect Benchmark set used for testing purposes in §7.4. In §7.5, we compare the profiling results from the standard tools to those calculated from the application code traces. We use the comparison of these results as a measure of the reliability of the trace data. In §7.6 we analyze the execution dynamics of the Perfect codes, primarily FLO52, on the Cray X-MP and Cray 2. The tracing information obtained from the two systems is also used to make architectural comparisons. Finally, we present some conclusions and directions for future research.

7.2 Standard Cray Tools

Two profiling tools are commonly used on Cray machines: *Flowtrace* and *Perftrace.* The *Flowtrace* tool [37] is available on the Cray X-MP, Cray Y-MP, and Cray 2. It purpose is to measure where time is spent in a program's execution and to generate a time profile based on program routines. Unlike sample-based, interrupt-driven profilers [72], *Flowtrace* calculates the profile dynamically by inserting profiling code at the beginning and end of each routine; similar approaches have been used in ELXSI systems [28]. This instrumentation is provided automatically by the Cray compilers. At the end of the program's execution, the time profile is generated and written to a file. The profile includes the number of calls to the routines, the time spent in each routine, and the average time per routine call. Information regarding the routine calling tree is also given.

The *Perftrace* tool [108] is available only on the Cray X-MP and Cray Y-MP systems. *Perftrace* computes all the statistics of *Flowtrace* in addition to generating a profile of hardware performance. Similar to the time profile measurement, *Perftrace* samples the counters of the hardware performance monitor (*HPM*) at routine entry and exit to determine the distribution of hardware performance across the program routines.[2] Because the *HPM* allows only one of four hardware counter groups to be monitored at a time (each group contains eight counters), *Perftrace* reports only the profiling statistics for the selected counter group. Multiple runs (up to four) must be made in the case hardware profiles spanning counter groups are desired. In addition to the time profile statistics, *Perftrace* reports the counter value of each hardware metric accumulated for each routine (e.g., floating point operations or memory references), the values shown in millions per second (e.g., megaflops[3]), the percentage of the hardware metric total the routine counter values represent, and different derived statistics depending on the counter group (e.g., memory references per floating point operation).

Both *Flowtrace* and *Perftrace* produce summary performance statistics — the statistics reflect performance totals accumulated for the entire program execution. Summary statistics reflect dynamic execution but only in an averaged form. The fundamental property of the *Flowtrace* and *Perftrace* statistics, from a performance characterization standpoint, is that they show only cumulative performance behavior.

7.3 A Cray Tracing Environment

We developed a tracing system for the Cray machines for purposes of capturing a history of a program's performance behavior. We were able to leverage the pre-existing Cray compiler support for *Flowtrace* and *Perftrace* to provide automatic instrumentation for tracing measurement. The tracing environment includes a low-level library for trace logging plus libraries that replace the standard *Flowtrace* and *Perftrace* routines.[4]

[2]The HPM is available only on the Cray X-MP and Y-MP machines. For further information on the *HPM*, see [107].

[3]Millions of floating point operations per second.

[4]The compiler instrumentation called these routines.

Routine	Description
tevent(*tid*, *eid*)	writes a timestamp and the event *eid* to the trace buffer of task *tid*
tdatas(*tid*, *eid*, *data*)	writes a timestamp, the event *eid*, and the integer data item *data* to the trace buffer of task *tid*
tdata(*tid*, *eid*, *n*, *ptr*)	writes a timestamp, the event *eid*, and the first *n* integer data items pointed to by *ptr* to the trace buffer of the task *tid*; *tdatas()* is faster for a single data item
texit()	writes the data currently in the trace buffers for each task to files task.*i* where *i* is the task id

Table 7.1: Cray Tracing Library Routines

7.3.1 Software Event Tracing

The basic function of a software event tracing facility is to record the occurrence of an event by writing a timestamped event identifier, with optional data, to a trace buffer. In the case of concurrent event tracing, a buffering scheme is needed that allows multiple tasks concurrent access without conflicts. In our tracing facility for the Cray machines, we allocate trace buffers statically, one for each possible task.

The routines in the Cray tracing library are described in Table 7.1. The event identifier and optional data are supplied by the user with the high-resolution system clock being retrieved from the Cray library routine *irtc()*.[5] Because the high-resolution system clock is a register shared by every processor in the machine, all tasks see a common global time value. The side-effect of using the high-resolution cycle counter is that it is real-time and, therefore, timing measurements are susceptible to multiprogramming influences. Thus, the current tracing library can generate accurate timing measurements only in dedicated mode.

The routine *texit()* is called at the end of program execution to save the trace data in trace files. If, during execution, the trace buffers overflow, the current trace data for the task whose buffer overflowed are written out to the associated trace file before the task's execution continues.

Although the tracing routines support the concurrent capture of trace events from multiple tasks, all the results reported in the paper are from scalar and vector executions. Analysis of

[5]In Cray's Fortran compiler,CFT77, *irtc()* is compiled into a single machine instruction.

Routine	Cray X-MP (cp = 8.5 ns)			Cray 2 (cp = 4.1 ns)		
	Mean	*Variance*	*Standard Deviation*	*Mean*	*Variance*	*Standard Deviation*
tevent()						
clock periods	142	33	6	335	728	27
nanoseconds	1207	281	51	1374	2985	111
tdatas()						
clock periods	155	64	8	351	923	30
nanoseconds	1318	544	68	1439	3784	123
tdata()						
clock periods	1102	428	20	2662	12427	111
nanoseconds	9367	3638	170	10914	50951	455

Table 7.2: Cray X-MP and 2 Tracing Execution Time Overheads

concurrent traces is complicated by the inability to efficiently determine the processing resource where the event was recorded. We currently are pursuing solutions to this problem.

In general, the tracing routines should be implemented to minimize execution time overhead. The current implementation of the tracing routines is in Fortran to enhance portability between the Cray machines.[6] The execution time overheads (mean, variance, and standard deviation) for the tracing routines on the Cray X-MP and Cray 2 are shown in Table 7.2. The *tdata()* numbers are for ten data items saved with the event identifier. In all cases, the subroutine invocation times constitute a significant portion of the total. The execution time overheads of the tracing routines are used during trace analysis to remove the intrusion due to instrumentation; see §7.5.

7.3.2 TraceFlow and TracePerf

We developed code that captures routine entry and exit events using the tracing library to record the trace data. The goal was to extend the existing, automatic *Flowtrace* and *Perftrace* instrumentation provided by the Cray compilers [37, 108]. Our approach replaced the *Flowtrace* and *Perftrace* routines with versions that generate traces.

We refer to the traced *Flowtrace* routines as the *TraceFlow* library. The trace *Perftrace* routines we refer to as the *TracePerf* library. The routines in each library are described in Table 7.3. The principal difference between the *TracePerf* and *TraceFlow* libraries is that

[6]In fact, there are no differences in the tracing library for the Cray X-MP and the Cray 2, and the library should port without change to the Cray Y-MP system.

Routine	Description	
	TraceFlow	TracePerf
flowentr()	records the routine entry event in the trace for the calling task; the routine identifier is saved	records the routine entry event in the trace for the calling task; the routine identifier and HPM counters are saved
flowexit()	records the routine exit event in the trace for the calling task	records the routine exit event in the trace for the calling task; the HPM counters are saved
flowin(*blockid*)	records the block entry event in the trace for the calling task; the block identifier *blockid* is saved	records the block entry event in the trace for the calling task; the block identifier *blockid* and HPM counters are saved
flowout()	records the block exit event in the trace for the calling task	records the block exit event in the trace for the calling task; the HPM counters are saved
perfon()	not defined	same as flowin()
perfoff()	not defined	same as flowout()
flowstop()	writes the trace files	write the trace files
flowoff()	disable tracing	disable tracing
flowon()	enable tracing	enable tracing

Table 7.3: Cray TraceFlow and TracePerf Routines

Routine	TraceFlow	TracePerf			
		Group 0	*Group 1*	*Group 2*	*Group 3*
flowentr	307	1493	1492	1492	1492
flowexit	204	1481	1481	1481	1481
flowin	230	1495	1495	1497	1496
flowout	204	1409	1409	1409	1409
perfenable	—	1534	1535	1534	1534
perfdisable	—	1460	1460	1460	1460

Table 7.4: Cray X-MP TraceFlow and TracePerf Execution Time Overheads

Routine	TraceFlow	
	Minimum	*Average*
flowentr	720	733.7
flowexit	476	491.96
flowin	514	535.12
flowout	480	495.04

Table 7.5: Cray 2 TraceFlow Execution Time Overheads

the *TracePerf* routines record current HPM counter values.[7] Note that for both *TraceFlow* and *TracePerf*, the routine name is saved in the trace only for "entry" events. We assume that entry/exit pairs can be determined from the trace during analysis. Because a frequently called routine can cause the trace to grow large, the *flowoff()* and *flowon()* routines have been implemented to control when trace data should be saved.

There is a significant practical difference between the overheads produced by the *TraceFlow* and *TracePerf* routines. Table 7.4 gives the mean execution time overheads for calling the *TraceFlow* and *TracePerf* routines for the Cray X-MP.[8] Table 7.5 shows the mean execution time overheads for *TraceFlow* on the Cray 2 machine.[9] Although *TracePerf* records approximately five times the amount of data as *TraceFlow*, the execution time difference is because *TracePerf* routines require an I/O operation to retrieve HPM counter values. The Cray UNICOS operating system [37] maintains HPM counter values only for user execution. In addition to the hardware performance counters, the *HPM* keeps a cycle counter. Fortunately, because I/O operations

[7]We used the hardware performance monitor of the CRAY X-MP/48 running UNICOS at the National Center for Supercomputing Applications in Urbana, Illinois.

[8]As in *Perftrace*, only one group of eight counters out of 32 are accessible during any execution. Thus, execution time overheads are listed for each of the four groups.

[9]Because no hardware performance monitor exists on the Cray 2, no *TracePerf* library can be constructed.

Routine	Counter							
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
flowentr	391	590	13	0	153	0	0	0
flowexit	385	572	14	0	149	0	0	0
flowin	397	589	13	0	158	0	0	0
flowout	378	543	12	0	146	0	0	0
perfenable	401	609	14	0	159	0	0	0
perfdisable	385	569	13	0	149	0	0	0

Table 7.6: Cray X-MP TracePerf HPM Counter Overheads - Group 0

Routine	Counter							
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
flowentr	3	3	229	306	0	0	0	38
flowexit	3	3	229	289	0	0	0	38
flowin	3	3	216	301	0	0	0	58
flowout	3	3	200	289	0	0	0	38
perfenable	3	3	231	306	0	0	0	58
perfdisable	3	3	229	288	0	0	0	37

Table 7.7: Cray X-MP TracePerf HPM Counter Overheads - Group 1

take place in system mode, we can use *HPM* cycle counter as a "virtual" high-resolution system clock in computing the execution time overheads in the trace.

For *TracePerf* there are execution time overheads as well as overheads in the HPM counters. That is, calling the *TracePerf* routines will affect the HPM counters. To accurately recover HPM counts during trace analysis, we measured the counter overheads for the *TracePerf* routines for each of the counter groups. These overheads are reported in Tables 7.6, 7.7, 7.8, and 7.9.

Routine	Counter							
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
flowentr	13	53	0	0	0	100	0	36
flowexit	14	51	0	0	0	98	0	34
flowin	13	52	0	0	0	106	0	42
flowout	12	48	0	0	0	98	0	34
perfenable	14	53	0	0	0	106	0	42
perfdisable	13	51	0	0	0	98	0	34

Table 7.8: Cray X-MP TracePerf HPM Counter Overheads - Group 2

Routine	Counter							
	0	*1*	*2*	*3*	*4*	*5*	*6*	*7*
flowentr	81	306	0	0	4	0	0	36
flowexit	79	302	0	0	4	0	0	34
flowin	81	309	0	0	7	0	0	42
flowout	76	298	0	0	4	0	0	34
perfenable	84	310	0	0	7	0	0	42
perfdisable	79	302	0	0	4	0	0	34

Table 7.9: Cray X-MP TracePerf HPM Counter Overheads - Group 3

7.4 Perfect Benchmarks

The applications codes used in our study (FLO52, OCEAN, and DYFESM) are benchmark programs from the Perfect Benchmark suite [19]. The baseline versions of the codes were compiled with default CFT77 compiler options and executed on one processor. As such, our results differ slightly from those presented in [152]. In particular, no preprocessors were used, all codes were run on a single processor, and no hand optimizations were performed.

The benchmark codes were chosen from different application areas. FLO52 and OCEAN are fluid dynamics programs. FLO52 is a two-dimensional analysis of the transonic inviscid flow past an airfoil and solves the unsteady Euler equations. OCEAN solves the dynamical equations of a two-dimensional Boussinesq fluid layer to study the chaotic behavior of free-slip Rayleigh-Benard convection. DYFESM is a two-dimensional finite element program for the analysis of symmetric anisotropic structures. An explicit leap-frog temporal method with substructuring is used to solve for the displacements and stresses, along with the velocities and accelerations at each time step.

7.5 Trace Verification

The primary motivation behind tracing application program execution is to capture dynamic performance behavior. The *TraceFlow* library can be used to understand the timing properties of routine execution. The *TracePerf* library can be used to obtain additional data on machine performance. However, with the use of these libraries comes perturbations in the program's behavior, both in execution time and in the HPM counter data. The severity of these perturbations and our ability to remove them will determine the reliability of the trace information.

Routine	Cray X-MP				Cray 2			
	Flowtrace		TraceFlow		Flowtrace		TraceFlow	
	Time	Percent	Time	Percent	Time	Percent	Time	Percent
ADDX	1.002	1.94	1.010	1.94	1.175	1.98	1.177	2.00
BCFAR	0.311	0.60	0.315	0.60	0.295	0.50	0.301	0.51
BCWALL	0.918	1.78	0.925	1.77	0.815	1.38	0.809	1.37
COLLC	0.611	1.18	0.615	1.18	0.702	1.19	0.694	1.18
CPLOT	0.128	0.25	0.129	0.25	0.297	0.50	0.282	0.49
DFLUX	10.898	21.13	10.986	21.05	11.684	19.74	11.67	19.84
DFLUXC	1.566	3.04	1.584	3.04	1.573	2.66	1.578	2.68
EFLUX	15.125	29.33	15.171	29.07	17.218	29.09	17.11	29.08
EULER	7.726	14.98	7.800	14.95	8.603	14.54	8.534	14.51
MESH	0.100	0.19	0.100	0.19	0.108	0.18	0.108	0.18
PRNTFF	0.113	0.22	0.119	0.23	0.223	0.38	0.266	0.45
PSMOO	10.370	20.11	10.627	20.36	14.583	24.64	14.314	24.33
STEP	2.593	5.03	2.617	5.01	1.749	2.96	1.756	2.98
TOTAL	51.570	99.78	52.184	99.64	59.186	99.74	58.826	99.60

Table 7.10: Flowtrace and TraceFlow Statistics for Scalar FLO52

It was shown in Chapters 3, 4, and 5 that perturbations due to trace-based monitoring can, in many cases, be modeled and, subsequently, removed during trace analysis [128]. We applied the perturbation analysis techniques in the development of our trace analysis programs. In particular, we developed trace analysis programs that compute the same set of profile statistics generated by *Flowtrace* and *Perftrace*. We used the results obtained from these programs as a measure of the reliability of the trace data after perturbation analysis has been applied.

7.5.1 Flowtrace Profiling versus TraceFlow Profiling

The *Flowtrace* statistics summarize the distribution of execution time across an application's routines. The total number of calls to a routine and the total execution time within a routine are reported. Information is also provided about the execution call graph. Our program to analyze an application trace produces similar statistics, plus additional execution time profiles, for the parents and children of each routine.

Table 7.10 shows the execution time in seconds for some of the routines in FLO52 when executed in scalar mode as calculated by *Flowtrace* and from a *TraceFlow* trace. As shown, the trace-generated statistics are very close to those produced by *Flowtrace*. Table 7.11 gives the same results for vector execution. Again, the statistics match extremely well. This result also

Routine	Cray X-MP				Cray 2			
	Flowtrace		TraceFlow		Flowtrace		TraceFlow	
	Time	Percent	Time	Percent	Time	Percent	Time	Percent
ADDX	0.152	2.29	0.154	2.27	0.181	1.90	0.182	1.90
BCFAR	0.043	0.65	0.043	0.64	0.034	0.36	0.035	0.36
BCWALL	0.116	1.75	0.116	1.72	0.107	1.13	0.108	1.13
COLLC	0.083	1.25	0.083	1.23	0.180	1.89	0.179	1.87
CPLOT	0.128	1.93	0.129	1.92	0.297	3.12	0.281	2.93
DFLUX	1.231	18.53	1.243	18.43	1.757	18.51	1.769	18.44
DFLUXC	0.214	3.22	0.216	3.21	0.324	3.41	0.323	3.37
EFLUX	1.787	26.91	1.803	26.71	2.165	22.81	2.158	22.51
EULER	0.999	15.05	1.019	15.09	1.054	11.10	1.076	11.22
MESH	0.101	1.52	0.102	1.51	0.113	1.19	0.108	1.13
PRNTFF	0.113	1.70	0.119	1.77	0.223	2.35	0.223	2.32
PSMOO	1.317	19.83	1.330	19.81	2.718	28.63	2.736	28.53
STEP	0.256	3.86	0.259	3.83	0.188	1.98	0.188	1.96
TOTAL	6.641	98.49	6.749	98.14	9.493	98.38	9.590	97.67

Table 7.11: Flowtrace and TraceFlow Statistics for Vector FLO52

holds true for both scalar and vector execution of the other two Perfect codes we tested.

There are two general conclusions from this analysis. First, we can accurately measure gross execution performance characteristics, in the form of execution time profiles, from traces of routine entry and exit by including perturbations introduced by the trace instrumentation. Second, we believe the dynamic performance behavior, as represented by the full event trace, should also be credible. Although this second conclusion is difficult to prove in practice, the analysis in the preceding chapters supports this conclusion.

7.5.2 Perftrace Profiling versus TracePerf Profiling

An analysis similar to that used with *Flowtrace* and *TraceFlow* can be applied to *Perftrace* and *TracePerf*. The corresponding results are given in Tables 7.12 and 7.13; only Cray X-MP results can be reported. In addition to the execution time measurements, some of the operation counts from counter data from group 0 are shown; *Madds* represents millions of floating point additions, *Mmult* is millions of floating point multiplies, *Mrecip* is millions of floating point reciprocals, and *Mflops* is millions of total floating point operations.

The statistics generated from the *TracePerf* trace are very close to those reported by *Perftrace*. The importance of the *TracePerf* data is its ability to associate dynamic machine opera-

Routine	Measurement	Time	Percent	Madds	Mmult.	Mrecip.	Mflops
EFLUX	Flowtrace	15.1	29.30	101.0	106.0	10.60	218.0
	TraceFlow	15.1	29.35	101.0	106.1	10.59	217.7
DFLUX	Flowtrace	10.9	21.08	72.3	54.8	6.07	133.0
	TraceFlow	10.9	21.08	72.3	54.8	6.07	133.1
PSMOO	Flowtrace	10.3	20.03	69.5	35.4	0.02	105.0
	TraceFlow	10.4	20.09	69.5	35.4	0.03	104.9
EULER	Flowtrace	7.81	15.14	35.0	46.3	6.59	87.9
	TraceFlow	7.76	15.03	35.0	46.2	6.59	87.8
STEP	Flowtrace	2.59	5.02	12.8	17.7	4.42	34.9
	TraceFlow	2.60	5.03	12.8	17.7	4.42	34.9
DFLUXC	Flowtrace	1.55	3.01	9.42	8.98	1.29	19.7
	TraceFlow	1.57	3.04	9.42	8.99	1.29	19.7
ADDX	Flowtrace	0.99	1.93	8.92	8.81	0.43	18.2
	TraceFlow	0.99	1.93	8.92	8.81	0.43	18.2
BCWALL	Flowtrace	0.88	1.71	4.89	6.00	0.96	11.9
	TraceFlow	0.92	1.78	4.90	6.01	0.97	11.9
COLLC	Flowtrace	0.61	1.19	3.90	3.30	0.47	7.67
	TraceFlow	0.61	1.19	3.90	3.30	0.47	7.67
BCFAR	Flowtrace	0.31	0.60	1.58	2.22	0.37	4.17
	TraceFlow	0.31	0.61	1.58	2.22	0.37	4.16

Table 7.12: Perftrace and TracePerf Statistics for Scalar FLO52 - Cray X-MP, Group 0

tion with the sequence of application routine execution. The high reliability of this data implies that a finer degree of performance characterization can be obtained than with summary results of hardware performance.

7.5.3 Caveats

It should be mentioned that *Flowtrace*, *Perftrace*, *TraceFlow*, and *TracePerf* are prone to generating errors when measuring routines with small execution times. This is mainly the result of instrumentation overhead and not timer resolution. Although the analysis programs attempt to remove the overhead, a few percent deviation in the overhead relative to the routine execution time could be significant.[10] From the viewpoint of performance analysis, however, achieving high measurement accuracy for routines representing a small fraction of total program execution time is not of primary importance.

[10]The Cray tools do not report routine timings if the average time per call is less than 0.001 seconds because the results are considered untrustworthy.

Routine	Measurement	Time	Percent	Madds	Mmult.	Mrecip.	Mflop.
EFLUX	Flowtrace	1.76	26.40	101.0	106.0	10.60	218.0
	TraceFlow	1.79	26.90	101.0	106.2	10.59	217.8
PSMOO	Flowtrace	1.29	19.38	69.5	35.4	0.02	105.0
	TraceFlow	1.33	20.03	69.5	35.4	0.03	104.9
DFLUX	Flowtrace	1.23	18.43	70.3	54.8	6.07	131.0
	TraceFlow	1.24	18.61	70.3	54.8	6.07	131.1
EULER	Flowtrace	1.07	16.08	35.4	46.3	6.59	88.3
	TraceFlow	1.00	15.06	35.4	46.2	6.59	88.2
STEP	Flowtrace	0.25	3.77	12.8	19.0	4.42	36.3
	TraceFlow	0.26	3.85	12.8	19.0	4.42	36.3
DFLUXC	Flowtrace	0.21	3.07	9.42	8.98	1.29	19.7
	TraceFlow	0.21	3.22	9.42	8.99	1.29	19.8
ADDX	Flowtrace	0.15	2.26	8.92	8.81	0.43	18.2
	TraceFlow	0.15	2.28	8.92	8.81	0.43	18.2
CPLOT	Flowtrace	0.13	1.92	0.00	0.07	0.00	0.07
	TraceFlow	0.13	1.92	0.00	0.07	0.00	0.07
PRNTFF	Flowtrace	0.11	1.70	0.04	0.13	0.00	0.17
	TraceFlow	0.11	1.70	0.04	0.13	0.00	0.17
MESH	Flowtrace	0.10	1.50	0.59	0.69	0.08	1.36
	TraceFlow	0.10	1.51	0.59	0.69	0.08	1.36

Table 7.13: Perftrace and TracePerf Statistics for Vector FLO52 - Cray X-MP, Group 0

7.6 Dynamic Execution Analysis

Detailed event traces hold promise for characterizing dynamic execution performance if perturbation effects can be understood and compensated. Earlier, we used the accuracy of execution profile statistics, calculated from an event trace, as a measure of the reliability of the trace data and the validity of the perturbation analysis techniques. Below, we demonstrate some of the dynamic execution behavior analyses possible by applying the *TraceFlow* and *TracePerf* tools. Although we show results for the thıee Perfect codes, for brevity's sake, we concentrate primarily on the FLO52 code.

7.6.1 Subroutine Events

Using *TraceFlow* trace data, one can analyze the dynamic characteristics of subroutine calls, beginning with a simple *procedure event graph* that shows the subroutine and function transitions during application program execution. In such graphs, the axes represent, respectively, the currently executing application procedure and the procedure execution time.

Routine	Event	Routine	Event
FLO52Q	1	STEP	9
COORD	2	EULER	10
GEOM	3	COLLC	11
MESH	4	ADDX	12
GRID	5	PRNTFF	13
XPAND	6	CPLOT	14
METRIC	7	GRAPH	15
INIT	8	RPLOT	16

Table 7.14: FLO52 Routine Names to Event Numbers

When the dynamic pattern of calls is correlated with application source code locations, these procedure event graphs can show the spatial and temporal patterns of control flow. More abstractly, the procedure event graphs show application execution stages (e.g., the functional combination of multiple application algorithms or the processing of multiple input data sets).

Generally, the dynamic pattern of procedure calls is identical for both scalar and vector execution.[11] Clearly, vectorization compresses the event time scale and may change the relative execution times of individual procedure invocations. These are seen as changes in the "shape" of the procedure event graph.

Figures 7.1 and 7.2 show the procedure event graphs for FLO52 executions on the Cray X-MP in both scalar and vector modes, respectively. To provide sufficient resolution to display all relevant calls, the procedure event graphs are drawn in eight sections. In each section, time increases from top to bottom and continues at the top of the next strip. In the figures, each procedure invocation is marked by a horizontal line (i.e., a state transition). Vertical lines denote the amount of time spent in the currently executing procedure. To present an entire execution history, while avoiding event clutter, some of the more frequent procedure events have been elided. Finally, Table 7.14 shows the association of procedure names and event numbers used in Figures 7.1 and 7.2.

From Figures 7.1 and 7.2, one can identify five major execution phases:

- Initialization
- Grid One
- Grid Two
- Grid Three

[11] Inline function substitution to increase vectorization is one possible exception.

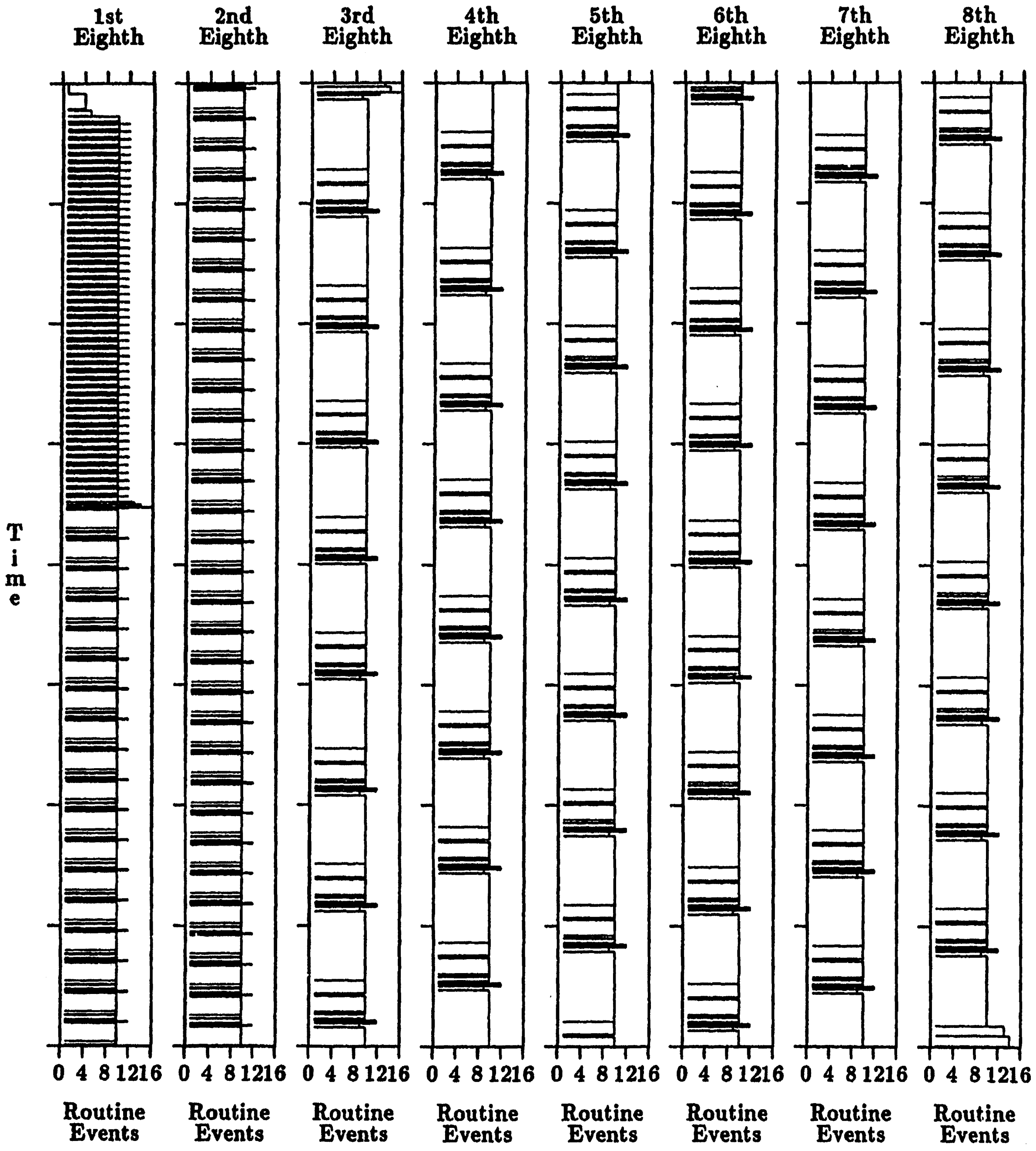


Figure 7.1: Event Graph for Scalar Execution of FLO52 on Cray X-MP (52.188527 seconds)

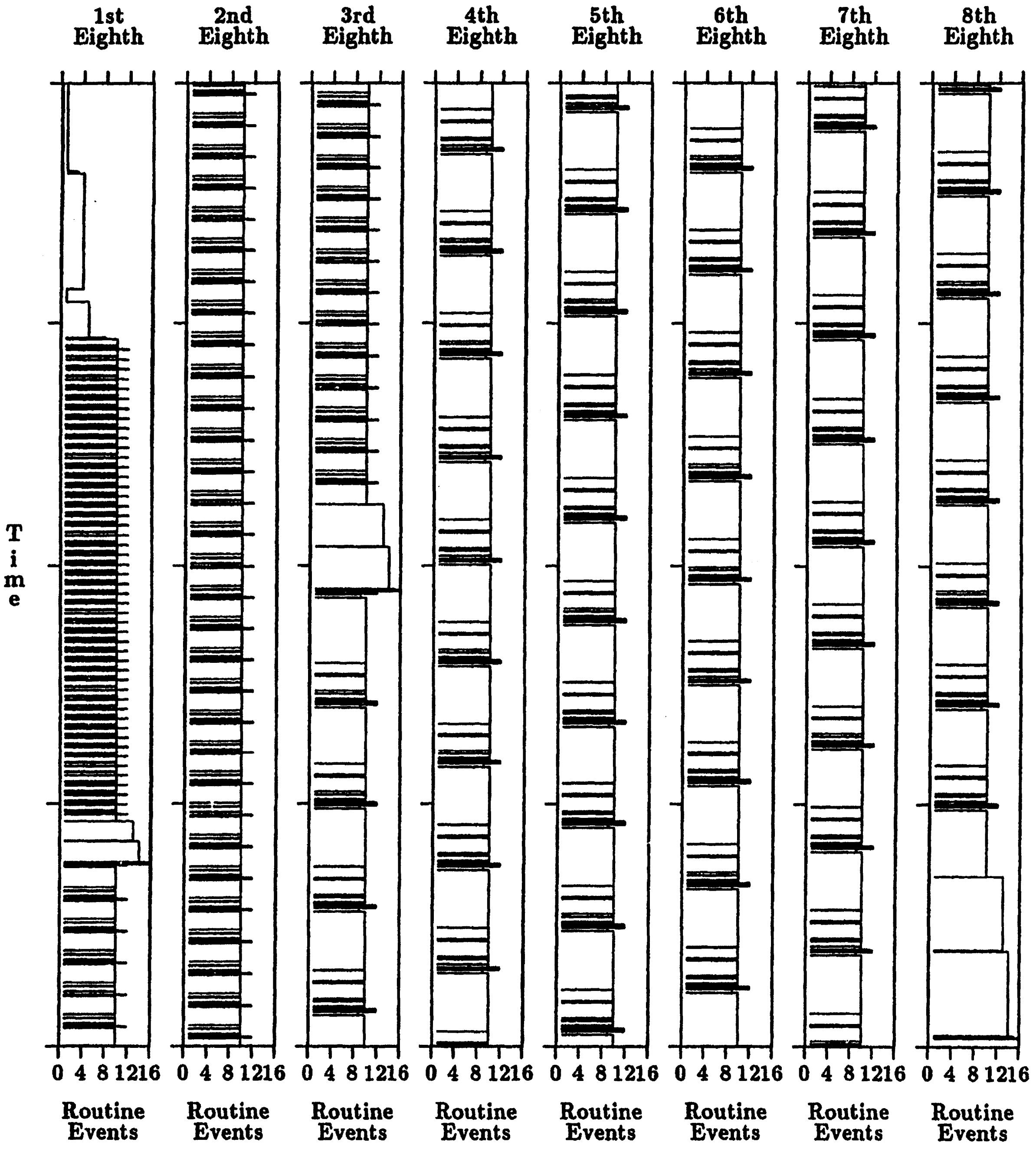


Figure 7.2: Event Graph for Vector Execution of FLO52 on Cray X-MP (6.749813 seconds)

- Termination

In the three grid phases, event analysis reveals a periodic sequence of procedure calls, where each phase contains 48 periods of the procedure invocation sequence. Obviously, the application phases, the repetition of procedure event sequences, and the procedure sequence itself are manifestations of the application program's call graph and the input data. Each grid phase reflects a dynamic instantiation of the static procedure call graph for a particular grid. Here, each "grid" phase represents a grid refinement of the FLO52 application's multigrid algorithm. In addition to the procedure calling behavior, Figures 7.1 and 7.2 show the changes in procedure execution times across application algorithm phases (i.e., grid refinements).

By comparing procedure event graphs for scalar and vector execution, one can quantify the interactions of vectorization, vector length, and grid size. For FLO52, the lack of vectorization in the initialization and termination phases increases their relative contribution to the vectorized version's total execution time. Moreover, grids two and three, with longer vectors, attain a substantially higher fraction of the Cray X-MP's peak performance in vector mode. This illustrates the value of time-dependent information. The average megaflop rate for this application is the weighted average of the megaflop rates for the three grid computations and depends on both the degree of application code vectorization and the range of vector lengths. These interactions are best understood by direct examination of the time varying application behavior, rather than attempting to infer interactions for first and second moments.

Although not shown, the procedure event graphs for the OCEAN and DYFESM Perfect applications show equally interesting, though different, procedure invocation behavior. As with the FLO52 code, these patterns of procedure calls can be correlated with the applications' algorithms and computation stages.[12]. Comparing procedure event graphs from the Cray 2 with those from the X-MP shows non-linear compressions of the event graph time scale. Although the general shapes of the graphs are similar, relative timing differences reflect architectural features, compiler optimizations, and the match of application code to the machines.

[12]Space constraints preclude a complete discussion of the procedure event graphs for the Cray 2 and for the other two Perfect codes.

7.6.2 Hardware Performance

In addition to generation and analysis of procedure call data the *TracePerf* traces permit analysis of dynamic machine performance, as captured by the HPM counters, during a application program execution. Below, we discuss a series of plots that display the the time-varying values of hardware performance metrics (in millions of events per second) for the entire execution of our example applications.

To simplify analysis and presentation, we divided each application program execution into five hundred, fixed size intervals of time. In each interval, we computed the average value for each hardware performance metric. Thus, applications that generate larger event traces likely will show less variation across adjacent time intervals (e.g., a trace for OCEAN contains ten times as many events as FLO52, and DYFESM contains twice as many as OCEAN). Also, because only one quarter of the HPM counters can be captured during a single program execution, we ran each application program eight times, once for each of the HPM counter groups in scalar and vector mode, to capture a complete set of hardware performance data. In the interests of space, we analyze only a small subset of the captured data.

Figures 7.3 and 7.4 show the data from HPM group zero for the scalar and vector execution of FLO52, respectively. Similarly, Figures 7.5 and 7.6 show the vector execution of OCEAN and DYFESM, respectively. In each figure's graphs, the horizontal axis is program execution time, and the vertical axis is the rate (in millions of events per second) of the associated hardware performance metric. Although the scalar and vector executions of FLO52 show similar behavior, the differences among the FLO52, OCEAN, and DYFESM codes are striking. The behavior of the vector DYFESM code is very regular; in contrast, the vector version of OCEAN code shows substantial variations in the floating point execution rate and the number of issued instructions. On a vector architecture, instruction issue rate and vector operations are inversely related (i.e., as the number of vector operations increases, the total number of issued instructions decreases – each instructions represents more useful computation). Thus, the behavior of the OCEAN code likely is attributable to frequent transitions to and from vector mode.

Although the application source code embodies a set of application algorithms and an associated number of arithmetic operations, a program compilation potentially can produce many different executable versions that are not work conservation equivalent. For example, a vectorized version might introduce redundant arithmetic operations to increase the vectorization level and execution performance. Thus, when comparing the performance of an application program across scalar and vector modes, one must verify that work is conserved. For the FLO52 code, the total number of floating point operations (6.43×10^8) is the same for both the scalar and vectorized versions.[13] Below, we compare and contrast the two versions of this code based on data obtained from the HPM counters.

In Figures 7.3 and 7.4, the three phases of grid evaluation are clear. In the vector case, the successively larger grids have longer vectors. This is reflected in higher megaflop rates (the ninth graph) and a lower instruction issue rate (the first graph). In both figures, "clock periods holding issue" is a measure conflicts for access to both registers and functional units that prevent release of an instruction to a functional unit. As vector length increases, more time is spent waiting for vector registers and vector functional units.

Because the Cray X-MP contains an instruction buffer, not every instruction fetch generates a memory reference. For example, if all code for a loop resides in the instruction buffer, instruction fetches from memory will cease during loop execution. As the third graph of Figures 7.3 and 7.4 suggests, the rate of instruction buffer fetches from memory is low, and the interference with operand memory fetches is small. However, examining both the scalar and vector executions of FLO52 shows that the number of instruction buffer fetches increases sharply during the transitions between grid sizes. In addition, the instruction buffer fetch rate is inversely proportional to megaflop rate for vector execution. Simply put, as the vector length increases, fewer instructions are needed to realize the same number of floating point operations.

Coupling our analysis of instruction buffer fetches with the graphs of CPU memory references confirms that most memory operations are for data access. In Figure 7.4, the memory reference rate is directly proportional to the megaflop rate.

Of the four floating point operation graphs, floating point reciprocals are the most interesting. Recall that the Cray X-MP does not implement division directly; instead, application program divisions require one reciprocal and three multiplication operations. In Figures 7.3 and 7.4 the floating point reciprocal rate, though small, is directly proportional to the megaflop, addition, and multiplication rates.

[13]The ninth graph of Figures 7.3 and 7.4 shows the time varying rate of these floating point operations in millions per second (i.e., megaflops).

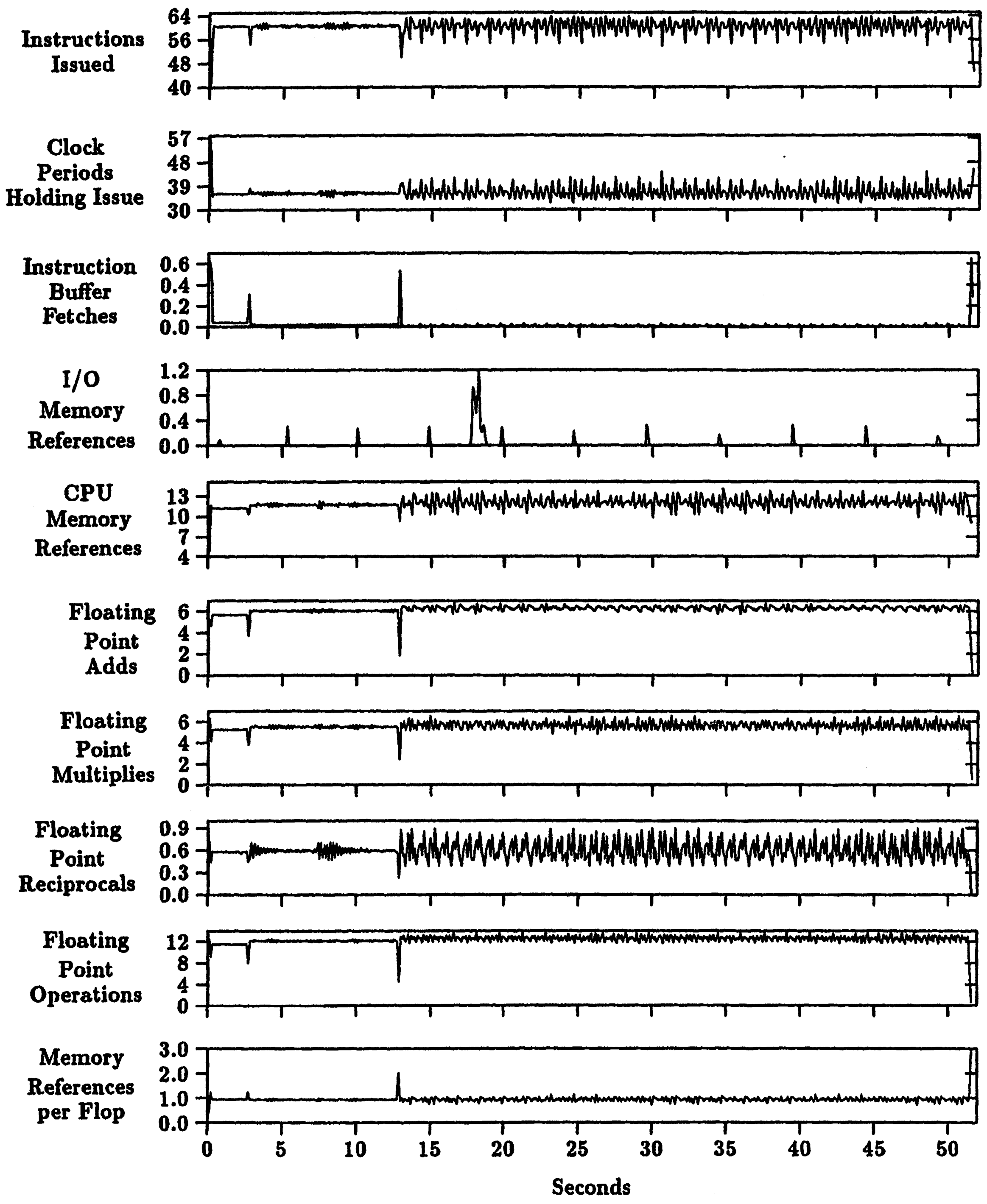


Figure 7.3: TracePerf Data for FLO52 Scalar Execution - Counter Group 0

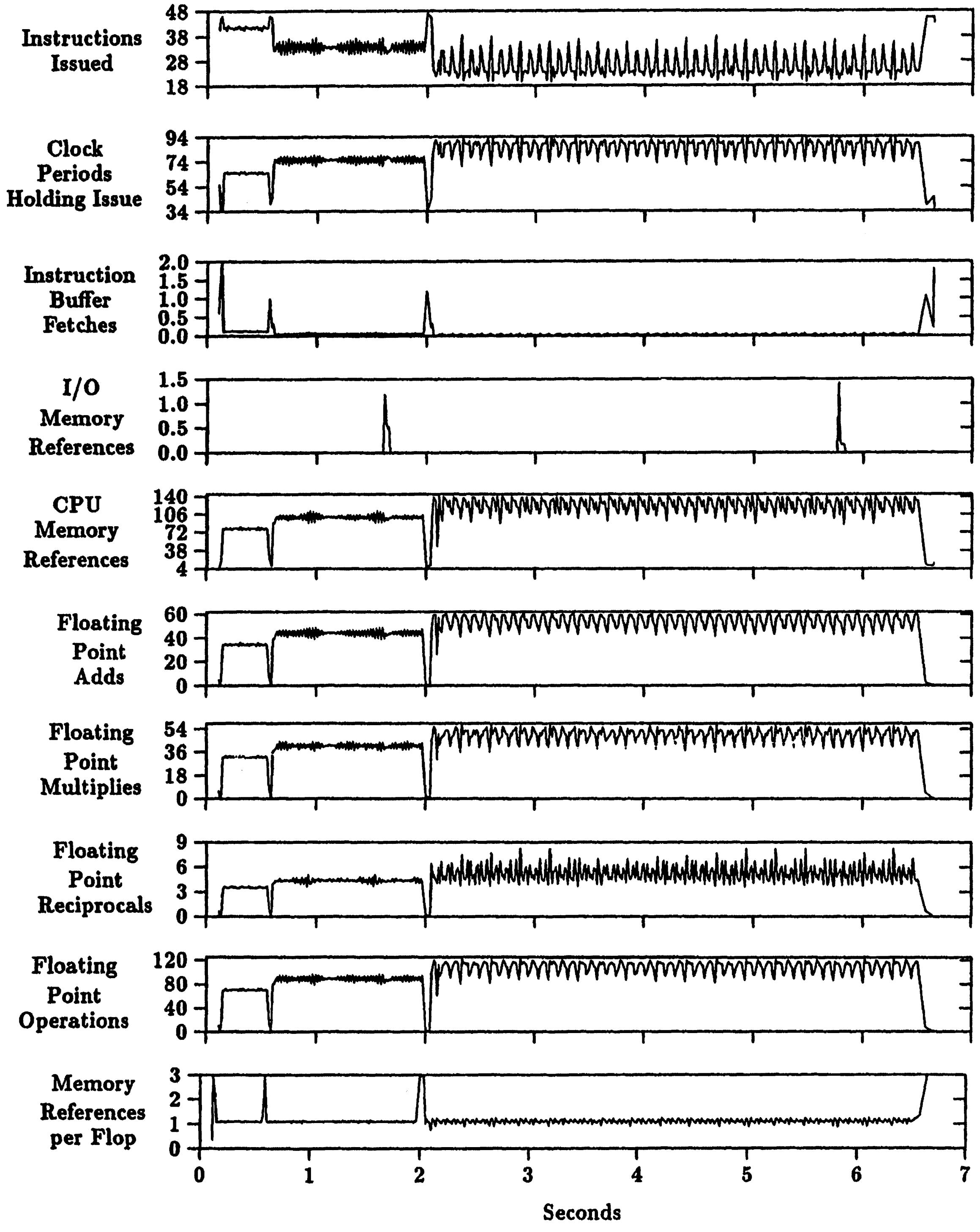


Figure 7.4: TracePerf Data for FLO52 Vector Execution - Counter Group 0

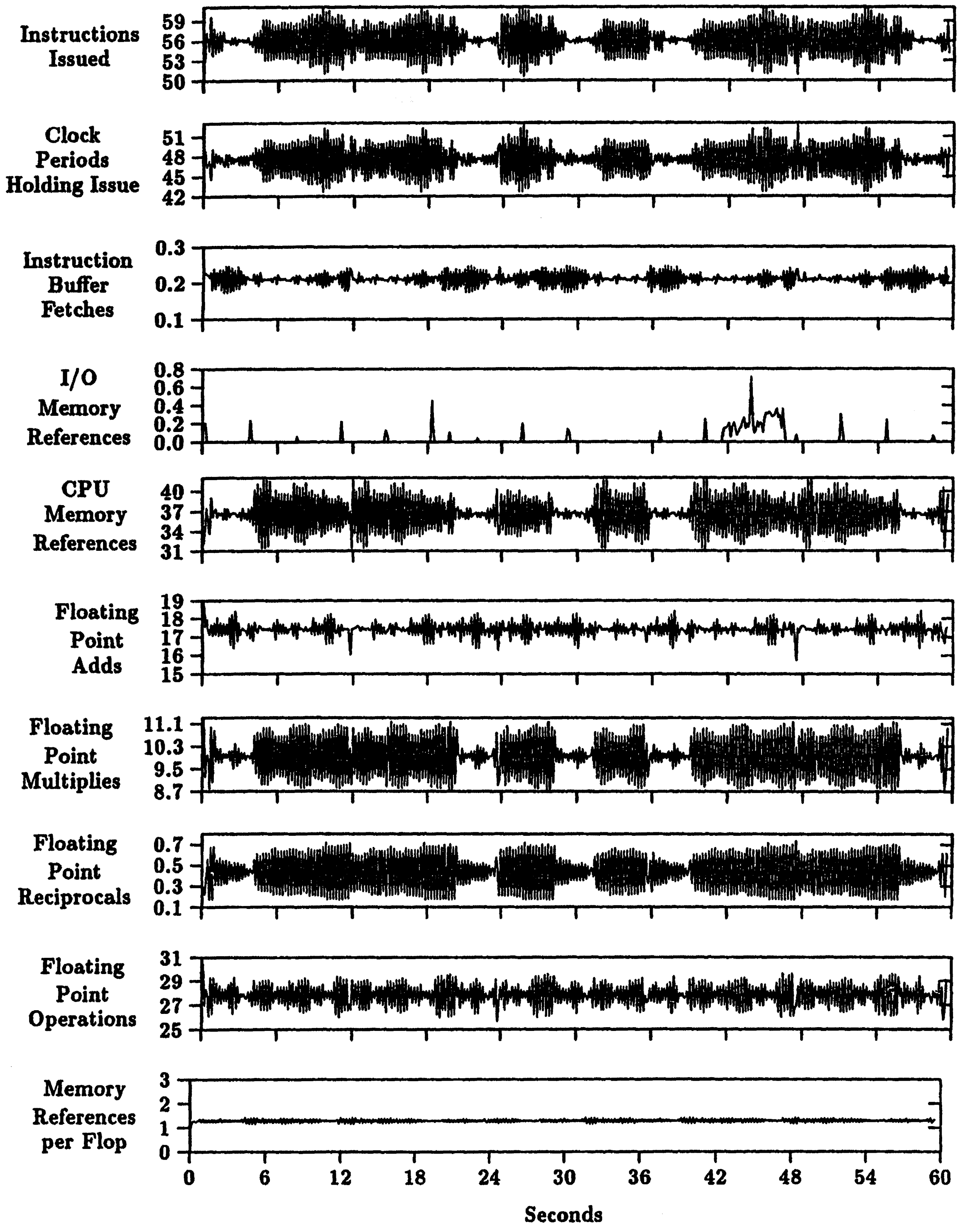


Figure 7.5: TracePerf Data for OCEAN Vector Execution - Counter Group 0

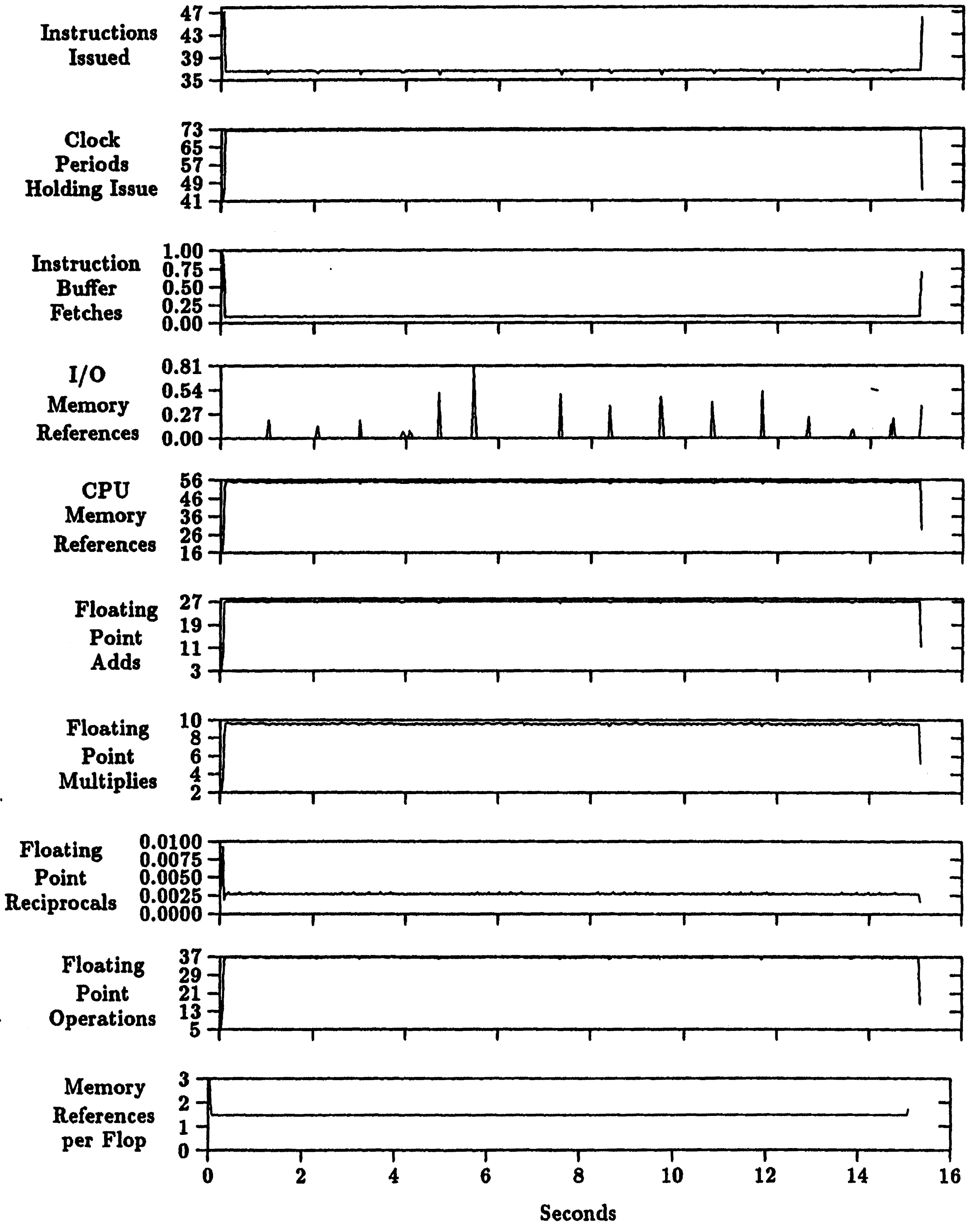


Figure 7.6: TracePerf Data for DYFESM Vector Execution - Counter Group 0

Routine	Event	Routine	Event
ADDX	1	EFLUX	7
BCWALL	2	PSMOO	8
FLO52Q	3	BCFAR	9
STEP	4	FORCF	10
EULER	5	COLLC	11
DFLUX	6	DFLUXC	12

Table 7.15: FLO52 Routine Names to Event Numbers for Event Correlation with Hardware Monitor Metrics

Finally, the average number of memory references for each floating point operation is near one. Although this might suggest that a single memory port, as found in the Cray-1, might suffice, the small-scale variance is large.[14] Without the Cray X-MP's three memory ports, performance would degrade substantially.

7.6.3 Procedure Execution and Hardware Performance

Although hardware performance graphs show the time-varying behavior of critical hardware metrics, it is important to correlate this behavior with the execution of particular application procedures. The event graph can be displayed next to the hardware performance graphs as done in Figures 7.7, 7.8, and 7.9 — the three figures correspond to the three phases of the vectorized FLO52 application where we have focussed in on one period of each phase. In each figure, the routine event graph is shown at the top with the counter group 0 graphs below. Because we are taking a more focussed view of the execution, all procedures events are shown. The routine-to-event mapping for these figures is slightly different than before and is given in Table 7.15.

[14]Recall that each graph point represents the average over 1/500th of the computation. The size of these intervals hides the most of the small-scale variance.

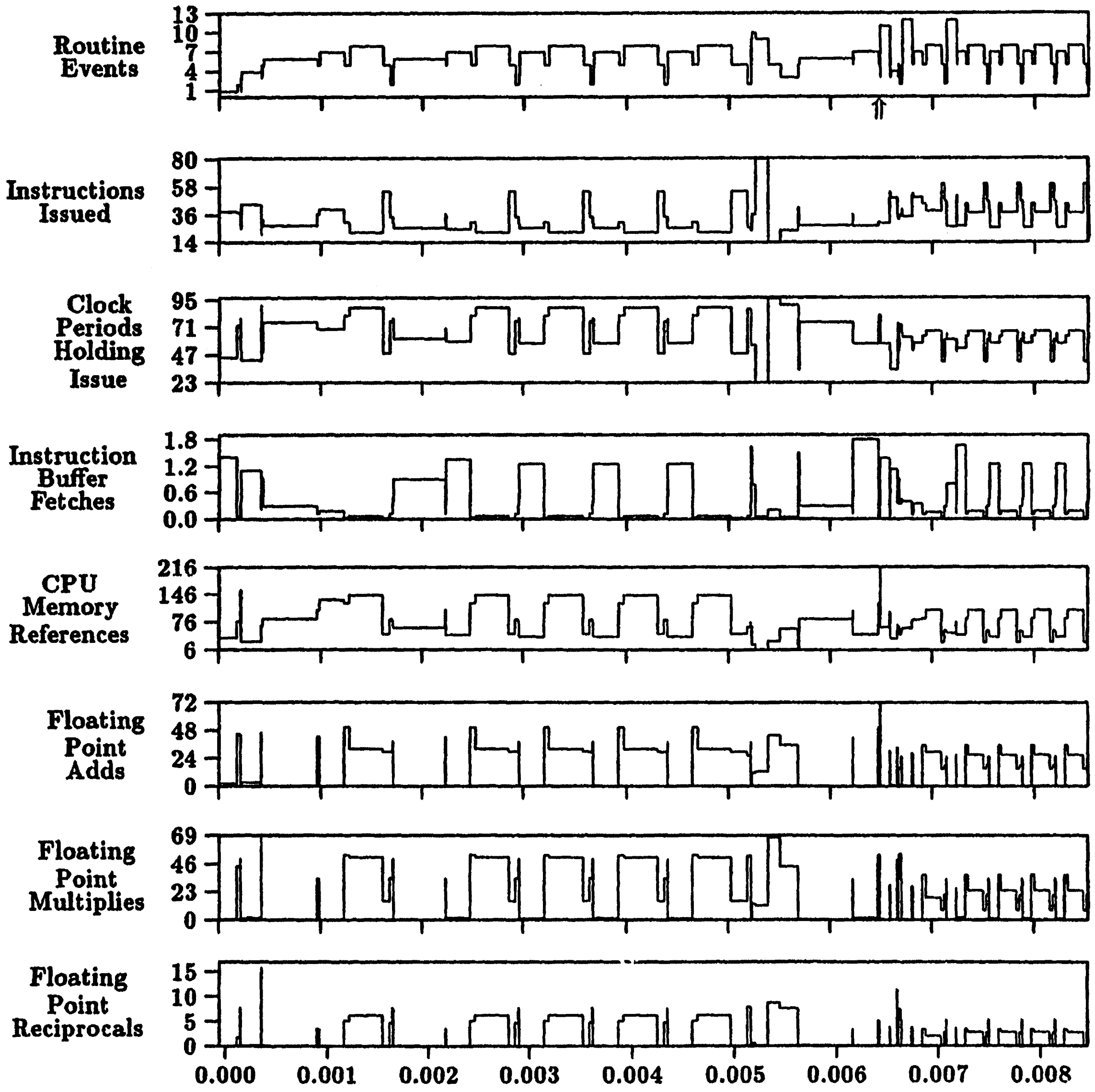


Figure 7.7: FLO52 Vector Execution - Group 0, Phase 1 (0.00849680)

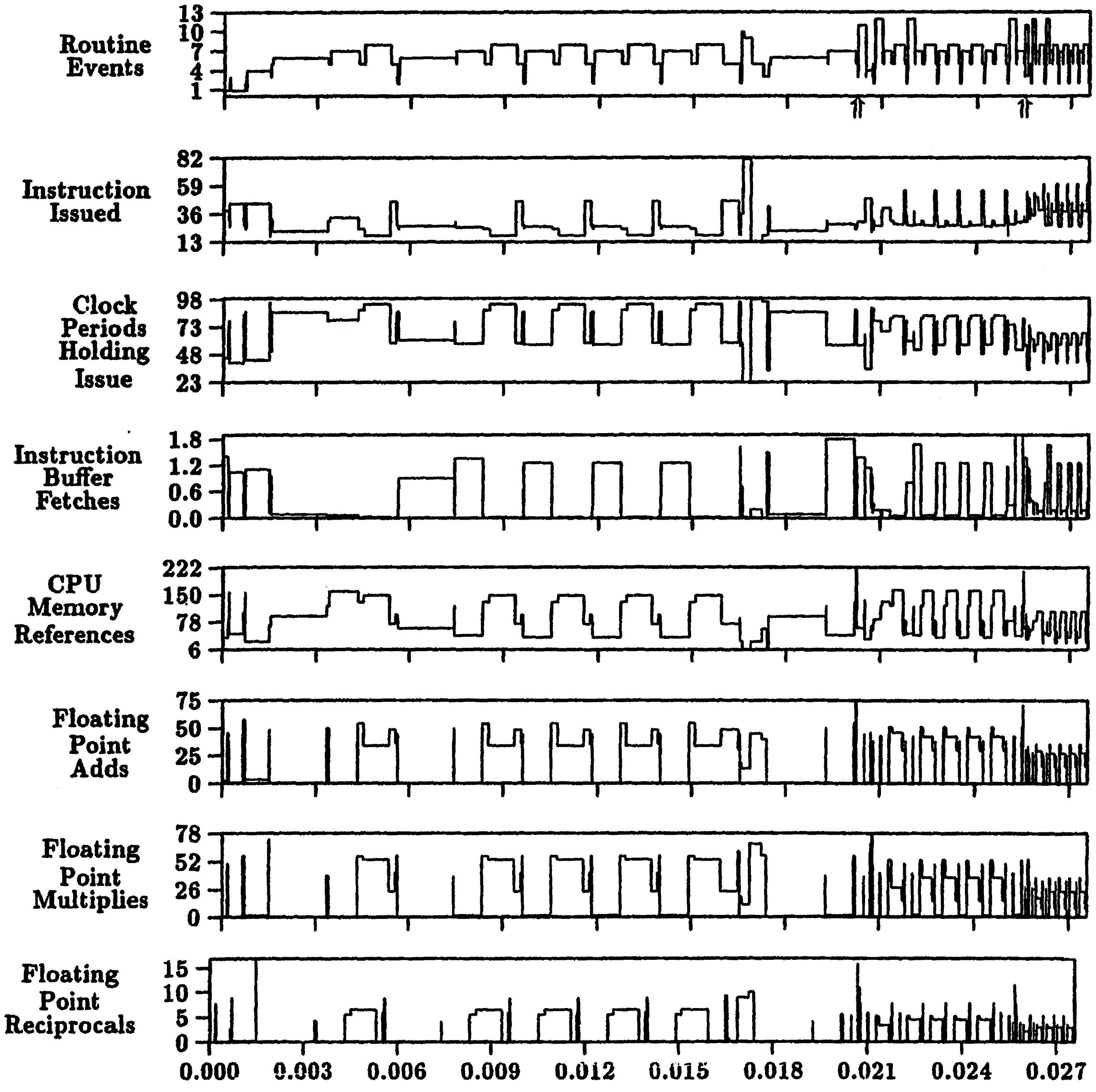


Figure 7.8: FLO52 Vector Execution - Group 0, Phase 2 (0.02751698)

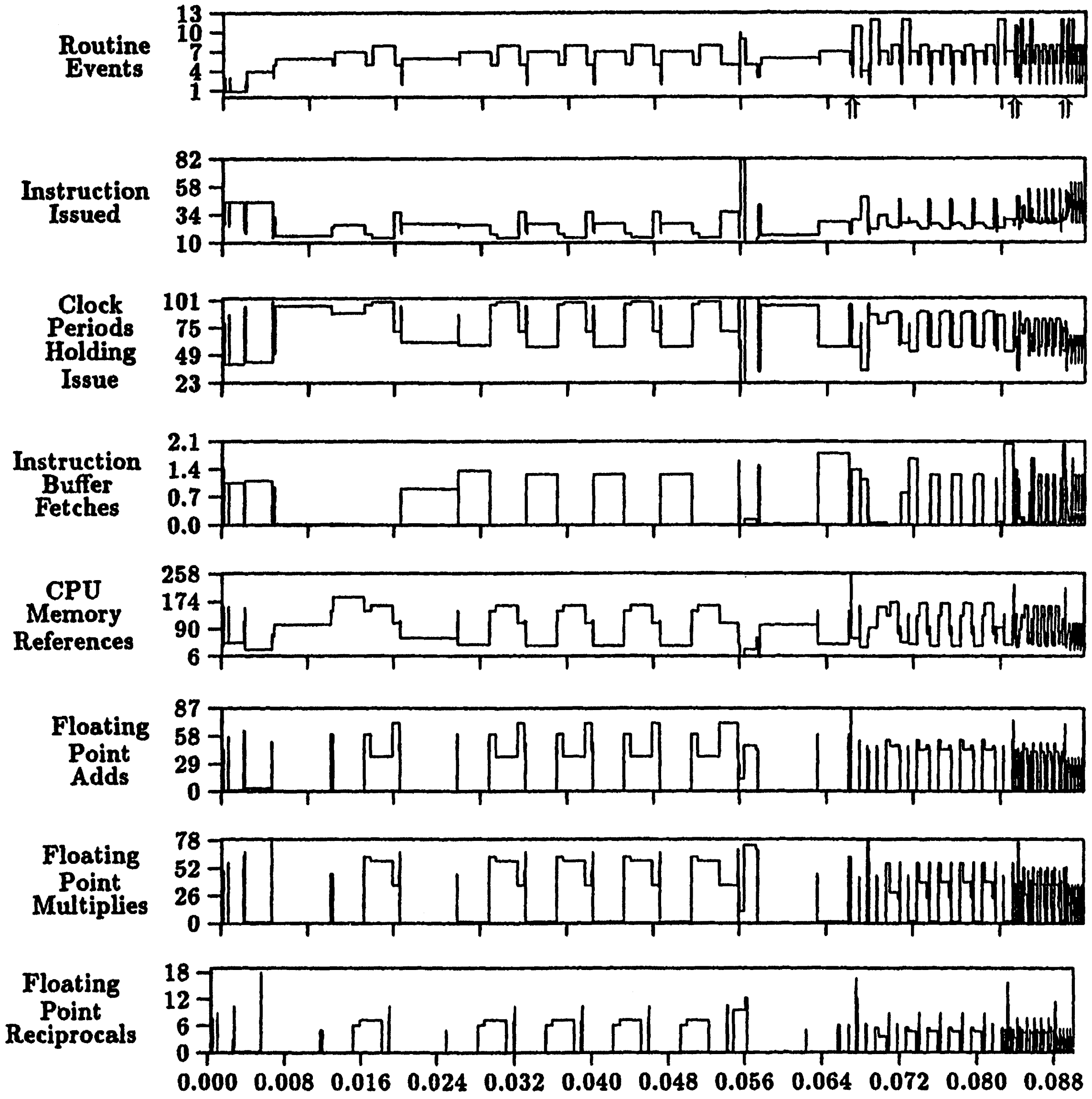


Figure 7.9: FLO52 Vector Execution - Group 0, Phase 3 (0.0897993)

To associate the hardware performance measures with the routine events at any point in time, a vertical line should be drawn at the point to intersect all graphs. The horizontal line on the event graph crossing this line identifies the currently active routine. The horizontal lines on the hardware performance graphs crossing the line show the level of the associated metric during the period the routine is active.

As is clearly evident from the three Figures, a significant amount of performance behavior detail can be observed. The perceived variations in the hardware performance curves of Figures 7.4 are more pronounced here and greater transitions in the value of certain performance metrics are seen. This is particularly true in the case of floating point adds, floating point multiplies, and CPU memory references. Moving from phase 1 to phase 3, the maximum values of most metrics increase, as expected, but so do the variations.

The increased detail of the graphs allows certain correlations with subroutine behavior to be made. The strong positive correlation between the CPU memory references and floating point operations suggests that the intervals these metrics are high correspond to periods of vector operation. This is confirmed by the counter 2 graphs (not shown) which give vector memory reference information. The routines generating this high level of vector floating point performance are EULER and PSMOO.

It is also interesting to observe additional micro-phase structure of routine execution. In Figure 7.7 there appears two sub-phases whose boundary is marked by $\Uparrow$. In Figure 7.8 there are three sub-phases and in Figure 7.9 there are four. The first sub-phase of each event graph follows the same sequence of routines as a solution to the current grid step is computed. The additional sub-phases correspond to an elongation phase that approximates from the fine grid structure the solution on the courser mesh. Because the latter major phases correspond to finer gridding, additional sub-phases are required at the end of a period to move back to the course grid; one extra sub-phase for the phase 2 period and two extra for phase 3 period. As the computation moves from the fine grid back to the course, there is a shortening of time for each successive sub-phase as well as a drop in performance because it takes less time to reach a solution (due to a reduction in required computations) and the vector calculations are less efficient because of a decrease in vector length. We can see this occurring in the performance curves.

7.6.4 Megaflop Distributions

The *Perftrace* summary statistics show hardware performance data for each procedure, averaged over the the computation lifetime. The performance curves above show the raw detail of hardware performance for every procedure transition, demonstrating the extreme variation of some performance metrics during execution. These variations can depend both on each procedure's computation and its input data. From the *TracePerf* traces, one can calculate these per-procedure statistics (as in Tables 7.12 and 7.13), plus determine variations of performance metrics per routine.

To understand the variations in measured megaflops for each procedure invocation, we generated *megaflop distribution graphs* that show the percentage of all calls made to a procedure that executed at a given megaflop rate. As an example, Figure 7.10 shows the megaflop distribution graph for the FLO52 code in both scalar and vector execution modes. Although the figures do not show the fraction of application execution time attributable to each procedure call instance, the megaflop variations are clearly visible. For some procedures, the megaflop range is small. For others, such as PSM00 and EULER, the range is quite large. As before, much of this behavior can be explained based on knowledge of the application program. The PSM00 and EULER procedures are primary components of the FLO52 grid computation, and the variations in megaflop rates reflect the changes in vector lengths across the three grids.

In addition to the producing megaflop distributions, using the *TracePerf* traces, it is possible to compare scalar and vector hardware performance for all instances of a procedure. For megaflops, this permits calculation of speedups for each procedure instantiation. In general, knowing the range of procedure's vector performance, rather than just the mean, provides greater insight into optimization potential.

7.6.5 Trace-Based Applications Modeling

For codes that have a periodic behavior, such as FLO52 , it is possible to use traced performance information to derive an abstract model of execution from event timings. The general idea is to apply signal analysis techniques to identify regular patterns in the event or performance curves that can be expressed in the form of an "execution" equation. The equation or sequence of equations would not only model the application's execution but also identify the

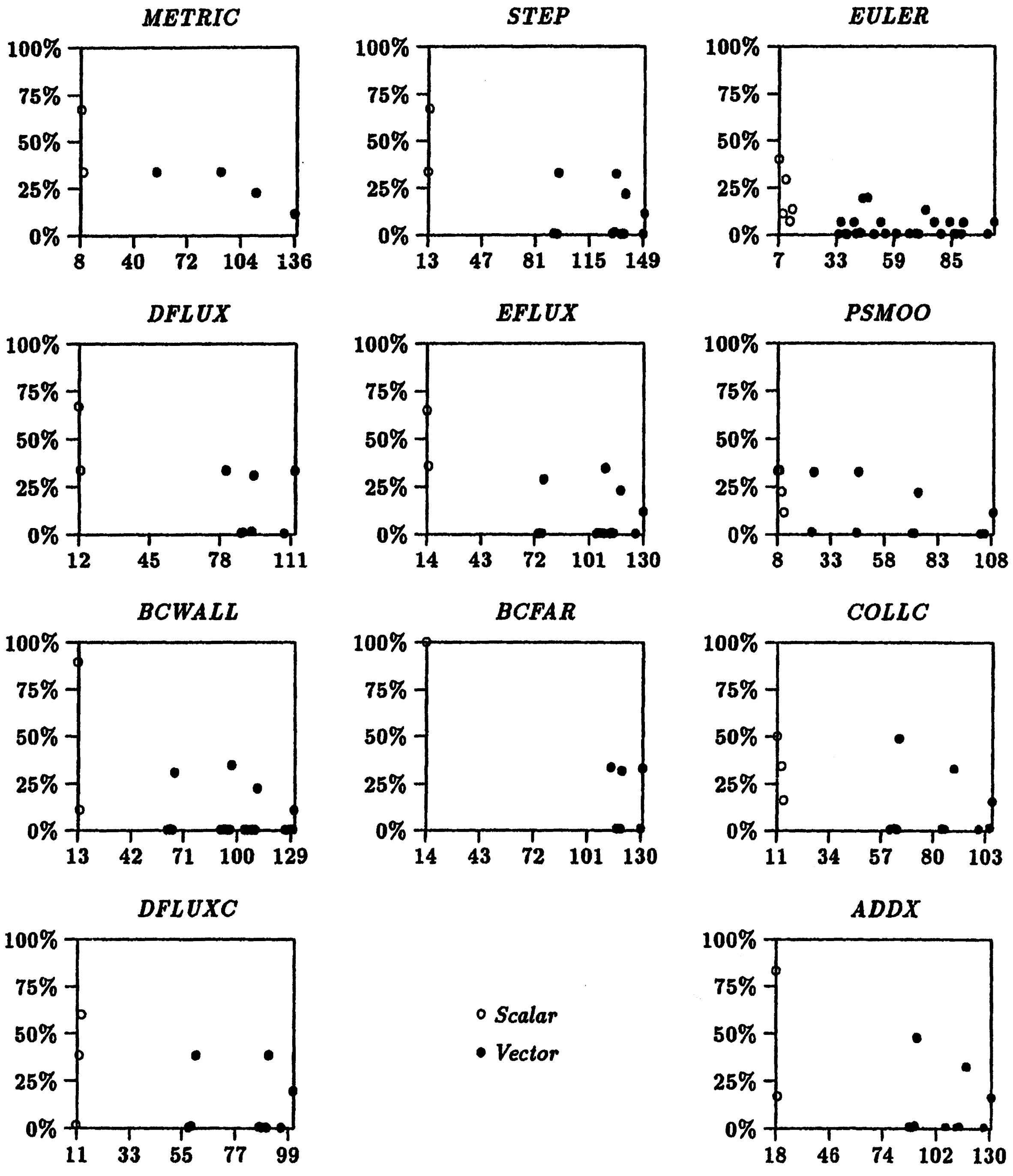


Figure 7.10: Megaflop Distribution for FLO52 Subroutines on the Cray X-MP

Term	Description	Scalar	Vector
T(init)	initialization execution time	0.27513765	0.28362232
η	number of phase periods	48	48
ρ_1	phase 1 period time	0.05249940	0.00849680
$T(1,2)$	phase 1 to phase 2 transition time	0.07645036	0.05366612
ρ_2	phase 2 period time	0.20451272	0.02751690
$T(2,3)$	phase 2 to phase 3 transition time	0.21706780	0.11549555
ρ_3	phase 3 period time	0.78509755	0.08967993
$T(term)$	termination execution time.	0.65705125	0.23356771
$T(FLO52)$	total modeled execution time	51.24699714	6.71964580
	actual	52.18852721	6.74981302
	error	-1.8%	-0.4%

Table 7.16: FLO52 Execution Model Results

critical performance measurements. Given the model, one could adjust certain coefficients to predict performance under a different set of execution conditions. It also allows a smaller set of measurements to be made from which execution behavior can be predicted.

We attempted a simple form of this more general approach in the modeling of the FLO52 code. Because of the clearly periodic behavior evident in the event graphs, we modeled the major phases of the computation by taking the execution time for one period in each phase and multiplying by the number of periods. The initialization and termination timings, plus the phase transition timings are treated separately. Our model of the FLO52 code is

$$T(FLO52) = T(init) + \eta * \rho_1 + T(1,2) + \eta * \rho_2 + T(2,3) + \eta * \rho_3 + T(term).$$

The results of applying this model to the scalar and vector execution are shown in Table 7.16.

It is surprising that the model can describe the detailed structure of the FLO52 computation yet require only one percent of the measured data to produce an accurate prediction — only one percent of the measured events were needed to calculate the values in Table 7.16. We caution against the conclusion that such a modeling approach is possible for all applications. Indeed, FLO52 has such regular behavior that it should be regarded as a special case. In general, an application's execution may be irregular, making repeated patterns difficult to identify. However, if any such patterns can be identified from the performance measurement, it might enable the application's performance behavior to be more concisely expressed and allow less detailed (and thus less intrusive) measurements to be made.

7.7 Comments

It is becoming increasingly apparent that to achieve high performance on supercomputers, the application programmer must have a better understanding of the dynamic execution characteristics of the computation and its interactions with the high-performance features of the machine. However, tools for capturing such data are not commonplace, and currently, users must instead rely on execution summary statistics provided mainly by routine-based profilers. The performance environment developed for the Cray X-MP and Cray 2 supercomputers applied several of the performance observability techniques described in earlier chapters, including timing-based perturbation analysis. The environment has been demonstrated in this chapter to provide important insights into time-dependent execution behavior and to provide greater performance detail while maintaining a high degree of accuracy.

In general, there are several potential advantages provided by dynamic tracing over conventional execution profiling. As a framework for future applications of our tracing facilities for the Cray machines, and for potential tracing implementations on other systems, we have listed some of the more important advantages below.

- A dynamic calling trace can be analyzed to show time-dependent calling relationships between program units.

- Program execution phases can be identified, enhancing the understanding of program function.

- Different modes of execution (sequential, vector, or multitasking) can be compared at a finer level to show the effects of different source code optimizations.

- Parameters for execution models of certain program structures, such as DO loops, can be derived from the trace data.

- A realistic evaluation of performance improvement potential can be made from a study of performance variability of routine execution.

- **Given access to hardware performance data, the high-performance features of the machine can be more accurately correlated with time-varying performance behavior.**
- **In theory, decisions regarding future parallelism choices (vectorization and multitasking) can be made at run-time based on previous trace-based performance knowledge of a program and its current performance behavior. This allows the system to reconfigure the parallelism of the program to better match the parallelism of the hardware.**
- **Comparisons between architectures based on specific, local information can be made. Timing and performance characteristics deemed intrinsic to the program may be used to directly evaluate the interaction of program constructs with architecture features.**
- **More appropriate selection of code characteristics for use in performance prediction model design can be made.**

Chapter 8

Conclusions and Future Work

Believe nothing, no matter where you read it,
   or who said it,
no matter if I have said it,
   unless it agrees with your own reason and
your own common sense.
— Buddha

The increasing complexity of computer systems necessitates a re-evaluation of the tools used for performance observation. This thesis proposes several methods and techniques for improving performance observability in the areas of performance measurement, performance perturbation analysis, and performance visualization. We have used prototype implementations of tools incorporating many of the ideas to study performance observability in real environments. The main conclusion of the thesis is that the requirements and approaches for improved performance observability depend significantly on context: the performance experiment being conducted, the accessibility of performance data in the computer system, the type and overhead of performance measurement, the perturbation sensitivity of the measured execution, the constraints on perturbation compensation, and the user's preference for performance data presentation.

Specifically, in this thesis, we have accomplished the following:

- We outlined capabilities required of performance instrumentation and data collection tools to support more precise performance measurements. We developed a hardware-based software event tracing facility for the Intel iPSC/2 multiprocessor to examine the practical requirements and constraints of a performance measurement system in a real computer system environment.

- We developed both timing-based and event-based models for removing performance perturbations due to instrumentation intrusion. These models are the first to be proposed that are able to approximate performance behavior from program event traces. We applied these models to both synthetic and actual testcases with highly successful results.

- We showed several applications of performance visualization techniques for different types of performance data. We proposed a general performance visualization architecture and presented results from a prototype implementation.

- We demonstrated the usefulness of new performance observability tools for the Cray X-MP and Cray 2 supercomputers in studying the performnace of the Perfect application codes. Many of the insights gained in these experiments would have been difficult to obtain using the standard performance measurement, analysis, and visualization tools for these systems.

Based on the thesis results, several avenues for future research are worthwhile. The integration of tools for performance measurement, analysis, and visualization into a performance environment will be important for their pervasive acceptance and use. The existence of performance environments across a broader set of systems (both research and commercial) will, in turn, allow more sophisticated performance benchmarking and characterization studies. Also, gaining experience with integrated performance environments will provide the needed feedback for the design of next generation performance tools. A particularly challenging proposition, in this respect, is the building of portable or standard performance environments that allow common performance experiments to be made on different computer systems using similar measurement, analysis, and visualization methods. The definition of standard performance measurements tools and interfaces, the adoption of common event trace formats, and the building of reusable visualization components have been by-products of this thesis research and are beginning to gain acceptance.

Research in performance perturbation analysis is just beginning. The perturbation models and experimental results presented in the thesis establish a foundation for future study, but need to be extended in several ways. The perturbation theory must be advanced to include a formulation of the errors incurred during perturbation analysis. Issues such as error accumulation and error propagation should have a formal basis in the models. The computing of confidence intervals both in the approximations and from multiple trial measurements must also be studied.

The event-based perturbation models should be expanded to include a larger variety of parallel execution paradigms and methods. For instance, the perturbation analysis techniques developed for a single-P, single-V type of semaphore might apply to message passing systems that use synchronous communications. However, asynchronous communications with typed messages and priority sending and receiving requires different perturbation analysis solutions. Similarly, there are many other forms of synchronization used in parallel computing that could pose unique perturbation analysis issues. A general question is whether it is possible to define a rigorous classification of synchronization techniques in terms of the perturbation analysis approach.

We also need a better understanding of perturbation effects in the context of nondeterministic execution. In many cases, the complete range of *feasible* executions will be restricted to a smaller set of *likely* executions due to the computational environment. That is, the probability distribution of feasible orderings of execution events is non-uniform. Not only is a better model of nondeterministic execution in a real environments needed, but we need to understand how instrumentation intrusion influences nondeterministic execution. There are two questions here. First, how does one determine the likely set of behaviors? Second, how is the likely set of behaviors perturbed by instrumentation? One might conjecture that the execution has been perturbed in a deterministic way, but it may lead to a narrowing, widening, or even a shifting in the set of likely executions. Even if these questions can be answered, there still remains the problem of compensating for the perturbation.

In the area of performance visualization, displaying performance data from massively parallel systems is an open research topic. The fundamental questions of what to show are the same, but the answers are sensitive to scale. Displays of processor utilization, network utilization, and load balancing over the whole system are likely to be more beneficial than showing detailed performance data for each individual node. The real-time aspects are performance visualization is another area that begs for further study. An investigation of the tradeoffs discussed in Chapter

6 in an actual system context would help evaluate present day capabilities. As processing and display technologies improve, it will also be interesting to track how the tradeoffs change.

At the heart of research in performance observability lies the goal of improving our ability to evaluate computer systems — their design, their operation, and their use. Performance observability is a means, not an end.

Bibliography

[1] M. Abrams. Design of a Measurement Instrument for Distributed Systems. Technical Report RZ 1639, IBM Research Division, Zurich Research Laboratory, October 1987.

[2] M. Abrams and A. Agrawala. Exact Performance Analysis of Two Distributed Processes with Multiple Synchronization Points. Technical Report TR-1845, University of Maryland, Department of Computer Science, February 1987.

[3] W. Abu-Sufah and A. Malony. Vector Processing on the Alliant FX/8 Multiprocessor. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 559–566, August 1986.

[4] S. Adve and M. Hill. Weak Ordering - A New Definition and Some Implications. In *Proceedings of the Seventeenth International Symposium on Computer Architecture*, pages 2–14, Seattle, Washington, May 1990.

[5] Alliant Computer Systems Corp. *Alliant FX/Series Architecture Manual*, 1986.

[6] G. Andrews and F. Schneider. Concepts and Notations for Concurrent Programming. *ACM Computing Surveys*, 15(1):3–43, March 1983.

[7] J. Andrews. A Hardware Tracing Facility for a Multiprocessing Supercomputer. Master's thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, May 1990.

[8] Z. Aral and I. Gertner. Parasight: A High-Level Debugger/Profiler Architecture for Shared-Memory Multiprocessors. Technical Report ETR 88-005, Encore Computer Corporation, 1988.

[9] N. Arenstorf and H. Jordan. Comparing Barrier Algorithms. *Parallel Computing*, 12(2):157–170, November 1989.

[10] R. Arlauskas. iPSC/2 System: A Second Generation Hypercube. In *Proceedings of the Third Conference on Hypercube Concurrent Computers and Applications, Volume I*, pages 38–42, Pasadena, CA, January 1988. Association for Computing Machinery.

[11] T. Axelrod. Effects of Synchronization Barriers on Multiprocessor Performance. *Parallel Computing*, 3(2):129–140, November 1986.

[12] U. Banerjee. *Speedup of Ordinary Programs.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1979.

[13] P. Bates and J. Wileden. Event Definition Language: An Aid to Monitoring and Debugging Complex Software Systems. In *Proceedings of the Fifteenth Hawaiin International Conference on System Sciences*, Hawaii, January 1982.

[14] P. Bates and J. Wileden. High-Level Debugging of Distributed Systems: The Behavioral Abstraction approach. *Journal of Systems and Software*, 3(4):225–264, April 1983.

[15] Gist User's Manual, 1988.

[16] T. Beck. A Hardware Histogramming Facility for a Multiprocessing Supercomputer. Master's thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1990. in preparation.

[17] Bell Laboratories, Murray Hill, New Jersey. *UNIX User's Manual*, 1979.

[18] D. Bernstein and K. So. Performance Visualization of Parallel Programs on a Shared Memory Multiprocessor System. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages II–1 – II–10, August 1989.

[19] M. Berry. The Perfect Club Benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[20] M. Berry. The Use of Matrix Visualization in Algorithmic Design. In *Computing Systems in Engineering*. Pergamon Press, Oxford, England, 1990. to appear.

[21] R. Brandis. IPPM: Interactive Parallel Program Monitor. Master's thesis, Oregon Graduate Center, August 1986.

[22] W. Brantley, K. McAuliffe, and T. Ngo. RP3 Performance Monitoring Hardware. In M. Simmons, R. Koskela, and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, pages 35–48. ACM Press, 1989.

[23] O. Brewer, J. Dongarra, and D. Sorensen. Tools to Aid in the Analysis of Memory Access Patterns for Fortran Programs. *Parallel Computing*, 9:25–35, 1988/1989.

[24] R. Brouwer and P. Banerjee. A Parallel Simulated Annealing Algorithm for Channel Routing on a Hypercube Multiprocessor. In *Proceedings of the International Conference on Computer Design*, pages 4–7, Rye Brook, NY, October 1988.

[25] M. Brown. *Algorithm Animation.* PhD thesis, Brown University, April 1987.

[26] M. Brown. Exploring Algorithms Using Balsa-II. *IEEE Computer*, 21(5):14–36, May 1988.

[27] H. Burkhart and R. Millen. Performance Measurement Tools in a Multiprocessor Environment. *IEEE Tranactions on Computers*, 38(5):725–737, May 1989.

[28] D. Carrington. Profiling Under ELXSI UNIX. *Software Practice and Experience*, 16(9):865–873, September 1983.

[29] S. Chen. Large-Scale and High-Speed Multiprocessor System for Scientific Applications: Cray X-MP-2 Series. In J. Kowalik, editor, *Proceedings of the NATO Workshop on High Speed Computations*, pages 59–67. Springer-Verlag, 1984.

[30] H. Chernoff. The Use of Faces to Represent Points in k-Dimensional Space Graphically. *Journal of the American Statistical Association 68*, pages 361–368, June 1973.

[31] W. Cleveland. *The Elements of Graphing Data.* Wadsworth, Monterey, California, 1986.

[32] W. Cleveland and M. McGill. *Dynamic Graphics For Statistics.* Wadsworth and Brooks Cole, Pacific Grove, California, 1988.

[33] P. Close. The iPSC/2 Node Architecture. In *Proceedings for the Third Conference on Hypercube Multiprocessors*, pages 43–50, Pasadena, CA, January 1988. Association for Computing Machinery.

[34] E. Coffman and P. Denning. *Operating Systems Theory.* Prentice-Hall, Englewood Cliffs, New Jersey, 1973.

[35] A. Couch. *Graphical Representations of Program Performance on Hypercube Message-Passing Multiprocessors.* PhD thesis, Tufts University, Department of Computer Science, 1988.

[36] A. Couch and D. Krumme. Projection, Pursuit, and the Triplex Tool Set for the NCUBE Multiprocessor. Technical report, Tufts University, Department of Computer Science, November 1989.

[37] Cray Research Inc. *UNICOS Performance Utilities Reference Manual*, May 1989.

[38] R. Cytron. *Compile-Time Scheduling and Optimization for Asynchronous Machines.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, October 1984.

[39] R. Cytron. Doacross: Beyond Vectorization for Multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 836–845, August 1986.

[40] J. Davies. Parallel Loop Constructs for Multiprocessors. Master's thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, May 1981.

[41] P. Denning. Working Sets Past and Present. *IEEE Transactions on Software Engineering*, 6(1):64–84, January 1980.

[42] P. Denning and J. Buzen. The Operational Analysis of Queueing Network Models. *ACM Computing Surveys*, 10(3):225–261, September 1978.

[43] J. Dongarra, O. Brewer, J. Kohl, and S. Fineberg. A Tool to Aid in the Design, Implementation, and Understanding of Matrix Algorithms for Parallel processors. *Journal of Parallel and Distributed Computing*, 9(6):185–202, June 1990.

[44] J. Dongarra and D. Sorenson. A Portable Environment for Debugging Parallel Fortran Programs. *Parallel Computing*, 5:175–186, 1987.

[45] J. Dongarra and D. Sorenson. SCHEDULE: Tools for Developing and Analysing Parallel Fortran Programs. In L. Jamieson, D. Gannon, and R. Douglas, editors, *The Characteristics of Parallel Algorithms*, pages 363–394. MIT Press, Cambridge, Massachusetts, 1987.

[46] H. Durrett. *COLOR and The Computer*. Academic Press, London, England, 1987.

[47] P. Emrath. An Operating System for the Cedar Multiprocessor. *IEEE Software*, 2(4):30–37, 1985.

[48] P. Emrath, S. Ghosh, and D. Padua. Event Synchronization Analysis for Debugging Parallel Programs. In *Proceedings of the 1989 International Conference on Parallel Processing*, August 1989.

[49] G. Estrin, D. Hopkins, B. Coggan, and S. Crocker. SNUPER COMPUTER: A Computer in Instrumentation Automaton. In *Proceedings of the 1967 Spring Joint Computer Conference*, volume 30, pages 645–656, 1967.

[50] Z. Fang, P. Yew, P. Tang, and C. Zhu. Dynamic Processor Self-Scheduling for General Parallel Nested Loops. In *Proceedings of the 1987 International Conference on Parallel Processing*, August 1987.

[51] D. Ferrari. Considerations on the Insularity of Performance Evaluation. In *IEEE Transactions on Software Engineering*, pages 678–683, June 1986.

[52] D. Ferrari, G. Serazzi, and A. Zeigner. *Measurement and Tuning of Computer Systems*. Prentice-Hall, Englewood Cliffs, New Jersey, 1981.

[53] C. Fidge. Partial Orders for Parallel Debugging. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 183–194. ACM Sigplan/Sigops, May 1988. University of Wisconsin.

[54] C. Fidge. Timestamps in Message-Passing Systems that Preserve the Partial Ordering. In *Proceedings of the 11th Australian Computer Science Conference*, University of Queensland, February 1988.

[55] L. Foulds. *Optimization Techniques: An Introduction.* Springer-Verlag, New York, New York, 1981.

[56] R. Fowler and I. Bella. The Programmer's Guide to Moviola: An Interactive Execution History Browser. Technical Report TR-CS-69, University of Rochester, Department of Computer Science, February 1989.

[57] R. Fowler, T. LeBlanc, and J. Mellor-Crummey. An Integrated Approach to Parallel Program Debugging and Performance Analysis on Large-Scale Multiprocessors. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 163–173. ACM SIGPLAN/SIGOPS, May 1988. University of Wisconsin.

[58] K. Frenkel. The Art and Science of Visualizing Data. *Communications of the ACM*, 21(2):110–121, February 1988.

[59] H. Fromm, U. Hercksen, U. Herzog, K. John, R. Klar, and W. Kleinoder. Experiences with Performance Measurement and Modeling of a Processor Array. *IEEE Tranactions on Computers*, 32(1), January 1983.

[60] S. Fuller, R. Swan, and W. Wulf. The Instrumentation of C.mmp, A Multi-Mini Processor. In *IEEE Compcon*, pages 173–176, 1973.

[61] J. Gait. A Debugger for Concurrent Programs. *Software Practice and Experience*, 15(6):539–554, 1985.

[62] J. Gait. A Probe Effect in Concurrent Programs. *Software Practice and Experience*, 16(3), March 1986.

[63] K. Gallivan, 1990. private communication.

[64] K. Gallivan, D. Gannon, W. Jalby, A. Malony, and H. Wijshoff. Behavioral Characterization of Multiprocessor Memory Systems: A Case Study. In *Proceedings of the 1989 ACM*

SIGMETRICS Conference on Measurement and Modeling of Computer Systems, pages 79–88, Berkeley, California, May 1989.

[65] K. Gallivan, G. Hung, and R. Saleh. Parallel circuit simulation using non-linear relaxation. Technical Report CSRD-1023, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, July 1990.

[66] K. Gallivan, W. Jalby, A. Malony, and P. Yew. Performance Analysis on the Cedar System. In J. Martin, editor, *Performance Evaluation of Supercomputers*, pages 109–142. North-Holland, Amsterdam, The Netherlands, 1988.

[67] K. Gallivan, W. Jalby, U. Meier, and A. Sameh. The Impact of Hierarchical Memory Systems on Linear Algebra Algorithm Design. *International Journal of Supercomputer Applications*, 2(1), 1987.

[68] D. Gannon, W. Jalby, and K. Gallivan. Strategies for Cache and Local Memory Management by Global Program Transformation. In *Proceedings of the 1987 International Conference on Supercomputing*, Athens, Greece, 1987. Association for Computing Machinery.

[69] H. Gantt. Organizing for Work. *Industrial Management*, 58:89–93, August 1919.

[70] M. Gardner. Mathematical Games. *Scientific American*, pages 120–123, October 1970.

[71] E. Gehringer, D. Siewiorek, and Z. Segall. *Parallel Processing: The CM* Experience.* Digital Press, 1987.

[72] S. Graham, P. Kessler, and M. McKusick. gprof: A Call Graph Execution Profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction*, pages 120–126, Boston, MA, June 1982. Association for Computing Machinery.

[73] S. Graham, P. Kessler, and M. McKusik. An Execution Profiler for Modular Programs. *Software Practice and Experience*, 13:671–685, 1983.

[74] F. Gregoretti and Z. Segall. Programming for Observability Support in a Parallel Programming Environment. Technical Report CMU-CS-85-176, Carnegie-Mellon University, Department of Computer Science, November 1985.

[75] D. Grunwald and D. Reed. Networks for Parallel Processors: Measurements and Prognostications. In *Proceedings for the Third Conference on Hypercube Multiprocessors*, Pasadena, California, January 1988.

[76] V. Guarna, D. Gannon, D. Jablonowski, A. Malony, and Y. Gaur. Faust: An Integrated Environment for Parallel Programming. *IEEE Software*, 6(4):20–27, July 1989.

[77] M. Guzzi. Multitasking Run-Time Systems for the Cedar Multiprocessor. Master's thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1986.

[78] M. Guzzi. Cedar Fortran Reference Manual. Technical Report CS-601, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 1987.

[79] D. Haban and D. Wybranietz. A Hybrid Monitor for Behavior and Performance Analysis of Distributed Systems. *IEEE Transactions on Software Engineering*, 16(2):197–211, February 1990.

[80] J. Hack. On the Promise of General-Purpose Parallel Computing. *Parallel Computing*, 10(3):261–275, 1989.

[81] P. Harter, D. Heimbigner, and R. King. IDD: An Interactive Distributed Debugger. In *Proceedings of the Fifth International Conference on Distributed Programs*, pages 498–506, 1985.

[82] V. Hasse. Real-Time Behavior of Programs. *IEEE Transactions on Software Engineering*, 7(9):454–501, September 1981.

[83] J. Hayes, T. Mudge, Q. Scott, S. Colley, and J. Palmer. A Microprocessor-Based Hypercube Supercomputer. *IEEE Micro*, 6(5):6–17, October 1986.

[84] P. Heidelberger and K. Trivedi. Queueing Network Models for Parallel Processing with Asynchronous Tasks. *IEEE Tranactions on Computers*, 31(11):1099–1109, November 1982.

[85] P. Heidelberger and K. Trivedi. Analytic Queueing Models for Programs with Internal Concurrency. *IEEE Tranactions on Computers*, 32(1):73–82, January 1983.

[86] D. Helmbold and D. Bryan. Design of Run Time Monitors for Concurrent Programs. Technical Report CSL-TR-89-395, Stanford University, October 1989.

[87] J. Hoeflinger, 1990. private communication.

[88] R. Hofmann, R. Klar, N. Luttenberger, B. Mohr, and G. Werner. An Approach to Monitoring and Modeling of Multiprocessor and Multicomputer Systems. In *Proceedings of IFIP Working Group 7.3 International Seminar on Performance of Distributed and Parallel Systems*, Kyoto, Japan, December 1988.

[89] R. Hon. A Simple Trace Interchange Format. Apple Computer, Inc., May 1989.

[90] A. Hough and J. Cuny. Belvedere: Prototype of a Pattern-Oriented Debugger for Highly Parallel Computation. In *Proceedings of the 1987 International Conference on Parallel Processing*, pages 735–738, August 1987.

[91] IEEE Software, July 1989. special issue on Parallel Programming, 6(4).

[92] J. Joyce, G. Lomow, K. Slind, and B. Unger. Monitoring Distributed Systems. *ACM Transactions on Computer Systems*, 5(2):121–150, May 1987.

[93] H. Katseff. The Software Oscilloscope: A Real-Time Execution Monitor for Multiprocessor Applications. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 316–318. ACM Sigplan/Sigops, May 1988. University of Wisconsin.

[94] T. Kerola and H. Schwetman. Monit: A Performance Monitoring Tool for Parallel and Pseudo-Parallel Programs. In *Proceedings of the 1987 ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1987.

[95] K. Kolence and P. Kiviat. Software Unit Profiles and Kiviat Figures. *ACM SIGMETRICS, Performance Evaluation Review*, 2(3):2–12, September 1973.

[96] C. Kruskal and A. Weiss. Allocating Independent Subtasks on Parallel Processors. *IEEE Transactions on Software Engineering*, 11(10), October 1985.

[97] D. Kuck. *The Structure of Computers and Computations*. Wiley, New York, 1978.

[98] D. Kuck. The Effects of Program Restructuring, Algorithm Change, and Architecture Choice on Program Performance. In *Proceedings of the 1984 International Conference on Parallel Processing*, pages 129–138, August 1984.

[99] D. Kuck, E. Davidson, D. Lawrie, and A. Sameh. Parallel Supercomputing Today and the Cedar Approach. *Science*, 231:967–974, February 1986.

[100] D. Kuck, R. Kuhn, D. Padua, B. Leasure, and M. Wolfe. Dependence Graphs and Compiler Optimizations. In *ACM Symposium on the Principles of Programming Languages*, July 1981.

[101] D. Kuck, D. Lawrie, W. Jalby, P. Yew, A. Malony, and A. Sameh. Methodology for Performance Evaluation for High Performance Computer Systems. Technical Report CS-725, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 1987. proposal to the National Science Foundation.

[102] D. Kuck and A. Sameh. A Supercomputing Performance Evaluation Plan. In *Proceedings of the 1987 International Conference on Supercomputing*, Athens, Greece, June 1987.

[103] M. Kumar. Measuring Parallelism in Computation-Intensive Scientific/Engineering Applications. *IEEE Tranactions on Computers*, 37(9):1088–1098, September 1988.

[104] L. Lamport. Time, Clocks, and the Ordering of Events in a Distributed System. *Communications of the ACM*, 21(7):558–565, July 1978.

[105] L. Lamport. How to Make a Multiprocessor Computer that Correctly Executes Multiprocess Programs. *IEEE Tranactions on Computers*, 28(9):690–691, September 1979.

[106] D. Lane. A Model of Time Dependent Behavior in Concurrent Software Systems. Technical Report TR-87-28, University of California, Irvine, Department of Information and Computer Science, November 1987.

[107] J. Larson. Cray X-MP Hardware Performance Monitor. *Cray Channels*, 1985.

[108] J. Larson and R. Lutz. Perftrace User Guide. Technical report, Cray Research Inc., August 1985.

[109] D. Lavery. The Design of a Hardware Performance Monitor for the Cedar Supercomputer. Master's thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, May 1989.

[110] T. LeBlanc and J. Mellor-Crummey. Debugging Parallel Programs with Instant Replay. *IEEE Tranactions on Computers*, 36(4):471–482, April 1987.

[111] T. LeBlanc, J. Mellor-Crummey, and R. Fowler. Analyzing Parallel Program Executions Using Multiple Views. *Journal of Parallel and Distributed Computing*, 9(6):203–217, June 1990.

[112] C. LeDoux and D. Parker. Saving Traces for Ada Debugging. In *Proceedings of the SIGAda International Ada Conference*, pages 1–12, 1985.

[113] T. Lehr. *Comensating for Perturbation by Software Performance Monitors in Asynchronous Computations.* PhD thesis, Carnegie-Mellon University, Department of Computer Science, April 1990.

[114] T. Lehr, D. Black, Z. Segall, and D. Vrsalovic. Visualizing Context-Switches of Parallel Programs Using PIE and the Mach Kernel Monitor. In *Proceedings of the 1990 International Conference on Parallel Processing*, August 1990. to appear.

[115] T. Lehr, Z. Segall, D. Vrsalovic, E. Caplan, A. Chung, and C. Fineman. Visualizing Performance Debugging. *IEEE Computer*, 22(10):38–51, October 1989.

[116] N. Lynch and M. Fisher. Semantics of Concurrent Computations. *Theoretical Computer Science*, 13(1):17–43, January 1981.

[117] V. Mak. Performance Prediction of Concurrent Systems. Technical Report CSL-TR-87-344, Stanford University, December 1987.

[118] A. Malony. Virtual High-Resolution Processing Timing. Technical Report CS-616, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 1986.

[119] A. Malony. Program Tracing in Cedar. Technical Report CS-660, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 1987.

[120] A. Malony. Multiprocessor Instrumentation: Approaches for Cedar. In M. Simmons, R. Koskela, and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, pages 1–33. ACM Press, 1989.

[121] A. Malony. JED: Just an Event Display. In M. Simmons, R. Koskela, and I. Bucher, editors, *Parallel Computer Systems: Performance Instrumentation and Visualization*. ACM Press, 1990. to be published.

[122] A. Malony, J. Larson, and D. Reed. Tracing Application Program Execution on the Cray X-MP and Cray 2. In *Proceedings of the 1990 Supercomputing Conference*, November 1990. to appear.

[123] A. Malony and K. Nichols. Standards in Performance Instrumentation and Visualization for Parallel Computer Systems: Working Group Summary. In M. Simmons, R. Koskela, and I. Bucher, editors, *Parallel Computer Systems: Performance Instrumentation and Visualization*. ACM Press, 1990. to be published.

[124] A. Malony and D. Reed. Visualizing Parallel Computer System Performance. In M. Simmons, R. Koskela, and I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, pages 59–90. ACM Press, 1989.

[125] A. Malony and D. Reed. A Hardware-Based Performance Monitor for the Intel iPSC/2 Hypercube. In *Proceedings of the 1990 International Conference on Supercomputing*, pages 213–226, Amsterdam, The Netherlands, 1990. Association for Computing Machinery.

[126] A. Malony, D. Reed, J. Arendt, R. Aydt, D. Grabas, and B. Totty. An Integrated Performance Data Collection Analysis, and Visualization System. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989. Association for Computing Machinery.

[127] A. Malony, D. Reed, and D. Rudolph. Integrating Performance Data Collection, Analysis, and Visualization. In M. Simmons, R. Koskela, and I. Bucher, editors, *Parallel Computer Systems: Performance Instrumentation and Visualization*. ACM Press, 1990. to be published.

[128] A. Malony, D. Reed, and H. Wijshoff. Performance Measurement Intrusion and Perturbation Analysis. Technical Report CSRD-923, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, October 1989. submitted for publication to the IEEE Transactions on Parallel and Distributed Computing.

[129] J. Martin, editor. *Performance Evaluation of Supercomputers.* North-Holland, Amsterdam, The Netherlands, 1988.

[130] C. McDowell and D. Helmbold. Debugging Concurrent Programs. *ACM Computing Surveys*, 21(4), December 1989.

[131] F. McMahon. The Livermore Fortran Kernels: A Computer Test of the Numerical Performance Range. Technical Report UCRL-53745, Lawrence Livermore National Laboratory, December 1986.

[132] U. Meier and R. Eigenmann. Parallelization and Performance of Conjugate Gradient Algorithms on the Cedar Hierarchical-Memory Multiprocessor. University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 1990.

[133] J. Mellor-Crummey. *Debugging and Analysis of Large-Scale Parallel Programs.* PhD thesis, University of Rochester, Department of Computer Science, September 1989.

[134] S. Midkiff and D. Padua. Compiler Algorithms for Synchronization. *IEEE Transactions on Computers*, 36(12):1485–1495, December 1987.

[135] B. Miller. DPM: A Measurement System for Distributed Programs. *IEEE Tranactions on Computers*, 37(2), February 1988.

[136] B. Miller and J. Choi. A Mechanism for Efficient Debugging of Parallel Programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 141–150. ACM Sigplan/Sigops, May 1988. University of Wisconsin.

[137] B. Miller, M. Clark, S. Kierstead, and S. Lim. IPS-2: The Second Generation of a Parallel Program Measurement System. Technical Report CS-783, University of Wisconsin at Madison, Department of Computer Science, August 1988.

[138] A. Mink and R. Carpenter. A VLSI Chip Set for a Multiprocessor Performance Measurement System. In M. Simmons, R. Koskela, and I. Bucher, editors, *Parallel Computer Systems: Performance Instrumentation and Visualization.* ACM Press, 1990. to be published.

[139] A. Mink, J. Draper, J Roberts, and R. Carpenter. Hardware Assisted Multiprocessord Performance Measurements. In *Proceedings of the 12th IFIP Working Group 7.3 International Symposium on Performance: Performance 87*, pages 151–168, Brussels, Belgium, December 1987.

[140] A. Mink and G. Nacht. Performance Measurement of a Shared-Memory Multiprocessor using Hardware Instrumentation. In *Proceedings of the Twenty-Second Hawaii International Conference on System Sciences*, 1989.

[141] M. Model. *Monitoring System Behavior in a Complex Computational Environment.* PhD thesis, Stanford University, January 1978.

[142] J. Mohan. *Performance of Parallel Programs.* PhD thesis, Carnegie-Mellon University, Department of Computer Science, July 1984.

[143] M. Morris. Kiviat Graphs — Conventions and Figures of Merit. *Performance Evaluation Review*, 3(3):2–8, October 1974.

[144] G. Nacht and A. Mink. Recommended Instrumentation Approaches for a Shared-Memory Multiprocessor. Technical Report NB-SIR 87-3663, National Bureau of Standards, Institute for Computer Sciences and Technology, October 1987.

[145] R. Netzer and B. Miller. Detecting Data Races in Parallel Program Executions. Technical Report CS-894, University of Wisconsin at Madison, Department of Computer Science, November 1989.

[146] R. Netzer and B. Miller. On the Complexity of Event Ordering for Shared Memory Parallel Program Executions. Technical Report CS-908, University of Wisconsin at Madison, Department of Computer Science, January 1990.

[147] K. Nichols and J. Edmark. Modeling Multicomputer Systems with PARET. *IEEE Computer*, 21(5):39–48, May 1988.

[148] G. Nutt. Tutorial: Computer System Monitors. *IEEE Computer*, pages 51–61, November 1975.

[149] R. Olson, B. Kumar, and L. Shar. Messages and Multiprocessing in the ELXSI 6400. In *Proceedings of the IEEE Spring COMPCON*, pages 21–24, 1983.

[150] R. Perron and C. Mundie. The Architecture of the Alliant FX/8 Computer. In *Spring COMPCON '86*, pages 390–393, March 1986.

[151] G. Pfister and V. Norton. Hot Spot Contention and Combining in Multistage Interconnection Networks. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 790–797, August 1985.

[152] L. Pointer. Perfect: Performance Evaluation for Cost-Effective Transformations - report 2. Technical Report CSRD No. 964, University of Illinois at Urbana-Champaign, Center for Supercomputing Research and Development, 1990.

[153] C. Polychronopoulos. *On Program Restructuring, Scheduling, and Communication for Parallel Processor Systems.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1986.

[154] C. Polychronopoulos and D. Kuck. Guided Self-Scheduling: A Practical Scheduling Scheme for Parallel Supercomputers. *IEEE Tranactions on Computers*, 36(12), 1987. Special Issue on Supercomputing.

[155] C. Ponder and R. Fateman. Inaccuracies in Program Profiling. *Software Practice and Experience*, 18(5):459–467, May 1988.

[156] V. Pratt. Modeling Concurrency with Partial Orders. *International Journal of Parallel Programming*, 15(1), 1986.

[157] H. Raveche, D. Lawrie, and A. Despain. A National Computing Initiative, The Agenda for Leadership. Society for Industrial and Applied Mathematics, 1987. Report of the Panel on Research Issues in Large-Scale Computational Science and Engineering.

[158] D. Reed and R. Fujimoto. *Multicomputer Networks: Message-Based Parallel Processing.* MIT Press, 1987.

[159] D. Reed and A. Malony. Integrated Performance Environments for Parallel Systems. proposal to the National Science Foundation, December 1989.

[160] D. Reed and A. Malony. Integrated Performance Environments for Parallel Systems. proposal to DARPA, May 1990.

[161] D. Reed and D. Rudolph. The Intel iPSC/2: An Approach to Performance Instrumentation. *International Journal of High Speed Computing*, 1990.

[162] M. Reilly. *A Hybrid Performance Monitor for Parallel Programs.* Academic Press, 1990. Ph.D. dissertation, Carnegie-Mellon University, Department of Electrical and Computer Engineering.

[163] D. Ritchie. The Evolution of the UNIX System. *Bell Laboratories Technical Journal*, 63(8), October 1984. Part 2.

[164] J. Roberts, J. Antonishek, and A. Mink. Hybrid Performance Measurement Instrumentation for Loosely-Coupled MIMD Architectures. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[165] D. Rudolph. Performance Instrumentation for the Intel iPSC/2. Master's thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, July 1989.

[166] D. Rudolph and D. Reed. CRYSTAL: Operating System Instrumentation for the Intel iPSC/2. In *Proceedings of the Fourth Conference on Hypercube Concurrent Computers and Applications*, Monterey, CA, March 1989.

[167] J. Saltzer and J. Gintell. The Instrumentation of Multics. *Communications of the ACM*, 13(8):495–500, August 1970.

[168] R. Scheifler and J. Gettys. The X Window System. *ACM Transactions on Graphics*, 5(2):79–109, April 1986.

[169] M. Seager, S. Campbell, S. Sikora, R. Strout, and M. Zosel. Graphical Multiprocessing Analysis tool (GMAT). Technical report, Lawrence Livermore National Laboratory, Computing and Mathematics Research Division, January 1988.

[170] Z. Segall and L. Rudolph. PIE: A Programming and Instrumentation Environment for Parallel Processiong. *IEEE Software*, 2(6):22–37, November 1985.

[171] Z. Segall, A. Singh, R. Snodgrass, A. Jones, and D. Siewiorek. An Integrated Instrumentation Environment. *IEEE Tranactions on Computers*, 32(1), January 1983.

[172] S. Sharma, A. Malony, M. Berry, and P. Sharma. Run-Time Monitoring of Concurrent Programs on the Cedar Multiprocessor. In *Proceedings of the 1990 Supercomputing Conference*, November 1990. to appear.

[173] D. Shasha and M. Snir. Efficient and Correct Execution of Parallel Programs that Share Memory. *ACM Transactions on Programming Languages and Systems*, 10(2):282–312, April 1988.

[174] A. Shaw. Reasoning About Time in Higher-Level Language Software. *IEEE Transactions on Software Engineering*, 15(7):875–889, July 1989.

[175] S. Sherman, F. Baskett, and J. Browne. Trace-Driven Modeling and Analysis of CPU Scheduling in a Multiprogramming System. *Communications of the ACM*, 15(12):1063–1069, December 1972.

[176] M. Simmons, R. Koskela, and I. Bucher, editors. *Instrumentation for Future Parallel Computing Systems*. ACM Press, 1989.

[177] M. Simmons, R. Koskela, and I. Bucher, editors. *Parallel Computer Systems: Performance Instrumentation and Visualization*. ACM Press, 1990.

[178] A. Smith. Cache Memories. *ACM Computing Surveys*, 14(3):473–530, September 1982.

[179] J. Smith. A Simulation Study of the Cray X-MP Memory System. *IEEE Tranactions on Computers*, 35(7):613–622, July 1986.

[180] R. Snodgrass. *Monitoring Distributed Systems: A Relational Approach*. PhD thesis, Carnegie-Mellon University, Department of Computer Science, December 1982.

[181] R. Snodgrass. A Relational Approach to Monitoring Complex Systems. *ACM Transactions on Computer Systems*, 6(2):157–196, May 1988.

[182] L. Snyder. Parallel Programming and the Poker Programming Environment. *IEEE Computer*, 17(7):27–36, July 1984.

[183] K. So, F. Darema, D. George, A. Norton, and G. Pfister. PSIMUL – A System for Parallel Simulation of Parallel Systems. In J. Martin, editor, *Performance Evaluation of Supercomputers*, pages 187–213. North-Holland, Amsterdam, The Netherlands, 1988.

[184] D. Socha, M. Bailey, and D. Notkin. Voyeur: Graphical Views of Parallel Programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 206–215. ACM Sigplan/Sigops, May 1988. University of Wisconsin.

[185] J. Stone. Visualizing Concurrent Processes. Technical report, IBM T.J. Watson Research Center, August 1987.

[186] J. Stone. A Graphical Representation of Concurrent Processes. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 226–235. ACM Sigplan/Sigops, May 1988. University of Wisconsin.

[187] C. Stunkel and D. Reed. Hypercube Implementation of the Simplex Algorithm. In *Proceedings for the Third Conference on Hypercube Multiprocessors.* Society for Industrial and Applied Mathematics, 1988.

[188] H. Su and P. Yew. On Data Synchronization for Multiprocessors. In *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pages 416–422, Jerusalem, Israel, May 1989.

[189] P. Tang. *Self-Scheduling, Data Synchronization, and Program Transformation for Multiprocessor Systems.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, January 1989.

[190] A. Thomasian and P. Bay. Queueing Network Models for Parallel Processing of Task Systems. In *Proceedings of the 1983 International Conference on Parallel Processing*, pages 421–428, August 1983.

[191] A. Thomasian and P. Bay. Analytic Queueing Network Models for Parallel Processing of Task Systems. *IEEE Tranactions on Computers*, 35(12):1045–1054, December 1986.

[192] R. Trammel. The Big Picture: Visualizing System Behavior in Real Time. In *Proceedings of the 1990 USENIX Summer Conference*, pages 257–266, June 1990.

[193] A. Tuchman and M. Berry. Matrix Visualization in the Design of Numerical Algorithms. *ORSA Journal on Computing*, 2(1):84–92, Winter 1990.

[194] E. Tufte. *The Visual Display of Quantitative Information.* Graphics Press, Cheshire, Connecticut, 1983.

[195] J. Tukey. *Exploratory Data Analysis.* Addison-Wesley Publishing Company, Reading, Massachusetts, 1977.

[196] J. Ullman. *Principles of Database Systems.* Computer Science Press, Potomac, Maryland, 1980.

[197] M. Wolfe. *Optimizing Supercompilers for Supercomputers.* PhD thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, 1982.

[198] C. Yang and B. Miller. Critical Path Analysis for the Execution of Parallel and Distributed Programs. In *Proceedings of the 8th International Conference on Distributed Computing Systems*, pages 366–375, June 1988.

[199] M. Zimmermann, F. Perrenoud, and A. Schiper. Graphical Animation of Concurrent Programs. In *Proceedings of the Workshop on Parallel and Distributed Debugging*, pages 342–344. ACM Sigplan/Sigops, May 1988. University of Wisconsin.

[200] M. Zimmermann, F. Perrenoud, and A. Schiper. Understanding Concurrent Programming through Program Animation. In *Proceedings of the Nineteenth ACM SIGCSE Conference*, February 1988.

Vita

Allen Davis Malony was born in Decatur, Alabama on February 11, 1958. Although his formative years were spent in Tennessee, it was easy to become accustomed to the sunny skies and warm beaches of southern California where Mr. Malony attended Pasadena High School. He took programming classes at Caltech and worked at JPL in the summers while still in high school.

Mr. Malony attended the University of California at Berkeley his Freshman and Sophmore undergraduate years. He then transferred to the University of California at Los Angeles, where he graduated Summa Cum Laude with a B.S. in Mathematics/Computer Science in 1980. He received the Computer Science department's senior prize and was elected to the Phi Beta Kappa honor society. He also received an M.S. in Computer Science from UCLA under the supervision of Dr. Stott Parker. His thesis was on the properties of regular interconnection networks for multiprocessor computer systems.

After receiving his M.S. degree, Mr. Malony went to work for Hewlett-Packard Laboratories where he was a principal designer of a high-speed computer network architecture. Among his many experiences while at HP, Mr. Malony cut his teeth on digital logic design. This new-found ability came back to haunt him during the Ph.D. thesis work.

Mr. Malony left Hewlett-Packard after four years to pursue a doctorate degree at the University of Illinois, Urbana-Champaign. He joined the Center for Supercomputing Research and Development soon after as a senior software engineer. At CSRD, Mr. Malony leads efforts in supercomputing performance measurement and analysis.

Mr. Malony completed his Doctoral dissertation in September of 1990 and continues to work at the Center for Supercomputing Research and Development.