COMPENSATING FOR PERTURBATION BY SOFTWARE PERFORMANCE MONITORS IN ASYNCHRONOUS COMPUTATIONS

PRESENTED BY
THEODORE FRANKLIN LEHR

ACCEPTED BY THE DEPARTMENT OF
ECE

MAJOR PROFESSOR

DEPARTMENT HEAD

APPROVED BY THE COLLEGE COUNCIL

PAUL WYMAN

DEAN
Compensating for Perturbation by Software Performance Monitors in Asynchronous Computations

April, 1990
Ted Lehr
Department of Electrical and Computer Engineering
Carnegie Mellon University
Pittsburgh PA 15213

Abstract

Designing performance efficient programs for parallel computing systems can benefit from running experiments for retrieving information about how computations execute. Retrieving this information requires some kind of data generation/collection facility that monitors computations as the computations execute. Unfortunately, monitoring an application almost always perturbs its execution profile.

The general problem is not the elimination of perturbations, but the detection, measurement and compensation of them. The perturbations considered here are those due to the presence of software sensors and independent monitoring threads that compete for processor resources. The best case results show that the difference between compensation estimates of the average maximum, minimum and mean executions are within 1% of their real values. The results of the work have potential applications for general research on performance prediction.

This research was sponsored in part by the Defense Advanced Research Projects Agency (DOD), Arpa Order No. 4864, monitored by the Space and Naval Warfare Systems Command under contract N00039-85-C-1034, and in part by the National Science Foundation, Grant No. CCR-86-02-143.

The views and conclusions contained in this document are those of the authors and and should not be interpreted as representing the official policies, either expressed or implied, of the Defense Advanced Research Projects Agency, the National Science Foundation or the U.S. Government.
Thesis Summary

Designing performance efficient programs for parallel computing systems can benefit from the use of "computational laboratories" with which programmers may design experiments to test the behavior of their applications and systems. Running experiments for retrieving information about applications requires some kind of data generation/collection facility that monitors applications and the host systems as the applications execute. Unfortunately, monitoring a application almost always perturbs the application's execution profile.

To address the problem of monitor perturbations, one may attempt to eliminate perturbations. Since the monitoring system considered in this thesis is portable, however, perturbation cannot be eliminated but can only be minimized through thoughtful coding of the monitoring constructs. Given that perturbation cannot be eliminated, this thesis examines an alterative solution consisting of the detection, measurement and compensation of perturbations.

The thesis presents three compensation modes. The first works well for sequential applications but presents semantic problems when applied to parallel applications. The latter two modes work well for asynchronous applications and applications which only synchronize on thread creation and termination. All three modes are restricted to compensating application executing on sequential or tightly-coupled machines with uniform access time memory topologies. The best case results show that, with 99% confidence, the difference between compensation estimate of the average mean (ie \( \mu \)) and the real average mean is 1.2 ± 0.9%.

With 99% confidence, it can be claimed that compensated mean minimum execution times are within 0.1 ± 0.9% of the real times. With 99% confidence, compensation estimates of the mean maximum execution times are within 0.4 ± 1.2% of the real times.

The contributions of this research are:

- The engineering of key features of a portable, parallel programmer's performance environment (PIE).
- The engineering and implementation of the monitoring systems integrated into Mach and PIE.
- The theoretical basis and engineering for compensation for monitor usage of computation resources.
- A demonstration that not only is an integrated computational laboratory feasible, but that such system could be fully implemented and evaluated.
- Providing assistance in debugging and understanding the performance of the Mach scheduler.
- The illumination of valuable ways to display performance data.
- The demonstration that software monitoring with compensation is a powerful and inexpensive alternative to more expensive but less intrusive hardware supported monitoring systems.
Although the focus of this research is the treatment of monitor perturbation problems, the proposed solutions have potential applications in a number of other areas such as determination of static scheduling policies, performance effects of increasing or decreasing the number of threads.
Table of Contents

1. Experimenting with Parallelism
   1.1. Running Computation Experiments: Perturbing the Data
   1.2. The Kind of Monitoring to be Considered
      1.2.1. What Can be Monitored
      1.2.2. The Danger of Perturbation
   1.3. The Kind of Systems to be Considered
   1.4. Recovering the Real Behavior of a Computation
      1.4.1. What Perturbations are Detected and Measured
      1.4.2. Compensating for Perturbations
      1.4.3. Confidence in Compensation
   1.5. Structure of this Thesis

2. Related Research
   2.1. Programming Environments
   2.2. Monitoring and Perturbation

3. The PIE System
   3.1. The Organization of PIE
   3.2. Setting Up an Experiment in PIE
      3.2.1. A Problem Application
      3.2.2. Setting up the Experiment
      3.2.3. Examining the Results
      3.2.4. Fixing the Application
      3.2.5. But Is the Environment Hiding Something?

4. Implementing a Monitor
   4.1. Basic Monitor Terms and Concepts
      4.1.1. Computation
      4.1.2. Events
         4.1.2.1. Data-driven events
         4.1.2.2. Control-driven events
      4.1.3. Sensors
      4.1.4. Monitor
   4.2. Attributes of a PIE-like monitor
      4.2.1. Inline code
      4.2.2. Monitoring Threads
      4.2.3. Multi-level Monitoring

5. The Monitoring Laboratory
   5.1. History of the Monitoring Laboratory
   5.2. User and Run-Time Monitoring
5.3. MKM: The Mach Kernel Monitor
   5.3.1. Mach
   5.3.2. General Architecture of the Mach Kernel Monitor
   5.3.3. Monitor System Calls
   5.4. Interpreting and Presenting the Data

6. Monitoring Examples: With Notes on Perturbation
   6.1. A Scheduler's Evolution: A Case History
       6.1.1. Spawning: The Advantages of the Prototype Lineage
       6.1.2. Time Hogs: Correcting the Problems of the Prototype Lineage
       6.1.3. Perturbation Considerations
   6.2. Saturating a Multiprocessor
       6.2.1. Background Information
           6.2.1.1. The Scheduler
           6.2.1.2. The Run-time
       6.2.2. Execution I Begins Spawning
       6.2.3. Execution II Begins Spawning
       6.2.4. Processor Assignments
       6.2.5. Improving the Execution of the Application
       6.2.6. Monitor Perturbation
   6.3. Gang Scheduled Computations
       6.3.1. Perfect Parallelism
       6.3.2. Incomplete Parallelism: When Threads > Processors
   6.4. When the Computation and Monitor "Resonate"
       6.4.1. Buffer Flushes and Context-Switches
       6.4.2. Who perturbed Whom?
   6.5. Accuracy Perturbation under High Resolution
   6.6. Summary of Perturbation

7. Software Monitor Perturbation: The Causes and Effects
   7.1. Orders of Perturbation
       7.1.1. First Order
       7.1.2. Second Order
       7.1.3. Third Order
   7.2. Recovering the Computation
       7.2.1. Determining the nature of the perturbation
       7.2.2. Monitor Trade-offs: Perturbation, Cost and Portability
   7.2.3. Compensating for Perturbations
   7.2.4. Confidence in Compensation

8. The Work
   8.1. Perturbation of the Sequential Computation
       8.1.1. First Order Perturbation by User Sensors
       8.1.2. Kernel Sensor Perturbation
           8.1.2.1. Perturbation of the Computation
           8.1.2.2. Overhead of Reading Event Information
           8.1.2.3. Sensor Perturbation of Scheduling
   8.2. Compensating for Perturbation of Sequential Computations
       8.2.1. Compensation by Trimming Trees
       8.2.2. The Accuracy of Compensation
   8.3. Some Problems with Trimming Trees of Parallel Computations
       8.3.1. Who Gets Moved Where?
8.3.2. And What Would It Mean When You’re Done
8.4. Computation Classes
8.5. Time Compensation Using Finite Elements
  8.5.1. Algorithm I: Processors versus Threads
  8.5.2. Algorithm II: Runnable Threads versus Running Threads
  8.5.3. Algorithm III: Processors, Runnable Threads and Running Threads
  8.5.4. Algorithm IV: Compensating Threads and Compensating Computations
8.6. Example of Compensation of Parallel Computation
8.7. Compensation Accuracy and Confidence
8.8. The Deficiencies of the Compensation Heuristics
8.9. Comments

9. Contributions and Future Work
  9.1. Contributions
  9.2. Extensions
  9.3. Future Work
    9.3.1. Compensation
      9.3.1.1. Generalizing Compensation
      9.3.1.2. Adaptive Compensation
    9.3.2. User Level Monitor
    9.3.3. Kernel Level Monitor
    9.3.4. PIE Visualization
  9.4. Acknowledgements

Appendix A. User-Sensor Times
  A.1. Measurement Methodology
  A.2. Results on Selected Machines
  A.3. Firing Statistics

Appendix B. Context-Switch Sensor Overhead
  B.1. Measurement Methodology
    B.1.1. Measuring Sensor Firing Times
    B.1.2. Measuring Constant Overhead
  B.2. Statistical Calculations of Overhead
  B.3. Results on Selected Machines
List of Figures

Figure 3-1: Organization of PIE 11
Figure 3-2: Part of a Matrix Multiply Program Text 12
Figure 3-3: Part of Visual Representation of Computation 14
Figure 3-4: Using the Visual Representation to Enable Sensors 15
Figure 3-5: Part of a Parallel Execution 16
Figure 3-6: Parallel Executions of the Computation With and Without the Sync 18
Figure 4-1: A Section of Execution History Top: without Kernel Information 22
Bottom: with Kernel Information
Figure 5-1: User and run-time monitor test-bed 24
Figure 5-2: Simple Application Using Monitor 27
Figure 5-3: Context-switch Event Data Structure 28
Figure 5-4: General Context-Switch Monitor Architecture 29
Figure 5-5: Monitor System Calls 30
Figure 6-1: Matrix Multiplication Executions: Top - Original Kernel; Bottom - 34
Prototype Kernel
Figure 6-2: Matrix Multiplication Executions: Top - Improved Kernel; Bottom - 35
Advanced Kernel
Figure 6-3: The Start of Executions: Original Kernel (top); Prototype Kernel; 37
Improved Kernel; Advance Kernel (bottom)
Figure 6-4: Top: Execution I, Bottom: Execution II 39
Figure 6-5: A closer look at the multipliers 41
Figure 6-6: Selected cpu-views of the Executions 43
Figure 6-7: Top: Execution III, Bottom: Execution IV 45
Figure 6-8: Execution and Cpu Barscopes of Complete Gang Scheduling (With 48
Successive Zooms)
Figure 6-9: Execution and Cpu Barscopes of Incomplete Gang Scheduling of 49
Execution
Figure 6-10: I/O Perturbation 50
Figure 6-11: Execution and Cpu Zoom Views Showing Scheduling Bug 53
Figure 7-1: Firing Times of 5000 sensors vs Time 55
Figure 8-1: Uncompensated and Compensated Views of Part a Matrix Multiply 66
Executed on a VAXstation II
Figure 8-2: A Parallel Execution View 69
Figure 8-3: Corresponding Parallel Cpu view with Threads 1, 3 and 8 69
Highlighted
Figure 8-4: Another Parallel Cpu view with Threads 1, 5 and 8 Highlighted 70
Figure 8-5: Hypothetical Monitored Parallel Computation 73
Figure 8-6: Another Hypothetical Monitored Parallel Computation 75
Figure 8-7: A More Complex Hypothetical Monitored Parallel Computation 77
Figure 8-8: Execution Barscope of Uncompensated Computation 79
Figure 8-9: Cpu Barscope of Uncompensated Computation 79
Figure 8-10: Plot of Selectively Compensated Computation 81
Figure 8-11: Plot of Uniformly Compensated Computation 82
Figure 8-12: Scatter Plot of Difference In Compensated and Unmonitored 84
Means
| Figure 8-13: | Scatter Plot of Difference in Compensated and Unmonitored Minimums | 84 |
| Figure 8-14: | Scatter Plot of Difference in Compensated and Unmonitored Maximums | 85 |
| Figure 8-15: | Execution and CPU Barscopes of Problematic Computation | 87 |
| Figure A-1: | One program for measuring user sensor firing times | 94 |
| Figure A-2: | One program for measuring user sensor firing times | 95 |
| Figure A-3: | Loop Times With Sensors versus Computation Time: 5000 iterations per loop | 98 |
| Figure A-4: | Loop Times With Sensors versus Computation Time: 5000 iterations per loop (minus gettimeofday) | 98 |
| Figure A-5: | Loop Times Without Sensors versus Computation Time: 5000 iterations | 99 |
| Figure B-1: | Code for measuring context-switch sensor firing times | 101 |
| Figure B-2: | Code for measuring kernel monitor overhead | 102 |
List of Tables

Table 8-1: Variable Definitions 67
Table 8-2: Execution Times for Two Computations 68
Table 8-3: Variable Definitions 73
Table 8-4: Program Configurations Used to Generate the Statistics 83
Table 8-5: Statistics keeping the sign of the differences 85
Table 8-6: Statistics using the absolute value of the differences 86
Table 8-7: Compensation Stats for Selected Threads of A Problem Computation 88
Table 8-8: Group Statistics of Degenerate Computation 89
Table A-1: User Sensor Firing Times: Program I 96
Table A-2: User Sensor Firing Times: Program II 97
Table B-1: Definitions 103
Table B-2: Overhead Results 103
Table B-3: Definitions 105
Table B-4: Overhead Results 105
Chapter 1
Experimenting with Parallelism

Aristotelians downplayed experiment and favoured deduction
from first principles. But the scientific revolution
of the seventeenth century changed all that forever.
Experiment was officially declared to be the royal road
to knowledge, and the schoolmen were scorned because
they argued from books instead of observing the
world around them.

- Ian Hacking [16]

In this thesis, the term "computation" covers both application and systems software. Constructing correct,
performance efficient parallel computations is often difficult. Although a programmer may be quite adept
at performance efficient computation design for unprocessors, generating performance efficient
computations for parallel computing systems is frequently more difficult (typically just because the
programmer is inexperienced in writing parallel computations). When designing parallel computations,
not only must programmers study what kind of parallelism lies in their computations, they ought to study
the machine on which the computations will run.

A primary problem of the parallel algorithm designer is to recognize the realizable parallelism inherent in
the problem, to define the problem data structure for maximum parallel data manipulation, and to order the
sequence of operations for maximum parallel functional operation. Parallel compilers can perform at least
some of these functions to transform a given algorithm from serial to parallel implementation, but parallel
software designers must know both the strengths and weaknesses of their parallel processor to effectively
capitalize on its full potential [42].

Unfortunately such a priori information can be incomplete, incorrect or so complex that programmers are
unable to productively manage developing parallel computations to be performance efficient. Originating
the design of parallel computations is like developing and testing scientific hypotheses: programmers
make hypotheses about how best to construct their computations. But just as a physical scientist
constructs experiments to test hypotheses and theories about physical systems, programmers must test
their designs by subjecting them to computational experiments.

To test such computations programmers need a kind of "computational laboratory" in which they can
design experiments to test the behavior of their applications and systems software. The a priori design
specifications must be evaluated in the light of empirical information gleaned from experiments.
1.1. Running Computation Experiments: Perturbing the Data

Measuring the performance of applications requires running them experiments on them. To run an experiment on a computation requires requires that the computational laboratory have some kind of data generation/collection facility that monitors the computation as it executes. Such laboratories are software development environments which make use of special development and data analysis tools for doing the monitoring. Such monitoring generates and records observable time-stamped objects (events) about the executions of programs. Unfortunately, collecting run time data about a computation almost always perturbs the computation's performance, not only altering its execution time\(^1\) but possibly distorting its behavior. The most ordinary (and trifling) effect of monitoring perturbation is the altering of execution time. This is not a particularly insidious effect as long as the computation behaves in a way not unlike it would behave unmornitored. At worst, the designer must wait longer before learning how the computation executes. Yet uniform extensions of execution times are so unlikely that non-uniform changes must be accepted as prevalent and inevitable consequences of monitoring. The more pernicious effects of perturbation are distortions in how computations execute, for example, changing the event order of computations. This is more problematic than a simple altering of execution times because if the behavior reported by monitoring is consistently and grossly different from that of the corresponding unmonitored computations the performance data may be useless.

1.2. The Kind of Monitoring to be Considered

The degree to which executions are distorted depend upon the kind of computation that is monitored, how it is monitored and what data is collected. This thesis examines perturbation by programming environments with portable and selective monitoring systems. "Portable" means no special hardware assistance is assumed to be available to such a monitor. Although additional hardware such as new disk controllers, memory mapped bus monitors or network monitors are permitted, they are not necessary. "Selective" means an interactive monitoring system which can choose an arbitrary subset of computation attributes to sample.

1.2.1. What Can be Monitored

Generally, the degree to which programming environments use portable and selective monitoring systems is also a measure of the degree they are built from primarily software objects and thus an indication of what kind of computational behavior they can observe. Monitoring systems without any supporting hardware can hardly be expected to accurately measure the miss ratios of instruction caches or the traffic on a shared system bus, for example. Software based monitors can monitor high level actions like thread creation and termination, \textit{for} loop iterations in a C program, context-switching and page faulting.

\(^1\)Although monitoring usually lengthens execution times, occasionally it can shorten them. See the results for the Encore Multimax in Appendix A
1.2.2. The Danger of Perturbation

The use of software monitoring objects such as sensors is also indicative of how much perturbation can be expected. Software based monitors perturb computations because they compete directly with the computation for the hardware and software resources of the system they are running on. Whereas the perturbation from hardware monitor sensors used to monitor can be insignificant, perturbation from software sensors can be quite severe if, for example, they are used to timestamp synchronization points of parallel computations executing on tightly-coupled multiprocessors. Many machines rely on system calls to retrieve timestamps of user-level events. Because of the overhead of most timestamping system calls, frequent and ubiquitous use of them risks distorting the executions of computations. Compounding this problem of distortion, additional high level software and independent monitoring processes are useful for support of interactive capabilities as well as to identify, tag and store measurement data as the computation executes. These additional monitoring objects compete with the computations for system resources. As more measurements are made, the risk increases of reordering events and altering execution times.

1.3. The Kind of Systems to be Considered

This thesis directs its attention to monitoring of sequential or tightly-coupled machines with uniform access time memory topologies. Examples of such systems are workstations like the Sun3 and the Encore MultiMax (a parallel, shared bus machine). Distributed systems with large local memories or systems with irregular or distributed memory subsystems are not considered.

1.4. Recovering the Real Behavior of a Computation

Although monitor perturbation raises the possibility that data retrieved by software monitors is untrustworthy, the complexities and uncertainties of parallel programming make monitoring parallel programs highly desirable. Accepting this possibility, the monitor problem becomes one of either attempting to eliminate perturbation or compensating for the errors caused by monitoring perturbation so that the real behavior of computations can be recovered. Since the monitoring system considered here is portable and selective, perturbation cannot be eliminated. At best, perturbation can be minimized through thoughtful programming of the monitoring code. Given that the perturbation exists, the general problem I tackle is not the elimination of perturbations, but the detection, measurement and compensation of them. There are three basic questions:

1. How does one detect and measure a perturbation?
2. How does one compensate for perturbation caused by monitoring?
3. How can one estimate confidence in the accuracy of the compensation?
1.4.1. What Perturbations are Detected and Measured

Chapter 6 illustrate three basic kinds of monitoring perturbation:

1. Perturbations that alter the execution times of computations and their constituent parts.

2. Perturbations that alter the order of events and scheduling of computations and their constituent parts.

3. Perturbations that distort the accuracy of reports of local timing (not order).

The fourth kind of perturbation consists of altering the workload of the computation and is usually derivative of the second kind described above. This thesis only treats perturbations of the first kind and relies on the self-detecting nature of monitoring objects to measure the effect of these perturbations. Examples of self-detecting objects are sensors and monitoring actions, such as the execution of a monitor thread, delimited by sensors. Measuring the immediate perturbation resulting self-detecting objects consists of knowing the duration of its immediate effect. In the case of a sensor, this is its firing time. For a monitor thread, this is a particular time slice of execution.

Because of limitations in the monitoring facility used in this work, this thesis does not consider detection or measurement of perturbations due to differences in program code size resulting from the inclusion of monitoring code. Nor does it consider perturbation of paging or I/O behavior. And, because the monitoring system is software based, it does not consider perturbation of low-level entities such as caches and busses where performance is measurable on the level of individual instructions. Nor does this thesis consider perturbation caused by uneven processor execution speeds of a parallel system.

Finally despite the evidence that the execution times of user-level sensors seems to depend upon the context within which they execute (see Appendix A), this thesis will treat them as constructs that unfaithfully execute in a fixed time. More realistic treatment of their effects is reserved for future work.

1.4.2. Compensating for Perturbations

Once the immediate effects of a perturbation are detected and measured, the problem of compensating for those effects remains in order to estimate what the execution of computation would have been like without monitoring. The compensation algorithms presented in this thesis address computations which are predominantly asynchronous. The only interthread synchronization they allow is the kind that may occur during thread creation and termination.

The algorithms do not detect event reordering and thus cannot determine whether the monitor altered the workload of the compensation. Finally the algorithms do not address the distortion of local timing accuracy of the measurements. Research into compensating for these kinds of perturbation is reserved for future work.
1.4.3. Confidence in Compensation

One can make informed judgements about an execution profile measured by monitoring only if one has some confidence in the measurements. The purpose compensating for monitor perturbation is to give confidence in the execution profile reported by the monitor. Thus compensating for perturbation must also be able to provide some kind of estimate of the confidence in the reported execution profile. Confidence would be high, for example, if compensation reports an execution profile that blends in with the indeterminate ambient system "noise" that varies the execution of computations even without monitoring. As argued by Schiffenbauer [38] compensation can be considered successful if the results fall neatly within the distribution of possible executions.

1.5. Structure of this Thesis

The thesis begins with a discussion in Chapter 2 of related work on monitoring and programming environments. Later Chapters of the thesis refer back the discussion of related work when it is necessary to compare the monitoring system and programming environment system used in this work with those used the other research. The discussion on related research is followed by a description of the programming laboratory used in this thesis, the Parallel Programming and Instrumentation Environment (PIE) in Chapter 3. The general attributes and architecture of its monitor are discussed in Chapters 4 and 5. Chapter 6 discusses several monitoring examples with notes on the perturbation present in each. Chapter 7 then gives a formal definition of the kinds perturbation followed by presentation and evaluation of three compensation algorithms in Chapter 8. Chapter 9 discusses the contributions of this research and future work to continue it.
Chapter 2
Related Research

Research related to this thesis can be divided into roughly three categories, programming environments and monitoring systems, and treatment of monitor perturbation. Discussion of each of these categories overlaps the other two. Thus, in this chapter, programming environments are discussed first followed by research on monitoring and perturbation. Although the research is presented here as part of an introduction to the problem of performance monitoring and perturbation, later chapters will reiterate some of this discussion for clarification.

2.1. Programming Environments

The Parallel Programming and Instrumentation Environment (PIE) is a software development environment for performance efficient parallel programs. Its support for monitoring program performance distinguishes it from environments that only support the development of program coding like the Code [7] and PV [6] projects. Like the Faust [14] project, one of PIE’s goals is develop techniques to efficiently map parallel applications onto specific architectures. Faust, however, is a more ambitious environment than PIE. It seeks to include expert system assistance in program development, code reusability and extensive use of database support for making queries about the structure and performance of programs. An ambitious environment by Aral et al., [2] based on PIE supports dynamic changes to the object code of a computation while it executes so that programmer can conveniently test coding changes during run-time instead of resorting to time consuming recompilation [2].

Parts of the PREFACE programming environment [4], which monitors an 8 processor shared-bus machine running a version of Mach, marks objects to be monitored during a pre-compilation stage. The pre-compilation instruments all parallel Fortran constructs. PREFACE provides information about system level work versus user level work.
2.2. Monitoring and Perturbation

It is difficult to discuss programming environments without discussing monitoring systems. There has been considerable work on monitoring over the years. Much of the research done on monitoring parallel computations has not been in performance monitoring, but in debugging for functional correctness, [22], [23], [27], [32], [40], [38]. The focus of these projects has been on reducing perturbation by distributed debuggers as well as how to ensure their correct replay of executions.

Several of the systems target loosely coupled distributed computations where kernel-level events are often not context-switches but messages between the distributed processes. The METRIC [26], DPM [30], IPS [29] and TMP [45] systems, for example, include monitoring facilities for distributed programs which trace actions between the distributed processes of a computation. The monitoring provides data which, when analyzed, not only yields basic communication statistics but also rudimentary information about execution histories. Because of the large granularity of the actions these monitoring systems examine, the performance perturbation has not been considered in the related literature.

Many systems, [31], [41], use extensive hardware resources to reduce monitor perturbation and get more precise timing measurements. Berg [3] uses a separate monitoring network and processor to toggle the kernels of distributed processors between a normal and monitoring mode in order to get very low level measures of processor behavior. The PEM system [8], permits monitoring of a single application using special hardware on a multi-configuration parallel machine, M^3. The TMP system also uses special hardware to support its distributive monitoring.

Although the M^3 machine is configurable in several ways, the monitoring is not portable in the sense that PIE demands. Certainly, the monitoring hardware in Berg's and Bhatt's systems renders them very powerful and unobtrusive, but that power makes them unportable and thus incompatible with these the kind of monitoring preferred by PIE. In addition many of these the monitoring systems can monitor only a single application at a time. PIE and MKM, as will be shown, permit simultaneous, protected monitoring of independent applications.

Many monitoring efforts have narrowly restricted the objects of interest. By doing so, designing the monitoring system to unintrusive is simplified. Kobayashi [20] discusses a kernel-level context-switch monitor that more closely resembles the monitoring of MKM and PIE than the systems just described. He uses a kernel trace facility to record the impact of context-switching on cache performance. Kobayashi's monitoring facility records the context-switches of all measurable threads and generates statistics about their effect on the caches. MKM differs in that it permits both user and system programmers to selectively track only those threads they purposively choose.

The TPORTSS project [35] uses hybrid (software and hardware) monitoring to measure a large set of operating system actions on a microprocessor. Although it may be possible for the measurements to be
used in ways as done in PIE, the results are used more for research characterizing operating system performance using equations derived from regression analysis of the measurements.

The SPAM [28] event tracer uses changes to the microcode of a VAX 8600 to trace interrupts, exceptions, system calls and context-switches. SPAM can measure behavior such as the number of machine instructions between context-switches.

The PREFACE environment monitors only the parallel segments of Fortran programs. The monitoring (the authors refer to the monitoring activity as "tracking") can be done in two modes, both of which assume that each process is responsible for its own monitoring. The first mode periodically flushes performance data to disk while the second writes only to memory. In order to reduce the monitoring perturbation of the important parallel components of the computations, the monitoring systems only update in various performance visualizations during the sequential parts of the computation,

Reilly [36] investigated hardware monitoring but, like Mink, laments the impracticality of a strictly hardware based approach to monitoring. He asserts that sensor execution times probably have a multi-model distribution, with with at least two means, one for cached sensors and the other for uncached sensors and perhaps one for those sensors which induce a page fault.

In the design of his monitoring system, Schiffenbauer [38] addresses monitoring perturbation by presenting an intuitive argument for distinguishing between improbable and probable executions, asserting that event orderings captured by a monitor are acceptable if they are an ordering that is not "surprising" or improbable. That is, monitor perturbation of a computation is acceptable if the monitored event orders and execution times can be considered to be not unlike what one could expect from an unmonitored computation. Although he restricts his study the effects of a distributed debugging monitor where time perturbation is not considered except as an effect on event ordering, when applied to performance monitoring, the notion of a probable execution must be
Chapter 3
The PIE System

Data processing frequently is a task of processing large amounts of all kinds of data. Nothing is more natural than performing that work interactively by dealing with pictures instead of numerical series.

- Haas, Mrva and Tengler [15]

The Parallel Programming and Instrumentation Environment (PIE) is a software development environment for debugging performance. It gives programmers ways to observe how parallel and sequential computations execute by making use of special development and data analysis tools. PIE is more than just a programming environment, as defined by Dart and others [10], which only supports the development of program coding like the Code [7] and PV [6] projects, for example. Nor is it a program simulation environment. Rather, PIE is a "computational laboratory" in which programmers can design experiments to visualize and evaluate real inprocess and interprocess behavior of computations. Like the Faust [14] project, one of PIE's goals is to develop techniques to efficiently map parallel applications onto specific architectures based on a philosophy of preventing, detecting and avoiding performance degradation.

PIE is implemented on top of the Mach [1] operating system but can be ported to other Unix-like operating systems. PIE supports languages such as C, MPC [44], C-threads [9], ADA and Fortran. Unlike some other performance measuring environments [8], [31], [41], PIE is a portable system whose basic platform is a workstation running the X Window System [37] with a minimum of 8 megabytes of RAM and a 70 megabyte disk. Its monitoring instrumentation runs on a diverse set of hardware platforms, including Vax and Sun workstations, Encore Multimax, and Warp.

3.1. The Organization of PIE

Figure 3-1 depicts the general organization of PIE. The environment presents the programmer with a consistent shared-abstract data type paradigm. In the framework of this paradigm a programmer assumes that each of his parallel actions has access to the same shared-abstract data type. PIE has developed a body of theory dealing with the prevention [43] and avoidance of performance bottlenecks. In order to detect the performance bottlenecks of an executing computation, and environment requires information about the structure of the computation, run-time information about how the computation executes, and a relational database that couples run-time data with the appropriate structural information.
The PIE environment consists of several subsystems for assisting a programmer in developing computations and observing their run-time behavior. These systems are:

- **PIEmacs**: an editor with special features for inserting monitoring hooks into a computation,

- **PIEmon**: a performance and correctness monitoring facility including the Mach context-switch monitor [24],

- **PIEman**: a database manager which correlates the text of a computation (development-time information) with information about the execution of the computation (run-time information) and,

- **PIEscope**: a graphics package for presenting the run-time and development information to the programmer.

In PIE, a programmer edits a computation using **PIEmacs**, an extension of Gnu-emacs. PIEmacs automatically marks several program constructs for affixing sensors later using a special compiler preprocessor. In addition to the automatically marked constructs, the programmer is permitted mark a computation in places of special interest to him. The development-time information PIEmacs generates as it marks a computation is to **PIEman**, a database manager. PIEman builds a database from this program development information which it later merges with data retrieved at run-time. The run-time information is retrieved by **PIEmon**, the PIE performance monitor. The information is presented to the programmer via **PIEscope**, a graphics package that displays several views of the structure of a computation as well as the how the structures were executed.
Parts of the PREFACE programming environment [4], which monitors an eight processor shared-bus machine running a version of Mach, are similar to components in PIE. Whereas PIE separates the selection and actual insertion of monitored objects into an editor and preprocessing stage, PREFACE combines them into a preprocessor which instruments all parallel Fortran constructs (ie, no selectivity). The visualization of performance data is more primitive than in PIE, but PREFACE provides information about system level work versus user level work. The differences in the monitoring policies is discussed in Chapter 5.

```c
multproc(x1, x2, y1, y2, mx, my, sz)
    int x1, x2, y1, y2, mx, my, sz;
{
    int ex, ey, i, j, k;
    float t, tmp, tmp2;
    multiply subtask;
    ex = x2 - x1 + 1;
    ey = y2 - y1 + 1;
    if (ex > ey) {
        if (ex > mx) {
            subtask (x1, (x1 + ex / 2 - 1), y1, y2, mx, my, sz);
            multproc((x1 + ex / 2), x2, y1, y2, mx, my, sz);
            join(subtask);
            return;
        }
        if (ey > my) {
            subtask (x1, x2, y1, (y1 + ey / 2 - 1), mx, my, sz);
            multproc (x1, x2, (y1 + ey / 2), y2, mx, my, sz);
            join(subtask);
            return;
        }
    }
}
```

**Figure 3-2:** Part of a Matrix Multiply Program Text
3.2. Setting Up an Experiment in PIE

Because PIE is the vehicle for investigating perturbation and techniques for compensation, it is important to be comfortable with frequent references to the environment later in this work. The following brief example of using PIE is condensed from the introduction to a paper in IEEE Computer [25] and should be read if greater familiarity with PIE is desirable. Otherwise, it may be skipped.

3.2.1. A Problem Application

Assume a programmer has a problematic matrix multiply application that achieves an average parallelism of only 2.1 although it spawns four multiplier threads\(^2\) on a 16-processor shared-bus machine. The basic structure of the application consists of passing well-partitioned parts of two matrices to several child threads. Each process first examines the parts of matrices it is passed and decides whether they are small enough to operate on without partitioning them further and passing them on to its own child process. After making the decision, each process iterates through its matrix parts, multiplying each pair of row and column and writing out the result.

Figures 3-2 and 3-3 showcase how PIE implements visualization of programming constructs. Figure 3-2 depicts parts of the text of the application via three windows of PIEmacs. The application is written in an extended C-language, called MPC [44], a multi-process C programming language which permits a programmer to write parallel applications unfettered by the actual synchronization and memory policies of the underlying target machine. It is not important to fully understand the semantics of the text but some elucidation of the more novel parts will be helpful.

The top window in Figure 3-2 shows a part of the definition of the application's multiplier procedure, multproc. It includes a variable declaration of the type multiply, an instance of what MPC calls an activity or act, as shown in the middle window. Activities are process-like units of parallel work which, when spawned from the same application, are able to share and operate on global memory. Notice that multiply contains a call to multproc. Multproc implements the basic matrix partitioning and multiplying functions described above. After the value of a matrix element is calculated, it is written out using put, shown in the lowest window. It is an instance of an MPC object called opr, used to operate on global memory. Entities of this type may be shared by several activities. The only feature of put that ought to be understood here is the sync, a special MPC function that enforces mutual exclusion on global operations. Here, sync ensures that only one result may be written back to global memory at a time.

\(^2\)Threads are "light-weight" processes that are required to save less state than normal processes when they context-switch
3.2.2. Setting up the Experiment

Figure 3-3 is an automatically generated PIEscope road map visualization of the application's principal constructs. The roadmap view is the first step in PIE for bridging what Yang and Miller [46] call the semantic gap between the semantics of the structures of computations and the semantics of the measurements retrieved by the monitoring system. Each box in Figure 3-3 has a corresponding textual entry in Figure 3-2. When a box is touch-selected by a mouse, as is shown by the enlarged border surrounding the box labeled [c] multproc, the PIEmacs window automatically moves its cursor to the head of the corresponding textual construct, in this case, a call to the multproc procedure as shown in Figure 3-2.

Having visualized the structures of the application, it is time to gather performance information. Like Kumar's [21] system for measuring parallelism of Fortran programs, PIE can generate performance views such as histograms, but these are ancillary to a more informative format which will be shown shortly. When a application with a potential parallelism of four runs only two or three threads simultaneously, the programmer knows that he should investigate any program construct that might force a multiplier to wait, namely the sync just discussed and the join (an example is shown in the top window in Figure 3-2) which a multiplier executes when it wishes to join its children. To get this information with PIE is simple.
Figure 3-4 shows a number of darkened boxes, [A] multiply, [S] Sync and several cases of [J] Join. The [A] multiply represents the multiplier threads and [S] Sync is the synchronization function in the put operation discussed earlier. Each [J] Join represents an instance of a join function. The darkening of these boxes indicates that the programmer, using a mouse-click, has selected them to be automatically observed during execution.

![Figure 3-4: Using the Visual Representation to Enable Sensors](image)

3.2.3. Examining the Results

PIE's theoretical foundations for instrumenting computations [12] includes software, hardware and hybrid event sensors. Currently, however, computations in PIE are instrumented using only software sensors. During run-time PIE ensures that when a selected construct executes, important information is automatically collected about the construct. An example of PIE's principal formats for visualizing performance data is shown in the upper two views of Figure 3-5. They are similar in intent to, but more richly implemented than the graphical views provided by the Faust and PREFACE environments as well as by a parallel debugging system developed at Los Alamos National Laboratory [13].

The top view of Figure 3-5 is called the **execution barscope** view. Time is measured in microseconds on the horizontal while the threads of the computation are ordered on the vertical. Although it is possible to
Figure 3-5:
Top: Part of a Parallel Execution of the Computation on 16-processor Machine
Middle: Corresponding Thread to CPU Assignments
Bottom: A PIE editor window

show any part of a computation, this particular view shows only tail end of the execution from about 2.6 to 2.8 seconds. The execution of each thread is depicted by the concatenation and occasional "overlap" of
several textured rectangles, each representing a particular episode in the thread's history. A rectangle is "in front of" another rectangle if the entity represented by the rectangle in the forefront is contained within the entity represented by the rectangle behind it. In the top view of Figure 3-5, for example, waits due to a \texttt{sync} show up as dotted rectangles alternating with several dark rectangles. The dotted rectangles are actually in front of a single dark rectangle representing the generic body of the thread. The slashed rectangles at the end of threads zero, two and three are instances of parents waiting to join a child.

The middle view of Figure 3-5 is called the \textit{cpu barscope} view. It shows the thread-to-processor assignment of the computation during the same execution period shown in the top execution barscope view. Time is measured in microseconds on the horizontal and the machine's processors are ordered and arbitrarily numbered on the vertical. Opposite each \texttt{cpu} are alternating sets of textured rectangles representing identifiable threads. White rectangles are periods when none of the computation's threads are running on the associated \texttt{cpu}.

When any rectangle in either barscope view is selected by the mouse, the cursor in the PIEmacs window is automatically moved to the head of corresponding construct in the program text. Here for example, a \texttt{sync-wait} rectangle has been clicked on in the execution barscope. Simultaneously, the PIEmacs cursor moves to the head of the appropriate \texttt{sync} definition window as shown in the bottom view of Figure 3-5. The semantic gap is now bridged allowing the programmer to analyze the computation's performance using data automatically projected onto the computation's structures. In addition, the visualization helps bridge what Yang and Miller call the \textit{functional gap}: the gap between the extent to which performance monitoring merely reports how computations behave and the extent to which it helps guide users to the source of their problems. PIE does not, however, supplant the visualization with automatic assistance to guide users in correcting the problems they find.

\subsection*{3.2.4. Fixing the Application}

Only some of the row/column multiplications performed by the entire application are shown executing in the top view in Figure 3-5. The \texttt{multiply} threads, numbered in the order they are spawned, execute the \texttt{multproc} procedure. A special PIE thread used in gathering performance data has been left out of this view in order to simplify this example (it would have been numbered as "1"). As expected, the figure shows that, at any given time, several of the threads are waiting on a synchronization barrier. If a vertical swath is drawn through the view at any point, it often intersects only two or three black rectangles, indicating that only those threads are doing useful work. The fact that the remaining threads are all executing \texttt{sync-waits} at these points (excepting \texttt{main}, of course), casts suspicion on the \texttt{sync} function in the \texttt{put} operation as the source of the performance degradation. The middle \texttt{cpu} barscope shows several breaks in the textured rectangles, indicating that the associated threads context-switch often, another symptom of synchronization overhead.

How does the programmer repair the application? Is the \texttt{sync} really necessary? Each time the \texttt{put}
Figure 3-6: Parallel Executions of the Computation With and Without the Sync operation is called, after all, it only fills a single location in the result matrix. Since the matrices are well partitioned, each call to put touches a unique location. Consequently, the sync is not necessary. The top view of Figure 3-6 shows a similar multiplying section of the application after removing the sync. The bottom view shows the corresponding cpu barscope view. As can be seen, the execution time has been cut from about 2.8 to 2.0 seconds and the heavy context-switching has disappeared.

3.2.5. But Is the Environment Hiding Something?

Over three thousand sensors fired in the corrupted application shown in the lower view of Figure 3-6, most of that total is equally divided among the four multiply threads. Only slightly more than two hundred sensors fired in the corrected application. Since these views are not compensated for the different number of sensor firings, it is not clear how much they accurately reflect the performance change.
Chapter 4
Implementing a Monitor

In order to better understand the context within which monitoring perturbation issues will be discussed, the type of monitoring used in this work must be discussed. In this chapter summary stipulations of monitoring terms and the attributes of the monitoring system used in this work are presented.

4.1. Basic Monitor Terms and Concepts

Parts of the following definitions and subsequent discussion is condensed from work by Gregoretti and Segall [12].

4.1.1. Computation

A computation is a set of given typed data, an effective procedure that operates on those data and a set of result typed data generated from the givens and the procedure. I use "computation" here in order not to lose any generality by using terms like "program" or "process." Thus, system and application software are computations.

4.1.2. Events

An event is an observable, time-stamped object occurring during execution of a computation; it is the unit of performance information (measure). Events consist of two basic types, control-driven and data-driven.

4.1.2.1. Data-driven events

A data-driven event is a time stamped modification of, or demand for, computation data. Data-driven events do not contain direct information about computation states, but en masse they describe data access patterns. Although inferences can be made about what computation states are possible for a specific data-driven event, they can be made only after comparing the event to where the datum is used in the computation's text and with an analysis of the execution history provided by control-driven events. Currently, PIE can not detect data-driven events.
4.1.2.2. Control-driven events

A control-driven event is designates a specific logical point (state) in a computation's control flow which includes the time when that state was reached and, occasionally, ancillary computation data requested by a user. Examples of control-driven events are the creation and termination of processes or the start of an iteration of a program loop.

4.1.3. Sensors

Sensors detect the events of a computation and prepare them for retrieval by collection instrumentation. This instrumentation is a software/hardware system which appends an event to the event record of the computation. After an execution terminates, Pieman selects and filters the events in the event record using a relational data base and any relevant performance or status goals requested by the user. The relational data base, constructed at development time, contains the static structures of a computation as well the semantic and temporal relations between them. The structures contain sensor marks so that events collected during execution can be mapped onto their corresponding computation.

4.1.4. Monitor

A monitor observes and records computation events. A monitor consists of sensors for marking events, instrumentation for event collection and a relational model that ties the execution data to the appropriate computation structures.

4.2. Attributes of a PIE-like monitor

While there are many ways to monitor computations, the PIE environment requires selective systems that easily port to different computing systems. "Portable" means no special hardware assistance is assumed to be available to such a monitor. These monitoring systems cannot require any architectural revision of the host computing systems. By "architectural revision," I mean any restructuring of a computing system's underlying processing, communication or memory hardware subsystems. Examples of architectural revisions are changes in word size or address space, additions of new instructions or new bus lines. Adding hardware such as new disk controllers, memory mapped bus monitors or network monitors are not architectural revisions. "Selective" means an interactive monitoring system which can choose an arbitrary subset of computation attributes to sample.

Although monitor implementations depend on what kind of architectural support a computing system provides, there are three attributes that broadly characterize any monitor I consider:

1. All implementations introduce some inline code, such as sensors, into a monitored computation.
2. All implementations must be able to handle special concurrent or parallel monitoring threads.

3. All implementations are multi-level. The monitoring system is hierarchical, consisting of data collection on levels such as the application, run-time and kernel.

4.2.1. Inline code

By permitting the monitor to insert additional code into an application, the selectivity of the monitoring system is improved: inline sensors can be arbitrarily placed within the computation. If the inline sensors are form of a high-level language, the portability of the monitor is enhanced.

4.2.2. Monitoring Threads

Environments with interactive run-time capabilities would allow programmers to peruse the execution of computations, moving their attention to different parts or asking for greater detail as the computations proceed through the different stages of their execution. In order for an environment to be able to adapt to the run-time vicissitudes of a programmer it must have an interactive interface. More ambitious environments might support dynamic changes to the object code of a computation while it executes so that programmer can conveniently test coding changes during run-time instead of resorting to time consuming recompilation [2]. The PIE environment allows programmers to observe computations while they execute but has only limited interactive features.

4.2.3. Multi-level Monitoring

A performance monitor which constricts itself to observing only a single level of behavior, user-level behavior for example, cannot retrieve sufficient information to adequately interpret a computation's execution. With only user-level information, programmers cannot always isolate the source performance deficiencies because the performance of parallel computations is not only affected by user designed attributes such the decomposition of the computations but also by operating system actions such as scheduling. One way of determining how computations are scheduled is to detect and time-stamp the context-switches of their threads.

By extracting context-switch information about computations, designers can disambiguate the user-level information they receives. For example, the top view of Figure 4-1 shows a section of an execution of a matrix multiplication computation. Each small, vertically textured rectangle delimits the execution of a single iteration of a \texttt{for}-loop. The left edge of a rectangle marks the begin time of an iteration; right edges mark the end times of iterations. The two time cursors mark the begin and end time of an iterations that appears to have executed for an unusually long time. Most of the other iterations executed in much shorter time. Since the iterations executed the similar code their execution times should be similar unless the operating system descheduled some of them.
Figure 4-1: A Section of Execution History
Top: without Kernel Information
Bottom: with Kernel Information

The effect of scheduling, however, is not shown in the top view of Figure 4-1. Despite the fact that this view suggests that all the threads ran in parallel at all times, without information about the scheduling of the computation, the real parallelism achieved is unknown. The bottom view of Figure 4-1 is the same as the top except that it includes context-switching information about of each of the threads. As described in Chapter 3, the white rectangular sections of a thread's execution represent periods when the thread was not running on a processor, that is, when it was blocked or ready to run. With context-switch information included, the reason for the longer execution times of the iterations is obvious.
Chapter 5

The Monitoring Laboratory

In order to do research into compensating for monitoring perturbation of computations, it is necessary to have monitoring "laboratory." This chapter describes the general architecture of the three important components of the monitor hierarchy: the user and run-time level monitor, the context-switch kernel level monitor and data interpretation system. It finishes with a summary description of the problems of interpreting monitor data and the solutions to them.

5.1. History of the Monitoring Laboratory

The monitoring laboratory used in this thesis is the PIE environment. When this research was first proposed, several important parts of PIE had yet to be implemented or developed fully enough to run experiments that could provide enough information to perform compensation calculations. The MPC monitoring run time and Mach context-switch monitor, for example, were either too primitive and unreliable for extensive use or were not installed on a full time basis. Although early versions of the Mach context-switch monitor has been running on PIE workstations since 1988, it did not become a regular feature on the MultiMax until early November, 1989. Thus much of the early experimental work could only be done on the single-processor workstations. PIE's visualization abilities have been periodically improved to give more ways to analyze computations. Finally, the compensation code had to be developed.

The original monitor system architecture and implementation has undergone several improvements those associated with PIE\(^3\). Figure 5-1 shows a block diagram of the PIE hierarchical monitoring system. The monitor hierarchy consists of three basic levels: the user, run-time and kernel levels. The user and run-time levels are well integrated and will be discussed together. The kernel level monitor communicates with the user and run-time levels, but is different in several respects due to operating system considerations not present at the higher levels.

\(^3\)See acknowledgements.
5.2. User and Run-Time Monitoring

The user and run-time levels of monitoring consist of software event sensors for writing records into memory buckets shared by a parallel collector thread (activity in MPC based versions) which writes each record to a designated file. There are approximately thirty user and run-time level sensors, each marking a specific event such as the beginning of a procedure or iteration of a loop. When a sensor fires, it executes a notifier which time stamps and stores information from the sensor into the memory bucket allocated for the thread executing the sensor. The packet of information stored by a notifier is called a notification. There are several notifier routines, but they all take parameters labeling the type of the event (e.g. the start of an activity) as well as unique information identifying where in the computation the sensor resides. The notifier routines are nearly identical, differing only in what auxiliary parameters they take. The format of the information provided by a sensor is that of a standard record followed by an optional auxiliary record consisting of character string, an integer or a floating point value. The standard record contains the type of sensor fired, a unique sensor ID (for use by the PIE environment) and an ID indicating the thread in which the sensor fired.

When a bucket fills, the sensor filling it allocates another bucket for its thread while the collector writes out the contents of the filled bucket to a file which PIE uses for performance/debugging analysis of the computation. If no buckets exists for a sensor to write, it creates a new bucket and writes a bit in the event record indicating that it was delayed firing. The collector's creation and termination is invisible to the user; it is spawned before any statements in the application's parent procedure execute and is joined after execution of its last statements.
5.3. MKM: The Mach Kernel Monitor

Oh great Mach, you are so very Huge.
Your mighty kernel, larger than life,
crushes mine enemies,
Yea while a host of #ifdefs confounds my foes.
Your source tree twists and contorts
through a thousand links;
And where one file is deleted,
Ten shall sprout to take its place.
Gosh, I'm just really impressed down here, I can tell you.

- Will Welch: On life in the trenches of Mach

PIE uses a low overhead kernel level context-switch monitor implemented within the Mach operating system [1]. Called the Mach Kernel-Monitor (MKM), it timestamps and records the context-switches of selectable threads and the processors they switch on. Before discussing the architecture of MKM, it is useful to be acquainted with the general architecture of Mach. Parts of the discussion of kernel monitoring is taken from [24], but is repeated here for convenience.

5.3.1. Mach

Mach is an operating system for integrating support of networks of uniprocessors and multiprocessors while presenting a Unix style software environment. The basic Mach primitives support Unix functionality by placing it outside the Mach kernel. Only the task and thread and port are discussed here because they the MKM interface is modeled after theirs. A task is an address space and a set of system resources (eg. file descriptors) while a thread can be thought of as a program counter and register set. Each task holds a large virtual address space which can be allocated and de-allocated by any thread running within that task. At least one thread is always associated with a task (a Unix process can be emulated as a single thread executing within a task), although several threads may share the task’s resources in parallel.

The basic Mach communication abstraction is a protected entity called a port for sending and receiving typed messages [47]. A port is basically a protected queue with associated send (enqueue) and receive (dequeue) rights. Interprocess communication (IPC) in Mach is transparently extendible over networks. The Mach kernel implements message passing between tasks on the same host. Messages sent to ports belonging to tasks executing on remote hosts are sent to a network server which forwards the message over the network. Mach provides an interface language [18] to generate the client/server interfaces that place network communication functionality outside the kernel in order to increase the flexibility of each host in choosing how data is represented, how network security issues are resolved and how communication protocols are implemented.

Because Mach is targeted to run on a variety of machines, close attention has been paid to ensuring that the Mach implementation is truly portable especially in the area of virtual memory support [34]. Because
virtual memory management units differ widely among different machines, Mach's virtual memory implementation is divided into machine dependent and independent parts.

5.3.2. General Architecture of the Mach Kernel Monitor

The Mach Kernel-Monitor is modeled after the structure of Mach abstractions such as tasks and threads. The inspiration for MKM arose out of the needs of PIE, and thus has a different focus than monitoring work aimed at gathering general usage statistics. The MKM scheduling monitor is a good example of how monitoring in PIE differs from other systems. MKM does not merely gather statistics on context-switching (although statistics can be generated). Statistics alone are often of little help in identifying where performance problems lie. Unlike Kobayashi’s context-switch monitor MKM permits both user and system programmers to selectively track only those threads they purposively choose. It supports multiple monitors for independent computations, storing the protected data into the appropriate buffers until each user reads them.

MKM orders context-switch data chronologically and finds what threads are associated with each switch. Within PIE, the context-switching patterns of a computation are mapped onto an execution graph which is then displayed by PIE using a number of visual formats. With MKM as its lowest level monitor, PIE gives users a way to disambiguate whether the sources of performance problems lie in the applications themselves or in something that the scheduler "does" to them.

An MKM monitor consists of:

- A data structure consisting of pointers to buffers for storing information about detected events (in the current implementation, a monitor can detect only context-switch events) and state information. The data structure and buffers are in kernel space.
- A list of threads that can be observed by the monitor.
- Event detection sensors within the Mach context-switching code.
- Entries within thread data structures for assigning monitors to threads.

These data structures are operated upon by several monitor system calls. Future versions of the monitor may contain data structures for monitoring message sends and receives or paging behavior. Currently, the only calls recognized by a Mach kernel monitor are ones concerned with detection of context-switches.

The monitor_create() call creates a monitor within the calling task. The call returns the monitor id and the size of the event buffer in protected kernel space. Monitor_create() returns the monitor in a suspended state in which all the necessary data structures and buffers are created but monitoring is prevented from detecting context-switches. Monitor_resume() starts context-switch detection by making the monitor leave the suspended state. Monitor_suspend() permits the user to resuspend monitoring if context-switch detection is not needed. Monitor_terminate() destroys the monitor, removing the buffers and other data structures in kernel space. monitor_read() reads the context-
switch event data from the buffer in kernel space and transfers the data to a computation supplied buffer in user space. The size of the buffer into which the context-switch data is transferred must be at least the size returned by `monitor_create()`. `monitor_read()` calls are valid as long as its arguments contain a valid, non-terminated monitor. Individual thread are enabled for context-switch monitoring using `thread_monitor()`. When a thread calls `thread.monitor()` the associated monitor will detect each time the thread context-switches. `thread_unmonitor()` disables a thread from being monitored.

```c
#include <mach.h>
#include <mach/kernel_event.h>
#include <mach/monitor.h>

main()
{
    int buffer_size;
    monitor_t my_monitor;
    int j, num_events = MONITOR_MIG_BUF_SIZE;
    kern_mon_buffer_t kernel_events;

    buffer_size = REQUESTED_SIZE;
    monitor_create(task_self(), &this_monitor, &buffer_size);
    monitor_resume(this_monitor);
    thread_monitor(this_monitor, UNIQUE_ID, thread_self());

    for(i = 0; i < 300; i++) sleep(1);

    thread_unmonitor(this_monitor, thread_self());
    kernel_events = (kern_mon_buffer_t)
        malloc(sizeof(kern_mon_data_t) * MONITOR_MIG_BUF_SIZE);
    while (num_events == MONITOR_MIG_BUF_SIZE) {
        monitor_read(this_monitor, kernel_events, &num_events)
        for (j = 0; j < num_events; j++) {
            printf("%8.8x %8.8x %8.8x %8.8x
",
                kernel_events[j].third_element,
                kernel_events[j].second_element,
                kernel_events[j].hi_time,
                kernel_events[j].lo_time,
                kernel_events[j].first_element);
        }
    }
    monitor_terminate(this_monitor);
}
```

**Figure 5-2: Simple Application Using Monitor**

Figure 5-2 shows an example application using context-switch monitoring. The application runs as a single thread. The constant `REQUESTED_SIZE` is assigned to `buffer_size`. This is the requested size of the monitor buffers allocated inside the kernel. After creating and resuming (starting) the monitor, the application enables itself for monitoring. `UNIQUE_ID` is some constant that the programmer knows is unique for each thread in the application. In this case, since there is only one thread, the value of `UNIQUE_ID` is arbitrary.

After executing a `sleep` statement 300 times, the thread is disabled for monitoring. Then a user routine is called, `print_events`, that repeatedly calls `monitor_read` and prints the events until the kernel buffer is empty. The size of the application's buffer is only `MONITOR_MIG_BUF_SIZE` events
because of a limitation of the current Mach Interface Generator (MIG) technology. Thus, the application must call monitor_read until it returns less than MONITOR_MIG_BUF_SIZE events. When this occurs, the application knows that the context-switch monitoring buffer is empty. When print_events returns, the monitor is terminated. Note that in this application, monitor_suspend is never called.

```
typedef
struct kernel_event {
    unsigned event_type;    /* type */
    unsigned first_element; /* stopped thread */
    unsigned second_element; /* started thread */
    unsigned third_element; /* flag and cpu */
    unsigned hi_time;       /* hi time stamp */
    unsigned lo_time;       /* lo time stamp */
} kern_mon_data_t, *kern_mon_buffer_t;
```

**Figure 5-3:** Context-switch Event Data Structure

The context-switch data structure saved for each relevant context-switch is shown in Figure 5-3. The members of the structure consist of the type of event (currently, only one type of kernel event is detected: context-switches), the stopped-thread and started-thread, the processor on which the threads switched, and a timestamp separated into seconds and microseconds fields. If one of the threads is unknown, its id is set to -1. The most significant bit of the processor field is 1 if that event overwrote a previous, unread event (i.e. overflow).

Figure 5-4 is schematic depicting a case in which two independent non-communicating Mach tasks have created separate MKM monitors. Each monitor is represented by a port in its parent task. Thus, the task that creates a monitor obtains rights to the port that represents the monitor; only tasks that possesses such rights can access the monitor. In our example, unless task B gives task A rights to the monitor created by B, task A cannot access it.

Figure 5-4 also shows non-intersecting sets of circular buffers allocated to each monitor for holding context-switch events. A buffer is assigned to each processor in order to eliminate contention between processors for buffers. When a thread context-switches, a software context-switch sensor detects which, if any, monitor is tracking the thread and writes an event to the appropriate buffer. Eventually, a task holding rights to a monitor will release those rights and terminate the monitor. This can be done either explicitly while the task is alive, or implicitly when the task terminates. In either case, the termination of a particular monitor is accomplished by the kernel. MKM maintains operating system integrity while providing fast context-switch sensor firing times.
Figure 5-4: General Context-Switch Monitor Architecture
5.3.3. Monitor System Calls

kern_return_t
monitor_create(my_task, new_monitor, requested_size)
task_t my_task;
monitor_t *new_monitor;
int *requested_size;

kern_return_t
monitor_resume(this_monitor)
monitor_t this_monitor;

kern_return_t
monitor_suspend(this_monitor)
monitor_t this_monitor;

kern_return_t
thread_monitor(this_monitor, unique_id, this_thread)
monitor_t this_monitor;
int unique_id;
thread_t this_thread;

kern_return_t
thread_unmonitor(this_monitor, this_thread)
monitor_t this_monitor;
thread_t this_thread;

kern_return_t
monitor_read(this_monitor, buffer, events_read)
monitor_t this_monitor;
kern_mon_buffer_t buffer;
int events_read;

kern_return_t
monitor_terminate(this_monitor)
monitor_t this_monitor;

Figure 5-5: Monitor System Calls

Figure 5-5 lists the syntax of the monitor system calls. Monitor_create() creates and initializes memory for a new context-switch monitor. It then returns the rights to the corresponding monitor port to the caller. The initial rights belong to the task that created it and any subsequent calls to the monitor will be valid only if the caller has rights to the monitor's port. The monitor_create() call is also passed the size requested for the monitor buffer used to hold context-switch data. The call returns a size rounded up to completely fill a set of pages in memory. A monitor and the size of the monitor buffer that was allocated is returned.

The created monitor is returned in a suspended state and does not record any context-switch events until monitor_resume() is called. Monitor_resume() sets the state of a monitor so that it may detect context-switches. It also sets special monitor flags of all threads associated with the monitor (see thread_monitor() below) to permit its context-switches to detected. Monitoring is suspended by monitor_suspend() which sets the state of a monitor so that no context-switches can be detected. It also sets the monitor flag of any thread associated with the monitor to indicated that its context-switches
cannot be detected. A monitor can be suspended and resumed repeatedly by successive calls to monitor_suspend() and monitor_resume().

To monitor the context-switches of a particular thread, thread_monitor() is called. The caller must pass a unique integer id to thread_monitor() in order to be able identify the thread when context-switch events are retrieved by monitor_read() (see below). The caller is responsible for keeping the id unique among threads. During context-switches a sensor checks the monitor state of each thread and fires if either thread is enabled for monitoring, timestamping and storing the event in respective buffer of the monitor observing each thread. A thread can be observed by only one monitor at a time. To disable monitoring of a thread’s context-switches, thread_unmonitor() is called. Both calls require a monitor and a thread.

To retrieve recorded context-switches, the user calls monitor_read(). This call reads all the relevant context-switches that occurred since the first time monitor_resume() was called or since the last time monitor_read() was called. Monitor_read() takes a monitor, a buffer for holding context-switch data and an integer into which the monitor returns the number of events that were read. If monitor_read() is not called often enough, some context-switch events might be lost if the context-switch monitor buffers fill and overflow. Correcting this requires either creating a new monitor with larger buffers or making more frequent calls to monitor_read(). The user can make as many simultaneous calls to monitor_read() as the user desires; the context-switch ensures that mutual exclusion of any critical sections. Each call returns no more than approximately 8k bytes of data, a limitation of the current MIG technology.

A monitor is terminated by monitor_terminate(). This call first breaks communication between the caller and the designated monitor. Then it places the monitor in a shutdown state which forces each thread holding a reference to the monitor to enter a null monitor state the next time it context-switches. At that time the thread removes its reference to the monitor. Before the last reference removed, the holder of the reference removes all the dynamic data structures belonging to the monitor.

5.4. Interpreting and Presenting the Data

Although the PIE retrieves a wealth of data about computations, correlating the kernel and user level data for display on PIEscope’s cpu and baroscope views has been complicated by possibility of both incomplete and excessive context-switch information. Depending upon precisely when a thread is enabled for context-switch monitoring, the monitor may retrieve data that makes it appear as if a thread has switched off/onto a processor without ever switching onto/off it, thus complicating the determination of what processors such a thread ran on. The basic theme of the solution to this problem of undetected and superfluous context-switches is that one can infer a context-switch from user level information. Thus, if some user level information indicates that a thread has begun but there is no kernel level information indicating a context-switch onto any processor for that thread, the PIEscope assumes that such a switch has occurred.
Aside from the problems of incomplete or unnecessary information, the sheer amount of information retrieved by the monitoring system presents problems for users in assimilating it all. Being able to pinpoint when and on what processor a context-switch occurs may be no value, for example, if a user merely wants to know how much parallelism a computation is able to muster over the course of its execution. Thus, in addition to the elaborate barscope and cpu views showing execution traces, PIEscope permits generation of more conventional performance views such as histograms and scatter-plots. Although the author was instrumental in the architecture of these solutions, the bulk of their implementation was done by PIE’s principle implementer of PIEscope, Eddie Caplan.
Chapter 6

Monitoring Examples: With Notes on Perturbation

In order to see more clearly how monitoring perturbs computations, several examples of monitoring are presented. The first is monitoring on a sequential machine (a VAXstation II) and the others on an Encore Multimax. The examples are convincing illustrations of the value and power of PIE, and of the problems of perturbation. With varying degrees of subtlety, each example discusses what is shown in the corresponding PIEscope views followed by comments on what perturbation is present. When studying the examples, it is advisable to distinguish between the performance of the application and the actions of the scheduler.

Views like those shown in these examples have been used by the designer of Mach schedulers\(^4\), to review and evaluate their performance. The first two examples are based upon discussions appearing in an article in IEEE Computer [25].

6.1. A Scheduler’s Evolution: A Case History

*Nothing else comes close to exposing the warts in the scheduler.*

- David Black on the Mach context-switch monitor

The four views in Figures 6-1 and 6-2 are PIEscope displays of a matrix multiplier application executing on four slightly different versions of Mach with context-switch monitoring. This example:

- Shows that PIE-like visualization and monitoring is useful for evaluating scheduling performance.
- Shows that while monitor perturbation of an application may be severe the simultaneous perturbation of scheduling can be minor.

Each version was booted on a VAXStation II. In the figures, black rectangles represent periods when the threads are running. White rectangles represent the periods when they are not. Because the views depict uniprocessor executions, cutting a vertical swath through a view at any point slices through only one running thread ... only a single black rectangle. Each view contains a pair of time cursors which

---

\(^4\)David Black, of the Mach project, was the principal architect of the schedulers.
Chapter 6

Monitoring Examples: With Notes on Perturbation

In order to see more clearly how monitoring perturbs computations, several examples of monitoring are presented. The first is monitoring on a sequential machine (a VAXstation II) and the others on an Encore Multimax. The examples are convincing illustrations of the value and power of PIE, and of the problems of perturbation. With varying degrees of subtlety, each example discusses what is shown in the corresponding PIEscope views followed by comments on what perturbation is present. When studying the examples, it is advisable to distinguish between the performance of the application and the actions of the scheduler.

Views like those shown in these examples have been used by the designer of Mach schedulers\(^4\), to review and evaluate their performance. The first two examples are based upon discussions appearing in an article in IEEE Computer [25].

6.1. A Scheduler’s Evolution: A Case History

*Nothing else comes close to exposing the warts in the scheduler.*

- David Black on the Mach context-switch monitor

The four views in Figures 6-1 and 6-2 are PIEscope displays of a matrix multiplier application executing on four slightly different versions of Mach with context-switch monitoring. This example:

- Shows that PIE-like visualization and monitoring is useful for evaluating scheduling performance.
- Shows that while monitor perturbation of an application may be severe the simultaneous perturbation of scheduling can be minor.

Each version was booted on a VAXStation II. In the figures, black rectangles represent periods when the threads are running. White rectangles represent the periods when they are not. Because the views depict uniprocessor executions, cutting a vertical swath through a view at any point slices through only one running thread ... only a single black rectangle. Each view contains a pair of time cursors which

\(^4\)David Black, of the Mach project, was the principal architect of the schedulers.
measure the time between them, and a monitor thread, the collector, which is created and terminated by main independently of the application using special libraries linked with the application at compile time. As noted earlier, the collector writes the data recorded by event sensors onto disk.

**Figure 6-1:** Matrix Multiplication Executions:
Top - Original Kernel;
Bottom - Prototype Kernel

The top view Figure 6-1 depicts an execution on a kernel officially labeled as **XF29** but which henceforth is referred to as **Original** for easier reference. The bottom view depicts the same application on a less primitive kernel designated as **CS3c**, hereafter referred to as **Prototype** since it has several advance features shared by the later two kernels. The top view of Figure 6-2 shows the application executing on a kernel very similar to Prototype but with an improved thread priority evaluation policy. Its official name is **CS5a**, but is referred to it as **Improved**. The official name of the kernel used in the bottom view of Figure 6-2 is **X96**, but is called **Advanced** hereafter.

---

5Because the VAXstation II clock is accurate to only 10000 microseconds, times displayed by PIEscope that are less than 10000 microseconds are estimates.
6.1.1. Spawning: The Advantages of the Prototype Lineage

The views show that, despite some similar behavior, the application does not behave identically on any of the four. Figure 6-1 shows quite clearly that Prototype, Improved and Advanced successively spawn the first three multiply threads more quickly than Original. They also show that the latter three kernels eliminate the context-switch "flutter" where a thread frequently context-switches to itself that occasionally occurs in Original just as a thread is forked off. The top view, for example, shows multiply threads 2, 3, and 4 undergoing considerable context-switching soon after they are spawned by Original (as evidenced by the "squiggles" in those threads). On each of the kernels, there is regular, uniform duration context-switching among all the threads the first moments after the multiply threads are spawned. After a while, however, Prototype, Improved and Advanced (Advanced, especially so) gradually schedules each thread for longer periods before switching them out. The schedulers of the latter three kernels use a progressive algorithm which gradually increases the time slices it allocates in order to diminish the number of context-switches.
6.1.2. Time Hogs: Correcting the Problems of the Prototype Lineage

Despite its advantages Prototype has problems equitably time sharing threads. The bottom view of Figure 6-1 shows that just before the first multiply thread is forked off, Prototype switches out the collector and does not run it again until nearly all the other children terminate. To compound the problem, the progressive algorithm used by Prototype, Improved and Advanced risks delaying thread execution. The first multiply thread is forked off by main, not by the collector. After spawning the thread, main does a join and switches out, waiting for its children to finish. Prototype does not permit the first multiply thread to run until after giving the collector an execution time-slice of over four seconds. Improved and Advanced are not as delinquent, they delay spawning by only about one and three seconds, respectively. Original, however, forks the first multiply thread after only three-fifths of second. The thread priority evaluation policy of Improved and Advanced compromises between Original and Prototype. The latter two time share the collector with the multipliers as does Original, but they attempt to give larger time slices like Prototype. As just noted, they extend the compromise by reducing Prototype’s delay in forking off the first multiply thread.

Although Original suffers from context-switch flutter just after spawning threads, Prototype and Improved seem to suffer similar flutter at different times. The views in Figure 6-3 show the first seconds of the start of each of the executions, revealing that the "squiggles" in Figures 6-1 and 6-2 are hiding different behavior on each of the kernels. The "squiggles" in the bottom three views correspond to context-switches, the duration of which are beneath the resolution of views as shown. The views show that although the progressive algorithms of Prototype and Improved fail to significantly reduce the number of context-switches, Advanced succeeds in doing so. Most of context-switching shown the the views for Prototype and Improved is primarily due to context-switch "flutter" where the threads context-switch to themselves. The remaining context-switches for these kernels as well as all the context-switches for Advanced are due to system interrupts (I/O with a console, for example) or genuine descheduling so that another thread may run. Advanced retains the advantages of progressive algorithms of Prototype and Improved while reducing the the number of context-switches.

6.1.3. Perturbation Considerations

These views show the simultaneous phenomena of significant monitoring perturbation of an application, and inconsequential perturbation of a kernel level action such as scheduling. The inline code and the collector of monitor perturb the application's performance by competing for the single processor of the machine. The effect of this competition is not only a dramatic increase in the application's execution time but also the possible re-ordering of the events of the application. The collector is the most significant source of the application's perturbation because of its relatively large execution time slices as well as its status as schedulable thread. Thus the collector risks perturbing the scheduling of an application. The collector does not, however, perturb the algorithm used in scheduling the threads. Because the collector is merely another schedulable thread, it does perturb the performance of the scheduler.
Figure 6-3: The Start of Executions: Original Kernel (top); Prototype Kernel; Improved Kernel; Advance Kernel (bottom)
Much of the monitor's perturbation of the application can be detected and compensated. The lengths of the time slices allocated to the collector are known as well as the firing points of context-switch and application sensors. Using approximate firing times of sensors and the scheduling of the collector, the execution times of the monitored sequential application can be compensated so that a more realistic estimates can be achieved. Later a technique is shown where PIEscope views of sequential applications are compensated by removing the collector and subtracting sensor execution times.

6.2. Saturating a Multiprocessor

Figure 6-4 shows two PIEscope views of parts of two matrix multiplier executions executing simultaneously on a 16-processor shared-bus computer, an Encore Multimax, running a version of Mach circa late 1988. Figures 6-5 and 6-6 show different views and periods of the same executions. The execution presented by the top views of the figures is referred to as Execution I. The execution in the bottom views of the figures is referred to as Execution II. Because the views depict executions on sixteen processors, cutting a vertical swath through a view may slice through as many as 16 running threads, 16 black rectangles. To facilitate the comparison of the views, the time-stamps of the executions are normalized to each other. That is, a time-stamp of 26.300000 seconds in one execution, for example, is simultaneous with a time-stamp of 26.300000 seconds in the other. The time stamps at the beginning and end of the PIE views in Figure 6-4 are identical, indicating that the views are snapshots of the same period in time. A similar pairing of time stamps is done in Figures 6-5 and 6-6. The interpretation of these views is restricted to how the run-time parameters of the executions affect how much parallelism they receive and how the kernel's scheduler reacts to the periods when the two computations spawn many more threads than there are processors available to run them. The discussion is not an exhaustive critique of the performance of the computation and scheduler. This example:

- Presents another case where PIE-like visualization and monitoring is useful for detecting degenerate scheduling behavior.
- Shows what can happen when applications execute with poorly designed thread creation code on a system with a weak scheduler.
- Shows a weak scheduler trying to time share many more threads than there are processors.
- Introduces the PIE cpu-barscope for viewing thread-to-processors assignments.
- Shows the perturbation of computations upon each other.
- Shows that a monitor thread does not always perturb computations when the thread is running.

6.2.1. Background Information

The discussion is restricted to the periods when the matrices are actually multiplied since these periods exhibit the most interesting behavior. The time cursors in the views in Figure 6-4 mark the approximate moment when Execution II commences multiplying its matrices. The figures show that various threads:

- Run without ever being switched out, especially in the later stages of the executions.
• Occasionally suffer context-switch flutter, as is clearly shown by the scattered periods of dense "squiggles."

• Are switched out for long periods before being allowed to run, as in the case of thread number 13 in Execution I in Figure 6-4.

6.2.1.1. The Scheduler

Although a thorough understanding of these phenomena is not possible without intimate knowledge of the scheduling policy and what other threads are running on the machine, to understand what follows only a few underlying characteristics of the kernel must be known:

• On this particular kernel one processor is designated as a master processor on which all thread creation and other special work, such as certain disk accesses, is done.

• If all processors are allocated and a new thread is spawned, the scheduler may not run the thread right away, apparently waiting to see if an existing thread finishes before adding the new thread to the run-queue.

• If at any time there are more threads than processors eligible to run, the scheduler will usually first context-switch threads that have been executing for the longest times.

By knowing these characteristics as well as the approximate number of threads vying for processors, we will be able to discern the cause some general phenomena like the context-switch thrashing occurring immediately after the time cursor in the top view of Figure 6-4.

6.2.1.2. The Run-time

By the time the two executions in Figure 6-4, terminate, each will have created 34 logical threads (main, collector and 32 multiply threads). Because of the architecture of the target system and the needs of the instrumentation a collector thread is needed), however, invisible PIE run-time parameters restrict the number of scheduled physical threads each execution may have to 17. As will be shown shortly, since the surplus logical threads must wait to execute until a physical thread becomes available (upon the termination of another logical thread, for example), the limit of 17 is too small.

6.2.2. Execution I Begins Spawning

Figure 6-5 shows zoom views of both executions. Execution I (in the top view) is the first to spawn its multiply threads. Let us begin by looking at Execution I at the time immediately after this spawning. Here, for about a second or two, several of its threads are executing unencumbered by context-switches, while others are either switched out, the collector and multiply thread number 13 for example, or switching repeatedly as in the case of threads 14 and 15. Execution I runs fourteen threads most of this time. The bottom view in Figure 6-5 shows that Execution II executes only two threads, its own main and collector, during this time. Together the two executions saturate the sixteen processors of the computer. The top view shows that Execution I has more threads eligible to run than actually can run. For example, after thread 13 is forked off, the scheduler immediately deschedules it. Since each of the other processors is occupied the scheduler hangs the thread. Whether or not the scheduler is doing a good job here is a question we address later.
Figure 6-5:
A closer look at the multipliers
Top: Execution I
Bottom: Execution II

We are examining both executions during the period immediately after Execution I spawns its multipliers. As threads are spawned it becomes more difficult for the scheduler to make equitable assignments. The top view of Figure 6-5 shows that soon after Execution I spawns its multipliers, the scheduler resumes running the collector and then thread 13. After resuming the collector, the scheduler iterates through the
threads of Execution I, switching out one thread at a time for short periods, all the while keeping thread 13
switched out. Eventually thread 13 is run just before Execution I’s initial group of multiply threads finish
executing. Since Execution II has not begun vigorously competing for processors (it has not yet spawned
its multipliers), the scheduler quickly begins assigning a new group of Execution I’s multipliers to
processors. Although the logical threads associated with these new multipliers were created about the
same time as the initial group of multipliers, each has been waiting all this time for one of the original
logical multiply threads to relinquish the physical thread it is tied to.

6.2.3. Execution II Begins Spawning

Figure 6-5 shows that Execution II begins spawning its own multipliers near the first time cursor in the
bottom view. This is shortly after Execution I begins spawning its second group of multipliers. At this
point, there are more physical threads vying to run on the Multimax than there are processors available to
run them. Since Execution I has been the dominant user up to this point, the scheduler firsts imposes a
heavy context-switching toll on it. Execution II’s threads are either switched out for a long time or run
unhindered. After a while, however, both executions begin to “thrash” quite heavily just after the second
time cursors in the top and bottom views of Figure 6-5. The term “thrash” is appropriate because it calls
to mind the thrashing that occurs in virtual memory systems when there are too many users demanding
too many pages. Such scenarios often confound paging algorithms and cause them to repeatedly swap
pages to and from disk. The heavy context-switching occurring here consists of three dominant
occurrences:

1. switching reminiscent of the “flutter” described in the uniprocessor executions,
2. interprocessor exchanges of threads where two processors simply swap the threads they
are running and,
3. "genuine" descheduling where a thread is switched out for a period of time.

Eventually, after this period of heavy competition the number of threads contesting for the processors
diminishes and the scheduler is able to assign the threads without thrashing them.

6.2.4. Processor Assignments

PIE supports a supplementary execution view, called the cpu barscope, that shows processor assignment
during the execution of a computation. The top view in Figure 6-6 is cpu barscope for Execution I during
the same block of time shown in Figure 6-5. The bottom view is the cpu barscope for Execution II during
the same period. Time is measured in seconds on the horizontal axis. The vertical axis orders the
processors used by the computation. Opposite each cpu are alternating sets of patterned rectangles. A
patterned rectangle represents an identifiable executing thread while a white rectangle is a period when
none of the computation’s threads are running on the associated cpu. PIE allows a user to arbitrarily
assign unique colors or patterns to as many threads per cpu-view as he wishes. Being able to assign a
unique color to a thread allows a user to visually track its individual scheduling history. Examining all the
threads in both views, it is obvious that before Execution II begins spawning its multipliers, Execution I
has almost exclusive use of the 16 processors on the machine. After the spawning, however, each view shows only a fraction of the processors are allocated to each execution.

**Figure 6-6:**
Selected cpu-views of the Executions
Top: Execution I
Bottom: Execution II
6.2.5. Improving the Execution of the Application

Many of logical threads spawned later in these executions have to wait to run until previously spawned logical threads relinquish the physical threads they are tied to. This problem is the result of the run-time libraries limiting each execution to only 17 physical threads. Several of the first multipliers in an execution spawn children for whom they must wait and thus can not relinquish their physical threads until their entire lineage terminates. This reduces the number of available physical threads, limiting the parallelism available to the several phases of the application. These PIE views led to a modification of both the application and the run-time support code in order to permit as much parallelism as the machine could provide.

Figure 6-7 shows two PIEscope views of parts of two improved matrix multiplier executions executing simultaneously on a Multimax running a version of Mach circa spring 1990. The execution in the top view is referred to as Execution III. The execution in the bottom view is referred to as Execution IV. The views show the effects of changes to two objects:

1. **Run-time libraries**: The run-time libraries used by these executions permit preprocessor specification of the maximum number of physical threads to be used by the application. This permits the logical threads to run as soon as they are spawned, correcting the delays experienced by Execution I and Execution II. The run-time libraries also use a different mechanism to assign execution time ids to threads. The mechanism accounts for most of the "squiggles" shown near the time cursor in Execution III. When a thread tries to get its execution time id, it has to test a counter lock. If the lock is free, the thread grabs the lock, reads and increments a counter and then releases the lock. If the lock is already held by another thread, the thread switches itself out and waits until the scheduler runs it again. In most cases, however, the scheduler switches the thread back in immediately so that the thread repeatedly switches in and out several times. This phenomenon is discussed more fully in Section 6.5.

2. **Mach Scheduler**: The scheduler which time-shares Execution III and Execution IV contains a number of improvements over the scheduler which scheduled Execution I and Execution II. It is more equitable in how it deschedules threads. There are no cases where one thread is switched out for a long time while others run for a long time. For example, neither Execution III nor Execution IV have threads suffering from the kind of temporary starvation experienced by threads 13 - 16 of Execution II shown in the bottom view of Figure 6-5. Although not clearly shown in Figure 6-7, the scheduler improvements reduced scheduling induced "flutter" as well as the number of interprocessor exchanges of threads where two processors simply swap the threads they are running.

6.2.6. Monitor Perturbation

In the cpu barscopes in Figure 6-6 for Execution I and Execution II, the respective collector threads have been assigned a unique pattern permitting easy intuitive evaluation of their perturbations. When the collector is running, it is possible that a non-running application thread could be running in its place. Even when the collector does run, it may not perturb an execution if there are no other threads eligible to run. Some of the scatter of collector slices that ran on the master processor, number 36, may not have perturbed the application at all since there seems to a number of cases where the master is barren,
Figure 6-7: Top: Execution III, Bottom: Execution IV
suggesting the application threads were not eligible to run on it. An application thread might have been a viable competitor for the master cpu if had some terminal or disk I/O to perform, just as the collector was doing. Collector perturbation is not the main source of performance degradation of Executions I, II, III or IV. The most damaging degradation comes from perturbation the executions impose impose upon each other.

These two factors, extra-computation performance degradation and the possibility that a running collector slice does not always perturb its application when there are application threads eligible to run, complicate reconstruction of the execution.

6.3. Gang Scheduled Computations

Mach's scheduler for the Encores has been improved considerably since then snapshots in Figures 6-4 through 6-6 were taken in late 1988. In addition to reducing the preponderance of the context-switch "thrashing" as shown in Figure 6-7, the current Mach kernel features a gang scheduler [5] which allows users to request sole use of up to three quarters of all the processors on a machine for varying lengths of time. This example:

- Helps to corroborate the performance of the gang scheduler.
- Shows that a monitor thread does not perturb computation scheduling if it does not compete for processors.
- Shows that when a monitor thread competes with a computation for the processors of a parallel machine the nature of the perturbation is ambiguous.

6.3.1. Perfect Parallelism

Figure 6-8 shows one execution and two cpu barscope views of the execution of matrix multiplication application that requests nine processors just before the multiply threads are spawned. The top view is the execution barscope showing a stage in which an input file is read (the long period when only main and the collector are executing) and multiplication stage of the execution. All "squiggles" in this view are context-switches. Aside from some switching at their start and end, the multiply threads execute without a single context switch. The context-switches in main while reading the input file are because main issues its read requests faster than they are satisfied by the disk controller. Thus the scheduler periodically switches out main until the controller catches up. In cpu barscope in the second view of Figure 6-8, main is the thread alternating between processors five and 13 during this file reading stage. The long, dark diagonally slashed bar during this stage of the computation is the collector.

The time cursors in this cpu barscope approximately delimit the block of time depicted by the cpu barscope view beneath it. The cpu barscope in view three of Figure 6-8 is a zoom view around the time when the nine processors are requested and the multiply threads are spawned. The view clearly shows the symptoms of the overhead of the processor allocation and thread spawning. They do not, however, reveal the specific nature of the causes behind this behavior.
Because the number of processors used by the execution matched the number threads of the computation and monitor, the collector does not compete for processor resources. Thus any perturbation stemming uniquely from its need for a processor, scheduling perturbation for example, is negligible. The collector could still perturb the computation if, for example, it were unable to keep event buffers unfilled thereby forcing a thread to occasionally pause during sensor firings. Even though collector perturbation is negligible, the execution still suffers from perturbation due to the overhead of normal sensor firings.

6.3.2. Incomplete Parallelism: When Threads > Processors

The top view of Figure 6-9 is an execution barscope of part of the multiplication section of a larger, more parallel execution of a matrix multiplication application that requests 12 processors for seventeen threads using the gang scheduler feature of the kernel. The bottom view of Figure 6-9 is the corresponding cpu barscope. As in the cpu barscopes of previous figures, collector slices are shown as dark diagonally slashed bars. Because only 12 processors are used for 17 thread, the collector periodically takes a twelfth of the processing resources allocated to the monitored computation.

The time cursors both views delimit the execution of one collector slice in order to point out a particularly onerous perturbation. Looking at the execution view, since only 12 processors are used, the scheduler switches out multiply threads 5 and 10, and postpones running multiply threads 15, 16 and 17 until previously spawned threads have executed for a while, including the collector slice delimited by the time cursors. Because thread 5 is switched out, its spawning of thread 17 (the "squiggle" right after thread 5 switches back in is the spawn event) is delayed. It is obvious that the collector is perturbing the computation, but it is unclear how to compensate for it. There are at least three scenarios possible if the collector were not present:

1. Thread 5 does not switch out, thereby spawning thread 17 earlier and making it eligible to run sooner. Since thread 5 would be the 12th thread, thread 17 probably would begin execution much sooner that it does in the monitored computation.

2. Thread 5 remains switched out, but thread 15 or 16 would be permitted to begin over one second earlier.

3. Thread 10 does not switch out. This probably would have little effect on the begin times of threads 15, 16 and 17.

Of course, all of these scenarios would have cascading effects upon the behavior of the remainder of the computation.
1: Execution Barscope Showing Gang Scheduled Execution

2: Corresponding Cpu Barscope With Time Cursors for View 3

3: Zoomed Cpu Barscope Showing Gang Invocation Overhead

Figure 6-8: Execution and Cpu Barscopes of Complete Gang Scheduling (With Successive Zooms)
Figure 6-9: Execution and Cpu Barscopes of Incomplete Gang Scheduling of Execution

6.4. When the Computation and Monitor "Resonate"

Occasionally, the actions of the computation and monitor can appear to move in synchrony. For example, whenever the computation does action C and the monitor does action M they execute in close proximity to each other, as if each exercises the "forcing function" of the other. This example shows:

- A perturbation case where the application is not only delayed but its some of actions seem to be timed in synchrony with monitor actions.

- A case where although the perturbation is detectable, the extent of the perturbation cannot be adequately determined because of insufficient information about monitoring and computation behavior. Since the extent of the perturbations cannot adequately determined, it is not practical to compensate for them.
1: Execution Barscope Showing Collector Flushes of Disk Buffers

Running on CPU 37 (ambiguous): activity 'collector', frame = 1, btime = 0.036096, etime = 277.056440, time = 277.092352
Running on CPU 29 (ambiguous): activity 'collector', frame = 1, btime = 0.036096, etime = 277.056440, time = 277.092352
Running on CPU 29 (ambiguous): activity 'main', frame = 0, btime = 0.000000, etime = 277.056440, time = 277.056440
Running on CPU 37 (ambiguous): activity 'collector', frame = 1, btime = 0.036096, etime = 277.056440, time = 277.092352

2: Same Execution Barscope with Context-Switch Information Added

Running on CPU 20 (ambiguous): activity 'collector', frame = 1, btime = 0.036096, etime = 277.056440, time = 277.092352
Switched from CPU 20 to CPU 31 kernel 'context switch'; frame = 277; btime = 190.945384, etime = 190.968592, time = 0.134401

3: A Zoom View of Execution Barscope Showing One Buffer Flush

Switched from CPU 20 to CPU 31 kernel 'context switch'; frame = 277; btime = 190.945384, etime = 190.968592, time = 0.134401

4: The Corresponding Zoom View of Cpu Barscope Showing One Buffer Flush

Figure 6-10: I/O Perturbation
6.4.1. Buffer Flushes and Context-Switches

The PIE views in Figure 6-10 show the I/O output stage of Execution I which ran on a 16 processor Encore Multimax. This stage is not shown in the earlier figures showing Execution I. In this stage, the collector is reading memory and writing to disk. Main, having joined all of its multiplex children, begins performing terminal I/O. The version of the Mach operating system on which Execution I runs requires that disk and terminal I/O be done on a master processor.

The top view of Figure 6-10 is PIE an execution barscope showing the main and collector threads. Although not clearly shown, main is printing data to a terminal. The collector is flushing buffered event data to disk. Each disk buffer flush by the collector is marked by a "squiggle". Because terminal I/O by main was not monitored there is no event information displayed for showing each time main writes to the terminal. The second view in Figure 6-10 is the same as the top view but with context-switch information added. It shows that main is often switched out (the amount of "white space" in its execution bar is a testament to this). The "squiggles" interspersed along main's execution are short periods when it briefly switched back in to run. Because of the added context-switching information in the second view, the "squiggles" in the collector's execution bar represent both buffer flushes and context-switches. The bottom two views of Figure 6-10 telescope to one of these periods. These are discussed shortly.

The time cursor in the second view marks the moment just after main has joined all its multiply children and begins to print results to the terminal. Before this point, the buffer flushes by the collector occur often and sporadically. After main begins printing to the terminal, the time between buffer flushes is longer. In addition the buffer flushes seem to occur in synchrony with each flurry of context-switches between collector and main.

The third view of Figure 6-10 is a telescoped execution barscope of one of the context-switch and buffer flush points. The bottom view is the corresponding cpu barscope view. The "squiggles" in the third view represent context-switches of either main or the collector from one processor to another. In the cpu barscope, main is represented by the light grey rectangles and the collector is represented by dark diagonally slashed rectangles.

The left time cursor marks in both views the time the collector begins a buffer flush. Both views show that it is quickly descheduled because it must run on the master cpu, CPU 36. The master processor is the processor on which all I/O is done; it is the processor that the threads are assigned to when they do their respective I/O. Although the buffer flush causes the collector to switch out it must wait until main relinquishes the master. Apparently main is doing terminal I/O at this time. It is not clear how the collector's buffer flush affects main except that the synchrony of the flush and main being run on the master cpu regularly recurs.
6.4.2. Who perturbed Whom?

It is not clear how the behavior of main would differ without collector. Do the buffer flushes cause main to context switch or is this just a coincidence? This thesis does not address perturbation of this kind for determining its extend requires information about the I/O activity of both the computation and the monitor: information not available with the current monitoring facility.

6.5. Accuracy Perturbation under High Resolution

This example shows:

- A perturbation case where event duration approaches the resolution of the sensors when sensor overhead constitutes a large proportion of the measured execution time of the phenomenon.
- That, despite significant perturbation, the sensors detected interesting and useful information.

The execution and cpu barscopes in Figure 6-11 are high zoom views of part of a 16 thread matrix multiplication execution that ran on an Encore Multimax. They show a degenerate interaction between the mpc run-time code and the scheduler in which several threads, 2, 3 and 5, "trash" on two processors. Here is an example of the problems of sensor perturbation under high zoom. Each of the slices shown in the cpu barscope are between 700 and 2000 microseconds long. The sensors that detected these, however, fire in about 100 microseconds\(^6\). One context-switch sensor fires per processor. Thus there is about 100 microseconds in each slice to be accounted for\(^7\). Although the behavior depicted in in Figure 6-11 is meaningful (there's a degenerate user/kernel interaction occurring), the accuracy of the reported local behavior is compromised by the firing time of the sensors. Despite the lack of accuracy, however, these views help to detect and isolate the source of a misbehaving run-time locking protocol.

6.6. Summary of Perturbation

These examples have illustrated three basic kinds of monitoring perturbation:

1. Perturbations that alter the execution times of computations and their constituent parts.
2. Perturbations that alter the behavior and scheduling of computations and their parts.
3. Perturbations that distort the accuracy of reports of local behavior.

One might argue, however, that the last perturbation represents only a limitation on the precision of monitoring and is not perturbation per se. Such a claim leads to a "slippery slope," where one could extend this sort of limit on "imprecision" as an apology for all sorts of monitor deficiencies. Consequently,

\(^6\) The timestamping call used on this Encore Multimax reads a bus based timer. When called infrequently, in returns a timestamp in about 5 microseconds. In cases as these, however, where sensors are firing often and close together, the timestamp requests can be delayed because the bus is sequential; only one timestamp can be retrieved at a time.

\(^7\) See Appendix B for firing times of context-switch sensors.
Figure 6-11: Execution and Cpu Zoom Views Showing Scheduling Bug

as with the two other types of perturbation just listed, these will be discussed more fully in the next chapter when a formal description of perturbation is presented.

Before moving on to the next chapter, however, it is important to note the utility of the execution and cpu views shown in the examples. Coupled with the other PIE tools discussed in Chapter 3, these views provide a very useful form of performance analysis to programmers, and scheduler designers.
Chapter 7

Software Monitor Perturbation: The Causes and Effects

A portable, selective monitoring facility like that in PIE is not merely a passive observer, unobtrusively watching computations go by. It is also an intrusive monitor, perturbing computations because it competes for hardware and software resources.

7.1. Orders of Perturbation

The examples Chapter 6 illustrated several different kinds of perturbation. In this chapter a more formal description of perturbation is given. As will be argued shortly, it is difficult, if not impossible, to completely eliminate monitor perturbations, but by measuring them, it is possible to estimate how computations would perform without them.

7.1.1. First Order

*Time* is the chief measure of how a monitor perturbs a computation. Time penalties, however, are only the immediate or first order effects of perturbations. Most commonly, first order perturbations are due blocks of inline monitor code such as sensors. Sensors may not be the only monitoring objects executing inline within a computation. A monitor may periodically perform inline bookkeeping chores or make system calls. In many cases, the perturbation of this additional inline code manifests as simple time delays, much like normal sensor perturbation. A single block of inline code is inconsequential in all but the shortest computations. In heavily monitored computations, however, pervasive sensor firings can substantially alter execution times. Because sensors are dispersed throughout a computation, the delay they inflict gradually accumulates as the computation executes.

Each time the same inline sensor code executes, its execution time may not be the same. Sensor code may or may not be cached every time a sensor fires, or worse, a sensor may invoke a page fault. Figure 7-1 is a plot of the firing times of blocks of 5000 sensors as a function of when in a computation they executed. As can be seen, the firing time of the first block of sensors is much greater than those of subsequent blocks. This disparity is most likely due to one sensor, the first one, which had to be paged in from disk before it could execute.
Figure 7-1: Firing Times of 5000 sensors vs Time

Sensor execution times probably have a multi-model distribution, with with at least two means, one for cached sensors and the other for uncached sensors [36] and perhaps one for those sensors which induce a page fault. Sensors may have "ripple" effects on the computation; when a sensor fires in the middle of a loop, code that might have otherwise remained in the cache may be flushed, only to be reloaded after the sensor finishes.

First order effects of sensors are not always a slowing down of computations. The measurements in Appendix A of the firing times of user-level sensors on an Encore Multimax show that the insertion of sensors into user code can even speed up a computation. The reasons for speeding up a computation instead of slowing it down could lie with decisions made at compile time. Software sensors may affect how sources are compiled, either benefiting or penalizing the performance of the resulting executables. More likely, however, is that sensors affect the interaction of the system busses and caches. This thesis does not solve the problem of matching sensor execution times with the kind of code it is embedded in. This is left for future work.

Monitoring code, whether inline or encased in a monitor thread executing concurrently with a computation, competes with the computation for processing time and other resources such as memory and I/O bandwidth. Concurrent or parallel monitor threads have their own states (separate program counters and local memories, for example) and thus are schedulable entities. Even if they do not inflict significant behavioral perturbation, monitor threads cause time perturbations that may be orders of magnitude greater than those wrought by inline monitoring code, especially on uniprocessor or small multiprocessor machines.
7.1.2. Second Order

Second order perturbations are event reorderings which often arise as cumulative effects of first order perturbations. As the magnitude and variance of inline delays in a parallel computation increases, the likelihood of shuffled event orders increases. Because the number of sensors per thread varies, the execution times of individual components are lengthened to different degrees and at different points in the computation. Consequently, different inline delays in the threads of a computation might corrupt the order of events and change work the computation does.

Inline monitor code can also distort paging, I/O, scheduling and synchronization behavior. For example, assume that uniformly distributed inline monitoring code increases a computation’s code size by 10%. This means that the entire text of the monitored computation requires 10% as many memory pages as it would unmonitored and only about 90% of each page contains executable computation code. Obviously, the computation will page fault more often and at different execution points relative to the text of the computation. This, of course, will affect the scheduling of the computation and, depending on its structure, quite possibly its synchronization behavior if the order of events is significantly altered.

Cumulative inline delays are not the only cause of second order perturbations. As shown in the examples in Chapter 6, second order perturbation can be caused by more pernicious sources than inline monitor code. Monitoring code may be encased in a monitor thread executing concurrently or in parallel with a computation competing with the computation the resources of the machine. In PIE, the collector thread runs concurrently with a computation. The system scheduler can cause the monitor thread to pre-empt one of the computation’s threads, causing a comparatively much longer delay than would a simple inline sensor. There is also a danger that the parallel monitor thread communicates with a computation, it might artificially sequentialize parts of the computation, causing both first and second order perturbations.

7.1.3. Third Order

Event shuffling is usually not a serious perturbation if the parallel threads of a computation do not exchange data and only synchronize during thread creation and termination. If, however, parallel threads exchange data or regularly synchronize, the accumulation of time delays and event shuffling may lead to distortions of the computation’s workload and to what Schifftanbauer calls improbable executions. Schiffenbauer [38] presents an intuitive argument for distinguishing between improbable and probable executions, asserting that event orderings captured by a monitor are acceptable if they are an ordering that is not "surprising" or improbable. Although he restricts his study the effects of a distributed debugging monitor where time perturbation is not considered except as an effect on event ordering, when applied to performance monitoring, the notion of a probable execution must be extended to include execution times that do not "surprise."

Event shuffling that distorts the workload and execution paths is call a third order perturbation. Third
order perturbations can so alter the performance of a monitored computation that it no longer is a realistic portrayal of behavior of the unmonitored computation.

Sometimes, second and third order perturbations are the result of more than just the gradual accumulation of previous perturbations. They can be induced immediately by events that are at once insidious and difficult to detect. Take, for example, the curious I/O behavior discussed in Section 6.4. It may have been that the I/O server assigned I/O time slots per task instead of per thread, so that each time the collector requested I/O, the server would also flush the computation’s I/O buffers. Conversely, computation I/O may force monitor I/O.

7.2. Recovering the Computation

Computation fragments for which monitor perturbation can be fully compensated are called recoverable; and those for which perturbation cannot be compensated are called non-recoverable. Generally, the degree to which monitored fragments can be recovered falls short of complete recovery.

7.2.1. Determining the nature of the perturbation

Most all the perturbation presented in the examples in Chapter 6 are first and second order perturbations. Determining the victims of each perturbation is often difficult because it often unclear when a perturbation has occurred as shown earlier in the discussion in Sections 6.4 and 6.3.2. It is not clear how the computation’s and monitor’s threads interact with each other. Does the collector perturb the computation? Which threads are its victims? Even if the effects of direct communication between the collector and computation are ignored, the collector may often preempt several of the computation’s threads. Would preemption have cascading effects later in the computation? Does it matter? Answering these questions requires information about parameters like the number of available processors and runnable threads as well as inter-thread communication patterns. Even if how is computation perturbed can be assessed, it is still necessary to determine whether and to what extent the perturbation can be compensated.

The ability to recover from a particular perturbation depends on how much information is known about monitoring actions like sensor firings. Because the first order effects of sensors seem to depend upon the code that surrounds it, information about sensor firings ought to include situation dependent firing times. Of course, recovering from perturbation also depends upon what kind of monitoring facilities are available. Recovering from the perturbation of a monitor thread like illustrated in Section 6.3.2 is nearly impossible without facility to monitor its scheduling. Other perturbations by software monitors are not as easily detected, limiting the recovery of computation fragments. Perturbations like altered cache hit-ratios due to inline monitor code, for example, are difficult to detect, measure and compensate, if the monitor facility is not supplemented by any special hardware components (as PIE is not). Even though the most notable perturbations arising from software monitoring come stem from competition for resources like processors,
other, more insidious perturbations can be inflicted. The addition of monitor software can change cache hit ratios and paging behavior. Even though a perturbation might certainly exist, if it cannot be detected, then it cannot be measured. If cannot be measured, it cannot be compensated. In addition, even if the local perturbations of each segment of inline monitoring code can be adequately detected, measured and compensated, if they precipitate changes in event order or scheduling behavior, their long term perturbing effects might cause large fragments of computations to be unrecoverable.

Three Basic Questions
If it is not possible to know the full extent of the perturbation by a monitor, what kind of confidence can one have in its performance reports? Certainly the confidence in the reports varies with what is demanded of them. One can usually be quite confident of the performance results of a monitor which merely measures the absolute begin and end times of a few parts of a computation. That confidence may wane, however, as the monitor tries to include more information by extending the vertical or horizontal scope of its observation. If the computation times are supplemented with information about lower level phenomena such as scheduling, cache performance and register use as well as greater detail about what the computation is actually computing, the monitor may introduce so many perturbation points, that it renders the computation unrecoverable. It seems then, that there are three basic questions:

1. What are the trade-offs in minimizing perturbation when monitoring?
2. How does one compensate for perturbation caused by monitoring?
3. How can one estimate confidence in the accuracy of the compensation?

The answer to each question is, as suggested earlier, dependent on how much parametric information is known about monitoring actions and on what kind of monitoring facilities are available. A full consideration of the first question is not object of this thesis. Rather, building upon earlier remarks, its reposing issues are discussed just enough to show that, indeed, some form of monitor perturbation is possible at any time. The issues then become moot and the answer reduces to a comment about cost-benefit analyses in monitor design and a claim about the necessity of compensation. The second and third questions are central to this thesis, and the remainder of the paper is devoted to addressing the issues constituting their answers. For the time being, however, the questions are fleshed out so that their scope is more fully understood.

7.2.2. Monitor Trade-offs: Perturbation, Cost and Portability

Perhaps the intuitive solution to minimizing or even eliminating perturbation is to build hardware support for monitoring so that inline software sensors and schedulable monitor threads are unnecessary [8] and [41]. The idea of a system of passive, non-intrusive monitoring units, quietly observing the flow of instructions and data over buses, in caches and through I/O is an appealing one. Unfortunately, as suggested by Mink and Nacht [31] and argued in detail by Reilly [36], it is an unrealistic one for a number of reasons. For existing and new architectures to be retrofitted with monitoring hardware, there are problems getting the hardware to the requisite locations. The instructions flowing into single-chip processors, for example, can only be observed from outside the chip. Many of them have on-chip
caches, like the Motorola 68030, or instruction pre-fetch buffers, making it difficult to know which instructions entering the chip are actually executed and which instructions are just along for the ride. Other difficulties in retrofitting existing architectures may be not be as mechanically overwhelming, but still quite intractable. In order to meet the goal of passivity, the monitoring may also require its own bus. The system power supply may have to be beefed up to handle the additional hardware; and the extra burden may change system timing. There may simply be a lack of slots in the cabinet rack for monitoring boards.

The opportunity may arise to design a system from chip to cabinet with fully non-intrusive monitoring. When faced with the question of what to do with a particular piece of electronic real estate, most system architects would opt for adding performance boosting hardware rather than monitoring instrumentation. In fact, if monitoring hardware itself imposes a performance penalty on the system, either in the form of slower cycle times or prohibitive cost, its prospects for inclusion into a production system are severely diminished. Reilly uses an example concerning the translation of virtual to physical addresses to illustrate the performance versus intrusion tradeoffs. Since architects probably believe that improving performance is more important than measuring it, the degree to which monitoring perturbation can be reduced depends on the architects’ cost-benefit considerations.

Finally, for both retrofitted and original equipment hardware assisted monitors, the more monitor perturbation is reduced the greater the danger of making the monitor architecture specific. As a monitor becomes more architecture specific it arguably becomes less useful as a system wide measurement tool because the expense of uniformly replicating its features over several architectures becomes prohibitive. Migrating monitoring functionality into software, on the other hand, decreases the financial outlays at the expense of greater perturbation and lower measurement resolution.

The amount of perturbation that can be tolerated and the resolution that is required depends upon the goals of monitoring. An architecture specific monitor for tracing cache footprints requires higher resolution timestamps and subtler perturbation than a monitor used for observing the creation and termination of threads or the execution times of large code segments. If both both high and low resolution events are to be observed, a hierarchy of monitors may be required. For just as it would be silly to use an electron microscope to determine whether a car is a Ferrari or a Ford, it would be unwise to invest in hardware instruction stream monitoring if one plans only to observe high level actions like thread creation. Conversely, just as one would not use a magnifying glass to identify imperfections in a silicon wafer, one cannot use low resolution software monitors to reliably measure cache performance.

7.2.3. Compensating for Perturbations

Perturbation compensation would not be necessary if perturbation could be completely eliminated or reduced in such a way that it blends in with the indeterminate ambient system "noise" that already perturbs computations. Eliminating perturbations is not viable. Reducing the perturbations so that they blend in with system "noise" like interrupts or executions of system processes is an option but is not
always easy to do. First order perturbations could blend in with this "noise" if, for example, the probability of a first order perturbation occurring at any given time equals probability of a noisy phenomenon occurring. Many times, however, monitor perturbation does not blend in with the ambient system "noise" and so must be taken into consideration when judging the performance data.

As noted earlier, this paper addresses monitors for gauging general program performance where most, if not all, of monitoring functionality lies in software. Thus perturbation such as alteration of cache hit rations is immeasurable although it is not important to compensate for such perturbation if the accuracy of monitoring is specified appropriately or it is subsumed by higher level perturbations. High level perturbation is present in such a high level monitoring system and is detectable and measurable. Therefore it can be compensated.

Some perturbations, however, can be compensated. In order to get a handle on how such a monitor perturbs a computation, several parameters about the monitor and computation must be known. For example, because a monitor may consist of one or more communicating threads executing in parallel with the threads of the computation, it competes with the computation for processors (as well as other resources). Consequently, some of the parameters which are needed in order to measure perturbation are when and how often a monitor thread context-switched, and on what processors it ran. Knowing how many sensors fire as well as their execution times is also necessary for assessing the total delay or other perturbation at any selected point in a computation.

The PREFACE environment [4] attempts two forms of compensation (the authors talk about "excluding" the overhead times) in their system. They do not count any time taken by the monitoring routine on each thread and they do the bulk of the monitoring overhead during sequential parts of the computation under the assumption that it is the performance of the parallel parts that must remain as unperturbed as possible.

In order to access whether the behavior of a computation was significantly altered by its monitor, inter-thread synchronization (communication) points must be examined. Delays caused by monitor sensors and threads are critically important at these points. By summing the delays caused by separate monitoring actions on two threads which synchronize with each other, it can be determined whether their synchronization had been interchanged by the actions of the monitor.

7.2.4. Confidence in Compensation

After compensating for perturbation, there is a question of how much confidence programmers can have in the performance information the monitors retrieve. One can make informed judgements about the performance of computations if one has some measured confidence in performance measurements. Looking at the execution times of the predicted and unmonitored asynchronous computation in Table 8-2, we can reject the predicted mean as an estimator of the actual mean with a confidence level of more than
99%. With even greater certainty, we can reject that the corresponding predicted mean of the synchronous computation because of event reordering or extended synchronization delays resulting from monitoring. Despite this, the predicted means are acceptable if they are errant to a consistent degree so that the predicted means that fall within a fixed percentage from actual means, especially if the algorithm errs to one side of the actual means. If it can be claimed that, for a selected confidence level, an algorithm like that above predicts mean execution times that approach values that are some small proportion greater than actual means, then we have a usable compensation algorithm. For programmers to have confidence in the behavior as well as the time predicted by compensation algorithms, the algorithms must be able to detect event reordering. This thesis addresses obtaining confidence in time predictions but leaves for future work methods for obtain confidence in behavioral predictions.
Chapter 8
The Work

In addition to the several kinds of perturbation, there are several classes of computations. Monitor perturbation of these is compensated in different ways. First, several broad classes of computations are presented, occasionally using excerpts from Chapter 6 as examples. After this, a compensation algorithm for sequential, asynchronous multi-threaded computations is presented. It is based on a manipulation of PIEscope execution views. Compensating sequential computations is discussed first for two reasons:

1. The reasoning behind compensation of sequential algorithms seems to be more intuitively appealing.

2. The algorithm attempts to recover both the time and behavior of computations, raising some important questions about the limits of behavioral recovery.

Then, after presenting a progression compensation algorithms for several categories of parallel computations, the results of compensation for a class of asynchronous/semi-synchronous computations executing on tightly-coupled, shared memory machines is presented. Lastly, some monitoring heuristics for getting good believable results using monitoring are discussed.

8.1. Perturbation of the Sequential Computation

Section 6.1 presented examples of the evolutionary stages in the development of the Mach scheduler.

The examples show that the most significant first order perturbations in a sequential computation occur when a monitor thread, the collector, runs in place of a computation thread. The monitor thread causes both time delays and, occasionally, event reorderings. In these examples, the monitor’s most significant perturbation on the user level is a first order one. The predominant delay from monitoring came from time slices used by the collector that would have otherwise been allotted to a computation thread.

Although, the monitor may have imposed second order perturbation on the computations, they were not significant (i.e. they were not third order perturbations) primarily because the computations only synchronized on thread creation and termination. If the computations were more synchronous and it could be shown that the monitor caused the computation threads to context switch in a different order or more often than they would have unmonitored then if the consequent non-uniform time delays assessed to the threads were large enough, the second order perturbations could have become third order.
8.1.1. First Order Perturbation by User Sensors

In order to get an intuitive feel for the extent to which user level sensors induce first order perturbations in asynchronous computations, consider a hypothetical case where a single thread computation executing on a VaxStation II consists of a loop iterating 1000 times. Let's say that the loop has enabled sensors that fire at the start and end of each iteration. There are no other sensors, enabled or disabled, in the loop. Thus, there will be 2000 user sensor firings. Let us assume that each iteration of the loop requires 200 milliseconds to execute. Let us assume also, that this computation never context-switches (an unrealistic assumption but it helps to simplify the example; context-switches will be considered shortly). Appendix A shows firing statistics for user sensors on a VaxStation II; a sensor executes in about 600 microseconds. Thus, the sensors will delay iteration of each loop by just over a millisecond (2 x 600 µs) for a degradation of about 0.25% for each iteration and thus the whole computation. The execution times of some computations, however, often vary by more than 0.25% so the delay due these sensors may be acceptable. Of course, these delay estimates are certainly not worse case, but they are an example of some the PIE group's experience in monitoring.

8.1.2. Kernel Sensor Perturbation

It is unrealistic to assume that sequential computations do not context-switch during execution, especially multi-threaded computations. Now let us consider a case where a multi-threaded computation is executing on a kernel with context-switch sensors. For simplicity, assume the scheduler is like the one used in the XF29 kernel in the example in Section 6.1, where each time slice is usually about 100 milliseconds. That is, the kernel regularly performs ten context-switches a second. Because scheduler's often suffer miscellaneous deviations from regular behavior, assume also that the kernel must context-switch an additional five times a second. A computation running under these circumstances on an MKM modified Mach kernel would incur two kinds of overhead, depending upon whether it is enabled for context-switch monitoring:

1. Sensor-firing overhead consisting of storing event information and
2. Constant overhead consisting of checking whether a switched thread is monitored.

An unmonitored computation would incur only constant monitoring overhead each context-switch while monitoring.

8.1.2.1. Perturbation of the Computation

Appendix B shows the constant overhead and firing statistics for context-switch sensors on a VaxStation II; shows that the average sensor firing time is about 191 microseconds and the constant context-switch overhead is about 17 microseconds.

What do these figures portend for the performance of typical computations? Let's assume that we are running computations on a kernel like the XF29 kernel. That is, the kernel regularly context-switches
threads ten times a second. Because there are frequently miscellaneous deviations from a scheduler's regular behavior, let's also assume that the kernel must context-switch an additional five times a second. An unmonitored computation running under these circumstances would incur only the constant monitoring overhead each context-switch. Thus, if it normally takes a computation an hour to execute, the constant overhead of the monitor code would delay it by only $15 \times 3600 \times 0.000017 = 0.92$ seconds, or less than 0.03\%. Now let's assume the same computation is running under the same conditions, but this time it's monitored. Realistically, if there are other computations running such as system utilities, not every context-switch will involve the computation. So let's say 12 out of 15 context-switches per second are monitored. Thus, if such a computation executed in an hour on an unmonitored kernel, it would require an extra $12 \times 3600 \times 0.000211 = 9.12$ seconds due to sensors firing and an additional $3 \times 3600 \times 0.000017 = 0.18$ seconds due to constant-overhead making up a total delay of about 9.3 seconds or only about 0.25\%.

These delay estimates are certainly not worse case. The experiments that measured the firing time and constant overhead, however, approached worst case scenarios. In the constant overhead experiments, for example, an average of over 2400 context-switches were estimated to have occurred in span of about 2.9 seconds (i.e., almost one every millisecond). In these experiments, the constant overhead yielded a penalty of about 1.4\%. When the sensors fired, the degradation rises to 14\%. Since most computations do not context-switch every millisecond (and if they do, there is probably something awry), severe performance penalties such as these are unlikely.

8.1.2.2. Overhead of Reading Event Information

Although delay due to the context-switch sensors is the most regularly occurring intrusion of the monitor upon computations, the events detected by the sensors must occasionally be retrieved from their buffers and stored in user space, either in physical memory or to disk. This requires making periodic monitor_read calls each requiring about 1.3 milliseconds. The effect of monitor_read calls on the computations discussed in this paper is negligible because if the case that the collector never called the routine, the collector would have merely busy waited. The length of time the collector executed in these computations was not dependent upon what it was doing, but rather upon how long the multiply threads executed. Thus the delays due to sensor firings were the dominant intrusion upon the computations. Work is underway in PIE to provides means to compensate for the intrusion of the sensor firings and monitor threads.

8.1.2.3. Sensor Perturbation of Scheduling

Do the context-switch sensors perturb the scheduler of the computation? When a context-switch occurs in an MKM kernel, the time penalty caused by the sensor can normally be assessed to the thread switching in because the clock timing its time slice starts before the context-switch sensor fires. That is, when a thread is allocated a time slice of $r$ microseconds, the amount of time it executes in user mode will be no greater than $r - t_{fs}$ microseconds ($t_{fs} =$ context-switch sensor firing time).
The sensors could then change the scheduling of a computation if enough of them fire so that the computation requires additional time slices. The number of additional slices, however, would be small. Consider the simple case where the average time slice is 100 milliseconds. If the context-switch sensors subtract about 200 microseconds of execution time from each time slice, the computation would have to undergo about 500 context-switches, that is, it would have to execute about 50 seconds, before the sensor firings would delay the computation long enough for it to require an additional 100 millisecond time slice. Thus perturbation of scheduling by the context-switch sensors can be considered small.

8.2. Compensating for Perturbation of Sequential Computations

Although individual sensors do not severely perturb execution times or scheduling of reasonably sized computations, they do alter the computation’s performance and therefore must be compensated. More severe perturbation of both the execution time and the scheduling of computation comes from schedulable monitor threads. Compensating for many firings of user and kernel sensors as well as execution of monitor threads requires a more formal treatment in order to be confident of the predicted execution times and behavior.

8.2.1. Compensation by Trimming Trees

How does one begin to compensate for monitor perturbation? Figure 8-1 shows two PIE execution barscopes depicting the same uniprocessor execution of a computation before and after compensating for perturbation using a fairly simple set of heuristic operations on the PIE data structure representing the computation. PIE represents the execution of a computation in sets of tree-like graphs. The graphs are not formally trees because they contain cycles, but since they have a hierarchy of nodes, thinking of them as trees has been helpful in visualizing their representation. Each tree represents the execution of one thread of the computation. The nodes of tree represent procedure calls, iterations, et cetera, depending upon what the user monitored. The leaves are context-switches. The tree is construct so that, for example, if one wanted to know how many context-switches occurred within a particular iteration, all one must do is traverse the tree to the node representing that iteration (there will be begin and end timestamps stored there) and then continue traversing under that node. Everything that occurred within that iteration is stored underneath the iteration’s node. The compensation heuristics presented here are based on trimming this tree of unwanted context-switches (context-switches with a monitor thread, for example) as well as adjusting the timestamps at the nodes due to sensor overhead. The time compensation component of the heuristics consists of merely adjusting the timestamp of each event by subtracting from its time stamp the sum of all previous sensor firing times plus any time consumed by monitor threads like the collector. The tree trimming component of the heuristics can be thought of as a rearranging of the scheduling by deciding whether certain context-switches by monitor threads should be removed or simply collapsed.

The top view in Figure 8-1 shows the tail end of a matrix multiplication executing on a VaxStation II. The
Figure 8-1: Uncompensated and Compensated Views of Part of a Matrix Multiply Executed on a VAXstation II

bottom view shows the same computation with the collector and sensor times removed after compensation. When subtracting time consumed by the collector, the heuristics either removes context-switches with it or simply collapses several switches together depending upon whether consecutive switches are with the same thread or different ones.

Compensating for temporal perturbation yields a time compression delta $\Delta$, formally expressed as:

$$
\Delta = \sum_{i=1}^{N_u} s_u + \sum_{i=1}^{N_k} s_k + \sum_{m=1}^{N_m} t_m
$$

where each of the variables are declared in Table 8-1. The views in Figure 8-1 show that the heuristics compensate more than just time, it removes and rearranges monitor thread context-switches from the execution views. Formally, the action on a context-switch is specified by:
### Table 8-1: Variable Definitions

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>$s_u$</td>
<td>Average user sensor firing time</td>
</tr>
<tr>
<td>$s_k$</td>
<td>Average context-switch sensor firing time</td>
</tr>
<tr>
<td>$t_m$</td>
<td>Length of monitor thread time slice for time slice $m$</td>
</tr>
<tr>
<td>$N_u$</td>
<td>Total number of user sensors that fired</td>
</tr>
<tr>
<td>$N_k$</td>
<td>Total number of context-switch sensors that fired</td>
</tr>
<tr>
<td>$N_m$</td>
<td>Total number of monitor thread slices</td>
</tr>
<tr>
<td>$\Delta$</td>
<td>Compression Delta: Time subtracted due to compensation</td>
</tr>
<tr>
<td>$SI_k$</td>
<td>Thread that switched in at context-switch $k$</td>
</tr>
<tr>
<td>$SO_k$</td>
<td>Thread that switched out at context-switch $k$</td>
</tr>
<tr>
<td>$M$</td>
<td>Monitor thread</td>
</tr>
</tbody>
</table>

**Remove switch** when \[
\begin{align*}
SI_k &= SO_k = \text{monitor thread} \\
SI_k &= SO_k + 1 = \text{monitor thread}
\end{align*}
\]

**Collapse switch** when \[
\begin{align*}
SO_k &= SI_k + 1 \quad \text{and} \quad SI_k = \text{monitor thread} \\
SI_k &= SO_k - 1 \quad \text{and} \quad SI_k - 1 = \text{monitor thread}
\end{align*}
\]

Figure 8-1 shows examples of both removing and collapsing context-switches. The collector time slices between the two time cursors in the top view have been removed in the bottom view. Examine the collector slices between the two time cursors. The first slice switches in as thread number 6 switches out. This collector slice, however, switches out later with an unknown thread. Thus, the second action is chosen. Both of the respective context switches of the other two slices, however, switch with unknown threads. Since the heuristics treats any unknown thread as the same thread, each context-switch is removed.

### 8.2.2. The Accuracy of Compensation

The heuristics are fairly successful at predicting unmonitored execution times for some computations. Table 8-2 shows two sets of execution statistics for two computations running on a VaxStation II. The top set of statistics represents a computation consisting of several concurrent threads that share no memory and synchronize only during thread creation and termination. The bottom set represents a similar computation consisting of several concurrent threads that not only synchronize during thread creation and termination but also when writing out results. The statistics show that, after compensation, the mean execution time predicted for the unmonitored asynchronous computation is almost two percent greater the
actual mean while the predicted time for the synchronous computation is almost three percent greater than its respective mean.

<table>
<thead>
<tr>
<th>Statistic</th>
<th>Mean</th>
<th>Std. Dev.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Asynchronous Matrix Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monitored Time</td>
<td>30.59 secs</td>
<td>1.26 secs</td>
</tr>
<tr>
<td>Predicted Time</td>
<td>19.35 secs</td>
<td>0.60 secs</td>
</tr>
<tr>
<td>Unmonitored Time</td>
<td>18.96 sec</td>
<td>1.01 secs</td>
</tr>
<tr>
<td>Synchronous Matrix Multiply</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Monitored Time</td>
<td>38.78 secs</td>
<td>1.11 secs</td>
</tr>
<tr>
<td>Predicted Time</td>
<td>23.72 secs</td>
<td>0.51 secs</td>
</tr>
<tr>
<td>Unmonitored Time</td>
<td>23.04 sec</td>
<td>1.13 secs</td>
</tr>
</tbody>
</table>

*Table 8-2: Execution Times for Two Computations*

The heuristics just presented can perform good temporal compensation for monitor perturbation of asynchronous multi-threaded sequential computations. Although the heuristics attempt to recover both the time and behavior of computations, it cannot guarantee that the scheduling of the computation, as predicted and portrayed in in the altered PIEscope views, represent anything that could have really happened without monitoring, suggesting that one must be careful to not read too much into the compensated views. Certainly, since the computation shown in Figure 8-1 is highly asynchronous, the scheduling shown in the compensated view is not an unreasonable one. If, however, the computation hit more synchronization points than just thread creation and termination, the compensated scheduling (and event order) would be highly suspect. In Chapter 9, additions to these heuristics for permitting detection of serious event-reordering of synchronous computations are discussed.

8.3. Some Problems with Trimming Trees of Parallel Computations

Running multi-threaded user-level computations on sequential machines in order to improve performance is like tuning a Subaru for the Indy 500; despite any improvement one makes, the computation is still second class. Most multi-threaded computations cry out to execute on parallel machines so that their concurrent execution can become truly parallel.

The example in Section 6.1 showed the most common and immediately damaging perturbation of sequential computations are the time delays (first order perturbations) resulting from a monitor thread, the collector, running in place of a computation's threads. In computations on sequential machines, when a monitor thread is allocated a time slice only it runs. No computation threads run. Consequently, assuming the presence of the monitor thread does not change the relative scheduling order of the computation's threads, each thread is equally perturbed and event order is unchanged. In parallel
computations, however, monitor threads are poised to inflict greater damage from second order perturbations because more than one thread may execute at a time, making the time delay of any one of them potentially calamitous with respect to maintaining a realistic event order.

![Figure 8-2: A Parallel Execution View](image)

Recall the event reordering discussed in the example in Section 6.3.2. The example pointed out only the most salient case of event reordering and suggested that there were at least three ways to compensate it. If the compensation algorithm described above were extended to compensate parallel computations, what criteria would it use to select from the three alternatives in the example? Examine the execution view in Figure 8-2; it shows part of the execution of a nine thread computation for which seven processors have been allocated. The respective processor assignments of the computation are shown by the cpu view in Figure 8-3. For a brief period after the second time cursor in Figure 8-3, the scheduler allocated an eighth processor to the computation but takes away shortly thereafter. As in the examples in Chapter 6, collector slices are represented by dark, diagonally slashed bars, like the one between the time cursors. Only the slices for two of the multiply threads are assigned special patterns: the bars with the wavy pattern is thread 8, and the one with vertical slashes is thread 3. They are high-lighted because they show some of the compensating difficulties using the algorithm described above.

![Figure 8-3: Corresponding Parallel Cpu view with Threads 1, 3 and 8 Highlighted](image)
8.3.1. Who Gets Moved Where?

Consider what ought to be done if the collector slice between the time cursors is "removed." It seems that thread 8 (the one with the wavy pattern) could be moved left to a point about midway between the cursors, so that it would have immediately switched from cpu 32 to cpu 9. This, of course, would leave some unfilled "space" where it used to "reside." What could be moved there? The solution cannot be simply sliding thread 3 to the left because it will then appear to execute on two processors (cpu's 8 and 9) simultaneously.

There is also the matter of trying to conform to the scheduling behavior already present. Figure 8-4 with different thread 5 highlighted instead of thread 3 and a pair of time cursors delimiting a period when the scheduler switched several threads during two brief periods near the first and second cursors. Perhaps, if the collector had not been present, thread 8 would have continued to run until about the first group of switches near the left time cursor. It then would have been rescheduled again a short time later near the second time cursor. It seems what would have to be done is divide up the slice of thread 8 executing on cpu 13 and move part of it to cpu 9 so that thread 8 appears to execute until around the left time cursor.

8.3.2. And What Would It Mean When You're Done

Of course, when compensating an entire computation, a set of heuristics for modifying timestamps and shuffling around threads would have to iterate and possibly backtrack in time over all the threads and processors, trying to avoid changing the measured order of synchronization points as well as trying to detect possible cases where if monitoring had been absent, the synchronization order would have been different. The result would be a rearrangement and resizing of the slices of the cpu and execution views. Although such a compensation scheme might yield possible (and even likely) execution and cpu assignments, the resulting altered view would be one of many possible views.

Extending the PIEscope based sequential compensation algorithm to parallel cases would require adding checks for synchronization order. Although PIE has means for determining dependencies for some
synchronization cases, they are inconvenient and tedious to implement. Even the synchronization dependencies that PIE can handle, such as joiner-joinee pairs are implemented well enough for easy determination of synchronization partners. Finally, because tree trimming based compensation is of dubious advantage, work on such a compensation scheme was suspended in favor of a method that can be roughly characterized as a compensation calculus based on finite element analysis.

8.4. Computation Classes

Before discussing compensation based on a kind of finite element analysis, a set of computation scenarios are described. There are several different kinds of computation scenarios which present different monitoring problems. Each of the scenarios assumes execution on tightly-coupled (shared-bus) machines with homogeneous processors and a large shared address space. Recall that gang scheduling refers to the allocation and scheduling of a computation to a "gang" of processors for variable time periods.

1. Gang scheduling, with no inter-thread communication or shared data: This is the simplest case. A fixed number of processors greater than or equal to the number of computation threads plus any monitor threads, is assigned to the allocated for the computation. Since there is no communication among the computation threads, event reordering cannot change the workload. The computation discussed in Section 6.3.1 is best approximated by this scenario because even though there is synchronization at thread creation and termination, they had little effect upon the performance.

2. Gang scheduling, with inter-thread communication or shared data: The computation is assigned a number of processors greater than or equal to the number of computation threads plus any monitor threads. Because the threads communicate with each other, event reordering may lead to unrecoverable work-load changes. If the computation of Section 6.3.1 had a more complex thread creation and termination protocol, it might have fallen into this category.

3. Partial gang scheduling, with no inter-thread communication or shared data: The computation is allocated a fixed number of processors, but the number of threads is greater than the number of processors. Because the computation threads do not communicate, event reordering cannot change the workload. Asynchronous multi-threaded sequential computations are a special case of this. The sequential, multi-threaded computations discussed in Sections 6.1 (the number of processors is one) and the parallel computation in Section 6.3.2 fall best into this category even though they experience some inter-thread synchronization during creation and termination.

4. Partial gang scheduling, with inter-thread communication or shared data: As above, a fixed number of processors is allocated, but the number of threads is greater than the number of processors. Communication among the threads means that event reordering can change the workload. a special case of this. The computations of Sections 6.1 and 6.3.2 would easily fall here if their synchronization behavior were more extensive.

5. No gang scheduling, with no inter-thread communication or shared data: No fixed number of processors allocated. Instead the computation is dynamically awarded processors based on how many are available and how many threads are competing for them. Because the threads do not communicate with each other, event reordering cannot lead to unrecoverable work-load changes. The computation discussed in Section 6.2 would fall here, but because its degenerate run-time prevent a sufficient number of physical threads to be spawned, it was particularly vulnerable to perturbation of its thread creation and termination points. Consequently, it would be better placed in the next category.
6. No gang scheduling, with inter-thread communication or shared data: No fixed number of processors allocated. The computation is dynamically awarded processors based on how many are available and how many threads are competing for them. Because the threads communicate with each other, event reordering may lead to unrecoverable work-load changes. See the comments for the preceding category.

The compensation discussed below addresses first order perturbation and does some limited treatment of second and third order perturbations regarding thread creation and termination. The heuristics address asynchronous computations and computations while synchronized only at thread creation and termination. The implementations of the algorithms handle only a subset of second order perturbations when threads are created and terminated, but there are no hard theoretical obstacles to extending the implementations to cover a wider range of second order perturbations.

8.5. Time Compensation Using Finite Elements

Compensating for time perturbation by rearranging and resizing the slices in PIEscope views is inefficient since much of the work goes into shuffling execution slices instead of deciding what threads could have run at any given time. This inefficiency could be considered a cost of "doing business" if the resulting shuffled compensation views held meaningful information. The questionable relevance of such shuffled views, however, suggests looking at other approaches to compensation.

Several compensation algorithms are discussed, each more sophisticated and general than the one preceding it. Instead of the shuffling of slices used in the PIEscope based algorithm above, they are numerical techniques which break up computations into many finite time elements for analysis. They all apply best to the first, third and fifth computation classes discussed in Section 8.4 (which include sequential computations under similar synchronization and data sharing conditions). Although the algorithms can be used to compensate computations fitting the second, fourth and sixth classes, their efficacy and reliability decreases with the amount data sharing and synchronization in the computation. Table 8-3 shows the variables used in the algorithms.

8.5.1. Algorithm I: Processors versus Threads

The premises behind the first algorithm presented are similar to those behind the algorithm presented by Sevcik [39] when he discusses "parallelism profiles" to determine the average parallelism present in a computation. Sevcik's algorithm is basically a sum of weighted parallelism ratios. For example, if half of a computation runs at a parallelism of three and half runs with parallelism of six, the average parallelism is $0.5 \times 3 + 0.5 \times 6 = 4.5$. The first algorithm here can be thought of as a modification of Sevcik's algorithm where the influence of monitoring threads is excised. As will be seen, the first algorithm has deficiencies. And as each of the parallel compensation algorithms are presented, they introduce finer methods of modeling the measured and compensated parallelism profiles.
<table>
<thead>
<tr>
<th>Compensation Variables</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A_{i}$</td>
</tr>
<tr>
<td>$A_{m_{i}}$</td>
</tr>
<tr>
<td>$N_{i}$</td>
</tr>
<tr>
<td>$S_{i}$</td>
</tr>
<tr>
<td>$S_{m_{i}}$</td>
</tr>
<tr>
<td>$s_{ki}$</td>
</tr>
<tr>
<td>$W_{s_{i}}$</td>
</tr>
<tr>
<td>$P_{m_{i}}$</td>
</tr>
<tr>
<td>$\Phi_{i}$</td>
</tr>
<tr>
<td>$i_{i}$</td>
</tr>
<tr>
<td>$I$</td>
</tr>
<tr>
<td>$T_{M}$</td>
</tr>
<tr>
<td>$p_{i}$</td>
</tr>
<tr>
<td>$\beta_{i}$</td>
</tr>
<tr>
<td>$\kappa_{i}$</td>
</tr>
<tr>
<td>$\kappa_{ki}$</td>
</tr>
<tr>
<td>$T_{c}$</td>
</tr>
</tbody>
</table>

**Table 8-3: Variable Definitions**

**Figure 8-5: Hypothetical Monitored Parallel Computation**
Figure 8.5 shows a bar-scope like view of hypothetical monitored parallel computation divided into four periods. The demarcation of the periods is based on how many threads are running. Context-switch behavior is not shown. Let's assume that the computation has exclusive access to four processors (N = 4) and none of them wait for any events (i.e., they're all crunching away). The ratio of parallelism to threads of the computation (plus monitor) in each of the four periods is:

\[ P_{m_i} = \min \left[ \frac{N_i}{A_i}, 1 \right] \]

The estimated ratio without the monitor is:

\[ \Phi_i = \min \left[ \frac{N_i}{A_i - A_{m_i}}, 1 \right] \]

Since allocating more processors than threads does not improve performance once there is a one-to-one assignment, parallelism ratios cannot be greater than one. Thus the execution time compression, \( T_{c'} \), of the entire computation after compensation can be calculated by:

\[ T_{c'} = \sum_{i=0}^{I} \frac{P_{m_i}}{\Phi_i} \cdot \frac{t_i}{T_M} \]

which, in the case of \( N_i < A_i \), is:

\[ T_{c'} = \sum_{i=0}^{I} \frac{A_i - A_{m_i}}{A_i} \cdot \frac{t_i}{T_M} \]

so that execution time of threads, \( c4 \) and \( \text{Main} \), for example, are compressed to 83% and 87% of their measured values.

**8.5.2. Algorithm II: Runnable Threads versus Running Threads**

Two problems with this first algorithm are that it does not look at the local scheduling behavior (context-switching) and that it does not deal with waiting threads. How does this algorithm handle, for example, if during the second and third periods of the computation in Figure 8.5, the machine's scheduler occasionally decided to switch out the monitor thread and give the free processor to the computation? How does it handle if one of the threads waited and switched out for a long period? In order to know how many processors a computation is running on at any given moment, you must have scheduling information about how the threads context-switch. The number of processors a computation (monitored or unmonitored) receives during any sufficiently small time interval \( t_i \) (it is important that \( t_i \) is small) is:

\[ N_i = A_i - S_i \]

thus the parallelism to thread ratio for the monitored computation becomes
\[ P_{m_i} = \frac{A_i - S_i}{A_i - (S_{m_i} + W_{s_i})} \]

The \(-W_{s_i}\) is present in the denominator because waiting threads that are switched out are not actively vying for processors. Similarly, the \(-S_{m_i}\) in the denominator is present because if a monitor thread is switched out in interval \(i\), it is not perturbing the computation in interval \(i\). The estimated parallelism to thread ratio for the unmonitored computation becomes:

\[ \Phi_i = \begin{cases} \frac{A_i - S_i}{A_i - (A_{m_i} + W_{s_i})} & \text{for } A_{m_i} + W_{s_i} \leq S_i \\ 1 & \text{for } A_{m_i} + W_{s_i} > S_i \end{cases} \]

\(\Phi_i\) goes to one when \(A_{m_i} - W_{s_i} = S_i\) because, as noted earlier, parallelism ratios greater than one do not improve computation speed. Thus the execution time compression becomes:

\[ \kappa_i = \frac{P_{m_i}}{\Phi_i} = \begin{cases} \frac{A_i - (A_{m_i} + W_{s_i})}{A_i - (A_{m_i} + W_{s_i})} & \text{for } A_{m_i} + W_{s_i} \leq S_i \\ \frac{A_i - S_i}{A_i - (S_{m_i} + W_{s_i})} & \text{for } A_{m_i} + W_{s_i} > S_i \end{cases} \]

Figure 8-6: Another Hypothetical Monitored Parallel Computation

8.5.3. Algorithm III: Processors, Runnable Threads and Running Threads

Although improved, this algorithm still has problems. Figure 8-6 shows a computation similar to the one depicted in Figure 8-5 except that it shows how the threads are scheduled. During the second period of the computation, the monitor thread runs without ever context-switching, while the computation threads are regularly switching, always with one switched out at a time. Despite how unrealistic such a scenario is, it represents a worst case intrusion by the monitor thread, so it is instructive to see how the algorithm compensates for it deal with it. We can see, that at any point in this period,
\[ W_{s_i} = 0 \]
\[ A_i = 6 \]
\[ A_i - S_i = 5 \]
\[ A_i - A_{mi} = 5 \]
\[ A_i - S_{mi} = 0 \]

thus the calculated time compression after compensation is:
\[ \kappa = 0.833 \]

This is an incorrect factor to apply to the computation. If the monitor thread were removed, the computation would literally have another processor available to it. Since one computation thread is always switched out, this additional processor would go to it. So instead of four threads running simultaneously, five threads would, making the compression factor 0.80.

The problem with the algorithm is its reliance upon an assumption about on how many processors the computation is running (it "throws away" the \( S_i \) variable). It also assumes that each thread has an equal likely-hood of running so that temporary switch outs are spread evenly among all competing threads.

To improve compensation, local, temporary fluctuations in the parallelism must be accounted for. Several basic quantities must be counted in each interval
- the number of running computation threads,
- the number of running and switched out monitor threads,
- the number of computation threads that are waiting and switched out
- the number of switched out computation threads that could be running

The number of computation threads running during monitored interval \( i \) is:
\[ A_i - A_{mi} - (S_i - S_{mi}) \]

while the number of computation threads that are not running but could be is:
\[ \rho_i = \min[(S_i - S_{mi} - W_{s_i}), (A_{mi} - S_{mi})] \]

Thus, the compression factor is:
\[ \kappa_i = \frac{A_i - A_{mi} - (S_i - S_{mi})}{A_i - A_{mi} - (S_i - S_{mi}) + \rho_i} \]

\[ \kappa_i = \frac{A_i - A_{mi} - (S_i - S_{mi})}{A_i - A_{mi} - (S_i - S_{mi}) + \min[(S_i - S_{mi} - W_{s_i}), (A_{mi} - S_{mi})]} \]

\[ \kappa_i = \frac{A_i - A_{mi} - (S_i - S_{mi})}{A_i + \min[(-A_{mi} - W_{s_i}), (-S_i)]} \]

\[ \kappa_i = \frac{A_i - A_{mi} - S_i + S_{mi}}{A_i - \max[(A_{mi} + W_{s_i}), S_i]} \]
8.5.4. Algorithm IV: Compensating Threads and Compensating Computations

As might be suspected, this formulation has its drawbacks too because, like the others, it is applied to all threads, whether they are running or not. Figure 8-7 shows yet another scheduling scenario where the non-waiting third computation thread has been switched out for a long period while the monitor and the remaining computation threads run. During each analysis interval of the period designated $\psi$, this algorithm would determine that the thread C4 could have run and that the compression factor is 0.80. This would be applied to each thread (whether it was running or not in the interval). The effect would be to under compress the third computation thread, but overly compress the others.

![Figure 8-7: A More Complex Hypothetical Monitored Parallel Computation](image)

What needs to be done is to apply a unique compression factor to each thread in each interval. For each interval of period $\psi$ in Figure 8-7, only the third thread should compensated. The compensation should consists of:

- Two time "pointers" per computation and monitor thread:
  - **measured time pointer (MTP):** This pointer always to points to a common time slice among all the threads. It points to the slice (the element) of the execution that is being analyzed. It is incremented by a constant analysis time granularity after the analysis of each slice until the compensation is completed.
  - **logical time pointer (LTP):** This pointer points to where the compensation algorithm thinks a thread would be in its execution if monitoring were absent. Like MTPs, it is updated after the analysis of each time slice. Unlike MTPs, however, it is updated based on the states of the threads at the times pointed to by the corresponding MTPs. An LTP always points to a nonswitched-out period of a thread unless a switched out period represents a time when a thread is waiting (to join another thread, for example).

- Compression factors: Compression factors is a fraction of the amount an MTP is
incremented. An LTP incremented by this fraction or by this fraction plus one depending its state as well as the compensation mode.

- The same amount as the corresponding MTPs if the state at both times show a thread is running. In such a case the compression factor is zero.

- By a compensation amount less than that the amount MTPs are incremented. This compensation amount depends upon the compensation mode selected by programmer and the state of the threads.

- By a period of time equal to a switched-out period. That is, since LTPs can never point to periods in a thread's execution when it is switched out, it is made to skip over switched-out periods and point to the next nonswitched-out time.

- Compensation modes: The compression factor is calculated by one of two user selected modes:

  - **Uniform**: It is assumed that the nonwaiting, nonswitched-out threads pointed to by the LTPs have equal chances of running. The compression factor is the ratio of the number of threads that are switched out at the time pointed to by the MTPs \((A_i-S_i)\) over the number of computation threads that are running or could have run \((A_i - A_{mi} - (S_i - S_{mi})-p_i)\). The LTPs of all threads that meet this criteria are incremented by the identical fractions of the analysis granularity. The determination of which threads could have run depends upon the states of both the LTPs and MTPs. For example, if an LTP shows that a thread is running, but the corresponding MTP shows that it is not running, it is factored in as a "could have run" thread. Formally, the compression factor for uniform compensation is:

    \[
    \kappa_{k_i} = \begin{cases} 
    \frac{A_i - S_i}{A_i - \max((A_{mi} + W_{si}), S_i)} & \text{for } s_{k_i} = \text{switched-out} \\
    \frac{A_i - S_i}{A_i - \max((A_{mi} + W_{si}), S_i)} & \text{for } s_{k_i} = \text{running} 
    \end{cases}
    \]

  - **Selective**: Like the Uniform mode, it is assumed that the nonwaiting, nonswitched-out threads pointed to by the LTPs have equal chances of running. The compression factor is the ratio of the number of threads that are switched out at the time pointed to by the MTPs over the number of computation threads that could have run as determined by the LTPs and MTPs. The difference between Selective and Uniform compensation modes, then, is that Selective mode only adjusts threads that could have run while Uniform adjusts threads that are both running and could have run. Like the Uniform mode, determining which threads could have run depends upon the states of both the LTPs and MTPs. The compression factor for Selective compensation is:

    \[
    \kappa_{k_i} = \begin{cases} 
    \min[(1.0), (A_{mi} - S_{mi})] & \text{for } s_{k_i} = \text{switched-out} \\
    0 & \text{for } s_{k_i} = \text{running} 
    \end{cases}
    \]

where \(\beta_i\) is determined by examining when a thread is spawned.

The original reason for two compensation modes was to test which would be better. The Uniform mode uniformly adjusts all LTPs. The Selective mode adjusts the LTPs of threads that are already running by incrementing by the analysis granularity and the LTPs of those that could have run by incrementing by some fraction of the analysis granularity. The Uniform mode assumes that in any given slice (element), the likely-hood that thread executes after compensation is equal to the execute likely-hood of each of the other runnable threads. The Selective mode, however, assumes that if a thread is running before compensation it is running after compensation. If a thread is runnable-but-not-running, that is, if it could
have run, before compensation then the likelihood that it executes after compensation is equal to the execute likelihood of each of the other threads that could have run.

In both compensation modes, sensor overhead is compensated as the sensors are detected based on the positive sensor firing times for delays of 100 in Table A-2 in Appendix A. This is an arbitrary decision but has yet to be seen to cause problems in getting accurate performance measurements. A better treatment of sensor overhead is discussed in Chapter 9.

8.6. Example of Compensation of Parallel Computation

Figures 8-8 and 8-9 are execution and cpu barscopes of an eight thread computation running on four processors. One thread is main, one is the monitor collector and the rest are child threads. As can been seen in Figure 8-8 the collector clearly competes with the children for the processors (main is switched out and waiting). In the cpu barscope in Figure 8-9 the collector is identified by the diagonally slashed rectangles.

![Figure 8-8: Execution Barscope of Uncompensated Computation](image)

![Figure 8-9: Cpu Barscope of Uncompensated Computation](image)

Figures 8-10 and 8-11 show the format in which PIE reports Selective and Uniform compensation results for parallel computations. The graphs are produced by running the csplot [11] program on data
generated by PIE compensation code. The graphs shows the measured, compensated and possible parallelism of the computation depicted in Figures 8-8 and 8-9. The begin and end times of each of the threads is reported as measured by monitoring and as estimated after compensation. The compression factor for each thread, indicating how much sooner each thread would have executed without the monitor is also shown. The graphs plot the measured parallelism of the computation threads (X's), the estimated parallelism without monitoring (O's) and the maximum possible parallelism that could have been achieved at any given point if there had been a sufficient number of processors to run the threads. The measured parallelism is always plotted a little below the compensated parallelism, even if the values are equal, in order to make it easier to read the graph.

The reason that the measured parallelism occasionally drops below four processors is that the collector was running at those points. As can be seen, removing the collector would have given the entire computation access to all four processors. The two compensation modes yield different end times, end time variance, and compression ratios for the computation. As will be shown shortly, Selective mode generally estimates faster executions and, thus, smaller compression ratios. As the graphs suggest, Uniform compensation seems to group the end times of the threads more closely around a specific time than does the Selective mode. As will be seen, Selective not only does a better job at estimating the mean execution time of each threads, it also is more true to the real variance in execution time of the threads.

8.7. Compensation Accuracy and Confidence

One application, run under three different configurations on a different numbers of processors was used to test compensation accuracy. Table 8-4 lists the configurations. The configurations consist of four, seven or nine threads, and are run a different numbers of processors. The first column of the table lists the program configuration designation (A,B or C). Program configurations with identical designations had identical spawning hierarchy. The designations are followed by parentheses containing the number application threads in the configuration and the number of processors on which the configuration and monitor thread ran. In all the configurations, thread zero did not run when its children ran. The second in Table 8-4 column describes the spawn (and join) history of the programs.

Each configuration was run 20 times\(^8\) with full PIE monitoring as well as low overhead begin and end timers on each thread. Then each was run 20 times with only the begin and end timers. The timestamp code added less than a millisecond to the total execution times of the threads. The statistics generated for each thread of a configuration were the mean, median, maximum and minimum execution times for the compensated and unmonitored cases. The statistics were generated for 58 different threads. For

\(^8\)More runs would have been desirable, but this is a time consuming process
Figure 8-10: Plot of Selectively Compensated Computation

each thread, the mathematical differences between means, medians, maximums and minimums of the monitored and unmonitored cases were calculated as percentages. For example, the percent difference, \( \Delta_c \), between the compensated maximum execution times and the unmonitored maximum execution times of a particular thread is: \( \Delta_c = \frac{C_C - U_C}{U_C} \) where \( C_C \) is the compensated time and \( U_C \) is the unmonitored time.

A thread's difference statistics are a measure of the compensation accuracy. A measure of the
Confidence in these statistics can be obtained by summing the difference statistics of many threads to determine the mean differences as well as confidence intervals around the means. For example, by summing the differences of all the compensated and unmonitored maximum execution times of all threads, an mean difference between the compensated maximum and unmonitored maximum can be calculated.

Table 8-5 lists average and median differences between compensated and unmonitored mean, minimum
<table>
<thead>
<tr>
<th>Program</th>
<th>Spawning Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>A (4,3)</td>
<td>Thread 0 ➔ 1 ➔{2, 3}</td>
</tr>
<tr>
<td>A (4,2)</td>
<td>Thread 0 ➔ 1 ➔{2, 3}</td>
</tr>
<tr>
<td>A (4,1)</td>
<td>Thread 0 ➔ 1 ➔{2, 3}</td>
</tr>
<tr>
<td>B (7,4)</td>
<td>Thread 0 ➔ 1 ➔{2,4,6} 2 ➔{3,5}</td>
</tr>
<tr>
<td>B (7,3)</td>
<td>Thread 0 ➔ 1 ➔{2,4,6} 2 ➔{3,5}</td>
</tr>
<tr>
<td>B (7,2)</td>
<td>Thread 0 ➔ 1 ➔{2,4,6} 2 ➔{3,5}</td>
</tr>
<tr>
<td>B (7,1)</td>
<td>Thread 0 ➔ 1 ➔{2,4,6} 2 ➔{3,5}</td>
</tr>
<tr>
<td>C (9,8)</td>
<td>Thread 0 ➔ 1 ➔{2,4,6} 2 ➔{3,5,7} 4 ➔8</td>
</tr>
<tr>
<td>C (9,7)</td>
<td>Thread 0 ➔ 1 ➔{2,4,6} 2 ➔{3,5,7} 4 ➔8</td>
</tr>
</tbody>
</table>

Table 8-4: Program Configurations Used to Generate the Statistics

and maximum thread execution times using Selective and Uniform modes of compensation. The confidence intervals around the averages are also shown. The values in Table 8-5 were calculated taking into consideration the sign of the difference. For example, if the compensated maximum execution time for a particular thread were less than its unmonitored maximum execution time, the calculated difference would be negative. By keeping the sign of the difference, the statistics give an indication of whether the compensated execution times cluster or err around their corresponding unmonitored values. For example, if the average signed differences are small, as in the case of the average difference in the maximums for the Selective mode, one might assume that the compensation is fairly uniform in whether it estimation errs to high or too low. Keeping the sign, however, ignores the effect of overly optimistic or pessimistic compensated execution times.

Figures 8-12, 8-13 and 8-14 plot percent difference between the compensated and unmonitored mean, minimum and maximum execution times versus the number of excess threads per configuration. Excess threads are determined by subtracting the number of processors that the configuration ran on from the number of threads in the configuration. The plots are based the best compensation modes for their respective data. For example, Figure 8-13 uses data generated from the Selective mode compensation. The Figures show that the greatest difference between the compensated times and the unmonitored times seems to occur when the number of excess threads is low.

Table 8-6 list similar statistics to those in Table 8-5. These statistics were computed using the same data except that the sign of the difference between the estimates and the unmonitored values was ignored by taking its absolute value. By ignoring the sign of the differences, the statistics are a measure of how far compensation for individual threads may stray. As can be seen, the statistics suggest that for some threads, the average accuracy of compensation can drift as much as 3% to either side of unmonitored execution times.
Table 8-5 shows that when the sign of the difference is considered, with 99% confidence the difference between Selective compensation mode’s estimate of the average mean (i.e., $\mu_1$) and the real average mean is $1.2 \pm 0.9\%$. Although the estimate of the average mean is good, the Selective compensation mode really shines when estimating the average mean (again, $\mu_2$) minimum execution times of threads. With 99% confidence, it can be claimed that compensated mean minimum execution times are within $0.1 \pm 0.9\%$ of the real times. Although the Selective mode beats the Uniform mode at estimating $\mu_1$ for the mean and minimum execution times, the Uniform mode does a better job at estimating $\mu_2$ for maximum execution times. As Table 8-5 shows, using the Uniform mode of compensation, with 99% confidence, the estimated the mean maximum execution times are within $0.4 \pm 1.2\%$ of the real times.
Figure 8-14: Scatter Plot of Difference in Compensated and Unmonitored Maximums

<table>
<thead>
<tr>
<th>Mode</th>
<th>$\mu$</th>
<th>m</th>
<th>$\sigma$</th>
<th>99% C.I.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percent Difference of Means</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selective</td>
<td>1.2</td>
<td>0.4</td>
<td>3.6</td>
<td>1.2 ± 1.2</td>
</tr>
<tr>
<td>Uniform</td>
<td>2.5</td>
<td>1.6</td>
<td>3.6</td>
<td>2.5 ± 1.2</td>
</tr>
<tr>
<td>Percent Difference of Minimums</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selective</td>
<td>0.1</td>
<td>0.4</td>
<td>2.8</td>
<td>0.1 ± 0.9</td>
</tr>
<tr>
<td>Uniform</td>
<td>2.5</td>
<td>1.2</td>
<td>3.5</td>
<td>2.5 ± 1.2</td>
</tr>
<tr>
<td>Percent Difference of Maximums</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selective</td>
<td>1.4</td>
<td>1.7</td>
<td>3.5</td>
<td>1.4 ± 1.2</td>
</tr>
<tr>
<td>Uniform</td>
<td>0.4</td>
<td>0.5</td>
<td>3.5</td>
<td>0.4 ± 1.2</td>
</tr>
</tbody>
</table>

Table 8-5: Statistics keeping the sign of the differences

As might be expected, Table 8-6 paints a gloomier picture of the compensation abilities of either mode. The picture is brighter only among the standard deviations and confidence intervals of the statistics. These statistics have a greater bearing on $\mu$ than those in Table 8-5. They suggest that although the statistics in Table 8-6 show that over many different threads, the compensation heuristics can estimate some times within 1% of the actual times, for certain individual threads, for average means, maximums and minimums of individual threads are accurate to within about 2% of the actual times.
<table>
<thead>
<tr>
<th>Mode</th>
<th>( \mu )</th>
<th>( m )</th>
<th>( \sigma )</th>
<th>( 99% ) C.I.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Percent Difference of Means</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selective</td>
<td>2.5</td>
<td>1.2</td>
<td>2.8</td>
<td>2.5 ± 0.9</td>
</tr>
<tr>
<td>Uniform</td>
<td>3.3</td>
<td>2.1</td>
<td>2.9</td>
<td>3.3 ± 1.0</td>
</tr>
<tr>
<td>Percent Difference of Minimums</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selective</td>
<td>1.9</td>
<td>1.2</td>
<td>2.0</td>
<td>1.9 ± 0.7</td>
</tr>
<tr>
<td>Uniform</td>
<td>2.8</td>
<td>1.3</td>
<td>3.3</td>
<td>2.8 ± 1.1</td>
</tr>
<tr>
<td>Percent Difference of Maximums</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Selective</td>
<td>2.9</td>
<td>2.4</td>
<td>2.3</td>
<td>2.9 ± 0.8</td>
</tr>
<tr>
<td>Uniform</td>
<td>2.2</td>
<td>1.0</td>
<td>2.7</td>
<td>2.2 ± 0.9</td>
</tr>
</tbody>
</table>

Table 8-6: Statistics using the absolute value of the differences

### 8.8. The Deficiencies of the Compensation Heuristics

As might be expected, the compensation heuristics do not work for all classes of computations. As asserted earlier, they have no features for handling highly synchronous computations. They also mishandle the computation shown in the execution barscope in the top view of Figure 8-15. The bottom view is its corresponding cpu barscope. Both views depict the same time period of the computation. The thread to pay attention to is number 17. In the cpu barscope, thread 17 is shaded light grey (the collector has its usual marking, the diagonally slashing pattern). Recall the example in Section 6.2 where the computations were limited to seventeen physical threads despite the fact that both spawned over thirty logical threads. The computation shown in Figure 8-15 is under the same limitation. Thus by the time thread 17 is spawned, there are no more physical threads available because the collector is holding one, forcing thread 17 to wait until thread 7 terminates and relinquishes its physical thread.

Obviously, the monitor severely perturbed this computation. But because the thread limitation is an imposition of a synchronization barrier, the heuristics cannot effectively compensate its threads. Table 8-7 lists the mean, maximum, minimum execution times and rank of selected threads as measured by the monitor, as compensated by the Selective and Uniform modes, and as measured by the low overhead timestamps. The statistics were computed on data generated from 20 executions of the computation. Some of the degenerate statistics are highlighted in boldface in Table 8-7. The monitored mean execution time for Thread 4, for example, is more accurate than the compensated execution time. Threads 14 and 16 have lower minimum execution times when monitored than when unmonitored. Thread 16 also has a lower maximum monitored execution time than when unmonitored.

Table 8-8 lists percentage differences after compensation averaged over the entire computation the group. They show that only one category, the minimum execution time predicted by uniform compensation, yields an acceptable estimate. The standard deviation, however, suggests that it isn't likely that any one thread was predicted correctly.
Figure 8-15: Execution and CPU Barscopes of Problematic Computation

8.9. Comments

The principal reasons the compensation modes did poor jobs of compensating the computation shown in Figure 8-15 is that thread 17 was not compensated correctly. The algorithms were able to shift the start time of thread 17 to left, but not quite enough. In addition, since the MTPs do not yet know about the thread the heuristics end up delaying the threads already executing. Future versions of the compensation heuristics will address these problems.
<table>
<thead>
<tr>
<th>Stat</th>
<th>Meas</th>
<th>Sel</th>
<th>Uni</th>
<th>Real</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Thread 0 (main)</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\mu$</td>
<td>21.37</td>
<td>19.78</td>
<td>19.61</td>
<td>19.18</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>0.55</td>
<td>0.87</td>
<td>0.75</td>
<td>0.36</td>
</tr>
<tr>
<td>min</td>
<td>20.22</td>
<td>18.65</td>
<td>18.53</td>
<td>18.54</td>
</tr>
<tr>
<td>max</td>
<td>22.60</td>
<td>21.17</td>
<td>20.97</td>
<td>19.97</td>
</tr>
<tr>
<td>rank</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
<td>0.0</td>
</tr>
<tr>
<td>Thread 4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\mu$</td>
<td>4.99</td>
<td>4.70</td>
<td>4.65</td>
<td>4.97</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>0.99</td>
<td>0.48</td>
<td>0.46</td>
<td>0.36</td>
</tr>
<tr>
<td>min</td>
<td>4.11</td>
<td>3.86</td>
<td>3.84</td>
<td>4.21</td>
</tr>
<tr>
<td>max</td>
<td>7.70</td>
<td>5.54</td>
<td>5.51</td>
<td>5.41</td>
</tr>
<tr>
<td>rank</td>
<td>6.85</td>
<td>7.75</td>
<td>6.80</td>
<td>5.15</td>
</tr>
<tr>
<td>Thread 10</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\mu$</td>
<td>4.52</td>
<td>4.65</td>
<td>4.53</td>
<td>4.65</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>0.80</td>
<td>0.57</td>
<td>0.54</td>
<td>0.40</td>
</tr>
<tr>
<td>min</td>
<td>3.21</td>
<td>3.75</td>
<td>3.69</td>
<td>3.73</td>
</tr>
<tr>
<td>max</td>
<td>7.09</td>
<td>6.43</td>
<td>5.88</td>
<td>5.09</td>
</tr>
<tr>
<td>rank</td>
<td>9.20</td>
<td>8.75</td>
<td>8.85</td>
<td>8.25</td>
</tr>
<tr>
<td>Thread 14</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\mu$</td>
<td>4.04</td>
<td>4.29</td>
<td>4.23</td>
<td>4.06</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>0.45</td>
<td>0.29</td>
<td>0.30</td>
<td>0.29</td>
</tr>
<tr>
<td>min</td>
<td>3.06</td>
<td>3.34</td>
<td>3.56</td>
<td>3.55</td>
</tr>
<tr>
<td>max</td>
<td>4.89</td>
<td>4.59</td>
<td>4.59</td>
<td>4.59</td>
</tr>
<tr>
<td>rank</td>
<td>12.15</td>
<td>11.15</td>
<td>13.25</td>
<td>12.80</td>
</tr>
<tr>
<td>Thread 16</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$\mu$</td>
<td>3.13</td>
<td>4.00</td>
<td>3.85</td>
<td>3.76</td>
</tr>
<tr>
<td>$\sigma$</td>
<td>0.31</td>
<td>0.48</td>
<td>0.49</td>
<td>0.31</td>
</tr>
<tr>
<td>min</td>
<td>2.98</td>
<td>3.05</td>
<td>3.03</td>
<td>3.27</td>
</tr>
<tr>
<td>max</td>
<td>4.11</td>
<td>4.64</td>
<td>4.60</td>
<td>4.34</td>
</tr>
<tr>
<td>rank</td>
<td>15.75</td>
<td>13.55</td>
<td>15.75</td>
<td>15.00</td>
</tr>
</tbody>
</table>

Table 8-7: Compensation Stats for Selected Threads of A Problem Computation
<table>
<thead>
<tr>
<th>Stat</th>
<th>$\mu_s$</th>
<th>$\sigma_s$</th>
<th>$\mu_u$</th>
<th>$\sigma_u$</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\mu$</td>
<td>4.47</td>
<td>5.59</td>
<td>3.63</td>
<td>5.24</td>
</tr>
<tr>
<td>min</td>
<td>-3.28</td>
<td>6.99</td>
<td>-0.13</td>
<td>6.49</td>
</tr>
<tr>
<td>max</td>
<td>20.45</td>
<td>14.34</td>
<td>17.19</td>
<td>13.9</td>
</tr>
<tr>
<td>$\mu$</td>
<td>5.68</td>
<td>5.73</td>
<td>5.17</td>
<td>5.47</td>
</tr>
<tr>
<td>min</td>
<td>6.13</td>
<td>11.96</td>
<td>4.74</td>
<td>8.21</td>
</tr>
<tr>
<td>max</td>
<td>20.50</td>
<td>14.34</td>
<td>17.24</td>
<td>13.91</td>
</tr>
<tr>
<td>rank</td>
<td>1.10</td>
<td>1.80</td>
<td>0.6</td>
<td>1.0</td>
</tr>
</tbody>
</table>

**Statistics: Signed Differences**

**Statistics: Unsigned Differences**

*Table 8-8: Group Statistics of Degenerate Computation*
Chapter 9
Contributions and Future Work

PIE is a practical, portable way to study and improve performance of computations. It is also a valuable educational tool for understanding the complexities of both sequential and parallel programming. It helps to remove the veil of mystery surrounding the execution of computations on complex systems. Even the scheduling behavior of an operating system, a seemingly esoteric activity over which the programmer frequently has no control, is revealed simply and elegantly in PIE. The research presented in this thesis, the run-time and kernel monitors, the compensation code and the ideas for different visual views has helped to place PIE in a position to remove additional shrouds of complexity from sequential and parallel computations, and the systems that run them. It has also helped to lay the foundation for future work.

9.1. Contributions

This research contributed to PIE’s efforts in proving that not only is an integrated user-level/kernel-level visualization system conceptually and practically feasible, but that such system could be fully implemented and evaluated.

- The Mach kernel monitor has and is providing insights into the design of the Mach scheduler and, when coupled with the user level monitoring system in PIE, helps with the design of concurrent and parallel applications.

- Although this research contributed little to the implementation of several of the PIE visualizations, it was instrumental in seeing the need for the cpu barscope and statistical plot views as well as in the in originating and planning of their structures.

- Finally, the compensation research helps to highlight the attractiveness of the software based monitoring used in PIE when contrasted with potentially less portable, less intrusive hardware based monitoring techniques. This thesis has attempted to change the focus of worries about perturbation from merely attempting to eliminate or minimize perturbation to accepting the perturbation and trying account for it post mortem. The compensation tools improve the PIE environment by giving programmers means to perform “sanity” checks on their performance measurements.
9.2. Extensions

In addition to the contribution to programming environments and performance monitoring, the results research may have extensions to areas such as evaluating any scheduling and execution time effects of changes in the "shape" (as defined by Sevcik [39]) of parallel computations. With further work on interthread overhead, this work may be extended to serve as a predictor of the performance effects of increasing or decreasing the number of parallel threads used by a computation. In parallel fault tolerant systems, for example, it is sometimes important to be able to predict the performance of a computation after the failure of a thread. Algorithm IV, if modified to represent different scheduling policies, may serve as a predictor of the performance computations with failed threads.

9.3. Future Work

There are a number of PIE features that were touched by this work. The PIE group had to develop the monitoring systems and performance visualization tools to capture and process the information for general performance evaluation as well as effective compensation. Each of these areas are fertile sources of future work.

9.3.1. Compensation

The current compensation tools do not handle highly synchronous computations or computations with "special" scheduling behavior. They do not identify possible cases of event order perturbation. If supplanted with synchronization checking, they may suffice for synchronous, sequential multi-threaded computations. Future versions should address these deficiencies. More research into the context-dependent behavior of user level sensors might yield algorithms that can estimate perceived sensor execution times based on the kind of compiler used, and the nature of the code the sensor resides in. The algorithms might be similar to that found using regression analysis in the TPORTSS project [35] where the effect on of several factors on a set of operating system actions is studied.

9.3.1.1. Generalizing Compensation

The compensation algorithms presented here concerned monitor competition for processor resources. The algorithms may have extensions to other computing resources, such as memory and I/O, for which monitoring competes with computations. By analyzing the resource usage by monitoring in a similar way as is done in this thesis for monitor processor use, compensation for the corresponding perturbation may be possible.
9.3.1.2. Adaptive Compensation

The current compensation tools are used in postmortem analysis of monitoring perturbation. More advanced systems could target real-time or "adaptive" compensation as a computation executes in order to perform dynamic adjustments of monitoring in an effort to reduce its perturbation of the rest of the computation. This kind of compensation effort would be part of PIE's research on performance bottleneck avoidance.

Finally, it might be fruitful to develop expert system techniques to help determine whether event order perturbation significantly changes the workload of individual computations. These techniques could consist of a semantic parser coupled with user supplied information about the role of various parts of his computation.

9.3.2. User Level Monitor

Adding interactive capabilities to the current user level monitor would make it a more productive tool for monitoring long computations. Interactive monitoring would allow programmers to peruse the execution of computations, moving their attention to different parts or asking for greater detail as the computations proceed through the different stages of their execution. In order for the monitoring environment to be able to adapt to the run-time vicissitudes of a programmer it must have an interactive interface. More ambitious environments might support dynamic changes to the object code of a computation while it executes so that programmer can conveniently test coding changes during run-time instead of resorting to time consuming recompilation [2].

The monitoring needs to be extended to distributed systems and there may be advantages to installing conditional sensors and sensor firing limits to increase the selectivity of the monitor. Finally, the number of implementations of monitoring needs to be increased. There seems to be some benefit to derived different implementations as suggested in [4].

9.3.3. Kernel Level Monitor

Context-switches are not the only kernel level events that have a bearing on a computation's performance. Other monitoring work consists of tracing file system management [33] or inter-thread actions in distributed computations as in METRIC, DPM and TMP. Being able to visualize disk accesses as in Kerola's Monit [19] or page fault behavior as done by Haikala [17], for example, would be useful for a designer investigating various memory management policies or for computations that are I/O bound. Kerola's and Haikala's work are limited to showing what the paging or I/O activity is at particular time. Monit, for example, does not know the context, in which the fault occurred or what the user is doing. MKM, when used with a PIE type system would be able to provide such information. MKM may need some additional trapping support. McDowell [27] argues that state-of-the-art (correctness) debuggers
may need to be able to trap on interthread communication. This may have utility in performance monitoring as well. The kernel-monitor will be extended to distributed environments as well, by adding a "monitor per host" feature.

9.3.4. PIE Visualization

One of PIE's long range goals is to develop a workstation computational laboratory using kernel and user level monitors to observe a sufficient variety of actions and a visualization environment which offers flexible views of performance data. Although the monitoring systems are extensible to a distributed environment, it is not clear how the additional behavior should be visually represented by PIE.

Work is underway to enhance the visualization formats in order to speed up the performance improvement process. To use a medical analogy, a thermometer can quickly measure whether a patient is sick, but it is almost useless in determining the reasons why. Taking a blood sample, however, can help to identify the causes but is often slower in sounding the alarm that the patient needs help. Similarly, some visualization formats are good for revealing that a performance problem exists, but do not help very much in finding the causes while other formats are good at helping find the cause but may be loaded with so much information that it cannot immediately warn users that a problem exists [25]. PIE can be extended to include visualization formats like those described separately by Kerola and Haikala, where performance data is presented on a histogram time line. Such views quickly tell, for example, whether the desired parallelism is being realized. More "blood sample" views can be added to PIE to help in the detected of the causes of performance problems. Currently, the two principle barscope formats only visualize either thread or cpu behavior versus time. Future formats might graph the actions on shared memory (on MPC frames, for example) or file systems versus time.

PIE will improve its performance analysis tools. As Miller and Yank [29] suggest, PIE can benefit from the kind of automatic assistance for performance analysis included in the IPS system.

9.4. Acknowledgements

Thanks go to: Zary Segall for being a patient, encouraging advisor; Dado Vrsalovic for the provocative discussions and excellent design guidance; Eddie Caplan for suffering through those long debugging sessions and just being an all around good guy; Alan Chung and Charlie Fineman for putting up with my impatient demands; David Black for his advice in developing the kernel monitoring; Jim Quinlan for asking questions and planting good ideas in my thick head; Joanna, my wife, the most important and best person to happen to me; and Raymond Young, my undergraduate advisor, for setting an example of gentle and thoughtful behavior.
Appendix A
User-Sensor Times

User-sensors add overhead to a computation in form of delays due to storing event information. The overhead is two tiered: 1) sensor-firing overhead consisting of storing event information to special monitor buffers and 2) constant overhead consisting of checking whether a sensor should fire completely. The measurements of both kinds of overhead are discussed below followed by a brief description of the measurement techniques.

A.1. Measurement Methodology

Several different MPC programs are used to gather the data points for determining sensor overhead. The executables were generated using the MPC-6.9f preprocessor and Unix C compiler without optimization turned on. As when benchmarking the performance of any system, rarely is there a single program type that completely reveals the behavior of the system. Measuring characteristics using more that one program helps to guard against making performance claims based on data that was generated using a program than unknowingly exercised a little used system feature.

```c
main()
{
  declarations;
  page_in_code();
  for (i = 0; i < EXPERIMENTS; i++) {
    block_of_code( /* with sensors */
      for (j = 0; j < sensors_divided_by_two; j ++){
        delay_code();
      }
    }
  }
  for (i = 0; i < EXPERIMENTS; i++) {
    block_of_code( /* without sensors */
      for (j = 0; j < sensors_divided_by_two; j ++){
        delay_code();
      }
    }
  }
}
```

**Figure A-1:** One program for measuring user sensor firing times
Although four programs were used to measure sensor firing time, only the pseudo-code for two of the programs are shown in Figures A-1 and A-2 because they represent the basic structure of all four. All consist of a single user thread with two major loops each nested within an **EXPERIMENTS** loop. One of the loops has a fixed number of sensors enabled while the other executes without any sensors. The thread is monitored by the PIE collector thread. The only observability code inserted into the C-code generated the MPC preprocessor is that needed for the selected sensors to fire.

```c
main()
{
  declarations;
  page_in_code();
  for (i = 0; i < EXPERIMENTS ; i ++) {
    block_of_code{ /* with sensors */
      for (j = 0; j < sensors_divided_by_two; j ++){
        delay_code();
      }
    }
    block_of_code{ /* without sensors */
      for (j = 0; j < sensors_divided_by_two; j ++){
        delay_code();
      }
    }
}
```

*Figure A-2: One program for measuring user sensor firing times*

The basic difference between the programs shown in Figures A-1 and A-2 is that the program in Figure A-1 first, times **EXPERIMENTS** experiments of loops with sensors and then times **EXPERIMENTS** experiments of loops without sensors while the program in Figure A-2 alternates timing loops with and without sensors each experiment. These two program structures yield strikingly different results on some machines as the kind and extend of `delay_code()` is varied.

### A.2. Results on Selected Machines

Tables A-1 and A-2 list the means, standard deviations and confidence intervals around the means for user sensor times for the first and second program structures, respectively. The delay code consisted of different number of memory moves. For the VaxStation II, the means are consistent across the two programs for low delays. The means diverge, however, as the delays increase. For the Sun3, the means are consistent across all delays and, as might be expected, the spread increases as the delay increase. The most curious measurements come from the Encore MultiMax. In program I, as the delay increases, the execution time of the sensors seem to be negative and almost linearly correlated with the increase in memory moves. The difference in execution times of the loops that measured these times, however, was
a nearly constant 4%. In program II, the execution times are all positive, but increase dramatically as the delay increases.

The reasons for these differences, and their bearing upon compensation methods is a subject for future work.

<table>
<thead>
<tr>
<th>Delay</th>
<th>μ</th>
<th>σ</th>
<th>99% C.I.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td>VaxStation II</td>
</tr>
<tr>
<td>0</td>
<td>580 μ-secs</td>
<td>25 μ-secs(σ_μ)</td>
<td>580 ± 15</td>
</tr>
<tr>
<td>100</td>
<td>570 μ-secs</td>
<td>39 μ-secs(σ_μ)</td>
<td>570 ± 23</td>
</tr>
<tr>
<td>1000</td>
<td>488 μ-secs</td>
<td>48 μ-secs(σ_μ)</td>
<td>488 ± 28</td>
</tr>
<tr>
<td>5000</td>
<td>153 μ-secs</td>
<td>64 μ-secs(σ_μ)</td>
<td>153 ± 17</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>Sun3</td>
</tr>
<tr>
<td>0</td>
<td>210 μ-secs</td>
<td>30 μ-secs(σ_μ)</td>
<td>210 ± 18</td>
</tr>
<tr>
<td>100</td>
<td>206 μ-secs</td>
<td>54 μ-secs(σ_μ)</td>
<td>206 ± 32</td>
</tr>
<tr>
<td>1000</td>
<td>205 μ-secs</td>
<td>39 μ-secs(σ_μ)</td>
<td>205 ± 23</td>
</tr>
<tr>
<td>5000</td>
<td>195 μ-secs</td>
<td>91 μ-secs(σ_μ)</td>
<td>195 ± 53</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>MultiMax</td>
</tr>
<tr>
<td>0</td>
<td>70 μ-secs</td>
<td>19 μ-secs(σ_μ)</td>
<td>70 ± 11</td>
</tr>
<tr>
<td>100</td>
<td>44 μ-secs</td>
<td>21 μ-secs(σ_μ)</td>
<td>44 ± 12</td>
</tr>
<tr>
<td>1000</td>
<td>-100 μ-secs</td>
<td>2 μ-secs(σ_μ)</td>
<td>-</td>
</tr>
<tr>
<td>5000</td>
<td>-584 μ-secs</td>
<td>22 μ-secs(σ_μ)</td>
<td>-</td>
</tr>
<tr>
<td>10000</td>
<td>-1239 μ-secs</td>
<td>28 μ-secs(σ_μ)</td>
<td>-</td>
</tr>
<tr>
<td>20000</td>
<td>-2838 μ-secs</td>
<td>53 μ-secs(σ_μ)</td>
<td>-</td>
</tr>
</tbody>
</table>

**Table A-1:** User Sensor Firing Times: Program I

### A.3. Firing Statistics

Figures A-3 through A-5 are example plots of statistics taken from the experiment designed to measure user-sensor firing times. Figures A-3, A-4 and A-5 plot the duration of 5000 iterations of various loops versus when they started. The context-switch time has been factored out so that the plots represent how long each set of 5000 iterations required. As can be seen in A-3 and A-4 the first set of iterations required abnormally long times. This is probably due to the fact that the code for these loops had not been paged into physical memory yet (the kernel-time data that was factored out only pertains to time when the computation was queued off the processor; it does not show "system time" interruptions of the
<table>
<thead>
<tr>
<th>Delay</th>
<th>$\mu$ (µ-secs)</th>
<th>$\sigma$ (µ-secs)</th>
<th>99% C.I.</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>VaxStation II</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>563</td>
<td>38</td>
<td>563 ± 22</td>
</tr>
<tr>
<td>100</td>
<td>565</td>
<td>20</td>
<td>565 ± 11</td>
</tr>
<tr>
<td>1000</td>
<td>660</td>
<td>40</td>
<td>660 ± 23</td>
</tr>
<tr>
<td>5000</td>
<td>1168</td>
<td>109</td>
<td>1168 ± 63</td>
</tr>
<tr>
<td></td>
<td>Sun3</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>200</td>
<td>30</td>
<td>200 ± 0</td>
</tr>
<tr>
<td>100</td>
<td>205</td>
<td>44</td>
<td>205 ± 26</td>
</tr>
<tr>
<td>1000</td>
<td>205</td>
<td>47</td>
<td>205 ± 27</td>
</tr>
<tr>
<td>5000</td>
<td>190</td>
<td>61</td>
<td>190 ± 35</td>
</tr>
<tr>
<td></td>
<td>MultiMax</td>
<td></td>
<td></td>
</tr>
<tr>
<td>0</td>
<td>49</td>
<td>10</td>
<td>49 ± 6</td>
</tr>
<tr>
<td>100</td>
<td>42</td>
<td>12</td>
<td>42 ± 7</td>
</tr>
<tr>
<td>1000</td>
<td>108</td>
<td>9</td>
<td>108 ± 5</td>
</tr>
<tr>
<td>5000</td>
<td>772</td>
<td>18</td>
<td>772 ± 10</td>
</tr>
<tr>
<td>10000</td>
<td>621</td>
<td>28</td>
<td>621 ± 16</td>
</tr>
</tbody>
</table>

Table A-2: User Sensor Firing Times: Program II

computation's execution). There is not a long first time for the plot in Figure A-5 probably because its iterations are small and first execute after the iterations in the other figures. It was probably paged in with the others.
Figure A-3: Loop Times With Sensors versus Computation Time:
5000 iterations per loop

Figure A-4: Loop Times With Sensors versus Computation Time:
5000 iterations per loop (minus gettimeofday)
Figure A-5: Loop Times Without Sensors versus Computation Time:
5000 iterations
Appendix B

Context-Switch Sensor Overhead

The context-switch sensor overhead is two tiered: 1) sensor-firing overhead consisting of storing event information to special kernel buffers and 2) constant overhead consisting of checking whether a switched thread is monitored. The measurements of both kinds of overhead are discussed below followed by a brief description of the measurement techniques.

B.1. Measurement Methodology

The programs used to gather the data points for determining context-switch sensor overhead consists of two concurrent/parallel processes passing messages back and forth. The programs are executed on the machines booted in multi-user mode because certain system services they used are unavailable in single-user mode.

B.1.1. Measuring Sensor Firing Times

One of the first programs to measure the firing time of the context-switch sensors used the Unix sleep() call in a loop to force threads to context-switch. Curiously, this experiment suggested that the sensor firing time was 5 microseconds! The experiment erred because the sleep time initiated by each call to sleep() consumed the sensor overhead; the sensors fired during the sleep time and so could not be assessed to the threads that context-switched. The programs using the sleep() call were replaced by using a two process program like that shown in Figure B-1. The underlying premise of its design is that identical loops will, on average, have the same number of context switches. The parent process continuously alternates between receiving and sending messages to the child. Inside a single outer for loop, the CHILD process nests two successive inner for loops, which make calls to send_and_receive_messages() each iteration. When either process attempts to receive a message from the other, it switches itself out until the message arrives.

The memory transfer loops are present to test the effect of intervening code upon the sensor firing times. If the sensor code is cached, for example, the memory transfers, if great enough, may flush the code from the cache, causing longer execution times the next time a sensor is fired. The number of transfers between each context-switch is set at run-time.
The first inner for loop is timed and the context-switch sensors are enabled so that the context-switches caused by send_and_receive_messages() are counted. The second loop it is only timed. By subtracting the time difference and dividing by the number of context-switches, an estimate of context-switch sensor-firing time is obtained.

```c
main()
{
 declarations;

 if (fork()) {
   while (child_done != CHILD_DONE) {
     receive_child_msg();
     send_child_msg();
   }
 } else {
   create_and_start_monitoring();
   for (i = 0; i < A_SMALL_CONSTANT; i++) {
     enable_thread_for_monitoring_and_get_start_time();
     for (j = 0; j < A_LARGE_CONSTANT; j++)
       for (k = 0; k < MEMORY_TRANSFERS; k++)
         move_a_block_of_memory();
     send_and_receive_messages();
   disable_thread_for_monitoring_and_get_end_time();
   count_context_switches_and_write_time_stamps();
   get_start_time();
   for (j = 0; j < A_LARGE_CONSTANT; j++)
     for (k = 0; k < MEMORY_TRANSFERS; k++)
       move_a_block_of_memory();
     send_and_receive_messages();
   get_end_time_and_write_time_stamps();
   }
 terminate_monitoring();
 }
}
```

**Figure B-1:** Code for measuring context-switch sensor firing times

### B.1.2. Measuring Constant Overhead

The measurement program shown in Figure B-2 is run on identical kernels, the only difference being that one is built with context-switch sensors included while the other is built with them commented out. The loop-limits are the same as in the first program so that it could be assumed that the number of context-switches are the same on average. There are no memory-transfer loops used here because the enable checks are performed every context-switch. By subtracting the shorter time from the longer and dividing by the expected number of context-switches, an estimate of the constant overhead is obtained.
main()
{
    declarations;

    if (fork()) {
        while (child_done != CHILD_DONE)
        {
            receive_child_msg();
            send_child_msg();
        }
    } else {
        get_start_time();
        for (k = 0; k < A_LARGE_CONSTANT; k++)
            send_and_receive_messages();
        get_end_time_and_write_time_stamps();
    }
}

Figure B-2: Code for measuring kernel monitor overhead

B.2. Statistical Calculations of Overhead

Table B-1 define the statistics measured in the child process used to determine the overhead parameters. Table B-2 lists the values for these statistics for a VaxStation II for the case when no memory block transfers were performed. The mean firing time is found by:

$$\bar{T}_{sensors\text{-fire}} = \frac{\bar{T}_{on}}{\bar{T}_{\text{sensors-on}}} - \frac{\bar{T}_{off}}{\bar{T}_{\text{sensors-off}}}$$

Because $\bar{T}_{\text{sensors-off}}$ could not be measured, $T_{\text{sensors-fire}}$ is approximated by:

$$\bar{T}_{\text{sensors-fire}} = \frac{\bar{T}_{\text{sensors-on}} - \bar{T}_{\text{sensors-off}}}{\bar{T}_{\text{sensors-on}}}$$

It turns out this approximation for the VaxStation II does not alter the estimate of the sensor firing time since its effects fall outside the accuracy of the measurements. The mean constant overhead incurred by the kernel because of the monitor code is approximated by:

$$\bar{T}_{\text{overhead}} = \frac{\bar{T}_{\text{enable-test}} - \bar{T}_{\text{sensors-out}}}{\bar{T}_{\text{est}}}$$
$T_{est}$ is presumed equal to $\bar{T}_{sensors-off}$

<table>
<thead>
<tr>
<th>Measurement Variables of Child Process Behavior</th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{sensors-on}$, $\bar{T}_{sensors-on}$ : Inner for loop time with sensors enabled</td>
</tr>
<tr>
<td>$T_{sensors-off}$, $\bar{T}_{sensors-off}$ : Inner for loop time with sensors disabled</td>
</tr>
<tr>
<td>$I_{sensors-on}$, $\bar{I}_{sensors-on}$ : Inner for loop iterations with sensors enabled</td>
</tr>
<tr>
<td>$I_{sensors-off}$, $\bar{I}_{sensors-off}$ : Inner for loop iterations with sensors disabled</td>
</tr>
<tr>
<td>$T_{sensor-enable}$, $\bar{T}_{sensor-out}$ : Inner for loop time with only enable tests performed.</td>
</tr>
<tr>
<td>$T_{sensors-out}$, $\bar{T}_{sensors-out}$ : Inner for loop time with sensors removed from code</td>
</tr>
<tr>
<td>$I_{est-out}$, $\bar{I}_{est-out}$ : Estimated Inner for loop iterations, sensors removed</td>
</tr>
<tr>
<td>$I_{loops}$ : Number of iterations of outer for loop</td>
</tr>
</tbody>
</table>

**Table B-1:** Definitions

<table>
<thead>
<tr>
<th>Statistic</th>
<th>$\mu$</th>
<th>$\sigma$</th>
<th>99% C.I.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Loop Statistics on VaxStation II (no block transfers)</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>$T_{sensors-on}$</td>
<td>3.495 secs</td>
<td>0.063 secs</td>
<td>3.495 ± 0.02</td>
</tr>
<tr>
<td>$I_{sensors-on}$</td>
<td>2408.1</td>
<td>4.0</td>
<td>2408.1 ± 1.3</td>
</tr>
<tr>
<td>$T_{sensors-off}$</td>
<td>3.069 secs</td>
<td>0.136 secs</td>
<td>3.069 ± 0.04</td>
</tr>
<tr>
<td>$I_{sensors-off}$</td>
<td>2412.3</td>
<td>14.0</td>
<td>2412.3 ± 1.8</td>
</tr>
<tr>
<td>$T_{sensors-out}$</td>
<td>2.866 secs</td>
<td>0.066 secs</td>
<td>2.866 ± 0.01</td>
</tr>
<tr>
<td>$I_{est}$</td>
<td>2412.3</td>
<td>4.0</td>
<td>2412.3 ± 1.3</td>
</tr>
</tbody>
</table>

**Table B-2:** Overhead Results

There are some assumptions about the calculations used to generate the numbers in Table B-2 in are not of central importance in this report. For those not concerned with the specifics, this section can be skipped.

Most of the data listed the two tables was calculated using ordinary statistical formulas. The exceptions
are $I_{\text{sensors-off}}$ and $\sigma_{\text{sensors-fire}}$ and $\sigma_{\text{const-overhead}}$ the standard deviations of the sensor firing time and constant overhead per context-switch. The mean number of context-switches occurring when the sensors were off is an estimate based on three considerations:

1. Aside from the firing of the sensors, the loops are identical.
2. The loop index limit was 1200, limiting the number of intentionally induced context-switches to 2400. The remaining switches resulted from other scheduling behavior.
3. Assuming this "other" scheduling behavior is roughly periodic, there will be a slightly lower mean number of context-switches when the sensors are off than when the sensors are on because the loop times are shorter.

The problem of not being able to get a count of loop context-switches in the unmonitored tests, required an approximation in the calculation of the standard deviation (and, thus, confidence interval) of the estimated sensor firing time.

The variance $\sigma_{\text{sensors-fire}}^2$ is approximated by:

$$\sigma_{\text{sensors-fire}}^2 = \left( \frac{1}{I_{\text{loops}}} - 1 \right) \sum_{i=1}^{I} \left( \frac{T_{\text{sensors-on}} - \bar{T}_{\text{sensors-off}}}{I_{\text{sensors-on}}} - \bar{T}_{\text{sensors-fire}} \right)^2$$

 Ideally, of course, it would be desirable to have a unique $T_{\text{sensors-off}}$ for each $T_{\text{sensors-on}}$ and $I_{\text{sensors-off}}$ But, the inability to retrieve an individual context-switch count with each unmonitored loop time (eg. $T_{\text{sensors-off}}$) necessitates using an approximate method to calculate the variance of $T_{\text{sensors-fire}}$. If the standard deviations for $T_{\text{sensors-on}}$ and $T_{\text{sensors-off}}$ were large and different, then it would not be likely that the approximation reflects a reasonable estimate of a hypothetical sum containing unique $T_{\text{sensors-off}}$. But, because the standard deviations for $T_{\text{sensors-on}}$ and $T_{\text{sensors-off}}$ are small and close, this approximation yields a viable estimate for $\sigma_{\text{sensors-fire}}^2$. An estimate for $\sigma_{\text{sensors-off}}^2$ is obtained in a similar way. The value for $\sigma_{\text{const-overhead}}^2$ is then computed using ordinary techniques.

The first columns in Table B-2 list the statistics of interest. Excepting a few notable cases, the standard deviation and confidence intervals of each statistic were obtained using ordinary techniques. It should be noted that $T_{\text{sensors-off}}$ and $T_{\text{sensor-enable}}$ are measurements of ostensibly equivalent phenomena, namely, the loop times with monitor sensors off. The two statistics were measured under different conditions; $T_{\text{sensors-off}}$ was measured during tests run overnight with the $\mu$Vax connected to the network. However, because it had been expected that the difference between the means of $\bar{T}_{\text{sensors-out}}$ and $\bar{T}_{\text{sensor-enable}}$ would be small, it was desirable to ensure that no other user loaded the machine so that the standard deviations of the two statistics could be kept small. To ensure this, the $\mu$Vax was disconnected from the network when $T_{\text{sensors-out}}$ and $T_{\text{sensor-enable}}$ were measured ensuring that $T_{\text{sensor-out}}$ would be less than $T_{\text{sensor-enable}}$. Actually, disconnecting the $\mu$Vax caused some network queries from the system to timeout, producing periodic outliers in the collection of time-stamps. This periodicity was so regular and the outliers so extreme, however, that those data points were easily detected and discounted.
B.3. Results on Selected Machines

Table B-3 defines the context-switch sensor parameters representing inline overhead. Table B-4 list the means, standard deviations and 99% confidence intervals for these parameters. The code constituting the constant overhead is identical across the machines while the actual firing code varies according to the time stamping facility used (the Encore Multimax, for example, uses a more efficient stamping mechanism than does the VaxStation II or Sun3). No value is listed for the constant overhead of the Encore Multimax and Sun3 context-switch sensor because of (scheduling) difficulties in arranging to boot a special monitored kernel with the sensor code removed. Thus, the constant overhead penalties are estimates. Since the sensor firing times on the Sun3 and MultiMax are similar, it is likely that their constant overhead penalties are also similar.

The values listed are those measured when no the number of memory moves were zero. The major effect of large memory block moves was to increase the standard deviation of the statistics.

<table>
<thead>
<tr>
<th>Sensor Parameters</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>$T_{\text{sensor-fire}}$, $\overline{T}_{\text{sensor-fire}}$</td>
<td>Time to fire a context-switch sensor</td>
</tr>
<tr>
<td>$T_{\text{const-overhead}}$, $\overline{T}_{\text{const-overhead}}$</td>
<td>Constant overhead incurred per context-switch</td>
</tr>
</tbody>
</table>

**Table B-3**: Definitions

<table>
<thead>
<tr>
<th>Context-Switch Sensor Firing Statistics</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Statistic</td>
<td>$\mu$</td>
</tr>
<tr>
<td>------------</td>
<td>------</td>
</tr>
<tr>
<td><strong>VaxStation II</strong></td>
<td></td>
</tr>
<tr>
<td>$T_{\text{sensor-fire}}$</td>
<td>191 µ-secs</td>
</tr>
<tr>
<td>$T_{\text{const-overhead}}$</td>
<td>17 µ-secs</td>
</tr>
<tr>
<td><strong>Sun3</strong></td>
<td></td>
</tr>
<tr>
<td>$T_{\text{sensor-fire}}$</td>
<td>46 µ-secs</td>
</tr>
<tr>
<td>$T_{\text{const-overhead}}$</td>
<td>3 µ-secs</td>
</tr>
<tr>
<td><strong>Encore MultiMax</strong></td>
<td></td>
</tr>
<tr>
<td>$T_{\text{sensor-fire}}$</td>
<td>37 µ-secs</td>
</tr>
<tr>
<td>$T_{\text{const-overhead}}$</td>
<td>3 µ-secs</td>
</tr>
</tbody>
</table>

**Table B-4**: Overhead Results
References


Faust: An Integrated Environment for Parallel Programming.  

Pictorial Man-Machine Communication.  
Siemens AG, Research Laboratories, Munich, Germany, November, 1981.

[16] Ian Hacking.  
*Science - Philosophy: Representing and Intervening.*  

[17] Iikka Haikala.  
ARMA Models of Program Behaviour.  

MatchMaker: An Interprocess Specification Language.  
In ACM (editor), *ACM Conference on Principles of Programming Languages*.  
Computer Science Department, January, 1985.

Monit: A Performance Monitoring Tool for Parallel and Pseudo-Parallel Programs.  
In *Proceedings of the 1987 ACM Sigmetrics Conference on Measurement and Modeling of  

An Empirical Study of Task Switching Locality in MVS.  


Event-Driven Monitoring of Distributed Programs.  
School of Information and Computer Science, Georgia Institute of Technology, May, 1985.

Debugging Parallel Programs with Instant Replay.  

*MKM: Mach Kernel Monitor; Description, Examples and Measurements.*  
Technical Report CMU-CS-89-131, Department of Electrical and Computer Engineering and the  
School of Computer Science, Carnegie Mellon University, April, 1989.

Visualizing Performance Debugging.  

METRIC: a kernel instrumentation system for distributed environments.  
Xerox Palo Alto Research Center (PARC), November, 1977.
[27] C.E. McDowell, D.P. Helmbold.  
Debugging Concurrent Programs.  

[28] Stephen Melvin, Yale Patt.  
The Use of Microcode Instrumentation for Development, Debugging, and Tuning of Operating  
System Kernels.  
In Proceedings of the 1988 ACM Sigmetrics Conference on Measurement and Modeling of  

[29] Baron P. Miller, Cui-Quing Yang.  
IPS: An Interactive and Automatic Performance Measurement Tool for Parallel and Distributed  
Programs.  
In Seventh International Conference on Distributed Computing Systems, pages 482 - 489.  
Computer Sciences Department, University of Wisconsin, September, 1987.

DPM: A Measurement System for Distributed Programs.  

[31] Alan Mink and George Nacht.  
Performance Measurement of a Shared-Memory Multiprocessor Using Hardware Instrumentation.  
In Proceedings of 22nd Hawaii International Conference on System Sciences.  

Monitoring system Behavior In a Complex Computational Environment.  

A Trace-Driven Analysis of the UNIX 4.2 BSD File System.  
Computer Science Division, Electrical Engineering and Computer Sciences, UC Berkeley,  

[34] Richard Rashid, Avadis Tevanian, Michael Young, David Young, Robert Baron, David Black,  
William Bolosky and Jonathan Chew.  
Machine-Independent Virtual Memory Management for Paged Uniprocessor and Multiprocessor  
Architectures.  

Measuring Operating System Performance on Modern Micro-Processors.  
In Joint Conference on Computer Performance Modeling, Measurement and Evaluation, pages  
Information and Computer Science Department, University of California, Irvine,  

Implementation and Evaluation of a Hardware/Software Performance Monitor for Parallel  
Programs.  
PhD thesis, Department of Electrical and Computer Engineering, Carnegie Mellon University,  
May, 1989.

The X Window System.  

[38] Robert D. Schiffenbauer.  
Interactive Debugging in a Distributed Computational Environment.  


