This is to certify that the

dissertation entitled

Instrumentation System Design, Modeling, and Evaluation

presented by

Abdul Waheed

has been accepted towards fulfillment
of the requirements for

Ph.d. degree in Electrical Engineering

Diane J Rover
Major professor

Date 5/1/97

Instrumentation System Design, Modeling, and Evaluation

BY

Abdul Waheed

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Electrical Engineering

1997

ABSTRACT

Instrumentation System Design, Modeling, and Evaluation

BY

Abdul Waheed

An instrumentation system (IS) is defined by this research as a set of modules and services for collecting, forwarding, managing, processing, consuming, and reacting to runtime information about a parallel or distributed system. Runtime information is essential for a variety of multidisciplinary applications of parallel and distributed computing, such as measurement-based tool environments, resource management for distributed real-time systems, adaptive control of distributed embedded systems, management of telecommunication networks, and administration of transaction processing systems. Despite fundamental differences among these application domains in terms of consuming runtime information, a number of features, requirements, services, and design principles of underlying IS modules are common. Recognition of these commonalities allows the IS taxonomy developed in this dissertation to represent a unified view of the IS design and modeling-based evaluation process. This unified view is essential to provide feedback to the system developers at an early stage and may result in a better understanding of distinctive IS features by the users of that system.

This research is the first to apply well-known computer system performance modeling and evaluation techniques to design and manage an IS. In addition, we develop and apply a Resource OCCupancy (ROCC) modeling technique, which facilitates IS modeling and captures the inter-dependences of multiple, interacting workloads. We apply the IS taxonomy and modeling-based evaluation approach to three extant ISs: PICL, Paradyn, and JEWEL. Evaluation of their alternative configuration options and management policies not only provides performance feedback to the developers but also substantiates the feasibility of the modeling-based IS evaluation methodology. Finally, we present the

Vista IS, which is an outcome of the design, modeling, and evaluation techniques developed by this dissertation research.

To my parents

ACKNOWLEDGMENTS

I would sincerely like to thank my advisor, Dr. Diane Rover for her guidance and assistance throughout my Ph.D. program. The accomplishment of this work would not have been possible without her openness toward innovative ideas and encouragement to undertake collaborative research efforts. In addition to directing my research, she provided me with excellent opportunities for my professional development. It has been a valuable experience to work under her supervision.

I thank the other members of my committee, Dr. Michael Shanblatt, Dr. Lionel Ni, Dr. Philip McKinley, and Dr. Vincent Melfi for their feedback about my research and other efforts on my behalf.

I would like to thank Defense Advanced Research Project Agency (DARPA), National Science Foundation (NSF), and Department of Electrical Engineering for funding this research.

I wish to thank my family for their love and support. Successful completion of my efforts as a Ph.D. student is an accomplishment for my parents also. I am grateful for their exceptionally strong support for all of my academic goals. I am also grateful to my wife, Tabassum, for her love, encouragement, and cooperation. I wish to thank our seven months old daughter, Haiba, for filling our lives with joy.

My colleague, Ken Wright, helped me to edit and improve the readability of this dissertation. I would like to thank for his efforts.

Table of Contents

List of Figures

Chapter 1

Introduction and Motivation

1.1 Introduction

Parallel and distributed computing is a cost-effective means of achieving high performance. A number of challenging scientific and engineering problems require an amount of computation that cannot be performed on sequential computing systems within a reasonable amount of time; these problems benefit from the high performance of parallel and distributed systems [40,226]. Use of parallel and distributed systems is not restricted to solving scientific and engineering problems; these systems are being used for multidisciplinary applications, such as those found in commercial transaction processing systems, multimedia systems, real-time systems, and embedded control systems.

Despite the potential for high performance, it is not trivial to achieve such high performance simply by executing an application program on a concurrent (parallel or distributed) system [57]. Measurement-based evaluation of an application can help users identify and remove performance bottlenecks; thus improving its performance. In addition to the so called *grand challenge problems* in science and engineering [70], measurement-based evaluation is being used for performance modeling and prediction [17,37], program debugging [8,25,52,90,105], scientific visualization [20,22,28,29,75,110,135,209], real-time application steering [15,65,71,111], testing of distributed real-time control systems [79], resource management for real-time systems [132], and administration of enterprise-wide transaction processing systems [13]. Measurement-based techniques involve collecting runtime information from a system and using this information to serve diverse needs such as system analysis, visualization, and control by using appropriate software tools. A common denominator among these tools is the need for collecting runtime information from the target system.

Evaluation and/or control of parallel and distributed systems is considered a difficult problem due to complex nature of the interactions among system components. In order to collect runtime information from a concurrent system, software modules are inserted into the target system or system under test (SUT). These modules execute concurrently with the target system and require runtime management of their operation. Therefore, insertion of instrumentation to a concurrent target system often increases its complexity. In this dissertation, we use the term *instrumentation system* (IS) to describe modules and services for collecting, managing, forwarding, processing, consuming, and reacting to runtime information in a concurrent system. Design, modeling, management, and evaluation of instrumentation systems is the primary focus of this research.

1.2 Problem Statement

It is a well-known fact that inserting instrumentation to obtain measurements from a system can adversely affect the behavior of the target system. Several terms are commonly used to describe this phenomenon, such as *intrusion*, *perturbation*, or *probe-effect* of instrumentation. The amount of intrusion is believed to increase with the amount of information extracted from a program; thus the problem of intrusion is considered analogous to the *Heisenberg problem* after the physicist who demonstrated that observing a phenomenon changes its nature [224]. Since measurement-based tools for parallel or distributed systems rely on runtime observations, the problem cannot be solved by reducing or eliminating the instrumentation; it has to be solved by minimizing intrusion without compromising the *observability* of the target system. The notion of observability comes from *Systems Theory*, which relates the design of a system to its other desirable characteristics, such as *controllability* and *testability* [27,175,180,232]. Design, modeling, management, and evaluation of instrumentation systems to minimize their overhead and intrusion to the target system is the primary problem that we have addressed in this dissertation.

There are no universal measures to determine the amount of intrusion due to instrumentation. In the case of conventional uniprocessor computing systems, intrusion can generally be measured as the amount of excess time to execute instrumentation code inserted in the actual application program. However, in the case of a parallel or distributed system, intrusion becomes a compounded problem. Correctness and performance of an application executing on a parallel or distributed system relies on message-passing and synchronization events that must occur in a specific order. The effect of any delay or deadlock due to IS tasks on one or multiple nodes can potentially cascade to other nodes, which may adversely impact the application behavior in an unpredictable manner. In fact, it is not practical to define a universal measure that quantitatively determines the amount of intrusion relevant in all cases. A performance study based on a model of the instrumentation system and its interactions with the target system can make the problem more manageable by allowing the system analyst to focus on the interesting behavior at a desired level of detail. Therefore, we develop models for instrumentation systems to determine their intrusion in the context of their domain-specific usage and requirements.

In addition to addressing the domain-specific requirements of the target system and application, a particular design of an instrumentation system may contribute to its intrusion and overhead. The design of a parallel or distributed instrumentation system is not restricted to merely interconnecting software modules to collect, buffer, and/or consume the runtime information. It also includes the development of one or more management policies to schedule IS-related tasks, configure the IS to allow interactions among its modules as needed, and maintain a steady flow of runtime information to the tools or applications that may need to consume this information. IS management and configuration policies have to be selected on the basis of their impact on intrusion and overhead to the actual application. Empirical comparison among possible IS management options may be carried out at an early stage of development but is not guaranteed to be reliable or accurate. This research shows the use of IS models and their (analytical or simulation-based) evaluation for comparison among management policies in a rigorous

manner and guidance to system developers to choose a policy that incurs minimum intrusion.

Despite its intrusion and complex interactions with the target system, an instrumentation system is often developed in an ad hoc fashion, without considering the impact of intrusion. In a favorable case, the overhead due to an IS may be restricted to degrading performance of instrumented applications (in terms of their execution time) from 10% to more than 50%, according to various measurement-based studies [71,134]. Unfortunately, the IS can perturb the behavior of the application [88,125,229], and data collected from such experiments may lead the users to incorrect conclusions. The user may be unaware of the impact of the perturbation resulting from contention for computing resources among application and instrumentation system processes.

Problems related to intrusion of an IS on the target system and to dynamic management of shared system resources are the main issues that motivate the research presented in this dissertation. Specific problems and issues addressed in this dissertation are summarized in the following thesis statement.

Thesis Statement: *Computer system performance modeling and evaluation techniques may be applied in a novel manner for design and runtime management of instrumentation systems (ISs). In particular, traditional computer system modeling techniques can be refined to facilitates the IS modeling effort and capture the inter-dependences of different types of interacting workloads often found in real systems. In addition, modeling-based evaluation is useful for the developers to make appropriate choices from a set of alternative IS configurations and potential management policies.*

In the following section, we discuss the motivation for adopting a modeling-based approach for designing and evaluating an IS.

1.3 Motivation toward Solving the Problem

Runtime measurements are used by many multidisciplinary applications of parallel and distributed computing systems. Despite differences in how those application domains use measurement-based information, a number of features, requirements, services, and design principles of the underlying IS are common. Although it is beneficial to recognize the commonalities among ISs for multidisciplinary parallel and distributed applications, no such effort aimed at unifying the design and evaluation of ISs is reported in the literature. The following are compelling advantages of unifying the design and evaluation process for ISs:

1. Recognition of commonalities among IS modules and services for different types of applications enhances the understanding of domain-specific requirements and related design issues.
2. Instead of designing a customized IS, it is possible to design a set of configurable IS components and an infrastructure to customize them as an IS for a specific application.
3. Individual modules can be evaluated with respect to domain-specific needs for improving their design and providing the application developers with useful information about the behavior of the IS.
4. A standard IS interface can facilitate the development of a number of measurement-based tools and applications.

This dissertation research is motivated due to the above potential benefits of unifying the design, evaluation, and usage of an IS for a given application. This research effort progressed on two tracks: the first track is related to synthesizing the insights about IS development and usage in diverse disciplines; and the second track involves original work in design, modeling, evaluation, and management of specific ISs. While the work on the first track contributed to our unified view of an IS for any application domain, the second track served as a "reality-check" on the insights gleaned from the first.

A number of experienced tool developers use empirical evaluation of IS design and management policies to develop the IS. While this approach may work in simple cases, it has a potential cost involved with it: if a performance bottleneck is discovered in an IS during production stage, it will have to undergo time-consuming upgrades and alterations.

This issue is of growing important for commercial tool developers who are reluctant to invest in tools for high-performance parallel and distributed systems [151]. It also motivates the need for evaluating the system at an early stage of development, which serves as a basis for the IS modeling proposed by this research.

Instrumentation systems used for applications in different disciplines have domain-specific requirements and constraints. For instance, an IS that collects performance data for a bottleneck searching tool is required to incur minimum overhead to an application process due to sharing of system resources. On the other hand, an IS that collects runtime information from the sensors of a distributed real-time control system is required to exhibit predictable behavior. Similarly, ISs used for embedded systems, pattern recognition systems, and commercial transaction processing systems have to address other domain-specific constraints. It is difficult to guarantee that an IS can meet the domain-specific requirements without carrying out a detailed performance study. System models with appropriate levels of detail can be used to analyze a system for its compliance to domain-specific constraints, even at an early design stage.

Currently, several ISs are developed as commercial, off-the-shelf software systems that can be used with a multitude of applications and tools [15,114]. The intrusion and overhead of these tools are not analyzed and documented under various possible operating conditions. The task of a tool developer using an off-the-shelf IS becomes harder in the absence of a performance study conducted on that IS. A performance study of an IS can help design test suites and benchmarks to investigate its operating characteristics. Application and tool developers can use this information to determine the suitability of an off-the-shelf IS for a particular application.

This research is also motivated by increasing sophistication of concurrent system software technologies (such as *multithreading* [159] and *microkernels* [200]). An IS-related task (a separate process or a thread) is expected to manage and regulate its use of the shared

system resources [162,163]. We model and evaluate an IS using a high-level workload characterization technique that adequately captures the non-deterministic interactions among IS and application tasks. We also apply established experiment design techniques to solve these models using simulation.

1.4 Objectives, Criteria, and Contributions

This section presents the objectives of this research and a strategy to evaluate its outcomes. These issues were considered at the time of planning and proposing this work and are noted here to serve as background information to interpret the results and their impact on the state-of-the-art. We also summarize the results and contributions of this work.

1.4.1 Objectives of the Research

Four objectives of the research presented in this dissertation are:

1. To characterize instrumentation systems used in a wide range of application areas in terms of generic data collection components and services. This enables tool developers and users to view an IS as a subsystem of a tool and is a prerequisite to understanding the intrusion of an IS to the target system and evaluating it.
2. To model and evaluate several ISs for investigating their performance under different operating conditions and intrusion to the target system. A model for an IS should be able to capture non-deterministic interactions among IS, target system, and application components and inter-dependences of their behavior.
3. To address the domain-specific requirements and constraints of an IS during evaluation so that results have relevance in practice. One of the goals of the evaluation process is to assist the investigations of "what-if" questions and scenarios regarding IS configuration, behavior, and performance.
4. To provide feedback and recommendations to the developers at an early stage of tool design and development. This feedback, regarding the IS performance and intrusion under alternative options for managing and configuring it, should assist the developers in making design decisions.

These objectives represent a balance between issues of academic interest, such as IS characterization, and issues of practical interest, such as addressing domain-specific constraints and feedback to tool developers. In order to assess the success in meeting these objectives, we set forth the following criteria:

- **Acceptability of the approach by tool developers**: We decided to apply the IS modeling and evaluation approach to the ISs of state-of-the-art parallel tools. On the one hand, our objective was to provide feedback to the tool developers. On the other hand, we wanted to put the modeling and evaluation approach into practice to determine its effectiveness over the current ad hoc IS development approaches. Having the approach and the results for a particular tool well-received by the developers indicates the promise of this approach in practice.
- **Ability to represent complex behavior and flexibility to analyze it**: The characterization and modeling of the IS must be able to represent the complex behavior of the IS under realistic operating conditions, providing insight into behavior not trivial to understand otherwise. Moreover, the evaluation approach should provide flexible support to analyze the behavior.
- **Accuracy vs. early feedback**: Workload characterization involves a trade-off between accuracy of the model and early feedback to the developers. Simple, yet adequate workload characterization is desirable for the success and acceptability of this research. Simple workload characterization is appropriate at an early stage of system development when extensive measurements are not available to complement this process. On the other hand, detailed measurements result in more accurate workload characterization and model-based evaluation. A proper balance should support the first two criteria.

We have applied these criteria throughout this research and in specific case studies.

1.4.2 Contributions of the Research

This research has made four notable contributions to the state-of-the-art in IS development and usage. These contributions are highlighted in this subsection:

1. **IS Characterization and Taxonomy**: We introduced a characterization of an instrumentation system consisting of six building blocks: *sensors* and *actuators*, *local instrumentation servers*, *instrumentation system managers*, *instrumentation data consumers*, *transfer protocol*, and *instrumentation system agents*. We also developed a taxonomy for instrumentation systems in terms of their on-line or off-line usage; hard-coded or application-specific design; and static, adaptive, or user-defined management policies

[213]. This characterization helped spawn other research efforts in the tool community to explore the feasibility of designing a generic, extensible, and retargettable IS that could be interfaced to heterogeneous measurement-based parallel tools.

2. **Resource Occupancy Modeling**: We developed a generic instrumentation system modeling approach based on the idea of resource occupancy to incorporate non-deterministic sharing of resources among different types of processes. The Resource OCCupancy (ROCC) models were applied to investigate intrusion of instrumentation system tasks to the target applications on several computer systems [214]. This modeling approach is a trade-off between (1) the ability to rapidly model complex dependences among IS and application processes in the absence of detailed measurements and, (2) the accuracy of conventional computer system modeling approaches based on extensive workload characterization [47,94].
3. **Evaluation of Extant ISs**: We carried out model-based evaluation of a number of instrumentation systems. To the best of our knowledge, this is the only documented effort in the current literature that models and evaluates several existing instrumentation systems according to their domain-specific constraints and features.
4. **IS Management Policies**: During modeling and evaluation of instrumentation systems that were at different stages of their development, we proposed several management policies in the context of domain-specific needs of these ISs. These policies were modeled and evaluated; feedback to the tool developers often resulted in actual implementation of a suitable policy.

Some of these contributions are in the context of modeling and evaluation of specific ISs. Nevertheless, we consider them as contributions to the state-of-the-art due to two reasons: (1) although a computer system performance study may be specific to a particular system, findings of one study are often applicable to other systems having similar characteristics; and (2) despite differences in architectures, operating systems, and domain-specific requirements of computer systems, runtime data collection and its intrusion is a common issue. Therefore, the contributions of this work add to the general knowledge in the field. In fact, this work is the first to study and optimize ISs using performance modeling.

1.5 Overview of Dissertation

In this dissertation, we address three specific areas that are of interest to developers and users of measurement-based tools and applications: IS design, modeling, and evaluation. Although the scope of our discussion is general, we focus on the instrumentation systems for parallel, distributed, and real-time system tools and applications as our case studies. As

noted in Section 1.3, use of these case studies is essential to illustrate the applicability of our modeling and evaluation approach to real systems.

The dissertation is organized into eight chapters. In this chapter, we presented an introduction to the problem and issues related to IS development and usage that are addressed in this dissertation. This chapter also outlined our approach to tackle the problem, criteria to evaluate the results based on this approach, and a summary of contributions of this work.

Chapter 2 presents the background of this research and other efforts related to IS development and usage. This chapter presents a historical perspective on the ISs that have been used in different areas. We discuss important issues related to IS design, implementation, and usage with respect to a broad range of current tools and applications. Related work presents the IS evaluation and modeling efforts that are of interest to this research.

Throughout this dissertation, we use three instrumentation systems as our reference systems that we have modeled and evaluated so far. These are: PICL, Paradyn, and JEWEL ISs. Chapter 3 presents an overview of these instrumentation systems in the context of their specific applications. PICL, Paradyn, and JEWEL ISs are considered in the following contexts: parallel program visualization; measurement-based identification of performance bottlenecks in parallel or distributed applications; and monitoring of distributed applications and adaptive control of real-time systems, respectively.

Chapter 4 focuses on characterization, design, and synthesis of instrumentation systems. It begins with a generic model and a taxonomy to describe and analyze an IS. Then we reflect on a number of design options that are available to the developers and require decision-making effort on their part to implement a particular IS. The generic IS model is

applied to the reference ISs to present their specifications and identify their components using our taxonomy. Topics of this chapter appeared in [212], [213], and [217].

Chapter 5 presents the resource occupancy (ROCC) modeling technique in terms of occupancy requests, shared system resources, management policies, and interacting workloads. We address general issues related to modeling an IS and demonstrate their relevance in the context of modeling three reference ISs. In addition to presenting models for the reference systems, we identify objectives of modeling in each case and present a set of metrics relevant for an IS and its specific domain of application. We introduced the ROCC modeling technique in [214].

Models for the reference ISs are evaluated in Chapter 6. We introduce the experiment design used in each case and setup of the simulation experiments. We use a ROCC simulator written in C++ with task library to model concurrent processes. Whenever feasible, we validate the simulation-based evaluation results using actual measurements. However, in certain cases when measurements are not possible because the system is not at prototyping stage, we use *operations analysis* techniques to provide back-of-the-envelope analytic results for comparison [117]. Some of the results presented in this chapter appeared in [211], [218], and [219].

Chapter 7 summarizes the outcome of this dissertation research in terms of three deliverables: modeling-based evaluation of the reference ISs, the ROCC simulator, and the Vista IS. In particular, we provide design and implementation details of the ROCC simulator and Vista IS. This information is beneficial for the potential users of the simulator and the IS to extend this dissertation research effort to other related areas. Parts of this chapter appeared in [215] and [216].

We conclude with consideration of future directions of this work in Chapter 8. The areas appropriate for the future extensions of this work include: design, modeling, and

evaluation of ISs for new applications, IS testing, adaptive control and management of ISs, and developing configurable IS kernels.

Chapter 2

Background and Related Work

In this chapter, we present the background of IS development and usage for different types of applications. Our primary objective is to systematically present some of the basic issues involved in IS development and usage that motivated this research. In Section 2.1, we introduce these issues to the reader. We present a historical background of instrumentation systems in Section 2.2 and three examples of IS usage in Section 2.3. We present the related work in Section 2.5, which has contributed not only to the maturity of the state-of-the-art in IS development and usage, but also influenced the progress and direction of this research.

A reader familiar with computer system modeling may choose to skip Section 2.4. The sections of primary relevance to this research include 2.1 and 2.5.

2.1 Introduction

An instrumentation system (IS) is defined as a set of modules and services used for collecting runtime information from parallel and distributed systems. This information collected by an IS can serve various purposes, for example, evaluation of program execution on high performance computing and communication (HPCC) systems [161], monitoring of distributed real-time control systems [66,221], resource management for real-time systems [132], and administration of enterprise-wide transaction processing systems [13]. These, as well as other applications, may place domains-specific requirements on the IS that directly impact its design, operation, and performance. This dissertation is a result of research efforts that directly address the above issues related to software ISs for multidisciplinary parallel and distributed applications.

Development of an IS can be considered as a two stage process (see Figure 2-1); the first stage consists of IS design, software development, and testing, whereas the second stage consists of evaluating the design and its prototype with respect to the system specifications and requirements. The activities under the first stage are well-known software development practices while those under the second stage are a consequence of this research effort. Evaluation of alternatives for configuring modules, scheduling tasks, and instituting policies for managing instrumentation tasks should occur early in IS development. Next, testing validates the evaluation results from the first stage and qualifies other functional and non-functional properties. Finally, the IS is built and used for real applications. Evaluation requires a model for the IS and adequate characterization of the workload that drives the model. During this phase, the model can be evaluated analytically or through simulations to provide feedback to the IS developers. The model and workload characterization also benefit testing by highlighting performance-critical aspects identified during the IS evaluation phase.

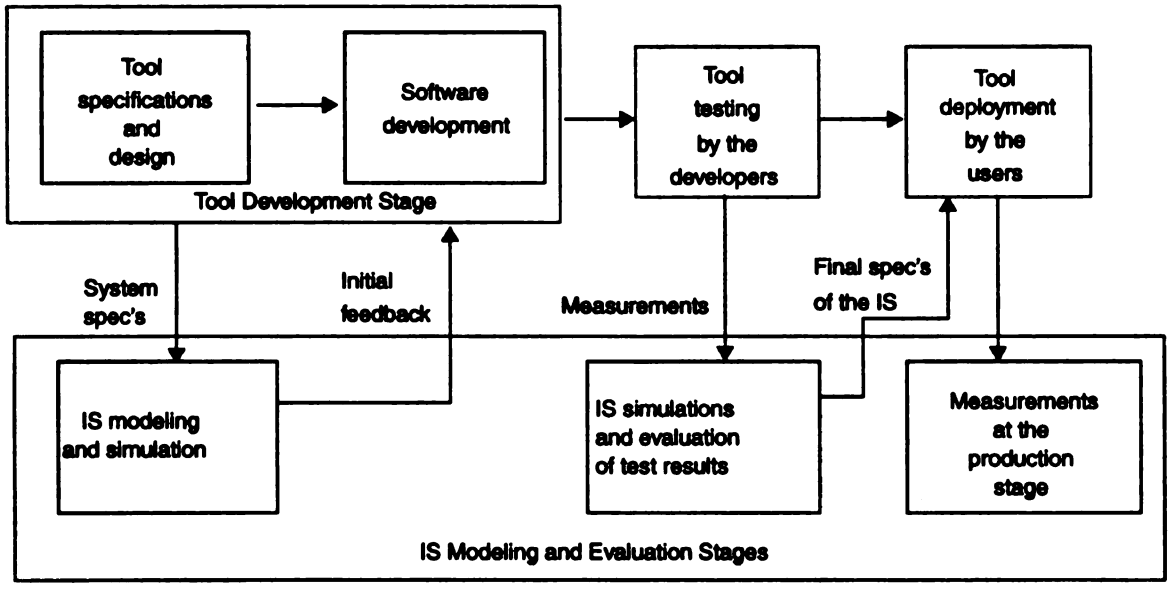


Figure 2-1. Various stages of design and usage of a typical general-purpose instrumentation system.

An IS is an enabling technology for developing measurement-based tools for parallel and distributed systems and is removed from the end-user. Tool developers can use the same IS

for various types of tools while hiding the implementation details from the users. Although, the usability and extensibility of a tool is thoroughly scrutinized, the IS and its overheads rarely receive any attention [213]. Consequently, an IS that is developed by ignoring the modeling and evaluation stages shown in Figure 2-1, may not meet domain-specific functional and performance specifications. Therefore, we consider the problem of instrumentation system design, modeling, management, and evaluation in this dissertation.

2.2 Historical Background

Instrumentation and measurements are the core of every engineering and scientific experiment. Measurements are made using suitable instruments in diverse scientific disciplines, such as physics, chemistry, biology, agriculture, and so on. While the methods and techniques of instrumentation and measurements may vary in different application domains, many of the objectives, principles, problems, and solutions are similar. Therefore, we present a unified treatment of instrumentation and measurement systems to put their application to the field of parallel and distributed systems in proper historical perspective.

We begin by introducing instruments and measurements in general terms, regardless of any particular application area. Prensky and Castellucis [155] classify various types of instruments as:

> *Instruments of many kinds have the common purpose of supplying information concerning some variable quantity (sometimes called a parameter i.e., a quantity that is experimentally varied in a series of steps) that is to be measured. This information is generally obtained as a deflection of a pointer on a meter or in the form of a digital readout, and in this general way the instrument performs an indicating function. In many cases the instrument also provides a chart record of the instantaneous indications, thus performing a recording function. A third important function, particularly in industrial-process situations, is accomplished when the information is used by the instrument to control the original measured quantity.*

Similarly, Schnell [184] characterizes the measurements and motivates the need for a model of the system under test in the following manner:

> *Measurement in a general sense is the procedure of obtaining information about the world by technical means. However, the surrounding world is extremely complex, and it is not possible to deal with every detail. Therefore, science uses simplified equivalents of real phenomena: it neglects unimportant aspects in order to catch the essential features. This simplification is called modeling.*

It is clear from the above excerpts that the measurements and measurement-based experiments have strong dependences on the system under test and the model used for representing the system. On the other hand, the characteristics of an instrument depend only on its functionality rather than the system under test. Since we focus on instrumentation systems in this dissertation, our discussion in this section is limited to instruments.

A varying quantity can reliably be represented as an electrical signal as compared to other means of representation such as mechanical. Due to remarkable advances in the technologies necessary to build these electrical devices, the precision and accuracy of processing these signals through electrical devices is far superior than any other means. Therefore, the term "instrument" has become synonymous to an electrical instrument. In case of quantities that are non-electrical, appropriate transducers are used to convert them to an equivalent electrical signal to benefit from electrical processing.

The types of instrumentation systems can be divided into three classes in terms of the key technologies used: electrical and electromechanical instruments, electronic instruments, and computer-based instruments. These classes of instrumentation systems have evolved over time and reflect our increasing capabilities to control electrons [155]. Table 2-1 provides details of instrumentation systems and techniques that each of these classes incorporates. The evolution of these systems over time is represented by the boxes from left to right and top to bottom. Note how technological trends dictated transitions from

electrical to electronic to state-of-the-art computer-based instrumentation systems. However, it is not correct to assume that the electrical, electro-mechanical, and electronic devices are no longer useful. In fact, all types of instrumentation systems listed in Table 2-1 are still being used for different applications because of their specific characteristics, such as precision, accuracy, sensitivity, applicability for a particular measurement-based experiment, and usability. Thus, Table 2-1 represents the entire spectrum of instrumentation systems that rely on electrical devices for measurements. Software instrumentation, which is the focus of this dissertation, is an emerging and comparatively less mature area among others in this spectrum. It is becoming popular due to an increasing number of computing-based applications that benefit from runtime measurements.

We can consider virtual and software instrumentation system as non-traditional instrumentation systems compared to the conventional instruments listed in Table 2-1. However, there are some similarities between the two types of instrumentation systems. These similarities include:

1. intrusion to the actual system due to non-ideal behavior of the instrument;
2. measurement errors due to less than perfect precision and accuracy; and
3. performability requirements of the instrumentation system are either relaxed or made stringent, depending on the target system and application.

Despite these similarities, software instrumentation systems posses distinct characteristics that cannot be found in conventional instruments. For instance, data collection at the operating system or application level is very flexible and user can easily manipulate the IS to gather useful information. Additionally, a software IS can be used in a "closed-loop" configuration where runtime information collected by the IS can be used to adaptively control the target system. Research presented in this dissertation is focused on these two distinctive functions of a software ISs. Therefore, the term *instrumentation system* refers to the software instrumentation systems in this dissertation unless noted otherwise.

Table 2-1. The spectrum of instrumentation systems.

Electrical and Electro-Mechanical Instruments	Electronic Instruments	Computer-Based Instruments
The basic instruments for direct-current measurements: • DC voltmeter • The Ohmmeter • The multimeter	Basic electronic instruments: • Single-tube direct-current VTVM • Electronic voltmeter for AC • Solid-state voltmeters	Analog computer-based instruments: • Summing amplifiers • Solvers for simultaneous differential equations • Integrating amplifier
The basic instruments for alternating-current measurements: • Rectifier-type AC meter • Dynamometer-type AC meter • Wattmeter • Moving-iron-type instruments	Recording instruments: • Electronic graphic recording systems • Self-balanced systems • X-Y recorders • Galvanometer recorders • Ink and inkless recorders	Basic digital instruments: • Binary counters • Frequency and period counters • Digital multimeters • Logic analyzers
Instruments for comparison measurements: • Potentiometer-null instruments • Wheatstone bridge	Cathode-ray oscilloscope-based instruments: • Voltmeters • Frequency meter • Waveform analyzer	Virtual instruments: • Data acquisition systems and bus interfaces • Hardware counters and monitors • Reconfigurable hardware instrumentation • Integration of hardware measurements and software tools
Instruments for impedance measurements: • Maxwell-bridge • Hay-bridge • Alternating-current null detectors • Phase-sensitive detectors	Special-purpose instruments: • Transistor testers and analyzers • Untuned-amplifier testers • Tuned radio-frequency testers • Nuclear-radiation detectors	Software instrumentation: • OS-level profiling • Application-level monitoring • Dynamic instrumentation • Numerous tools for performance analysis based on measurements

2.3 An Overview of IS Development and Usage

Instrumentation systems are being used to serve the data collection needs for diverse parallel and distributed environments, applications, and systems [188,189,190,174]. Tool environments consisting of debugging, performance analysis, bottleneck searching, modeling, and prediction tools rely on runtime measurements supplied by an IS [78,82]. Multidisciplinary applications, such as administration of commercial transaction

processing systems [13], measurement-based testing of complex military systems [9,79], and resource management of distributed real-time systems [132] consume the runtime information obtained from an IS. A variety of distributed systems, such as pattern recognition systems [99] or embedded real-time controllers [66,198] require continuous data collection for either measuring the features of an object for its appropriate representation or adaptively controlling a device or process, respectively.

In this section, we survey the use of an instrumentation system, as defined by this dissertation, for three types of applications: scientific applications that execute on high performance computing and communication (HPCC) platforms, commercial transaction processing, and distributed real-time computing. Use of ISs in these application domains serves the common objective of getting runtime information from the applications and implementing management policies based on the collected information. This information can be used for diverse purposes, including capacity planning, system administration, system performance and workload evaluation, performance prediction, system resource monitoring and testing, and visualization. Table 2-2 summarizes the key functions that an IS supported in each of the three application areas and outlines related research issues. The following subsections present the state-of-the-art with respect to the issues identified in the table.

In the following subsections, we present research and development efforts and usage of instrumentation systems for scientific applications, commercial transaction processing systems, and real-time systems. We elaborate the use of ISs in these areas through appropriate examples derived from an application areas.

2.3.1 High Performance Scientific and Engineering Applications

The "grand challenge" applications in science and engineering are distinctive due to their need to achieve high performance [70]. Due to their requirements for high performance, we refer to these applications as High Performance Computing and Communication

Table 2-2. Application areas and related issues of designing and using *ISs*.

Application Area	Key Functions Supported by IS	Relevant Issues
High performance scientific and engineering applications	• Performance evaluation and tuning • Application steering • Debugging • Modeling and prediction • Performance and data visualization	• Intrusion of data collection modules to the application behavior • Contention for shared system resources • Lack of generalized performance evaluation methodology that necessitates application-specific/user-defined ISs
Commercial transaction processing systems	• System administration and capacity planning • Distributed enterprise integration through a single point of control • Monitoring of resource usage • Management of resources to implement *workflow*	• System is required to be robust and fault-tolerant • System is expected to run over an exceedingly long period of time for a particular application • Security of various databases need to be ensured
Real-time systems	• On-line monitoring of system health • Data logging for off-line failure analysis • System testing with or without fault injection • Adaptive control and steering	• Scheduling of application and IS tasks having unequal priorities • OS support for real-time behavior • Scheduling of periodic and aperiodic tasks for adaptive steering

(HPCC) applications. In practice, it is very difficult to write an application to fully utilize the peak performance capabilities of the underlying parallel or distributed system architecture. An HPCC application program usually undergoes a tedious process of debugging, bottleneck evaluation, performance visualization, performance prediction, etc. before it can be optimized for a particular platform. Measurement-based tools are used at each step of this process. A number of efforts focused on the use of measurement-based information for HPCC applications as well as the theoretical issues of presenting this information are reported in literature (such as, [21,42,43,44,82,149,150,167,168,170,171, 204,205,206]).

2.3.1.1 Research and Development

Instrumentation systems for the tools to develop HPCC applications are designed to collect and manage the data, which are consumed in an on-line or off-line fashion. ISs for these HPCC applications are often developed in an ad hoc manner. Nevertheless, a number of research efforts can be found in literature that emphasize systematic characterization of an IS and careful considerations of its design alternatives. Ogle, Schwan, and Snodgrass characterize an IS in terms of a *local* and *central monitor* and present the Issos monitor, which is an extensible, application-specific IS [146]. Schroeder characterizes an IS in terms of five applications that require runtime data collection: debugging and testing, performance evaluation, correctness checking to ensure consistency with the formal specification, security monitoring attempts to detect unauthorized accesses to system resources, and control where monitoring is part of the target system [185].

Researchers have considered the problem of IS perturbation to HPCC applications. Malony characterizes an IS in terms of an *Instrumentation Uncertainty Principle*, which is based on three observations [127]:

- instrumentation perturbs the system state;
- execution phenomena and instrumentation are logically coupled; and
- volume and accuracy of instrumentation are antithetical.

This perturbation may result in one or multiple of the following: loss of performance (slowdown or direct perturbation or timing perturbation), event reorder, and parallelism constraints. Perturbation is known to be affected by event frequency, event size, synchronization requirements, storage limits, and communication mechanisms of IS and application processes. Malony et al. present time-based and event-based models to recover accurate timing information and remove effects of instrumentation on event ordering, respectively [125]. Performance evaluation tools, such as AIMS, use specific models to correct the instrumentation data by compensating the effects of the perturbation to

represent an approximate behavior of the program [229]. There is an important distinction between these efforts and the research presented in this dissertation; these efforts do not consider early evaluation of an IS to reduce or eliminate the perturbation by appropriately selecting the IS modules and configurations.

More recently, several other researchers have given special attention to the instrumentation system overheads of their tools. Miller et al. present measurements of overheads of the IPS-2 tool and compare them with the overheads of a functionally similar tool, *gprof* [134]. Gu et al. use synthetic workloads to exercise specific features of the Falcon IS and measure its performance [71]. Haakee, Schauser, and Scheiman develop an IS for Split-C parallel programming language and quantitatively evaluate its profiling overhead for several applications [74]. The management policy for the IS, called flush-on-barrier, is an extension of our work on modeling and evaluation of the PICL IS management policies, which will be presented in this dissertation.

2.3.1.2 Usage of Instrumentation Systems

The purpose of using an IS for HPCC applications is to supply the runtime information to a multitude of on-line or off-line tools in the environment. These tools may include: debuggers, visualizers, and performance analyzers. Table 2-3 summarizes various tools for scientific and engineering HPCC applications that need an IS to perform their specific task. Cheng provides a more extensive survey of numerous commercial and research tools [36].

2.3.1.3 Example of an IS for an Integrated Parallel Programming Environment

In the parallel system tool environment example given in this subsection, we consider the architecture of ParAide integrated tool environment for Intel's Paragon system [165]. Figure 2-2 illustrates the overall architecture of this environment. Commands are sent to the distributed instrumentation system, called Tools Application Monitor (TAM). TAM

Table 2-3. Tools for scientific and engineering HPCC that use an IS for runtime data collection.

Tools	Type of the tool(s)	Description of the IS functions
ParAide [165]	Performance evaluation	See Section 2.3.1.3.
Paradyn [136]	Bottleneck identification	Paradyn uses W3 search model [] to identify performance bottlenecks in programs on a CM-5 and cluster of workstations. Instrumentation is dynamically inserted by Paradyn daemons at each node of the system, as needed by the search algorithm.
VIZIR [76,77]	Debugging	This debugger consists of an integrated set of commercial sequential debuggers. Its *IS* synchronizes and controls the activities of individual debuggers that run one of the concurrent processes. *IS* also collects data from these processes to run multiple visualizations.
Falcon [71]	Steering	In order to steer the application during its execution, the *IS* plays a dual role. First collecting the desired runtime information to allow the human user to analyze it. Then it interacts with the application processes to control their execution according to the user input, in order to enhance its performance.
AIMS [230], Lost Cycles Toolkit [45]	Performance modeling and prediction	These tools integrate monitoring and statistical modeling techniques. Measurements are used to parameterize the model, which is subsequently used for predicting the desired performance metric. *IS* performs the basic data collection tasks.
ParaGraph [81], POLKA [195]	Performance and program visualization	*IS* collects runtime data in the form of time-ordered *trace records*. These trace records are used to drive hard-coded (in the case of ParaGraph) or user-defined (in case of POLKA) visualizations of program's behavior. IS can collect performance data to graphically represent various performance metrics to aid the visual evaluation of the program. IS can also be configured to collect program information (program variables, objects, arrays, lists, etc.) to visually represent the information from application domain.

consists of a network of TAM processes arranged as a broadcast spanning tree with one TAM process (part of the IS) at each node. This configuration allows broadcasting monitoring requests to all nodes. Instrumentation library calls generate data that are sent to the event trace servers, which perform post-processing tasks and write the data to a file or send them directly to an analysis tool. To minimize perturbation, trace records are stored locally in a trace buffer that is periodically flushed to the local trace server.

2.3.2 Commercial Transaction Processing Applications

Transaction processing systems consist of sources of data and services distributed throughout an enterprise with a consistent set of management policies across the system. These systems usually operate under a client-server paradigm using distributed databases

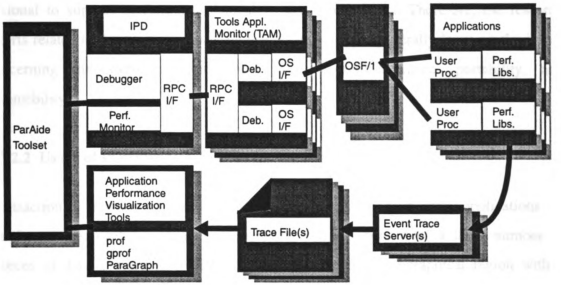


Figure 2-2. ParAide integrated tool environment for Intel Paragon [165].

and services [2]. In such systems, the data and control transfer mechanisms play an important role in integrating and managing the enterprise-wide resources. For such systems, it is often desirable to maintain a single point of control that can replicate a set of operations across a large number of data sources to allow system administration. Instrumentation data are collected to analyze the long-term trends of system resource usage for capacity planning purposes as well as tuning the performance of individual applications. For an IS in this application domain, it is necessary to guarantee that the security of the databases will be maintained. Additionally, ISs need to conform to well-known standards, not only to be useful for a large number of client-server systems but also to be able to interact with a number of heterogeneous information systems within an enterprise.

2.3.2.1 Research and Development

There is an important difference between the ISs for tools that support HPCC applications and ISs for transaction processing systems. The ISs for transaction processing systems, as well as those for several other types of systems, are developed as a part of the system itself

to support important management tasks. The use of ISs for HPCC applications is often optional to support performance evaluation and tuning tasks. Therefore, the research efforts related to the ISs for transaction processing systems generally focus on the issues concerning the software development process, fault-tolerance, portability, and extensibility.

2.3.2.2 Usage of Instrumentation Systems

Transaction processing is one of the most important commercial applications of distributed computing. Transaction processing systems consist of a large number of sources of data and services distributed throughout some geographical region with a consistent set of management policies across the system. The large size of the distributed system and the difficulty inherent in managing it with some acceptable quality of service makes it a complex system. In such systems, the data and control flow mechanisms play an important role in integrating and managing the enterprise-wide distributed resources. An IS helps establish a continuous flow of information to a central or distributed points of control to manage the entire system. Table 2-4 lists only a few of several commercially available tools that monitor transaction processing systems using an instrumentation system.

2.3.2.3 Example of an IS for a Commercial Transaction Processing System

In the example transaction processing system given in this subsection, we consider a *network management and operation support* (NMOS) system used for providing commercial telecommunication services. Telecommunication NMOS systems, such as those used for *AT&T*'s World Wide Intelligent Network, are integrated systems composed of subsystems, each of which may be an NMOS system itself or a generic component. Development of an NMOS system is a multi-phased effort with parts of the system in production while other parts are being developed or deployed. Figure 2-3 depicts the architecture of a typical NMOS system [13]. In this example, the IS-provided services

Table 2-4. Tools for transaction processing systems with IS support.

Representative Tool	Functionality	Description of Key IS Functions
AT&T's NMOS	Telecomm. network management	See Section 2.3.2.3.
A+OpenWatch	Network management	The instrumentation system uses a CMG standard called *Universal Measurement Architecture* (UMA) to model the instrumentation system for both its development and usage. It allows the user to collect application-specific data from Unix-based, enterprise-wide distributed client-server systems. This information can be used with several A+ tools developed by Amdahl. A+OpenWatch is one such tool used for distributed threshold monitoring to allow exception-based network management. **Reference:** http://www.amdahl.com/doc/products/oes/pm.oes/perfhome.html
NonStop TUXEDO	Enterprise transaction processing monitoring	Instrumentation system collects data for monitoring enterprise transactions. This facility is available for heterogeneous Unix-based transaction processing systems. **Reference:** http://www.tandem.com/INFOCTR/HTML/PROD_DES/NSTXDOPD.html
Sybase SQL Server II	Workload adaptability and tunability of applications	Data collected through instrumentation system is used workload adaptability through optimized usage of memory resources. This data is also used for tuning the performance of individual applications also. This facility is available on multiple platforms, ranging from PCs to HPC systems. **Reference:** http://www.sybase.com/Offerings/System11/sqlsrv11.html
DataHub	System management	The tool supports management of the system from a central point of control and performance analysis using DATABASE 2 monitor. Data collected by the instrumentation system is useful for supporting both centralized system management decisions as well as tuning the performance of various transaction processing applications. This facility is available for OS/2 and Unix-based workstations using multiple databases. **Reference:** http://www.software.ibm.com/data/db2/b41amgmt.html
Encina	Administrative services	This tool is used by IBM's AIX-based networked computing environments. Encina monitor uses several AIX features to collect the data and uses it for system administrative services. **Reference:** http://www.austin.ibm.com/software/encina.html

include: *transaction monitoring*, *decision support*, *data streaming*, *communication*, *alarm*, *audit*, *resource management*, *security*, and *visualization*.

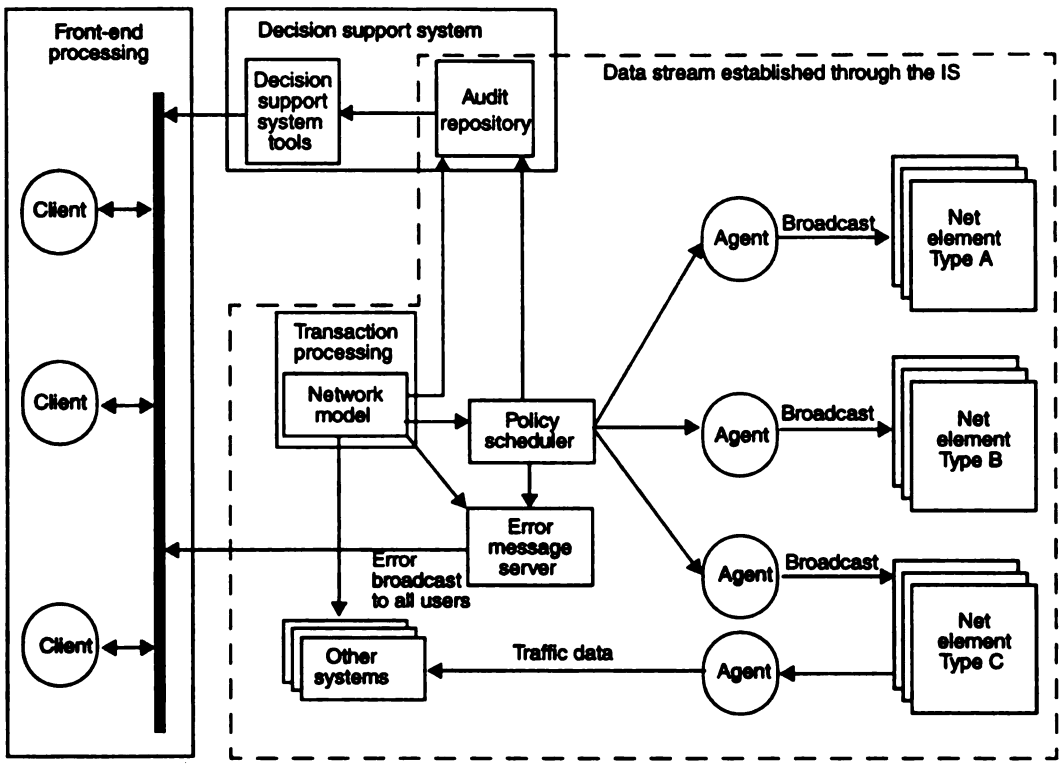


Figure 2-3. AT&T's Signal Operation Platforms-Provisioning (SOP-P) architecture for network management and operations support[13].

2.3.3 Distributed Real-Time Computing Applications

Real-time systems usually interact with real world phenomena that involve time. These systems usually follow a stimulus-response paradigm where the real-time system needs to respond to an external stimulus within a pre-determined deadline for correct overall operation. Distributed real-time systems can be considered a special case of distributed systems that are becoming popular due to the increasing use and availability of distributed computing hardware and software. Unlike HPCC applications, distributed real-time systems are specialized and often embedded. Therefore, the issues related to this application domain are more constrained and better understood. Often, distributed real-time systems are designed as embedded computers in control systems.

Distributed real-time applications also rely on runtime information collected from the distributed real-time tasks. In case of real-time systems, correct real-time behavior of the system is of critical importance in most of the applications instead of the high performance as in case of HPCC systems. If some of the tasks miss their deadlines, it may be of critical importance to either immediately report this information to the system operator or else log this information in a database for a detailed off-line analysis. Such runtime information is perhaps the only reliable source for analyzing failures in highly complex and equally critical real-time subsystems.

2.3.3.1 Research and Development

Distributed real-time systems present a number of interesting as well as non-trivial research issues. For instance, design specification, static analysis, scheduling, fault-tolerance, operating systems, and software tools are some of the active areas of research. From the perspective of instrumentation system development and usage, however, our scope of real-time applications is limited. We consider applications that require runtime data collection for performing real-time tasks. Some of the example include: dynamic resource management, distributed real-time control systems, algorithmic steering of application programs, correctness checking, and monitoring of system health.

Instrumentation can be inserted into a distributed, real-time operating system for dynamic resource management. Mercer et al. apply this technique for dynamic resource management using RT-Mach operating system [132]. Adaptive control of real-time systems is commonly applied in the area of embedded systems. Typical examples of such systems include military combat systems, safety-critical systems, and aircraft and automobile control subsystems. Welch, Masters, and Harrison explore a path-based paradigm for developing a control system software architecture for large-scale, ship-board, distributed, real-time control systems [221]. Gergeleit et al. describe the use of JEWEL IS for a distributed, real-time control system [66].

Reed et al. identify a number of national challenge applications that exhibit irregular structure, data-dependent execution behavior, time-varying resource demand [163]. These applications can benefit from real-time adaptive control of their dynamic behavior. For instance, parallel file system management policies that take the application input/output access patterns into account can increase performance by more than an order of magnitude [92]. Schwan et al. distinguish between two types of adaptive control techniques at an application level: algorithmic steering and interactive steering [50]. Algorithmic steering controls the execution behavior of an application by implementing an algorithm and dose not require the human user to be a part of the closed-loop control system. Interactive steering is based on the explicit user intervention to modify the application behavior. Applying interactive steering approaches and tools, a user can modify an executing application parameters in real-time to improve its performance (i.e., steer it), based on graphical feedback of system states and application behavior [111].

2.3.3.2 Usage of Instrumentation Systems

Distributed systems are becoming more common in safety-critical on-board control systems in the transportation industry, e.g., in aircraft and automobiles. Some systems are faced with real-time constraints. Whereas a missed deadline in a real-time, multimedia system such as on-line video conferencing can result in poor quality voice or video, in a safety-critical system, it could lead to unpredictable, catastrophic behavior. On-board distributed systems used in mission-critical applications in the military often involve highly stringent real-time requirements. Many subsystems may interact to accomplish a series of tasks on time, requiring finely-tuned local and global resource management.

Table 2-5 lists some of the tools for real-time systems that rely on runtime data collection. While most of the existing tools use runtime data to test, analyze, and visualize the real-time subsystem under study, the emerging applications require additional sophistication from these tools by using this data to adaptively and dynamically control that system. System resources are dynamically scheduled among various tasks based on the state of the

system determined from the information collected by the instrumentation system. Scheduling of resources for at least one system node is a well-understood problem and the instrumentation system only provides a bi-directional link between the system (actually sensors and actuators) and the controller (real-time system). It is still a challenging task to implement the controller as a distributed real-time system rather than centralized to make it robust and fail-safe.

Table 2-5. Tools for real-time systems with IS support.

Representative Tool	Functionality	Description of Key IS Functions
SPI [15]	Correctness checking	Scalable Parallel Instrumentation (SPI) is Honeywell's real-time IS for testing and correctness checking on heterogeneous computing systems. SPI supports user-defined application-specific instrumentation development environment, which is based on an event-action model and event specification language. **Reference**: http://www.sac.honeywell.com/
PGRT [174]	Testing and visualization	Instrumentation system collects runtime information (user-specified trace records as well as pre-defined events) from a heterogeneous, distributed real-time embedded system. IS supports an integrated environment consisting of off-the-shelf visualization and analysis tools. Data collected by the IS during testing of an embedded system is used for both on-line and off-line analyses. **Reference**: http://web.egr.msu.edu/VISTA/Pgrt/pgrt.html
JEWEL [114]	On-line monitoring	JEWEL is a commercial, off-the-shelf software product from The German National Research Center for Computer Science. It has been used to setup and control user-defined measurement experiments for embedded real-time platforms, such as Ultrix 4.2 on a MIPS processor, Amoeba and VxWorks on FORCE VME-bus M68030 board, and MACH 3.0 on an i386 single and multiprocessor. **Reference**: http://borneo.gmd.de:80/RS/Papers/JEWEL/JEWEL.html
DIRECT [66]	Adaptive steering	Runtime information collected by the instrumentation system is fed to a dynamic scheduler. Scheduler uses this information to adaptively control the real-time system to make it responsive to the variation of important system variables. **Reference**: http://borneo.gmd.de:80/RS/Papers/direct/direct.html
RMON [132]	Dynamic resource scheduling	RMON monitors the resource usage for distributed multimedia systems running RT-Mach. Information collected by the instrumentation system is used for adaptively managing the system resources through real-time features of the operating system. **Reference**: http://www.cs.cmu.edu/afs/cs.cmu.edu/user/cwm/www/publications.html
Commercial off-the-shelf tools	Military control systems	See Section 2.3.3.3.

2.3.3.3 Example of an IS for a Military Control System

In the military control system example given in this subsection, we consider the shipboard computing system envisioned by the *HiPer-D* Program (*Hi*gh *Per*formance *D*istributed computing Program). The program is conducted jointly by the Department of Defense Advanced Research Projects Agency (ARPA) and the Aegis Shipbuilding Program. It consists of simultaneous top-down engineering studies and large-scale experiments involving mission-critical systems using off-the-shelf computing products. The architecture of a HiPer-D distributed, embedded control system for the Aegis weapon system is shown in Figure 2-4 [79]. It is based on a generic control system architecture. Sensors, e.g., satellites and radar and sonar units, provide sensor data to be processed by the *sense elements* of the system (shown on the left side of the figure), which include radar systems, identification systems, the electronic sensing system, navigation systems, and sonar systems. The sense elements provide data to the *command and decision elements*, which evaluate the data and decide what actions should be taken and when. Actions are carried out by various *act elements* (shown on the right side of the figure), such as gun weapon systems, fire control systems, and launch systems. Act elements schedule actuators and other resources to perform actions and monitor progress of the action. Compute-intensive functions are handled by a mesh-based parallel computing system, which is connected with the rest of the control system through various subnetworks.

In each of the above three areas, ISs impact the behavior of the actual system. However, this impact is not easy to understand due to the following reasons:

1. there is no simple model that can characterize the measurement system and exactly account for the IS intrusion;
2. an IS usually intrudes the SUT behavior in a non-deterministic manner, which makes it even harder to account for the intrusion; and
3. an IS is designed to provide the measurements regarding the states and behavior of the SUT and not to measure its own overhead.

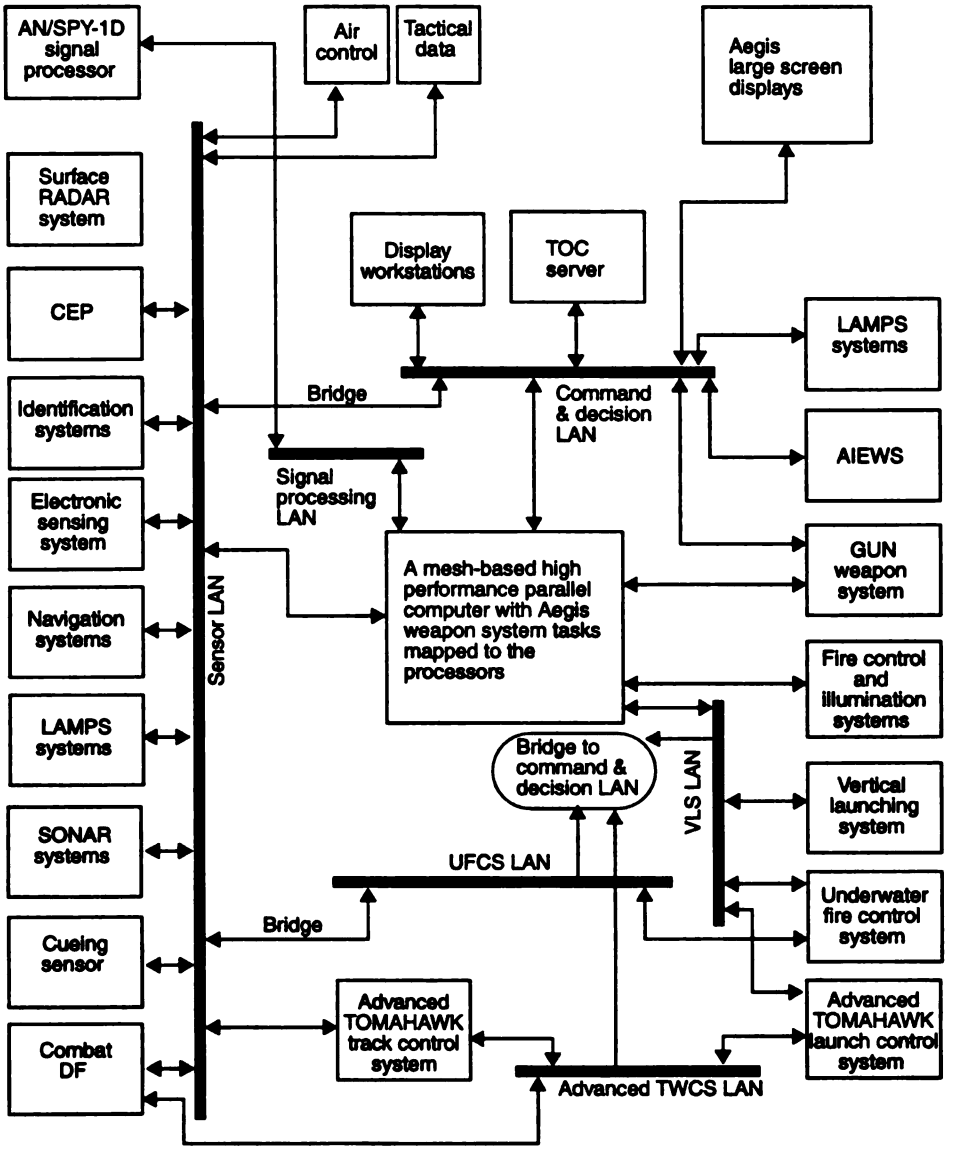


Figure 2-4. Aegis weapon system based on HiPer-D shipboard computing system [79,221].

Therefore, unless an IS is designed with proper evaluation of its overhead and intrusion, it can cause undesirable problems that may range from poor performance to catastrophic failures depending on the type of the target application. Application of the design, modeling, and evaluation methodology presented in this dissertation can help the IS developers to avoid such problems.

2.4 An Overview of Computer System Modeling Techniques

In general, computer system performance modeling is considered a multidisciplinary area. Information from diverse disciplines such as computer architecture, operating systems, stochastic processes, operations analysis, and statistics contribute toward the modeling and evaluation of different types of computer systems. Since the research presented in this dissertation is based on some of the above disciplines, it is appropriate to present an overview of the well-known performance modeling and evaluation techniques in this section.

Performance analysts believe that the pace of developments in computer system design has always overwhelmed the development of adequate and unified theoretical characterization efforts for these systems [53,129]. Nevertheless, importance of appropriate analysis methods becomes obvious when the system complexity increases [54]. Analytical methods are based on building appropriate mathematical models for the computer system (and computation) to better understand the system and provide insight to the designer. Such models are most appropriate for making various decisions at the design stage of such a system, by analyzing the behavior of the system using alternative architectural choices. The complexity of analytical techniques may result from the abstract nature of the performance evaluation objectives, such as, optimizations applied to hardware, operating systems, compilers, networks, and so on. A number of commercially available parallel and distributed systems have been developed under different design and performance goals. Their relative merits are not yet fully understood to enable the system designer to use appropriate models for the system to analyze and predict the performance to compare various design alternatives. Therefore, the evaluation of analytical performance models is still an actively researched problem.

We review the discipline of analytical modeling based on the models/methods that are often applied to parallel systems. Ferrari [53] divides an analytic study of a computer system into three parts: (1) model formulation; (2) model solution (or simulation); and (3)

model validation. Analytical models are solved either symbolically or numerically, in order to calculate desired metrics. A model of a parallel system consists of two parts [129]:

1. description of the architecture, and
2. definition of the workload under which performance predictions are to be obtained.

Figure 2-5 represents a general framework for evaluating analytical models according to the steps described above. It should be noticed that various architectural components are expected to behave in a non-deterministic manner under a given workload. Therefore, a realistic analytical model has to be stochastic, in general.

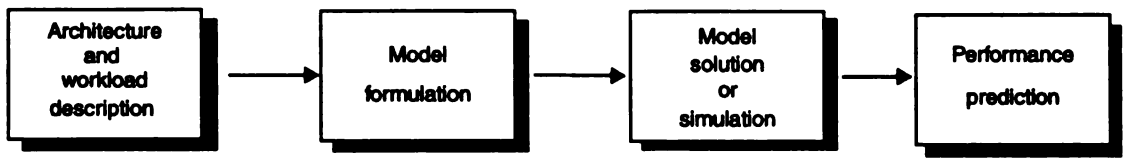


Figure 2-5. Phases of an analytical study of a parallel system.

In the following subsections, we survey four generic computer system modeling techniques: Markov models, queuing models, Petri nets, and simulation models; and survey a few tools based on these modeling methods.

2.4.1 Markov Models

Markov models (also known as *Markov processes*) are a compromise between the real-world dependences among various physical phenomena and the theoretical requirement of "independent" phenomena to make the calculations tractable [39,164]. The class of Markov processes that is often used for computer system modeling is called *Markov chain* processes. Markov chains are built by considering the stochastic process as a set of states that are visited once or repeatedly over time. This set of states is termed as the *state-space* of the system. A Markov chain process is defined as one whose transition to a future state

from its present state depends only on the present state, and not the complete past history of its states up to the present state. This type of dependence structure proves a powerful tool for building models of complex real-world phenomena that can be solved.

As noted by Sauer and Chandy [182], there are three issues involved in modeling a computer system as Markov chain processes:

1. defining the process as a Markov chain according to the definition of a Markov chain given above by analyzing the dependences of various states of the process;
2. mapping computer system models to Markov processes; and
3. solving Markov model.

Since a multicomputer system exhibits highly dynamic behavior that strongly depends on contention for and sharing of system resources at a given time, a model of such a system relies on representing the internal states. A detailed representation of internal states of the system (i.e., dependences among them) is sufficiently complex and leaves us without any methods to solve the model other than simulations. This will happen when we want to analyze the states of the system in response to instruction and data streams of individual programs. The representation of current and past states provides a "memory" of current and past states of the system. If this memory includes very few details, then it will be difficult to predict the future states accurately. However, inclusion of a number of states makes the numerical solution of the model impractical. We can make up a Markov model of a given computer system as described (in words) in the following [182]:

> *Assume that we represent all possible states of the system by a set of mutually exclusive and collectively exhaustive states. Also assume that the future behavior of the system depends only on the current state of the system and is independent of previous states of the system. The times between corresponding entrances to and departures from ("holding times" or "transition times") are independent and identically (exponentially) distributed. Then the states of this system with a set of transition probabilities between these states correspond to a Markov chain process.*

After setting up this model, the next step is to represent the state transitions in a matrix form. This representation reduces the model-based calculations to simple matrix

manipulations [3,164]. The transition probabilities represented by the matrix can also be represented graphically as shown in Figure 2-6. The circles show the states of the system during the execution of a program and p_{ij} are the probabilities of transition from state i to j.

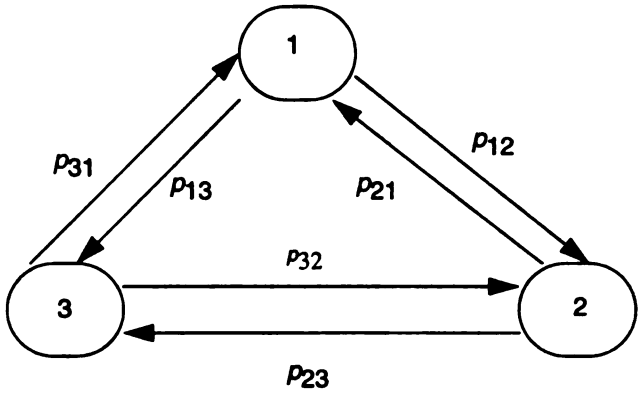


Figure 2-6. Markov Chain representation of the system-program behavior.

The solution of Markov models for more complex structures has been an active area of research. Iazeolla et al. [96] describe a technique of solving these models utilizing fully symbolic and computer-algebra based methods at earlier stages of the Markovian process analysis and using numerical methods at the latter stages. Anton and Rorres [6] provide some examples of solving these models by purely numerical techniques, which work well for predicting future states of the system with simple models.

Markov models have been applied to various aspects of parallel and distributed computer system behavior and performance. Ahluwalia and Singhal [3] analyze the performance of the interprocessor communication architecture of the CM-2 by developing a discrete-time Markov chain model of its network architecture. The model shows the use of simplifying assumptions in a practical situation and predicted results compare favorably with simulation. Abrams et al. [1] analyze the time-dependent behavior of programs using the program execution sequence to create a homogeneous semi-Markov chain model. This

model can predict the system performance with program parameters different than those used in the input program execution sequence.

2.4.2 Queuing Models

Queuing models are useful analytical tools for systems that develop conflicts when multiple entities simultaneously try to access the same resource. Queuing models are frequently applied to multicomputer networks, communication protocols, contention of resources among various jobs in a multiprogramming environment, and so on [33,68,106,119]. The study of queuing models is a discipline in its own right, known as *queuing theory.*

Queuing models are based on two abstractions: *servers* and *customers*. Servers are the resources of a system that need to be shared or utilized by customers. A *queue* is another abstraction in this model where customers arrive and wait for the service, while the servers are busy serving other customers. The basic structure of this model is represented by Figure 2-7, which shows servers as rectangles and customers as circles. A queue is represented by a line of customers with vertical bars on both sides. Queuing models consisting of more than one server are referred to as *queuing network* models. There are two stochastic processes that have to be defined to parameterize this model. These processes are: (1) inter-arrival time distribution of customers (input process), and (2) service time distribution of each of the customers. The probabilistic definition of these parameters (i.e., their probability distributions) determines the type of queuing model for such a system. Once this model is determined, it can help answer the following type of questions:

1. What is the average waiting time for a customer in the queue?
2. What is the average queue length?
3. What is the optimal number of servers needed for this system?
4. What about having a separate queue for each server? Will it be optimal compared to a single queue?

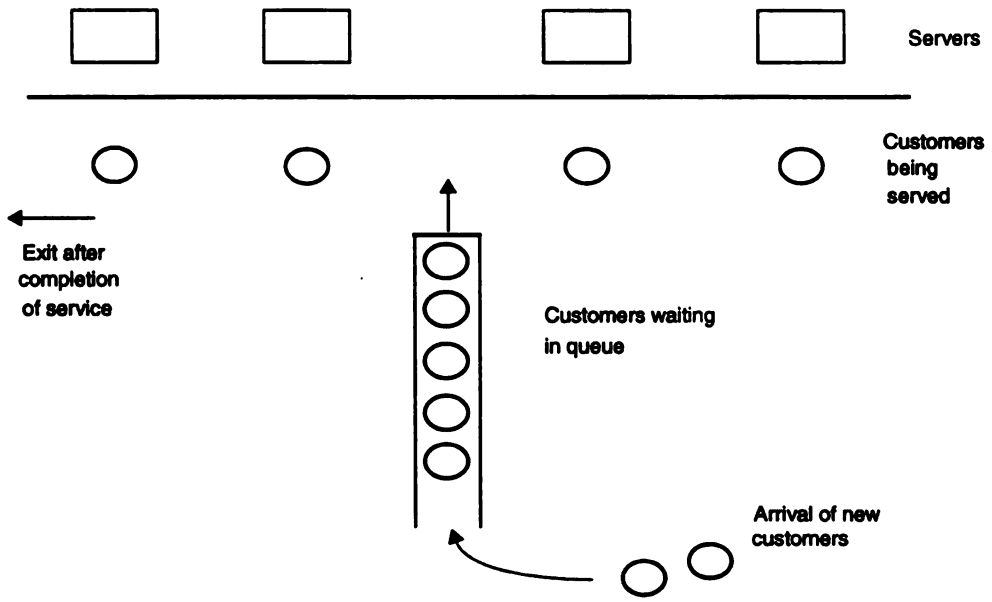


Figure 2-7. Ingredients of a basic queuing model.

It can be appreciated that these questions are of a general nature and can arise in any phenomenon that is being analyzed by queuing models. The answers are important with respect to computer system performance (e.g., its throughput and response time). For modeling a computer system, often three types of devices are encountered [100]. These devices are:

- Devices that have a single server, whose service time does not depend on the number of jobs in the device. Such devices are called *fixed-capacity service centers*. For instance, CPUs in a system may be modeled as fixed-capacity service centers.
- Devices that do not exhibit "queuing" behavior, and jobs spend the same amount of time in the device regardless of the number of jobs in them. These devices can be modeled as a service center with infinite number of servers and are called *delay centers*. A group of dedicated terminals is usually modeled as a delay center.
- Devices whose service rates may depend on the load (i.e., number of jobs in the device) are called *load-dependent service centers*. An interconnecting network between the nodes of a parallel computer system is an example of a load-dependent service center.

Tsuei and Vernon [203] use a queuing network to model a commercial multiprocessor bus. Important characteristics of the bus such as asynchronous memory write operations, in-order delivery of responses to processor read requests, priority scheduling of memory responses, upper bound on the number of outstanding processor requests, and so on, can be modeled accurately. Kleinrock [106] discusses several computer time-sharing and networking examples using queuing models to evaluate their performance. Gelenbe [68] presents an application of regenerative processes to simplify the queuing models that arise in various computer systems.

2.4.3 Petri Nets

A Petri net is a graphical modeling tool for description and analysis of concurrence and synchronization in parallel systems. Petri nets were introduced by C. A. Petri in 1962 and are widely used to model asynchronous systems and concurrent processes. The theoretical problems associated with Petri nets have been thoroughly investigated, and therefore, it has sufficient mathematical structure to support formal analysis of parallel systems. The success of Petri nets is mainly due to the simplicity of the model that works well to depict complex large-scale systems. Many extensions have been added to the basic Petri net model to facilitate their use for different application fields. These extensions include timed Petri nets for quantitative performance analysis of systems, stochastic Petri nets which uses random variables to specify the behavior of the model with time, as well as others [129]. Stochastic Petri nets are considered more attractive as a modeling tool for analysis of multiprocessor systems.

The structure of a standard Petri net is a graph that consists of *places*, a set of *transitions*, and a set of *directed arcs*. A place represents some conditions and a transition represents an event. Arcs connect places and transitions to each other. A place is an input to a transition if an arc exists from the place to the transition, and is an output of a transition if an arc exists from the transition to the place. Therefore, the set of arcs can be partitioned into a set of transition input arcs and a set of transition output arcs.

Figure 2-8 is a graphical representation of an example Petri net, taken from [129]. Circles (or nodes) represent places and bars represent transitions. Tokens are placed on place nodes to represent that certain condition is held on that node. A transition bar (an event) can fire (occur) if all the nodes (conditions) that input to that transition bar have tokens (i.e., are holding conditions). When a transition bar fires, it removes one token from each of its connecting input nodes and places one token on each of its connecting output nodes [31]. This firing of a transition is called the *execution* of the Petri net. For instance, if P_1 in Figure 2-8 contains a token, then transition t_1 will be enabled. States of a Petri net can be determined by observing the collection of names of the nodes that hold tokens. The number of tokens a node holds is equal to the number of instances of a node name in the state. This procedure is called *marking* of Petri nets.

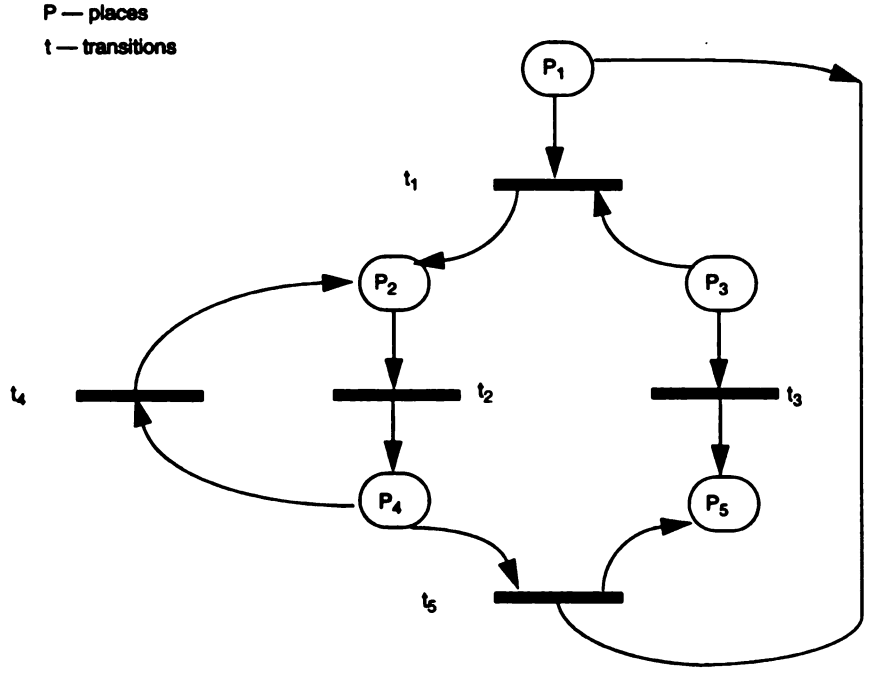


Figure 2-8. Example of a Petri net.

Petri nets and its variations are graphical, mathematical tools that can model the following characteristics of a concurrent computer system [31]:

- representation of concurrent execution of multiple processes;
- representation of nondeterministic and asynchronous executions;
- decomposition and composition into many graphs; and
- representation of model's structure and dynamic behavior.

Petri nets are also used to model the behavior of a processing unit that needs a resource to perform some action [129], the behavior of file systems [31], etc.

2.4.4 Simulation Modeling

Simulation models are used to model the operation of a system, rather than the structural components of the system [12,116,123,131]. Simulation is a useful technique to solve an analytical model of a complex system that is impractical to be solved by analytical techniques. Usually, simulation is used in conjunction with the analytical modeling of a system to verify results. Simulation models have their own advantages, in addition to being a tool for verifying analytical models. Simulation allows a more detailed study of a system than analytical models. In this case, a more detailed model is considered better as it makes fewer assumptions. Simulation studies are useful for analysis of systems even after they are realized because these models do not perturb the system behavior for analyzing it.

There are several types of simulations used in computer system performance analysis. These include emulation, Monte Carlo simulation, trace-driven simulation, and discrete-event simulation. Simulation models are translated into simulation programs that generate statistical information regarding the model. This simulation data is analyzed by various statistical techniques [100].

2.4.5 Workload Characterization

Workload characterization complements a computer system model. After a suitable modeling technique has been selected for a performance study, we have to characterize the

work performed by that system. A computer system model is solved analytically or simulated with respect to a particular workload characterization.

Workload characterization is the process of appropriately representing and modeling a set of programs that a computer system executes. It complements the physical description of the architecture of a computer system and represents the system behavior due to the activities of operating system and user processes that utilize system resources. Workload characterization is used for both analytical solution as well as simulation of a model.

Workload characterization is usually the most time consuming aspect of a typical system modeling effort. Large volumes of low-level measurement data are used to identify clusters of interesting system activity and transitions from one type of activity to another using a representative mix of programs (for instance, see the studies conducted by Dimpsey et al. [47] and Hughes [94]). Current software technology and rapid-prototyping tools have greatly reduced the turn around time of a software system development project; therefore, a prolonged workload characterization process may yield accurate results but those results may no longer be useful for the developers. A workload characterization effort with the following features is desirable to evaluate the performance at an early stage of development: (1) short turn around time; (2) applicability to only a specific application instead of targeting generality; and (3) less dependence on low-level measurement data and more dependence on the knowledge about the application-domain. This type of workload characterization is increasingly becoming popular for performance prediction studies that use a simulation model, which is parameterized for a particular parallel or distributed system using only high-level, coarse-grained measurements [37,231].

2.5 Related Work

In this section, we put the design, modeling, and evaluation approach presented in this dissertation into proper perspective by reviewing the related work. We have categorized the related work into three areas according to the foci of this research: IS characterization

for diverse application areas of parallel and distributed computing, IS design and development, and IS modeling and evaluation efforts.

2.5.1 IS Characterization

In order to present an overview of IS characterization efforts, we have to use the definition and scope of an IS as presented in this dissertation. Several related research efforts have focused on characterizing an instrumentation system either without considering it as an entity independent of the rest of the system, or else without putting the IS into a broader perspective of diverse application areas. Additionally, the terminology used by different researchers is not consistent. These efforts, however, have contributed to the state-of-the-art and cannot be overlooked.

At a gross level, Jain categorizes an IS into three types: *hardware monitor*, *software monitor*, and a *hybrid monitor* [100]. This terminology was originally meant for uniprocessor systems but parallel and distributed systems inherited it without any modifications. Since this characterization is widely accepted, tool developers or users do not have any problem categorizing an IS in terms of these three types. The trade-offs among these techniques are also well understood. A hardware monitor is least intrusive but it is not suitable for providing information about process level events, such as context switches, page faults, etc. A software monitor can collect application level information but its penalty for the target application is usually very high.

Hollingsworth et al. characterize hardware instrumentation based on monitoring low-level activity on the processor bus into two types: *passive hardware monitors* and *trace co-processors* [88]. Passive hardware monitors are implemented as programmable counters, which count the events on the bus, in the memory hierarchy, and on the processor. Many state-of-the-art processors provide hardware counters and operating system level interfaces to implement a hybrid instrumentation system. Zagha et al. describe an integrated performance evaluation setup based on hardware monitoring support in MIPS

R10000 processor, operating system abstractions, and performance tools [233]. Multikron is also an example of a custom VLSI co-processor for monitoring, which appears as a memory mapped device to the main processor [138]. The co-processor collects the event data and sends it to a central data reduction station via a special-purpose monitoring bus.

Although the gross level characterization of an IS in terms of hardware, software, and hybrid instrumentation is useful for a uniprocessor system, it is not sufficient for a parallel or distributed system. A parallel or distributed system consists of a number of physically distributed processes with complex interactions with one another; such systems require many more IS modules and services for runtime data collection than deemed necessary on a uniprocessor system. Therefore, we should view an IS as a distributed system itself within the target parallel or distributed system. Joyce et al. recognize the instrumentation needs of a distributed system and present a generic IS architecture based on a distributed programming environment, called Jade [101]. This work also recognizes the fact that the data collected by the IS can be used for different types of tools, such as debugging, performance evaluation, and testing in the Jade environment. The generic IS architecture consists of monitorable processes and events, channels and controllers, communication mechanism, and consoles. An application process is characterized as *monitorable* or *unmonitorable*. A *monitorable event* is defined as any process operation that may have an effect outside of that process. A *channel* process resides on each machine being monitored and collects runtime information from locally executing processes. User can introduce a *controller* process, downstream from a channel process, in order to control the order of events. Communication of the monitoring information can be handled in one of two possible ways: use of customized inter-process communication (IPC) mechanisms; and use of the same IPC mechanism, which is used by the application processes. A *console* receives monitoring information from one or multiple channels, examines and interprets it, and finally presents it to the user. A console may be running on a different machine than the channels. Although this characterization is fairly detailed, it may be difficult to apply it to an IS in an environment or application other than Jade.

Ogle, Schwan, and Snodgrass [146] characterize a distributed instrumentation system in terms of a set of *resident monitors* and a *central monitor* in Issos environment. A resident monitor resides on each system node, collects and analyzes the runtime information about local application processes, and reports this information to a central monitor. The central monitor executes on a system node on which a *monitoring database* is stored. The central monitor collects the distributed information, interacts with the tools in Issos, and provides a user interface. Data is collected through *sensors* and *probes*. Sensors are defined as small pieces of code residing within the instrumented application processes. Sensors are applied to event-driven data collection. Probes are defined as code fragments residing within a resident monitor rather than the application process. Probes can directly access the address space of an application process on a system node and are useful for sampling-driven data collection. Collected information is stored in the monitoring database with time-stamps and locations of occurrence. The main goal of this characterization is to support application-dependent data collection. This characterization is sufficient for ISs that operate in an "open-loop" configuration; however, it does not consider the applications where an IS operate in a "closed-loop" as a part of an adaptive real-time control system.

Lange et al. characterize the IS of JEWEL distributed measurement, analysis, and visualization system as a Data Collection and Reduction System (DCRS) [114]. The architecture of the JEWEL IS (i.e., DCRS) consists of sensors, collectors, evaluators, and mediators. See Section 3.3 for details about the JEWEL IS architecture. Although the developers of JEWEL did not originally intend to characterize a generic IS, the architecture of JEWEL IS has the potential to be extensible and retargettable to diverse applications and systems.

Hollingsworth, Lumpp, and Miller [88] term a measurement-based experiment as *instrumentation*, and characterize it as opposed to an instrumentation system. While an IS characterization focuses at its architecture in terms of modules and services, a characterization of the instrumentation is concerned with the use of an IS for

measurement-based experiments. Instrumentation is characterized at an abstract level in terms of following six attributes:

1. *Program Instrumentation*: it refers to the insertion of instrumentation code in the application program. Program instrumentation can be accomplished in one of four ways: direct insertion of instrumentation into the source code; automatic insertion of instrumentation by a modified compiler; linking the object code with a runtime library for instrumentation; and modification of the linked executable.
2. *System Instrumentation*: instrumentation data can be collected at the system level rather than the application level. It can be accomplished through dedicated monitoring processes or instrumenting the operating system.
3. *Hardware Instrumentation*: instrumentation data can be collected in a non-intrusive manner using hardware instrumentation. Hardware instrumentation is categorized in terms of using passive monitoring with hardware counters, trace co-processors, and hybrid monitoring.
4. *Event Specification*: it refers to the specification of the information to be collected related the occurrence of an application event. Event specification languages are commonly used to provide a convenient mechanism for a user to specify the useful information with respect to an event.
5. *Event Filtering*: this mechanism is important to limit the volume of performance data that should be collected during a measurement-based experiment. One event filtering technique is to use predicates and actions to recognize the desired events. Another approach is to apply dynamic instrumentation control mechanisms [85].
6. *Perturbation Compensation*: there are four ways to handle intrusion due to instrumentation: realize that intrusion affects measurement and treat the instrumentation data as an approximation to the actual execution; leave the added instrumentation in the final implementation of the target application; try to minimize the intrusion; and quantify the intrusion. Perturbation compensation techniques are based on quantifying the effects of intrusion and correcting the collected information.

An instrumentation system is used to conduct measurement-based experiments. Characterization of instrumentation is helpful for the user of an IS for designing a measurement-based experiment. Although the scope of the above characterization is limited to performance measurement of parallel programs, it complements the IS characterization efforts presented earlier in this subsection.

The IS characterization efforts presented in this subsection have influenced our IS characterization that will be presented in Section 4.1. Our IS characterization effort differs

from earlier efforts in two respects: it considers multidisciplinary applications rather than restricting to a particular application such as parallel program performance measurement; and it is useful for developing as well as evaluating an IS.

2.5.2 IS Design and Development Efforts

A software instrumentation system is designed to work at one of three possible levels: system level, runtime library level, and application level. System level instrumentation support is usually built into the operating system and can be enabled for an application. It results in low-level information, such as execution times of various functions, count of function calls in a program, process and or thread state transitions, thread scheduling, system call entry and exit events, page faults, address space swapping, and I/O operations. Examples of such ISs include Unix gprof, Solaris *Trace Normal Form* (TNF) kernel probes, and IBM AIX tracing facility. Runtime library level instrumentation is used for collecting information related to high level functions, such as unicast and multicast communication, synchronization, and I/O. Examples of ISs developed at this level include PICL [62], PVM [63], and various implementations of MPI [69,133] that support instrumentation. Finally, the ISs that provide application level instrumentation, allow the user to select instrumentation points or events of interest that must be monitored. User can access information that is directly useful from the perspective of a given application. Examples of such ISs include Pablo [161], AIMS [230], JEWEL [114], and SPI [15]. The level of instrumentation plays a major part in designing, managing, and evaluating an IS.

Several recent efforts are focusing at the design of an instrumentation system as an independent subsystem that can be configured or retargeted for different types of applications and systems. These efforts include standardization of IS management and control, standardization of instrumentation data representation, standardization of IS interfaces, languages for performance metric specification, and development of configurable IS modules.

Standardization in terms of software development is an essential requirement for commercial computing to ensure flexibility and interoperability of the product. The Performance Management Working Group (PMWG), which is a part of the Computer Measurement Group (CMG), was formed to address the needs for collection, management, and distribution of performance data. The resulting specification is called the (proposed) Universal Measurement Architecture (UMA) standard. The objectives of UMA include shared instrumentation system management and control facilities, transparent access of various application to the IS, common data storage, and extensibility. Although this standard focuses more specifically on Unix-based commercial client-server systems to collect performance and accounting data, the software architecture is equally relevant for other application domains.Figure 2-9 shows various layers and interfaces of UMA model.

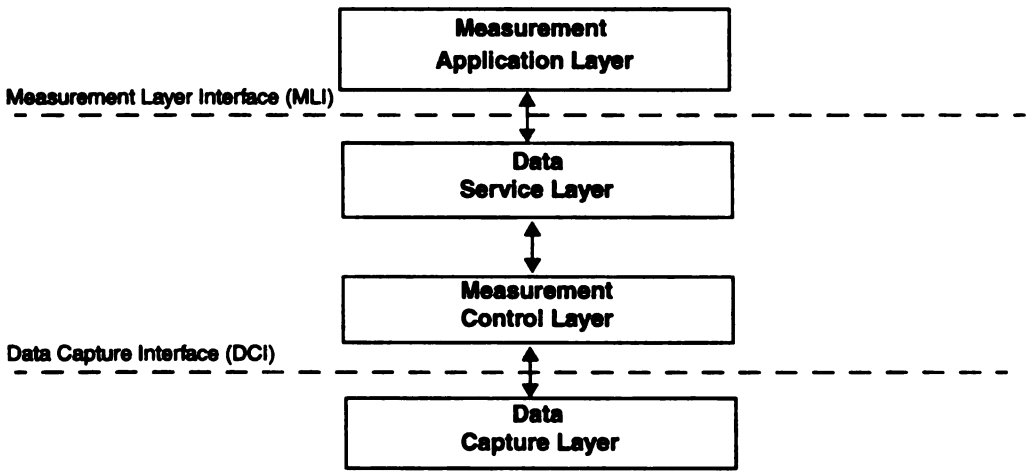


Figure 2-9. Universal Measurement Architecture (UMA) layers and interfaces.

The data capture layer of the UMA is responsible for collecting the runtime information from the target system. This architecture allows the data to be collected from heterogeneous sources by a single layer. The data capture interface (DCI) joins the measurement control layer and the data capture layer. This interface allows the dynamic addition of new data sources in the IS. The measurement control layer schedules and synchronizes data collection and management activities. Data service layer accepts data

collection requests from the applications through MLI. The MLI allows a transparent interaction between application and the rest of the IS.

A similar standardization effort is On-line Monitoring Interface Standard (OMIS) [148]. This effort is more specifically focused on the standardization of the interface between an on-line IS and a tool. The objective is to facilitate the use of the same IS for different types of tools. An IS can be developed such that it conforms to the OMIS specifications to make it usable by different application and tool developers.

A problem related to the use of an IS as a plug-and-play module is the specification of a standard instrumentation data representation. Such a standard facilitates the task of application and tool developers to use the instrumentation data, which might be generated by different ISs. The Pablo Self-Defining Data Format (SDDF) is a notable effort in this regard. SDDF is a data description language that specifies the instrumentation data record structure and instances of data records (i.e., the actual instrumentation data) that can be used according to the specifications [161].

Specifications for IS to indicate what data should be collected are an important part of the design of an IS. Usually, this information is "hard-coded" in the IS and tool environments. For instance, the JEWEL IS allows the user to specify a measurement-based experiment using *aspects* [114]. An aspect is defined as a measurement-based data collection abstraction to identify specific events of interest in a distributed system. The JEWEL IS has to be customized (i.e., recompiled and relinked) to include this aspect information in an experiment. In other cases, it is possible to flexibly specify this information through a description language. For instance, Issos [146], SPI [15], and Paradyn [89] use language-based approaches to specify the data to be collected.

IS design and development efforts are influenced by the emerging software development techniques as well as the standardization efforts in other areas. As new technologies and

standards are emerging, IS design and development methodologies also continue to evolve.

2.5.3 IS Modeling and Evaluation

There are very few examples where tool developers either perform, provide, or document an evaluation of their IS overheads through testing with real applications or synthetic programs. In particular, we are not aware of this type of evaluation being performed concurrently with tool design and implementation processes. Paradyn is a notable example in which tool developers provide an adaptive cost model to predict the overhead to an application program due to the IS [87]. This cost model is continually updated in response to actual measurements during instrumented program execution. SPI ensures that the invasiveness of its IS is accountable [15]. It measures the instrumentation load on nodes and links in each specified window of time to evaluate the degree of invasiveness relative to an application program.

Falcon [71] is perhaps the only tool that supports a thorough evaluation of the modules of its instrumentation system. Perturbation to programs is measured under different conditions of tracing rates, event record lengths, and event buffer sizes. On-the-fly ordering of event records, which is needed for meaningful visualization, is evaluated as a ratio of out-of-order events that need to be "held back." This *hold-back ratio* is found to be sensitive to the size of trace data buffers. Additionally, IS performance is compared with other standard instrumentation tools, such as *Gprof,* using the same metrics for overheads. Such meticulous and practical evaluation of IS performance by the developers provides essential information to the users, especially when an IS is used under real-time constraints. IPS-2 [134] also reports overhead measurements for application programs in comparison with *Gprof.*

Work has been done on compensating for the effects of program perturbation due to instrumentation. The goal of perturbation compensation is to reconstruct the actual

program behavior from the perturbed behavior as it may be recorded by the IS. Malony et al. [125] describe a model for removing the effects of perturbation from the traces of parallel program executions.

Presently, it is not standard practice to formally evaluate the performance and functionality of a tool early in its development. Usability and efficiency studies of prototypical tools are emerging to alleviate this situation. However, the underlying IS is removed from the end-user and is part of system infrastructure, thus necessitating more rigorous evaluation. Moreover, contemporary approaches to evaluate IS overheads and perturbation do not adequately consider the nondeterministic nature of these effects. The approach introduced in this paper has addressed these issues. Table 2-6 summarizes the different evaluation approaches that are being used for existing ISs.

Table 2-6. Classification of IS evaluation approaches.

Extant tools	IS Evaluation Approach	Description of the Evaluation Approach with respect to the Tools
Several existing tools [213]	Ad hoc	Tools are developed and then incrementally modified to correct any performance problems that are discovered during production.
ChaosMON and Issos [104,146]	Heuristic	Tools developers can use their knowledge of target architecture as well as the design of their *IS* and conjecture about overheads.
Falcon [71]	Benchmarking	Synthetic benchmarks can be used to exercise various data collection, forwarding, and management services of an IS. A running version of the tool is required to adopt this evaluation approach.
IPS-2 [134]	Measurement-based	*IS* overheads due to a tool can be assessed by comparing the performance of instrumented version of a program with an uninstrumented version of the same. Then overheads can also be compared by using a different *IS* to collect performance data.
[125,127]	Analytical modeling	Analytical perturbation analysis can be used directly on the source code. This analysis can be used by the IS or tools to remove the effect of perturbation for the program behavior being represented by the instrumentation data.
Paradyn [136]	Performance modeling and evaluation	Well-known performance modeling techniques can be used to model the *IS* and then evaluate the desired performance metrics using analytic and/or simulation techniques. This evaluation can be performed at the time of designing an *IS* before it is actually coded or put in production.

Development and usage of instrumentation systems is directly related to the development of new measurement-based tool environments and applications that need data collection. Software tool environments for parallel and distributed systems is known to be an area of growing importance to the users [151,162]. Similarly, the application of parallel and distributed computing paradigms to diverse, complex systems that use runtime collected data is expanding [174]. Therefore, the background and related work presented in this chapter depicts the current state of knowledge in IS design, development, usage, and evaluation. The state-of-the-art in this area continues to evolve and mature as a discipline in its own right.

In this chapter, we provided a comprehensive background of the research presented in this dissertation. In addition, we also surveyed two areas related to this research: IS development and usage; and techniques for computer system performance modeling. We also presented an overview of the efforts related to three goals of this research: IS characterization; IS design and development; and IS modeling and evaluation.

Chapter 3

Reference Instrumentation Systems

In this chapter, we introduce the instrumentation systems used in the subsequent chapters of this dissertation as case studies of our design, modeling, management, and evaluation approaches. Although the choice of these ISs may appear somewhat arbitrary, they represent a broad range of the state-of-the-art ISs for parallel tools [217]. These ISs include: PICL [61], Paradyn [136], and JEWEL [114] ISs. In general, these tools support measurement-based evaluation of parallel and distributed systems. At the time of their modeling and evaluation, these tools were at different stages in their development and usage life-cycle.

In the following sections, we briefly overview the tools with respect to the functionality of our reference ISs. We also discuss the domain-specific requirements and constraints that the design of these tools and their ISs should address.

3.1 PICL IS

Portable Instrumented Communication Library (PICL), designed at Oak Ridge National Laboratory, provides efficient communication functions that are easily portable to various multicomputer and distributed computing platforms [61]. Instrumentation is an additional feature, and when combined with a tool such as ParaGraph, it supports program performance analysis and animation [81].

3.1.1 Overview of Functionality

In order to instrument an application program, PICL library functions are inserted into the program before compilation. During program execution, calls to these functions generate

instrumentation data in a particular event record format and log these data to a local buffer at each node of the parallel system. The user can specify the size of the buffer. These buffers are typically flushed at the end of program execution and merged into a single trace file at the host system. Figure 3-1 illustrates the functionality of PICL instrumentation system.

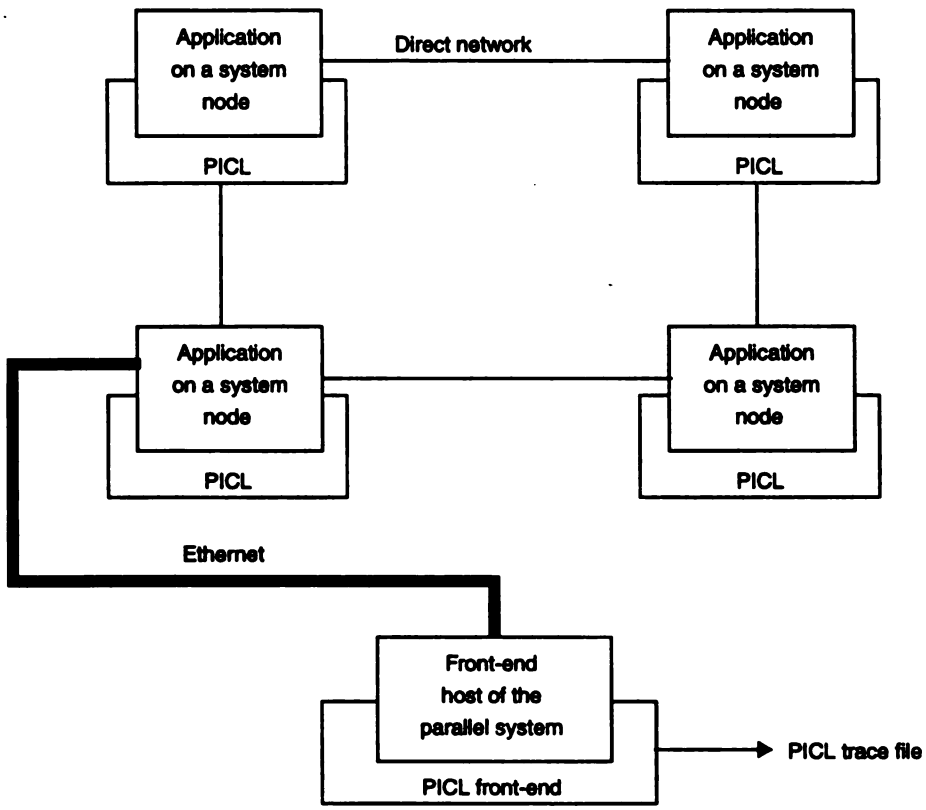


Figure 3-1. Overview of PICL IS functionality.

When a PICL library instrumented program executes, it generates a single trace file consisting of trace records in ASCII format. Each line of this file (i.e., trace record) corresponds to a traced event of interest at one of the processors in the multicomputer system. Each trace record has the following fields: record type, event type, timestamp, processor number, process number, and the number of additional data fields associated with the trace record. Figure 3-2 shows a part of a typical PICL trace file.

```
-3 -2 0.000360 1 -1 0
-3 -2 0.000362 3 -1 0
-3 -52 0.000476 3 -1 1 2 0
-3 -2 0.000512 0 -1 0
-3 -21 0.000647 0 -1 3 2 4 0 1
-4 -21 0.000965 0 -1 0
-4 -52 0.001007 1 0 3 2 4 0 0
-3 -21 0.001108 0 -1 3 2 4 0 2
-4 -2 0.001323 1 -1 0
-4 -21 0.001416 0 -1 0
-3 -2 0.001531 1 -1 0
-3 -21 0.001547 0 -1 3 2 4 0 3
-4 -52 0.001553 2 0 3 2 4 0 0
```

Figure 3-2. Example of a PICL trace file.

3.1.2 Domain-Specific Requirements

PICL instrumentation system is used for event-driven tracing of parallel programs written according to the Single-Program, Multiple-Data (SPMD) paradigm. These programs are often numerical solvers for scientific problems or simulations of physical phenomenon that often run for long periods of time. In order to trace such long-running programs, the IS is required to handle large volumes of data. In case of the PICL IS, every node stores the trace records in a local trace data buffer. Appropriate buffer management policies must be adopted by the PICL IS to be useful for long-running practical applications on parallel systems. Performance of such applications is often sensitive to IS intrusion; therefore, PICL IS should ensure minimum intrusion even for tracing long-running programs.

3.2 Paradyn IS

Paradyn is a tool being developed at the University of Wisconsin for measuring the performance of large-scale parallel programs. Its goal is to provide detailed, flexible performance information without incurring the space and time overheads typically

associated with trace-based tools [136]. The Paradyn parallel performance measurement tool runs on TMC CM-5, IBM SP-2, and clusters of Unix workstations. The tool consists of the main Paradyn process, one or more Paradyn daemons, and external visualization processes.

The main Paradyn process is the central part of the tool, which is implemented as a multithreaded process. It includes the *Performance Consultant*, *Data Manager*, and *User Interface Manager*. The Data Manager handles requests from other threads for data collection, delivers performance data collected from the Paradyn daemon(s), and distributes performance metrics. The User Interface Manager provides visual access to the system's main controls and performance data. The Performance Consultant controls the automated search for performance problems, requesting and receiving performance data from the Data Manager.

3.2.1 Overview of Functionality

Paradyn daemons are responsible for inserting the requested instrumentation into the executing processes being monitored. The Paradyn IS supports the W^3 search algorithm implemented by the Performance Consultant for on-the-fly bottleneck searching by periodically providing instrumentation data to the main Paradyn process [86]. Required instrumentation data samples are collected from the application processes executing on each node of the system. These samples are collected by the local Paradyn daemon (Pd) through Unix pipes, which forwards them to the main process. Figure 3-3 represents the overall structure of the Paradyn *IS*. In the figure, p_j^i for j =0,1,..., n-1 denote the application processes that are instrumented by a local Paradyn daemon at node i, where the number of application processes n at a given node may differ from another node.

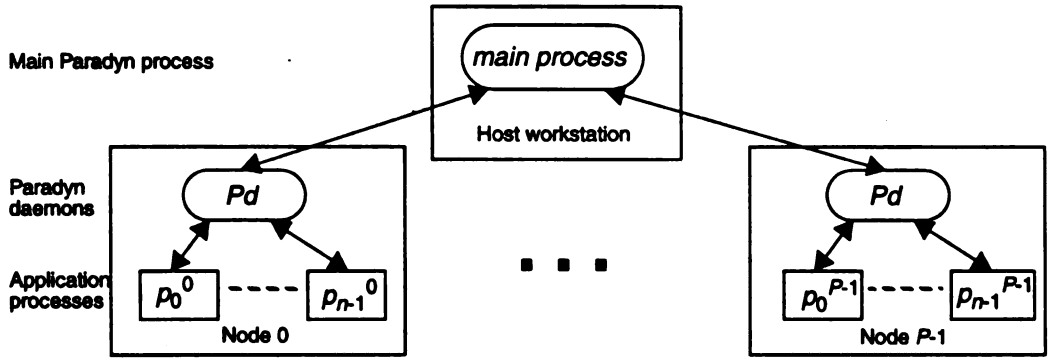


Figure 3-3. An overview of the Paradyn IS [136].

3.2.2 Domain-Specific Requirements

Design of Paradyn IS benefits from typical operating system based monitoring techniques for sequential systems (such as profiling with *gprof*), which uses sampling-driven data collection. Sampling-driven data collection usually generates fixed volume of data per sample and may incur lower overhead compared to event-driven data collection. However, sampling-driven data collection overhead is sensitive to the sampling rate. Therefore, a sampling-driven IS is required to use a suitable sampling rate, which is a compromise between delivering adequate number of samples within a fixed period of execution time and a reasonable sampling rate that incurs low overhead to the System Under Test (SUT). Hence, the IS should be designed to maintain a steady flow of instrumentation data with minimum overhead.

3.3 JEWEL IS

JEWEL is a commercial, off-the-shelf software product from the German National Research Center for Computer Science. It has been used to setup and control user-defined measurement experiments for embedded real-time platforms, such as Ultrix 4.2 on a MIPS processor, Amoeba and VxWorks on FORCE VME-bus M68030 board, and MACH 3.0

on an i386 single and multiprocessor, in addition to several other general-purpose distributed computing platforms. Following are some of the features of the JEWEL IS:

- The JEWEL IS is general-purpose as opposed to problem-specific ISs for collecting, evaluating, and presenting the runtime information.
- It consists of an integrated set of reusable, flexible, and adaptable components with well-defined interfaces.
- IS is useful for diverse problems related to system and application development and management for a broad spectrum of distributed platforms.
- It provides a central point of control to the experimenter in a distributed computing environment.
- It uses a high resolution global time base for global ordering of events.
- It can support both on-line monitoring and off-line analysis.

JEWEL consists of three main parts (see Figure 3-4): Data Collection and Reduction System (DCRS), Experiment Control System (ECS), and Graphical Presentation System (GPS). The DCRS module is responsible for collecting runtime information from the SUT and transfer it out of the context of the instrumented application process. The ECS configures the distributed modules of the IS for a measurement-based experiment. In order to integrate and control the JEWEL IS components from a central point of control, ECS assumes that all of the JEWEL components support a generic set of control commands. The GPS provides an on-line facility to visualize the performance data. It is loosely coupled with the driving system (i.e., the IS or a trace file) and independent of the data source.

3.3.1 Overview of Functionality

The data collection and reduction system (DCRS) and experiment control system (ECS), introduced in the preceding subsection, constitute the JEWEL IS. Similar to the PICL IS, the JEWEL IS also uses a library linked with the application processes to allow access to runtime information of the SUT. Figure 3-5 represents the architecture of the JEWEL IS to support measurement-based studies in the context of a heterogeneous, distributed

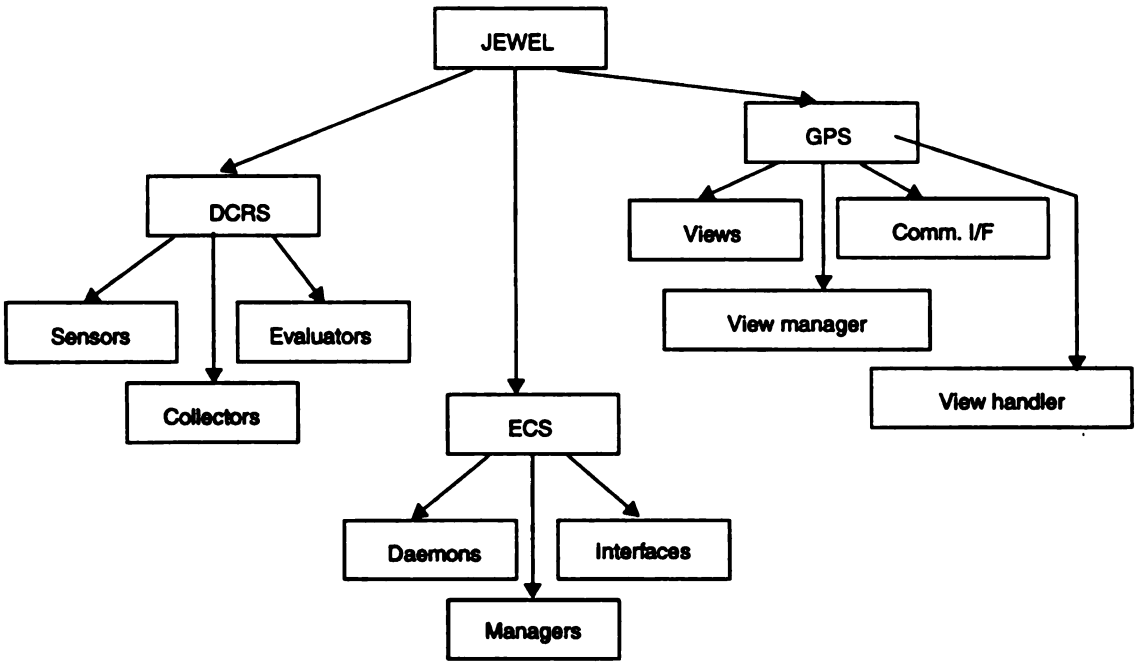


Figure 3-4. Modules of JEWEL measurement and visualization system.

computing system. Instrumentation is inserted as *internal sensors* to the SUT processes running at physically distributed locations. These internal sensors collect the measurements as a set of integers, which is passed to an *external sensor* via shared memory. The external sensor collects this information and creates a corresponding *measurement data record* (MDR) in *external data representation* (XDR [196]) form. An MDR is a standard data representation for all JEWEL system modules. MDRs are forwarded to a hierarchy of data collection and evaluation components. These components are responsible for collecting, sorting, merging, and reducing the instrumentation data. Under the control of the ECS, the MDRs can directly be forwarded to the GPS for on-line visualization. If a separate network is available for measurement data and ECS-related data, JEWEL IS communication does not produce network contention for the SUT traffic.

Since JEWEL IS is a general-purpose system, it is being applied to several measurement-based studies, such as testing of high-performance distributed combat systems (HiPer-D project [9]), monitoring of embedded systems [224], and adaptive control of real-time

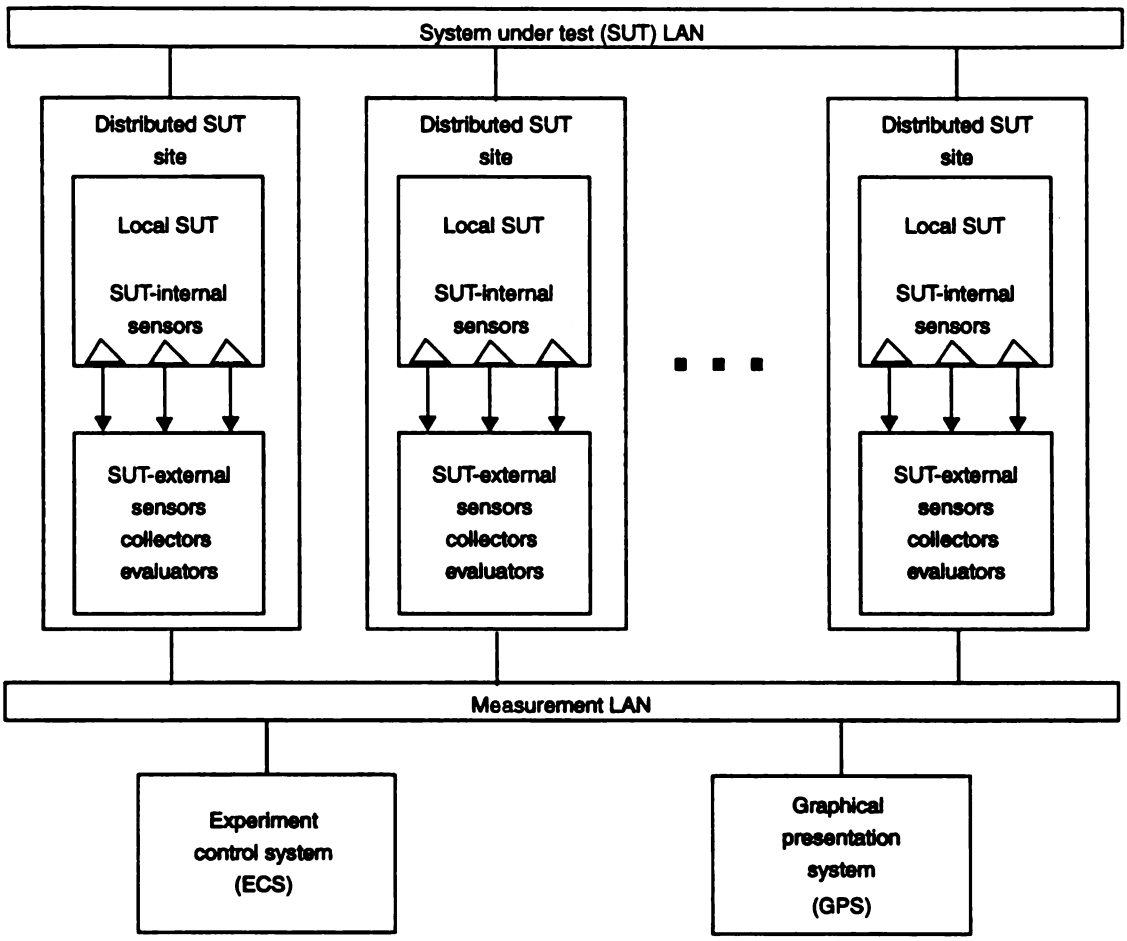


Figure 3-5. Architecture of the JEWEL IS to support a measurement-based experiment in a distributed, heterogeneous system.

systems [66]. We are using JEWEL IS to collect data from a distributed video conferencing application. The purpose of this data collection is to support real-time resource management and adaptive control of the video application. Figure 3-6 presents an overview of the functionality of JEWEL IS for this application. Video conferencing application uses a client-server paradigm to multicast the successive frames of a scene captured by the server using a camera to its clients in real-time. Runtime information is collected by a JEWEL collector arriving from a JEWEL sensors embedded in clients and the server that are running at physically distributed sites. Collector can share this information with a resource manager, which analyzes it and makes appropriate resource

management decisions for the server as well as clients. Thus, resource manager adaptively controls the clients and server using its agents and specific interface for interacting them.

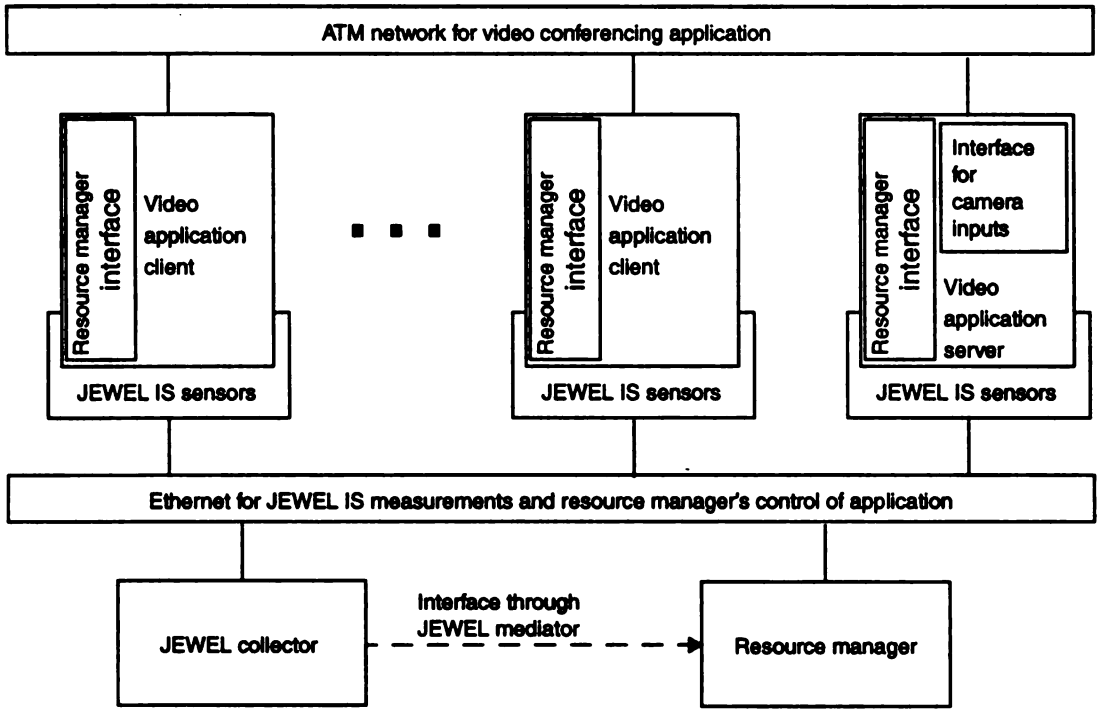


Figure 3-6. Overview of JEWEL IS functionality for adaptive control of a video conferencing application.

3.3.2 Domain-Specific Requirements

Video application is one example of a real-time system that has stringent timing constraints for tasks involved in sending and receiving video frames. It is a requirement of the client to receive and display 30 frame per second to represent a dynamic scene in real-time. However, the quality (smoothness of changes) will be lost if the frame rate reduces from this value. Since JEWEL IS components are sharing resources with the clients and the server, they can potentially aggravate the problem if the real-time tasks do not have adequate laxity in a particular setup. Modeling and evaluation of the JEWEL IS in this scenario should focus on its intrusion to the real-time behavior of the SUT.

In addition to considering potential intrusion to the real-time behavior of the video conferencing application, impact of the JEWEL IS to the resource manager tasks should also be considered. If the IS cannot deliver runtime information to the resource manager within a pre-calculated limit of time after it was generated by a client or the server, it can cause "oscillations" of the system (under test) as the adaptive control system may continue to steer the system from one nominal point of operation to another in the available space of operating conditions. The JEWEL IS should either guarantee delivering the runtime information before this time limit expires or discard it. Resource manager design should also incorporate certain degree of "hysteresis" to make it less sensitive to the transient conditions.

This concludes our introduction to the ISs used as reference for the case studies presented in this dissertation. It should be noted that these ISs are presented in this chapter in the order in which they were modeled and evaluated. Therefore, this order also illustrates the milestones in the progress of this research as well as its possible future directions.

Chapter 4

Instrumentation System Characterization, Design, and Synthesis

In this chapter, we begin by extending the discussion of instrumentation system characterization from Section 2.5.1. Our objective is to develop a generic model for an instrumentation system that is independent of any specific data collection applications. Additionally, such a generic model is useful for the IS modeling and evaluation studies presented in this dissertation. The reference ISs for this dissertation are used to serve different types of applications, therefore, a consistent taxonomy is necessary to put the overall modeling and evaluation process in proper perspective.

The process of developing an instrumentation system begins with its specification and design and culminates into writing the code for its modules. We refer to the task of IS code writing as *synthesis*. Contrary to the instrumentation system modeling and evaluation processes, which involve dealing with predominantly quantitative issues, design and synthesis involve application-specific, qualitative considerations. In addition, design and synthesis require decision-making on the part of developers to choose among a number of available alternatives to configure and manage the instrumentation system. We address IS design and synthesis issues in this chapter by synthesizing the approaches found in state-of-the-art tools and applications. Our intention is to provide the reader with the background, necessary for selecting among available IS design alternatives and synthesis methods, based on qualitative considerations.

In order to develop an IS in a structured manner, we proposed a two-level approach, which is depicted in Figure 4-1 [212,213]. On a higher-level, requirements of an IS are either

determined by the developer or specified by the users. These requirements are transformed to detailed lower-level system specifications, which are subsequently mapped to a model representing the structure and dynamics of the IS. This model is parameterized and evaluated with respect to chosen performance metrics that reflect the critical IS overheads to the application program as well as the target system. The evaluation results are then translated back to the higher-level, so that conclusions can be drawn by developers and users regarding IS performance. Feedback from the IS evaluation process is used to modify either the requirements or the system specifications to obtain desired performance. Finally, the model becomes the blueprint for actual software synthesis for the IS.

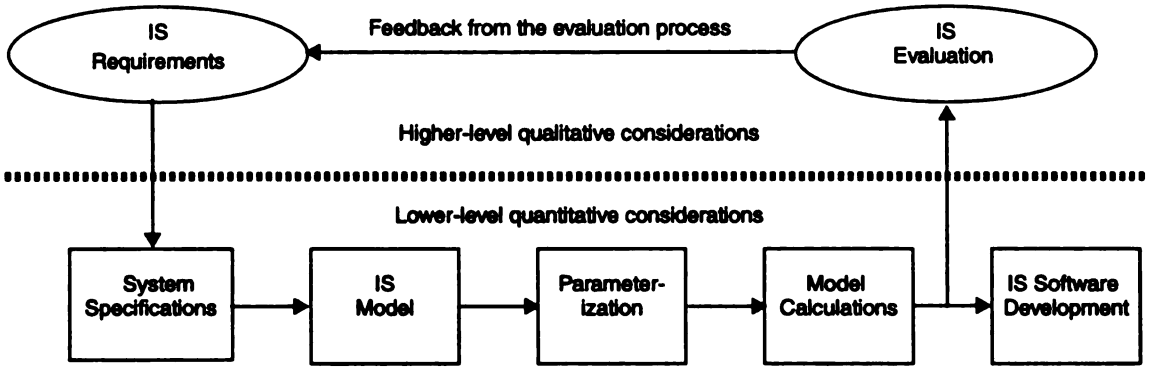


Figure 4-1. Two levels of a structured IS development approach.

We introduce a taxonomy for IS modules and services based on a generic IS model in Section 4.1, which are applicable to both IS design and synthesis as well as modeling and evaluation tasks. Subsequently, we discuss IS specifications in Section 4.2; IS design and synthesis decisions in Section 4.3; and design and synthesis of reference ISs in Section 4.4.

4.1 A Taxonomy of IS Modules and Services

Viewed from a higher level of abstraction, the purpose of an instrumentation system is to establish a continuous flow of runtime information from distributed sources to one or

multiple, often centralized, consumers. The exact nature of sources and consumers of runtime information is determined by the application for which an IS is being used. Establishment of a continuous flow of runtime information, however, is the common thread that can be found across several applications that consume runtime information. This common thread is the primary motivation behind the IS taxonomy presented in this section.

Viewed from the perspective of runtime data collection, an IS is synonymous to a system *monitor*. However, state-of-the-art in software environments for parallel and distributed systems that consume runtime information spurs the need for considering data collection from a broader perspective. A number of current environments that rely on runtime data collection put a greater emphasis on real-time management of this information for different reasons such as on-line performance analysis, debugging, visualization, and controlled overhead. Emerging applications, such as application steering, dynamic resource management, and distributed real-time control, dictate that the data collection modules are part of a closed-loop system. The scope of the term *instrumentation system* takes these new requirements into consideration. We use the term *instrumentation data* to account for both *execution* information (messages, memory references, I/O calls, etc.) and *program* information (variables, arrays, objects, etc.).

We have developed a taxonomy for an IS that represents a majority of components and services supported in extant ISs and omits unnecessary implementation details. This taxonomy can be represented with the help of a generic IS model, which is depicted in Figure 4-2. The model defines six components of an IS that supports an integrated environment: (1) sensors; (2) local instrumentation server (LIS); (3) instrumentation system manager (ISM); (4) instrumentation data consumers (IDC); (5) transfer protocol (TP); and (6) instrumentation system agent (ISA). In Section 4.4, we study the ISs of selected IDCs with respect to this taxonomy.

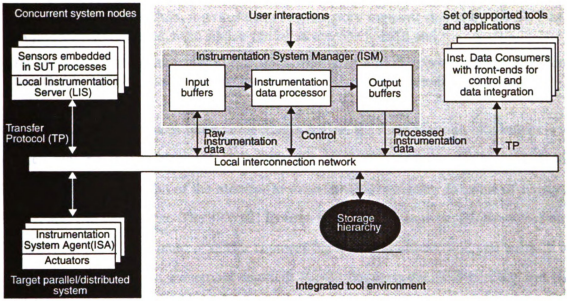


Figure 4-2. Components of a typical instrumentation system supporting an integrated tool environment.

4.1.1 Sensors

A *Sensor* is a piece of code that is inserted into the SUT code at the time of compilation or during its execution. When this part of the SUT code is executed, it generates instrumentation data, which indicates:

1. the locality of the code in the instrumented SUT program currently being executed; and
2. application-specific information related to the system performance or program data.

These two pieces of runtime information are used by different types of measurement-based tools and applications (IDCs), such as debuggers, performance analysis tools, performance bottleneck searching tools, modeling and prediction tools, steering tools, and program and performance visualization tools. A sensor acts as an interface between the SUT and the rest of the IS. There are at least three different ways to implement a sensor:

1. sensor is inserted in the code at or before the compile time (i.e., automatically or explicitly by the user) and it sends the instrumentation data to the LIS or ISM using a transfer protocol;

2. sensor is implemented as in (1) but instead of explicitly sending the instrumentation data to another IS module, it writes them in a memory segment shared by the SUT and IS processes (Ogle et al. term such a sensor as a *probe* [146]); and
3. sensor is inserted dynamically in the binary image of a SUT process during its execution (Hollingsworth et al. term this mechanism as *dynamic instrumentation* [86]).

Regardless of the differences in implementing a sensor, the main function of a sensor is to transfer runtime information out of the context of a SUT process. Ogle et al. [146] describe the sensor portion of the monitor in their Issos environment in terms of *sensors, probes*, and *tracing buffers*. The JEWEL IS uses two types of sensors: *SUT-internal* and *SUT-external* [114]. The internal sensor captures the specified information and writes it in a ring buffer, which is in a memory segment shared between the SUT process and the external sensor. The external sensor can collect this information from the shared memory ring buffer.

Regardless of the differences in terminology and implementations by different developers, a data collection components can be found in an IS, which can be inserted in a program to collect desired runtime information. We shall refer to this component as a sensor.

4.1.2 Local Instrumentation Servers

The *Local Instrumentation Server* (LIS) collects instrumentation data captured by the sensor and forwards them to the ISM. Additionally, the LIS can provide additional functionality to control the measurement-based experiment. Typically, an LIS uses local buffers for temporarily storing instrumentation data, a management policy to accomplish data collection and forwarding functions, and an interface interact with other IS modules (i.e., sensors and ISM).

In some cases, the sensor and LIS are not implemented as distinct modules and their functions are combined into a single module. As in PICL, for example, an LIS can simply comprises of instrumentation library calls responsible for storing data in the local buffers

or forwarding data to the ISM. Or, as in Paradyn, the LIS may consist of a separate process on each node of the concurrent system, which handles instrumentation data management independent of the application processes. The JEWEL IS uses external sensors to collect the data captured by the internal sensor and forwards them to the ISM. It uses a separate module, called the *Experiment Control System* (ECS) *daemon* for interacting with the ISM. Existing monitoring systems use varying terminologies for the LIS; for example, Paradyn calls it a *Paradyn daemon*, Issos, a *resident monitor*, and JEWEL, an *external sensor*. The term LIS, however, is an abstraction for specific implementations of data capturing and forwarding functionality.

4.1.3 Instrumentation System Manager

The LIS forwards instrumentation data from the concurrent system nodes to a logically centralized location called the *Instrumentation System Manager* (ISM), which manages the data in real-time. The functions of the ISM include temporary buffering of data, storing of data on a mass-storage device, and pre-processing of data for IDCs (e.g., causal ordering). Functional requirements of an ISM that supports on-line data consumption are different in nature than for one that supports off-line consumption. Similarly, different requirements are associated with an integrated tool environment versus a stand-alone tool. For instance, on-line tool usage may require the ISM to order data on-the-fly before submission to a tool; whereas an ISM for off-line tool usage may only need to merge data from various application processes, performing event-ordering off-line. We reflect this programmability by defining an *instrumentation data processor* module within the ISM in Figure 4-2. IDCs receive instrumentation data from ISM output buffers or a mass storage device, depending on-line or off-line usage, respectively.

The ISM components in Paradyn, Issos, and JEWEL ISs are known as the *main Paradyn process*, the *central monitor*, and the *collector*, respectively. Many tool developers, such as Ogle, Schwan, and Snodgrass [146], favor a different partitioning of pre-processing

functions, implementing data reduction/analysis in the LIS rather than in the ISM. The definitions of the LIS and the ISM do not preclude this.

4.1.4 Instrumentation Data Consumers

Instrumentation data collected by the ISM can be consumed by one or more measurement-based tools, resource managers, decision makers, or adaptive controllers. We abstract these tools and applications using the term *Instrumentation Data Consumers* (IDC). An IDC is typically part of an integrated environment, and therefore, it must have a well-defined interface for data and control communications with the ISM. Apart from this interface, the design and implementation of an IDC can be carried out independent of the rest of the IS. In fact, an IDC is part of the target system or tool environment rather than the IS. We consider it as a module in the IS for two reasons:

- it is on the information (i.e., runtime instrumentation data and control messages) flow path, which spans distributed processes, sensors, LISs, ISM, and finally the IDC; and
- it can dynamically control the target parallel or distributed system via the IS-supported information flow path, which can affect IS intrusion to the target system.

Examples of different types of IDCs include: decision support system in AT&T's network management and operations system (see Figure 2-3 in Section 2.3.2.3); signal processing, identification, command and decision tasks in HiPer-D/Aegis weapons system (see Figure 2-4 in Section 2.3.3.3); and a variety of measurement-based parallel tools such as debuggers, performance analyzers, and visualizers [208].

4.1.5 Transfer Protocols

Instrumentation data are transferred from the LIS to the ISM and further to the IDC(s) to establish an information flow path. Data transfer to an IDC (a tool or an application) is typically accompanied by an exchange of control signals between the ISM and the IDC. Additionally, control messages may need to be sent in the other direction, back from the

IDC to the ISM, and then to concurrent application processes or actuators (via IS agents), to control program execution or the target system [71]. Usually, a consistent instrumentation data and control *Transfer Protocol* (TP) is used for IS-related communications.

The majority of existing monitors uses operating system-supported interprocess communication abstractions. For instance, *sockets*, *pipes*, and *remote procedure calls* have been used in Paradyn [136], Issos [146], JEWEL [114], TAM [165], and Pablo [161] ISs for their implementations on Unix-like operating systems. Some monitors, such as VIZIR [77], implement customized high-level protocols, developed on top of system library functions, to enhance the flexibility and portability of the instrumentation data transfer and control messaging mechanisms.

4.1.6 Instrumentation System Agents

An *Instrumentation System Agent* (ISA) extends the functionality of an IS to control the execution of data collection modules as well the application processes. Thus, the IS can steer the application as well as interact with its own modules to adaptively control them. In the case of steering (i.e., algorithmically controlling the execution [50,163]), the ISAs are embedded in the application processes. A number of emerging distributed and embedded real-time system management and control applications (such as AT&T's Network Management and Operations System [13] and NSWC's Aegis Weapons System based on HiPer-D computing fabric [79]) use information about the current system states for making control decisions. In order to incorporate such systems into the generic IS model, we consider *actuators* separate from the instrumented application processes. The term actuator is borrowed from the feedback control systems theory to represent modules that can modify the state of the system according to a command input [222]. Thus, an ISA/ actuator combination can be implemented such that:

1. it is embedded in an instrumented application process that generated instrumentation data; or
2. it is embedded in an independent process, which is responsible to take specific actions.

The first implementation corresponds to the control of data collection modules and application steering while the second implementation corresponds to a real-time control system. In either case, the ISM or an IDC (e.g., a resource manager [219] or decision-making module [79]) can send a command to the ISA using the transfer protocol to implement a specific steering or system control function.

The taxonomy of an IS is evolving with emerging applications in parallel and distributed computing. It is interesting to note that the taxonomy initially did not include sensors and ISA/actuator components [213]. However, several applications of parallel and distributed processing to real-time adaptive control systems started using runtime information and tools that consume this information [163]. Thus, we had to extend the taxonomy to include these modules. We expect that the scope of an IS will continue to expand due to these applications as well as the use of integrated tool environments; therefore, the taxonomy of an IS will continue to evolve.

4.2 Design Specifications

IS specification is one of the low-level task as identified in Figure 4-1. System specifications are determined from the high-level system requirements. Requirements of an IS are primarily determined by the specific needs of the application (or a range of applications) that are to be supported by the IS. For instance, if the IS is to support on-line program visualization (e.g., as in case of XPVM [64]), requirements include: data collection from the application processes, continuous data flow from the application processes to the visualization tool, and the decoupling of IS from the actual program activities. These requirements are then transformed into specifications that may include: library functions to insert instrumentation in the user programs, local buffers to temporarily hold the data, one or more IS processes to forward the data to the tool, and an

interface with a visualization tool to collect, sort, merge, and then visualize the instrumentation data. Transformation of IS requirements to specifications depends on the nature of the measurement-based experiment supported by the IS.

4.3 Design and Synthesis Decisions

Design of an instrumentation system involves choices based on its use for a specific application. These design decisions cannot be justified by any quantitative measures; experience of tool development and intuition about the future needs of a tool is necessary in order to make these decisions. In this section, we outline three issues related to the design of an IS that require decision-making effort for choosing a suitable option.

4.3.1 Selection of an Instrumentation Data Format

In an integrated environment, integrated data is to be used by the tools designed by different developers. There is usually one common denominator among all these tools: use of the same instrumentation data for different types of analyses. Minimally, these tools should use a common data representation to share data and have a well-defined interface with the IS to receive this data. Additionally, the interface between the IS (or ISM) and the tools should incorporate a control messaging mechanism to allow interaction between tools and the IS. Instrumentation data format is an important consideration when the environment relies on the tools developed by different developers. In case of the tools developed by the same developers, the question of sharing the same data by all the tools is resolved by defining a consistent data format.

Several existing ISs support consistent data format to support integrated tools. Pablo's *Self-Defining Data Format* (SDDF) is a notable effort in this regard. The SDDF is a performance data description language that specifies both data record structures and data record instances. It can *describe* general data records, as opposed to a predefined set of records; therefore, the SDDF is best viewed as a data meta-format. Intuitively, the format

supports the definition of records containing scalars and arrays of the base types found in most programming languages (i.e., byte/character, integer, and single and double precision floating point). SDDF was originally developed to link Pablo IS with the data analysis environment. However, a number of integrated tool environments are using SDDF as a consistent instrumentation data format. Examples include ParAide performance environment [165] and XPVM extension of PVM message-passing library [63]. JEWEL IS uses *External Data Representation* (XDR [196]) format to share data among several of its modules.

4.3.2 Sampling-Driven vs. Event-Driven Data Collection

There are two distinct approaches of collecting the instrumentation data from a target system: *sampling-driven* and *event-driven*. In the case of sampling-driven approach, an instrumentation data sample is collected after a specified period of time. Thus, the data collection occurs *periodically* and IS modules do not require the use of system resources before a sampling period ends. On the other hand, data collection occurs *aperiodically* under the event-driven approach. In this case, the data are collected only when a sensor in the instrumented code is executed.

The sampling-driven and event-driven approaches are different in terms of the type of data they provide to the tools in an environment. Instrumentation data collected under a sampling-driven approach usually consists of the values of IS-defined *timers* and *counters* that are embedded in the instrumented SUT code. For instance, operating system supplied profiling tools (such as, *gprof*) use sampling-driven techniques to count the frequency of a function call in an instrumented program and time spent in that function. Such information is useful to identify bottlenecks in a program and programmer can analyze the parts of the program where most of the time is being spent. Instrumentation data collected under an event-driven approach usually consists of information of interest to a user. For instance, ISs used for testing distributed real-time control systems (such as, HiPer-D [9]) use an event-driven approach to analyze that the real-time task deadlines are met in critical

sections of the code. These data are almost always time-stamped to identify a "time-line" of occurrences of the events in a system. Therefore, the selection between the sampling- and event-driven approaches is based on the nature of the application for which an IS is being designed.

4.3.3 Global Time and Event Ordering

If an IS for a parallel or distributed system is developed using an event-driven data collection approach, it has to deal with the classical problems of globally consistent time-stamps and event ordering. A concurrent system consisting of multiple nodes with independent local clocks may experience discrepancies among values of these clocks. If a sensor or LIS assigns time-stamps to the locally collected data and forwards them to the ISM, it is likely that the resulting "global time-line" does not represent the actual sequence of event occurrences. The event ordering problem becomes important for message-passing activity where sending of a message by one node and receiving of that message at another node should have a *causal* relationship.

Several efforts have addressed the problems related to global time and consistent event ordering in parallel and distributed systems. Lamports's work in this regard is a basis of some of the solutions that have been used to tackle this problem [112]. Lamport divides the event ordering problem into two parts: *partial ordering* and *global ordering* using a *happened before* relationship among the events. A partial order is achieved if we can establish the happened before relationship for all the events occurring at a particular node. Then a global order can be achieved if happened before relationship is established for the events representing interactions among different nodes.

A number of ISs for different types of IDCs have benefited from these results. O'Donoghue and Plunkett use the Network Time Protocol (NTP) for determining global time-stamps to test distributed real-time systems using commercial off-the-shelf software tools [147]. The main problem with this approach is the coarse granularity of time

measurements that the NTP can support. Ellwood and Heath describe a postprocessing mechanism to fix the clock offsets and event inconsistencies in the trace records for PICL IS implementation for MPI [51]. The postprocessing is implemented as a parallel algorithm that executes after the actual program execution is completed. PICL uses barrier synchronizations at the start, end, and specified instrumentation points in the program. The postprocessor then uses this information to correct any clock drift. Partial and global ordering techniques are used to check the global consistency of the events and the sequence of events is adjusted to fix any inconsistencies. This approach is of limited use for an IS that supports on-line analysis or visualization tools because instrumentation data are needed before the program execution finishes. Additionally, the overhead due to excessive barrier synchronization operations may be undesirable for long-running application programs. JEWEL IS developers advocate the use of a high-resolution global clock for consistent time-stamps and event ordering. A global clock can solve the problems related to clocks and event ordering but most of the parallel and distributed systems do not support a global clock as a standard feature.

In some cases, IS ensures a causal order of the events while the time-stamps are only logical (as opposed to physical time-stamps assigned at local nodes). For instance, VIZIR ISM assigns the time-stamps to the instrumentation data received from the LISs [78]. The received data are processed in such a manner that the processed data records do not have any event ordering problems and an on-line tool can use them. Falcon IS also uses an on-line event ordering algorithm for its steering tool [50].

Based on the above discussion, it is clear that the clock drift and event ordering is still an open problem in the area of ISs for parallel and distributed systems. However, as we noted above, there are solutions that ensure consistency on the cost of other factors. These factors include accuracy of time measurements, intrusion to the target system and application, cost of specialized hardware, and limited functionality of an IDC supported by the IS.

4.3.4 Hard-Coded vs. Application-Specific Synthesis

In terms of actual coding of an IS, hard-coded and customizable or application-specific software development techniques reside on opposite ends of a spectrum of possible development approaches. Tool developers have to choose a suitable approach that can be applied consistently to all the components in an environment.

Most of the ISs found in literature are designed to complement the specific functions of IDCs in an environment. Therefore, the primary concern of the software development process is to provide the user with a self-contained tool rather than to optimize the extensibility. Consequently, the majority of extant ISs are "hard coded" into the tool environments. The configuration and functions of the IS remain unchanged for different application programs, which may have entirely different data collection, management, and usage requirements. For instance, a parallel program using PICL must initialize instrumentation functions and buffers at all nodes in a multicomputer system, even if only a subset of the nodes is of interest to the user. This may incur undesirable overhead. Similarly, a user can not change the configuration of the IS from one application to another. This further reduces the flexibility of the IS.

Development of application-specific ISs is a relatively recent phenomenon. Honeywell's Scalable Parallel Instrumentation (SPI [15]) supports customized synthesis of ISs. Its IS synthesis approach is based on an Event-Action model. IS functions are specified by the user as actions taken by the IS, in response to the occurrence of specific events. A user specifies the events and actions in terms of an *Experiment Specification Language* (ESL). Therefore, it is possible for the user to specify customized event buffering and instrumentation data forwarding actions that are optimized (with respect to instrumentation overhead) for a particular application. Ogle et al. [146] present a similar approach of developing application-specific monitoring functionality in their Issos parallel programming environment. Paradyn supports the *Paradyn Configuration Language* (PCL) for describing its target architecture and operating system and the language-dependent

characteristics of the application and platforms [136]. Tuning and Analysis Utilities (TAU [83]) is an integrated performance evaluation environment based on the pC++ system [126]. Instrumentation is specified by the programmer via a pC++ class library and Sage++ library, and collected data may be analyzed by on-line or off-line tools supported by the TAU environment.

Application-specific approaches represent the trend of IS synthesis technologies. Although these approaches are very promising, they may be counterproductive for a user having little experience with performance issues of an IS. Therefore, the developers have to choose between the hard-coded and application-specific approaches. It is also possible to use a hybrid of two approaches by making specific components of the IS application-specific while other components are developed as hard-coded software modules.

4.4 Reflections on the Design and Synthesis of Reference ISs

Determining the IS design specifications and a taxonomy of its modules and functions is simply an effort to collectively consider the features of a broad range of existing ISs. In this section, we work backwards to show the applicability of the taxonomy to the selected ISs and examine their specifications. We discuss the reference ISs in more detail and summarize the design considerations of a number of other ISs.

4.4.1 PICL IS

PICL IS is used for collecting instrumentation data from distributed-memory parallel systems. It supports an off-line performance visualization tool, called ParaGraph [81]. The IS does not require a continuous flow of instrumentation data from the LISs to the ISM and the tool during the program execution. Therefore, the IS collects the instrumentation data at each node and merges them as one trace file at the end of the program. The ParaGraph tool consists of a rich set of visualizations to represent the computation and message-passing. The IS is required to be event-driven to capture the data related to

interesting message-passing activity. Clock drift problem is handled by synchronizing the clocks at the start of the program execution and providing a barrier synchronization function that can be called by the user to synchronize clocks during the execution. If the resulting trace file shows a receive event on a node before a corresponding send event occurs on another node, it is said to have a "tachyon" in it. Tachyons can be removed from the resulting trace files using a postprocessor that sorts the trace records in a globally consistent order and re-adjusts the time-stamps. PICL modules represent a typical example of a hard-coded instrumentation system.

Table 4-1 presents the specific modules of the PICL IS according to the IS taxonomy defined in Section 4.1. The LIS is implemented with an instrumentation library, and the ISM and TP provide a means to merge the data as a trace file. Due to off-line usage of the instrumentation data, there is no need for an ISA to control the PICL IS.

Table 4-1. Specifications characterizing the PICL instrumentation system.

Sensor/LIS	ISM	IDC	TP	Actuator/ISA
Instrumentation library with trace data buffers at each node	Instrumentation library with merging distributed buffers as a trace file	ParaGraph tool	Parallel I/O	None (open-loop system)

4.4.2 Paradyn IS

Paradyn IS addresses the requirements of the main analysis component of the environment, called *Performance Consultant* [136]. It uses a W^3 search algorithm for on-the-fly location of bottlenecks in the code of parallel programs. The algorithm uses the information about resource usage (such as, synchronization primitives, message-passing, I/O calls, CPU usage, etc.), which is supplied by the LIS at regular sampling intervals. Therefore, the IS is required to support on-line analysis and maintain a steady flow of information to the tool. Due to sampling-driven approach, Paradyn IS does not require any clock synchronization or event ordering algorithm. Software development approach for

Paradyn IS can be considered a hybrid between hard-coded and application-specific because some of its modules are configurable using its configuration and specification languages.

Table 4-2 presents the specifications of the Paradyn IS. A local Paradyn daemon works as an LIS, which inserts the sensors into the running application code, on-demand from the ISM (i.e., the main Paradyn process).

Table 4-2. Specifications characterizing the Paradyn instrumentation system.

Sensor/LIS	ISM	IDC	T P	Actuator/ISA
Local daemon process for each node that collects samples from application processes and forwards data	Main Paradyn process that accepts data from daemons and uses data for analysis	Performance consultant	Unix-based interprocess communication	Paradyn daemon is used to adaptively control the overhead through dynamic insertion/ removal of instrumentation

4.3 JEWEL IS

JEWEL IS is a commercial off-the-shelf software. Therefore, it can be customized for different types of target applications and systems. It can support both on-line as well as off-line processing, analysis, and visualization of the instrumentation data. It supports an event-driven data collection approach. JEWEL IS does not provide any mechanism to correct the problems due to clock drifts or out-of-order events. It provides customizable modules that can be modified by the user to implement an appropriate technique to fix these problems. Thus, JEWEL IS assumes a globally consistent clock as a part of its default target SUT setup. JEWEL is developed as a fully customizable IS because a user can modify its modules to suit particular platforms and applications.

Table 4-3 presents the specifications for the JEWEL IS. JEWEL provides internal and external sensors as its sensor and LIS modules, respectively. A collector component acts as an ISM. In order to support a heterogeneous distributed system, JEWEL supports the

notion of a hierarchy of ISMs to collect and reduce instrumentation data from different parts of the system. All of the JEWEL IS modules use instrumentation data in the form of *Measurement Data Records* (MDRs) using XDR format. Different adaptations of JEWEL IS for different platforms uses sockets and remote procedure calls as a part of its TP. It supports on-line control of the instrumentation system through its *Experiment Control System* (ECS) components.

Table 4-3. Specifications characterizing the JEWEL instrumentation system.

Sensor/LIS	ISM	IDC	T P	Actuator/ISA
Local daemon process for each node that collects samples from application processes and forwards data	Main Paradyn process that accepts data from daemons and uses data for analysis	Generic, therefore, can be retargeted to different applications	OS-based interprocess communication (using sockets or RPC)	Paradyn daemon is used to adaptively control the overhead through dynamic insertion/ removal of instrumentation

4.4.4 Overview of other ISs

In this subsection, we apply the IS taxonomy beyond the reference ISs to cover a broader range of IDCs. Many parallel programming tools use an IS. We introduce the IS design approaches of selected IDCs, according to our IS taxonomy. These are summarized in Table 4-4.

This overview shows that the IS design requirements and taxonomy presented in this chapter is applicable to the extant ISs. IS design issues such as those requiring decision-making on the part of the developers cannot be based on formal quantitative analysis. While the research work presented in this dissertation focuses more on the quantitative aspects of IS design, these issues are also essential to be well understood for a balanced design. In the following chapters, we address the IS modeling and evaluation aspects in the light of insights gained from the discussion of IS design issues in this chapter.

Table 4-4. Summary of IS features of some representative parallel IDCs.

IDC (tools and applications)	Sensor/LIS	ISM	Actuator/ISA	Synthesis Approach
Jade [101]	Channels and controllers	Consoles	None	Hard-coded
AIMS	Library	Trace file	None	Hard-coded
Pablo	Library	Trace file	Adaptive control	Hard-coded
Falcon/Issos/ ChaosMON	Resident monitor	Central monitor	Interactive steering	Application-specific
ParAide (TAM)	Library	Event trace server	OS-based interface to gang scheduled the flushing of the filled trace buffers	Hard-coded
SPI	Library	Event-Action machines	Adaptive control is possible through its experiment specification language	Application-specific
VIZIR	Library	VIZIR front-end	None	Hard-coded
Aegis and HiPer-D	software (JEWEL) sensors and embedded subsystems working as sensors	JEWEL collector and decision-support tasks	Actuators for the weapons and fire control system	Application-specific, using off-the-shelf software

Chapter 5

Instrumentation System Modeling, Management, and Workload Characterization

In this chapter, we focus on IS modeling and management issues and present models for the reference systems. After designing an instrumentation system, performance feedback may be valuable for tool developers. While the tool development is at an early prototype stage, it is convenient for the developers to modify the design according to the performance feedback. Since the IS is not fully developed at this stage, a measurement-based performance study is not feasible. It is possible, however, to model the system based on its design and specifications. The model can be solved analytically or through simulations; thus early feedback can be provided to the developers.

Modeling-based evaluation of computer systems, compared to an ad hoc measurement-based evaluation, is often regarded as a careful and rigorous approach that is not widely practiced [54]. One may ask if such rigor is needed in IS development. The IS represents enabling technology of growing importance for effectively using parallel and distributed systems. The IS is often used by application developers or system administrators; the user typically sees a tool and not the IS. Consequently, tools applications are scrutinized, and the IS and its overheads receive little attention. Users may be unaware of the impact of the IS to the SUT. Unfortunately, the IS can perturb the behavior of the application, degrading the performance of an instrumented application program from 10% to more than 50% according to various measurement-based studies [71,134]. Perturbation can result from contention for system resources among application and instrumentation processes. With increasing sophistication of system software technologies (such as *multithreading*), an IS process is expected to manage and regulate its use of shared system resources [162,200].

Toward this end, tool developers have implemented adaptive IS management approaches; for instance, Paradyn's dynamic *cost model* [87] and Pablo's user-specified (static) *tracing levels* [161]. With these advancements come increased complexity and more design decisions. Modeling and early evaluation facilitates dealing with these design decisions.

This chapter consists of two parts: the first part presents a methodology of dealing with specific issues involved in instrumentation system modeling, management, and workload characterization; and the second part presents models for the reference ISs. We discuss modeling and management issues related to an IS in Sections 5.1 and 5.2, respectively. The resource occupancy modeling technique is presented in Section 5.3 and workload characterization for such models is considered in Section 5.4. Finally, we present the models for the three reference ISs in Section 5.5.

5.1 Instrumentation System Modeling Issues

This section focuses on specific issues that should be addressed while developing a model for an IS. Some of the issues are general and relevant for modeling any other system; however, we consider these issues from the perspective of modeling an IS for parallel and distributed systems. Modeling issues discussed in the following subsections are: defining level of abstraction and objectives for an IS model; system level considerations; patterns of instrumentation data flow from distributed producers to centralized consumers; and selection of performance metrics for IS evaluation.

5.1.1 Abstraction and Objectives of Instrumentation System Modeling

Defining the level of detail that the model should try to capture helps determine the complexity of a solution technique and, therefore, its suitability for a particular study. In case of an IS, the level of detail is determined by the functionality and requirements of the instrumentation data consumer that it supports. If the IDC uses the instrumentation data as a trace file for off-line analysis, only high-level details such as data collection and

forwarding are of interest. This level of details will have to be enhanced if the IDC supports on-line analysis of long-running programs and may selectively enable or disable the instrumentation. The model incorporates many more details when the IS supports an IDC that adaptively controls the IS or steers the parallel or distributed application in real-time. As our reference ISs address this range of IDCs (tools and applications), level of detail is an important consideration for the models of these systems presented in this chapter.

Before developing a model to study a computer system, it is essential to spell out the modeling objectives. Although the modeling objectives are expected to vary from one tool to another, a common set of goals of modeling ISs includes:

- evaluation of IS overhead and intrusion to the SUT or target system under various operating conditions, according to the selected metrics;
- comparison of IS overhead and performance using available options to configure and manage the IS modules;
- determination of sensitivity of the selected IS overhead and performance metrics to the operating parameters and factors; and
- investigation of "what-if" scenarios by allowing a specific configuration, IS task schedule, or management approach.

The above list provides only a general set of goals that should be addressed in a modeling based evaluation of an IS. However, these objectives are specifically spelled out for a given IS and SUT combination as the definition of overhead and performability is application-specific.

5.1.2 System Level Considerations

In Chapter 4, we defined the scope of an IS and a generic model to identify its components. It is possible to model an IS that restricts to these modules and services provided by them. However, one objective of IS modeling is the evaluation of intrusion to the SUT or target system due to contention for and sharing of system resources between

instrumentation system tasks and the target system. This objective can not be accomplished by modeling an IS in isolation from the target system. The IS should be considered a part of the entire system to model the interactions among IS and target system components. Due to system level considerations, models for the reference ISs are coupled with the functions of their respective target systems.

5.1.3 Data Flow Patterns

Regardless of the level of sophistication, the overall objective of an IS is to maintain a steady flow of runtime information from the SUT to the supported IDC(s). A model of the overall system includes a number of data handling modules that perform one of the following functions: data collection, processing, reacting, buffering, merging and sorting of data arriving from multiple source, and consumption of the data by an IDC. Depending on the level of detail, modeling these components may require the knowledge of data flow patterns. There are three data flow patterns of interest: *independent and identically distributed* (*iid*) arrivals of instrumentation data samples; *bursty* arrivals; and *correlated* arrivals. We overview these patterns in the following with respect to the levels of detail at which they are applicable.

5.1.3.1 IID Arrivals

Consider the instrumentation data samples that arrive at an IS buffer at instants of time t_0, t_1, t_2 and so on as shown in Figure 5-1. Then the stochastic process $\{X_n = t_{n+1} - t_n: n>=0\}$ represents the inter-arrival time of data samples. For analytically solving the model, it is often convenient to assume that the inter-arrival times $\{X_n\}$ is independent, and identically (often exponentially) distributed arrivals.

Considering the arrival patterns to be IID simplifies the analytical solution of the IS model relying only on the probabilistic calculations. However, this approach is rarely useful from a practical point of view because it is difficult to prove that the inter-arrival times are

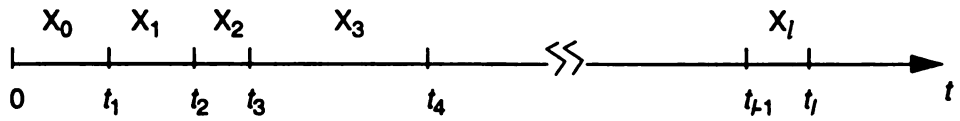


Figure 5-1. IID arrivals at an IS buffer.

independent. Therefore, we need to consider other possible arrival patterns that are more useful in practice. In addition, this approach is applicable when we are interested in low level details of a particular stochastic process. For instance, number of waiting instrumentation data samples at in an LIS that are to be forwarded to the ISM regardless of the application program behavior that generated these data.

5.1.3.2 Bursty Arrivals

In may applications and systems, instrumentation data arrives at a collection stations in batches (or bursts) of samples. Each burst has an inter-arrival time, which is the time between the end of burst to the start of the next burst. Other parameters associated with a burst include the number of arrivals within a burst and duration of a burst. Figure 5-2 shows burst of arrivals at a buffer, where number of samples in each buffer is deterministic.

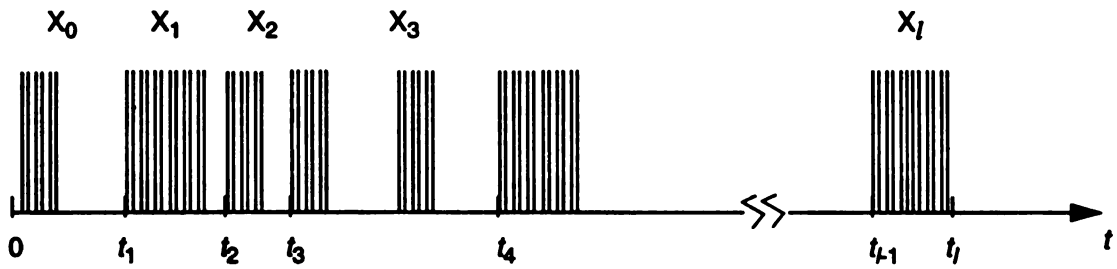


Figure 5-2. Bursty arrivals at an IS buffer.

Bursty arrival pattern are often more realistic than the IID arrivals because establishing lack of independence between individual arrivals is not trivial. Bursty arrivals are also

useful with low-level considerations of the IS behavior and difficult to directly map to a higher-level operation of the IS. While bursty arrival patterns have been successfully applied for modeling I/O access patterns to improve file system performance [191], the approach is not appropriate to model the data arrivals in response to the high-level operations of an instrumented program. Therefore, in the following subsection, we consider the correlated arrival patterns that relates the arrival of instrumentation data to the higher level instrumented program behavior and its interactions with the IS.

5.1.3.3 Correlated Arrivals

Arrival of instrumentation data to an IS buffer is closely related to the functionality of the instrumented program and the behavior of the IS sensor that collects the data in the SUT. This is true for both sampling- and event-driven instrumentation approaches. Representing the high-level behavior of a program as a set of interacting states, including one or multiple data collection states, is a possible approach to incorporate correlation. For instance, Figure 5-3 shows an example of an instrumented program that performs two functions: it computes a result and then multicasts it to a set of other processes. Instrumentation data activity of an LIS (shown as shaded oval) is related to these two program activities. If we establish that the "holding time" in each of these states is independent and exponentially distributed, these states can be considered to form a Markov chain and an analytical solution of the IS model may be possible. On the other hand, we can make the IS model specific to a particular instrumented application by fitting and parameterizing the distribution of holding times in each of the instrumented SUT state separately. However, simulation is the only practical approach to solve the IS model, in this case.

5.1.4 Metrics

During the initial phase of this research, we tried several performance metrics to suite the IS evaluation objectives of individual case studies [211,215]. However, experience with

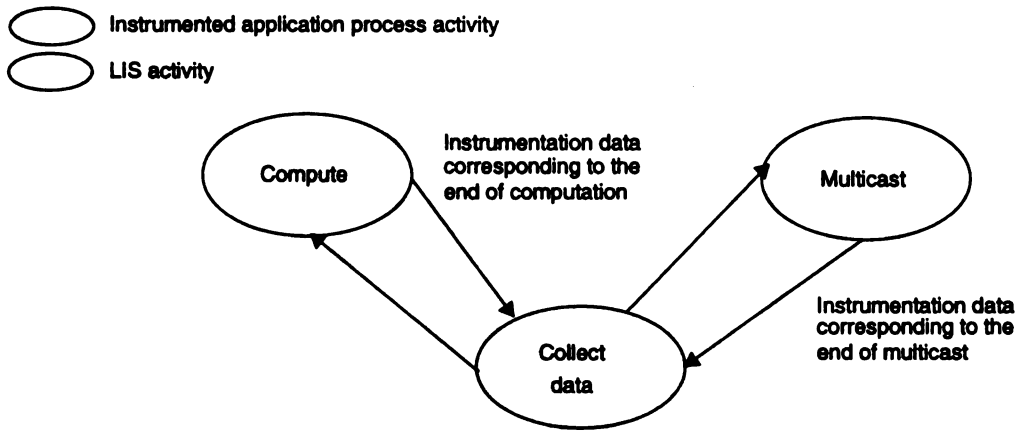


Figure 5-3. An example of a correlated pattern of instrumentation data arrivals at an LIS.

these initial case studies and feedback from collaborators on later IS modeling and evaluation studies resulted in a unified set of three types of metrics for this type of evaluation:

1. **Direct Overhead Metrics:** these metrics are related to the utilization of the bottleneck resource by IS processes. A bottleneck resource is one that has maximum utilization among all the system resources [117]. Specific metrics belonging to this class may include: *CPU utilization*, *network bandwidth usage*, *I/O device utilization*, etc.
2. **Data Flow Metrics:** these metrics provide a quantitative measure of the steady flow of runtime information from the instrumented processes to the ISM or IDCs. Specific metrics related to this category may include: *monitoring latency* i.e., wall-clock time taken by a sample to reach IDC after it was generated by a sensor; *hold-back ratio* i.e., the ratio of the number of samples received by an IDC to the number of total number of samples generated by all the distributed sensors in the system; and *number of received trace records*, etc.
3. **SUT or Target System Intrusion Metrics:** these metrics directly or indirectly quantify the impact of instrumentation to the system under test. Some of the metrics belonging to this class include: CPU or other shared resource utilization by the application with and without instrumentation inserted in the program; and application quality-of-service (QoS) metrics [207]. Note that the uninstrumented case provides a baseline measure to compare the intrusion using metric values obtained from an instrumented version of the SUT or target system.

The definitions of individual metrics belonging to each of the above three classes of metrics fall short of being directly useful for different IS modeling and evaluation studies. For almost every case, the analyst has to choose a specific metric from each of the above three classes, which are best suited to the application at hand. Selection of metrics for the

studies of reference ISs is based on three types of metrics presented here. These metrics are presented in the context of specific case studies in Section 5.5. Notice that in Section 5.5, the specific nature of the metrics is determined only after settling on IS management issues and the system modeling technique. Only generic nature of the metrics can be determined early at the time of considering modeling issues, as in this subsection.

Modeling issues discussed in this section are in fact high-level qualitative considerations before proceeding to the actual development of the model. Clarity of objectives of the study, consideration of dependences among IS and target system activities, and definitions of suitable IS performance metrics contribute toward an accurate and careful study that can provide useful feedback to the developers.

5.2 Instrumentation System Management Issues

Instrumentation system management is necessary to maintain a steady flow of instrumentation data from the target system to the ISM and IDCs in the presence of system resource sharing and contention. Steady flow of information is required while keeping the overhead and intrusion to the target system to a minimum level. There are no generic IS management policies that can be applied to all measurement-based experiments. Instead, an appropriate management policy is selected for a particular IS and SUT combination based the overhead and intrusion due to that policy. There are two issues that should be addressed by a management policy: scheduling of IS-related tasks and adaptability of the IS modules with dynamic states of the target system or SUT.

5.2.1 Scheduling of IS-Related Tasks

IS management involves scheduling instrumentation data collection and forwarding tasks using a policy, which could be a trade-off between two conflicting requirements: maintaining a steady flow of runtime information from the target system to down-stream modules and maintaining low overhead and intrusion to the target system due to IS

functions. In case of sensors inside the application processes, the instrumentation data collection can be scheduled as a *periodic* or *aperiodic* process for the cases of sampling-driven and event-driven instrumentation, respectively. Data collection and forwarding by the LIS can be scheduled using a variety of policies, such as *collect-and-forward* or *batch-and-forward* policies, which have different implications for IS overhead, intrusion, and performance.

5.2.2 IS Adaptability

It is possible to manage an IS in response to the time-varying operating conditions of the target system or SUT and volume of information being generated. This is possible through an adaptive controller for the IS. Figure 5-4 illustrates the operation and modules of a typical adaptive controller that can be used for an IS or a distributed application. The objective of an adaptive controller for an IS is to make it exhibit a desired response. The actual response of the system is observed by sampling this information after pre-determined discrete intervals of time (i.e., sampling period). This discrete system state and response information is compared with the desired IS operating requirements and based on any discrepancies between the two, the controller decides to take an appropriate action. This decision is dispatched to the distributed actuators that can directly control the local application and IS processes. The actuator implements the changes to the system, application, and IS operating parameters according to the command input from the controller.

Contrary to the modeling issues presented in Section 5.2, IS management issues are not generic. Selection of appropriate IS management or adaptation policy usually requires domain-specific knowledge of system level details of establishing a runtime instrumentation flow path from data sources to IDC(s).

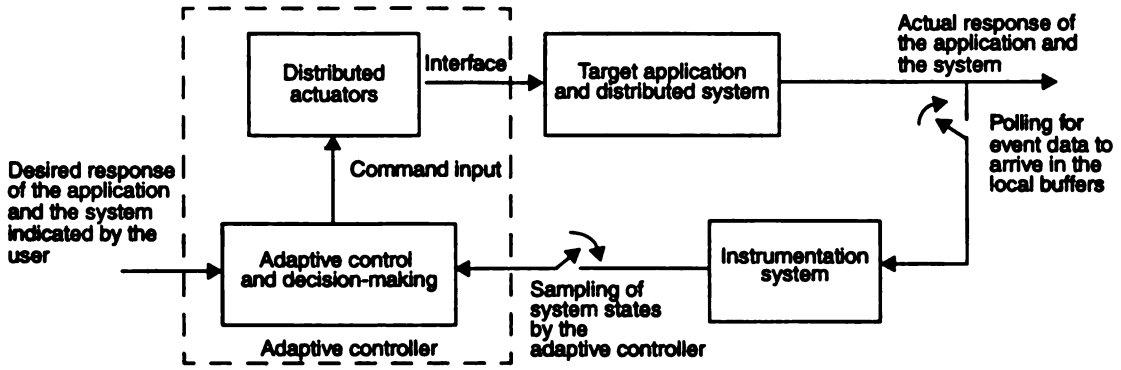


Figure 5-4. A generic model of an adaptive controller for managing an IS.

5.3 Resource Occupancy Modeling

The survey of computer system modeling techniques presented in Section 2.4 concluded that the tractability of solving a model depends on the complexity of the system. Since an objective of IS modeling is to provide an early feedback to the developers, it is important that model solution should be tractable. Nevertheless, interactions among IS and target system activities that contribute toward the complexity of the model cannot be ignored for the sake of analytically tractable solution. Thus a modeling approach that can compromise the conflicting requirements of tractability and accuracy of representing system behavior is desirable.

Our work in the area of instrumentation system modeling resulted in a novel approach of considering the contention for and sharing of system resources among various processes, termed as *Resource OCCupancy* (ROCC) modeling [214]. This method is suitable for parallel and distributed systems with multiprocessing and multithreading support to allow multiple processes, threads, and users to share the computing resources on the same node. The ROCC modeling approach is particularly suitable for evaluating the intrusion of an IS to the target system because processes belonging to both types of systems share the computing resources, such as CPU, I/O, network, shared bus, display, and other specialized devices interfaced to the system.

Unlike usual stochastic models, the ROCC model does not rely on the assumption of independence workloads (i.e., processes). Instead, it fully supports complex interactions and inter-dependences among workloads to closely model the actual behavior of target system and IS processes. This is an important feature of the ROCC modeling because any analysis based on the assumption of independent workloads cannot lead to reliable results beyond "back-of-the-envelope" calculations.

A ROCC model for an IS and target system combination is based on four components: resources, requests, management policies, and interacting workloads. Figure 5-5 presents an example of a ROCC model for a system node with four resources being shared by three processes via resource occupancy requests. The figure also illustrates the interactions among the processes. Additionally, processes may be synchronized with the resources or be blocked due to a swamped system resource. Such interactions with the resources are indicated by dotted lines from resources to the processes.

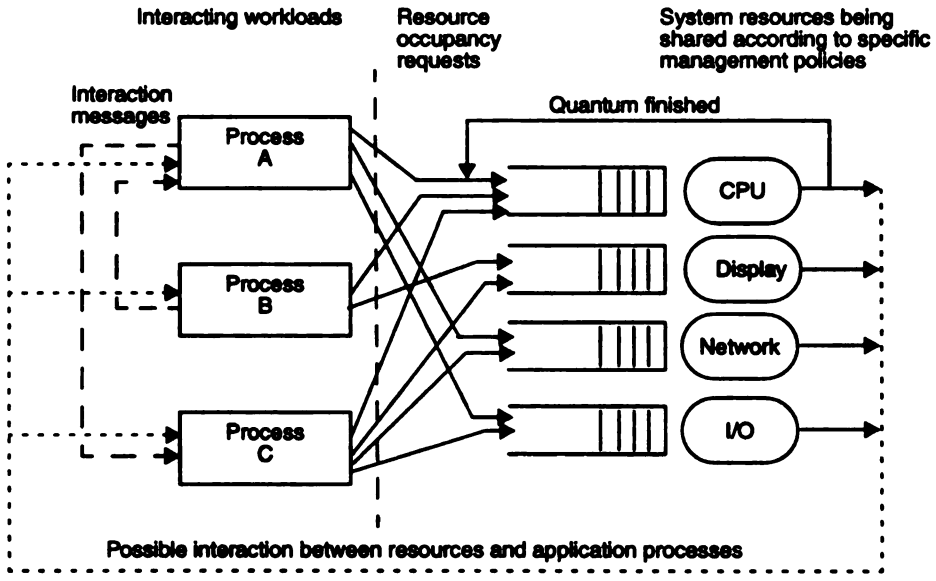


Figure 5-5. A Resource OCCupancy (ROCC) model consisting of shared resources, occupancy requests, management policies, and interacting workloads.

5.3.1 Components of a ROCC Model

Before applying ROCC modeling technique to an IS, it is essential to understand the details about individual components of a ROCC model. The four components of the model are elaborated in the following subsections.

5.3.1.1 Resources

In general, a computing system can be considered as a collection of resources that can be exploited to perform useful work for the users. These resources can be used by a single process or can be shared by several processes belonging to multiple users, depending mainly on the level of sophistication of the operating system [194]. Since state-of-the-art operating systems support multiprogramming and multithreading, we consider the resources to be shared among multiple user and system processes. Exclusive use of resources by a single user or process can be handled as a special case of the shared resource scenario in ROCC modeling.

In addition to sharing of system resources from the operating system perspective, state-of-the-art computer architectures are based on hardware level sharing of resources. Examples of hardware level resource sharing are pipelining and caching [102]. However, we cannot consider resource sharing at such low level of details because it will be almost impossible to collect measurements related to pipeline and cache states without using specialized hardware monitoring features available only on a few processors. These measurements are necessary to parameterize the ROCC model. Therefore, we restrict to the operating system level resource sharing, which requires comparatively simple measurements for parameterization purposes using commonly available monitoring and resource usage functions of the operating system.

For the application of ROCC modeling technique to an IS, resources are shared among (instrumented) application processes, other user and system processes, and IS processes.

The exact nature of IS processes depends on IS design and synthesis. In some cases, sensors and LIS are implemented as separate processes while others may have an LIS embedded in each instrumented application process as a separate thread. However, regardless of these differences, a set of system resources can be identified, which is used either by the application processes, IS processes, or both (i.e., shared among them).

5.3.1.2 Requests

A request is defined as an abstraction, through which a process can occupy a resource to accomplish useful work. A request is similar to a job submitted by an interactive terminal workload to a time-sharing computer system [4,106,115,117]. However, in the case of a ROCC model, there is no think time involved. A process generates a request to a resource; waits for the completion of the request or continues to generate subsequent requests, depending on workload characterization; and interacts with other processes as required by the workload characterization.

Requests are demands from different types of processes (workloads) to occupy the system resources during the execution of an instrumented application program. A request to occupy a resource specifies the amount of time needed for a single (coarse-grain) computation, communication, or I/O step of a process. We call this time as *occupancy time* of a resource due to a particular request. The occupancy times are often non-deterministic and can be characterized by an appropriate *probability density function* (*pdf*). Such characterization can be used for analytical or simulation-based evaluation of the model.

The nature of a request is specified by the workload characterization for the ROCC model. We address this issue in Section 5.4.

5.3.1.3 Management Policies

System resources are occupied by the requests according to the management policies followed by the resources. These management policies are often dictated by the operating system for accomplishing different goals, such as fair scheduling of a number of processes, obtaining high throughput, or reducing the response time. Emerging standards for operating system interfaces allow the user processes to handle system resource scheduling to a limited extent [97]. Therefore, management policies can also determine the sequence of resource usage by different processes and interaction or synchronization with other processes.

The goal of a management policy involves scheduling of system resource to fulfill the occupancy request of the processes. We identify a series of coarse-grained states to characterize each process, their dependences on the states of other processes, and occupancy requirements corresponding to each state.

5.3.1.4 Interacting Workloads

In addition to generating resource occupancy requests, a process in a ROCC model can interact with other processes. Taking interactions among processes into account addresses at least two aspects of multiprogramming: process *synchronization* and *blocking*. Multiple processes or threads require synchronization to force an explicit sequence of operations or order of accessing shared data structures. Different types of synchronization primitives include: mutual exclusion *locks*, *semaphores*, and *barriers*. Behavior of the processes that participate in these synchronization operations depend on one another and cannot be fully characterized in isolation. Message passing among processes may result in blocking of the sender. For instance, if two Unix processes interact using a pipe, the sender is blocked when the pipe becomes full to its capacity. This behavior is particularly worth considering for IS processes or threads that forward data to other down-steam modules in the

information flow path; intrusion of a sensor or LIS to the target system is affected due to this type of blocking.

Consideration of interacting workloads distinguishes a ROCC model from other computer system modeling approaches. Many workload models can adequately consider a single workload but try to deal with multiple workloads as a collection of identical workloads without any dependences among one another [37,38,45,231]. This approach can provide approximate results but may entirely overlook potential performance bottlenecks due to workload interactions, such as synchronization and blocking.

5.3.2 Characterization of the Queuing Network

The ROCC model is a queuing network. Depending on the workload characteristics, it can be an open or a closed queuing network. Often, it is not possible to characterize a ROCC model as an open or closed network because it works as a closed network for some workloads and open network for the others. Usually under application workload, the ROCC model is viewed as a closed network because after finishing one request the corresponding application process generates a subsequent request for the same or a different resource. The IS workloads that periodically schedule their data collection and forwarding tasks can make the ROCC model appear as an open queuing network. A request from an LIS may result in forwarding data sample(s) to the ISM; requests from the ISM may result in initial processing and forwarding of data sample(s) to an IDC; and requests from the IDC may result in the consumption of data sample(s), which is equivalent to an exit from the queueing network. Therefore, we consider that a ROCC model to be a hybrid of both open and closed queueing networks.

5.3.3 Dealing with Concurrence

So far in this section, we have restricted the scope of our discussion of the ROCC modeling technique to a single node. However, an IS for a parallel or distributed system is

a concurrent system itself and resides on multiple system nodes. The ROCC model can be extended for multiple nodes through interconnection resources (e.g., a network or a bus), requests to occupy interconnection resources (e.g., messages, packets, cells, flits, etc.), management policies for using the interconnect (e.g., ethernet or switch characteristics, protocols, etc.), and interacting workloads (e.g., collisions, stalls, deadlocks, etc.).

As an example of ROCC model functionality for a concurrent system, consider that the workload characterization dictates one process to pass a message to another process on a different node. The sender requires CPU time corresponding to the system call overhead for sending a message. Therefore, it composes and issues a CPU occupancy request and occupies the CPU according to the scheduling policy enforced by the operating system. After this request is serviced by the CPU, the sender process composes a network occupancy request and puts it at the tail of the network queue. If the message send operation is asynchronous, the sending process can continue its work by generating subsequent resource occupancy request; otherwise it will remain inactive until the message is received by the receiving process on another node. The network handles the occupancy request according the network bandwidth, switching characteristics, and protocol being used by the system. After the request has received the required occupancy time (not including the waiting time in the queue), the network prompts the receiving process residing at the receiving node about the arrival of a message. The receiving process composes a CPU occupancy request corresponding to the system overhead for retrieving a message from the network interface buffer. After this request is serviced by the local CPU, the message is considered to have been received. We can consider any blocking due to a swamped network resource in the path of a message.

Using the above technique, the ROCC model can closely follow the message-passing behavior of the actual system. Therefore, it can model an entire parallel or distributed system without over-simplifying the characteristics of the actual system.

It is clear that the ROCC modeling technique is depends on a workload characterization that is simple to carry out yet sophisticated enough to capture the complex dependences among interacting workloads. The number and type of shared resources also differ for each target system and IS combination. The behavior of the application, IS, and other user or system processes is determined through workload characterization, which is considered in Section 5.4.

5.4 Workload Characterization

Workload characterization is the most time consuming aspect of a typical system modeling effort. Large volumes of low-level measurement data are used to identify clusters of interesting system activity and transitions from one type of activity to another using a representative mix of programs (for instance, see the studies conducted by Dimpsey et al. [47] and Hughes [94]). Current software technology and rapid-prototyping tools have greatly reduced the turn around time of a software system development project; therefore, a prolonged workload characterization process may yield accurate results but those results may no longer be useful for the developers. In the context of ROCC modeling, a workload characterization effort with following features is desirable to evaluate the performance at an early stage of development:

1. short turn around time;
2. applicability to only a specific application instead of targeting generality; and
3. less dependence on low-level measurement data and more dependence on the knowledge about the application-domain.

This type of workload characterization is increasingly becoming popular for performance prediction studies that use a simulation model, which is parameterized for a particular parallel or distributed system using only high-level, coarse-grained measurements [17,181,231].

In order to model the behavior of an application, we first divide it into smaller and manageable modules that may have complex patterns of interdependence. A particular module of an application can be a subroutine or a series of high-level functions distinguished by their requirements of occupying specific system resources. These coarse-grain modules may be available as prototypes at an early stage of system development. For each module, we consider two aspects of its behavior relevant to a ROCC model of the system:

1. system resources that the module occupies to perform its specific function; and
2. its interactions with other modules belonging to a process in the same or a different class, such as application, IS, user process, etc.

A system resource is occupied by a process, corresponding to a coarse-grain module via issuing an *occupancy request* to the resource, indicating the length of time for which it needs to occupy the resource. Resources may use their own management policies to service these requests, such as first-come first-server, unequal priorities, preemption, etc. A module can interact (asynchronously) with another by sending a *message* to the other module. Interactions among modules are analyzed through the knowledge of the application. Messages are passed from one module to another using *message queues*. Such messages do not need to occupy any system resources.

This workload characterization strategy is best explained by its application to IS modeling of reference ISs in Section 5.5. While modeling of Paradyn and JEWEL IS benefits from the coarse-grain workload characterization strategy, the PICL IS study is based on conventional workload characterization techniques.

5.5 Results: Modeling and Management of Reference ISs

In this subsection, we apply the IS modeling methodology developed in Sections 5.1–5.4 to the reference ISs. In each case, we consider modeling and management issues before developing a model and characterizing workload for it. After workload characterization,

we decide IS performance metrics of interest that will be used at the evaluation stage. As the study of each reference IS has specific objectives, we consider each reference system in a separate subsection.

5.5.1 PICL IS

Modeling of PICL IS is the first application of the ideas promoted by this research. Compared to the later studies including Paradyn and JEWEL ISs, we did not directly collaborate with the PICL IS developers. The primary objective of the study was to provide a proof-of-the-concept for the ideas of early modeling and evaluation of instrumentation systems. At that time, we had not worked out the ROCC modeling methodology, therefore, the modeling and workload characterization for the PICL IS case uses conventional techniques. However, the results of this study are still relevant and extended by others [74], therefore, it is beneficial present it here.

5.5.1.1 IS Modeling Issues

In addition to its primary function as a portable communication library, PICL is often used for instrumenting the execution of parallel programs on distributed-memory parallel systems. In order to instrument an application program, PICL library functions are inserted in the program by the user before compilation. During program execution, calls to these functions generate instrumentation data in a particular event record format and log the data in a local buffer of each node. The user specifies the size of the buffer. These buffers are typically flushed at the end of program execution and merged into a single trace file at the host system.The objectives of modeling PICL IS are:

1. to optimize the use of limited resources at each nodes of the parallel system, such as local memory; and
2. to minimize the adverse effects of excessive intrusion due to trace data flushes that occur during the execution of long-running instrumented programs.

To effectively meet these objectives, concurrent IS is modeled to evaluate the merits of various management policies (presented in Section 5.5.1.2). The results of this modeling and analysis effort will have direct utility to system software designers who can incorporate appropriate management policies in the runtime environments.

5.5.1.2 IS Management Issues

Management of PICL IS is essential for a long-running program because local buffers will overflow with the immense amount of instrumentation data generated during program execution. By default, data collection stops after a buffer becomes full. Local buffers need to be flushed to allow continued data collection. We have identified two management policies for the PICL IS: *Flush One buffer when it Fills* (FOF) and *Flush All the buffers when One Fills* (FAOF). Neither of these is the default policy; and only FOF is actually supported as a PICL option, however, other IS developers have favored FAOF. The objective of modeling and evaluating this IS is to analyze the overhead of each policy and guide in the selection of an appropriate policy.

5.5.1.3 IS Model

We consider a distributed-memory parallel system consisting of P processors that have been allocated to execute a particular instrumented program. The concurrent IS consists of a set of trace data buffers, one at each processor, as shown in Figure 5-6. The performance trace data arrives, in response to the occurrence of an *event of interest*, at a local processor. Possible events of interest include communication, computation, local memory references, input/output device references, and a number of other program-specific activities. Ensuing trace data arrivals are stored in the local buffers of the corresponding processors as *trace records*. The number of arrivals stored in a local buffer of a processor i at a time t during the execution of a program will be denoted by $Q_i(t)$. Suppose that the capacity of each local buffer is l records and it can not allow any further arrivals of new trace records once this limit is reached. The inter-arrival times at each of these buffers are assumed

independent and exponentially distributed with rate α. The data in local buffers need to be transferred dynamically to the host system when the buffers become full. Storage at the host system is the next level of trace data storage hierarchy after the local buffers. This level is a larger buffer in the main memory of the host and is called the *main buffer*. At the end of the program, all the trace records must be transferred to the main buffer. The main buffer, in turn, may also have to be flushed to the subsequent level of the storage hierarchy, for example, a disk. The storage capacity is assumed to increase as we go further in the storage hierarchy. The scope of the model for this PICL IS study is restricted only to the local buffers, but it can readily be extended to higher levels of the storage hierarchy.

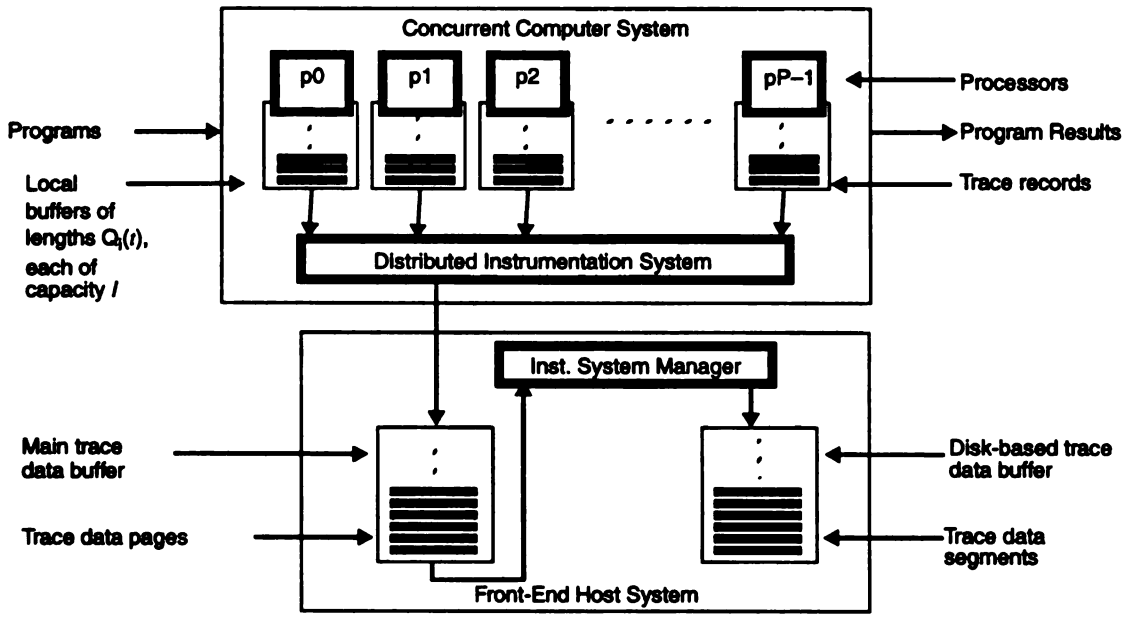


Figure 5-6. Model for a concurrent program instrumentation facility.

5.5.1.4 Workload Characterization

Since for this case study, we are considering the lower-level IS details by focusing on the arrivals of individual tracer records, we assume that these arrivals are independent and exponentially distributed to simplify the analytical solution of the model. In order to justify the assumption of exponential inter-arrival times, we examined time-stamped PICL trace records from a particular node of nCUBE-2 system. Figure 5-7 shows the frequency

distribution histogram of inter-arrival times between successive records during a particular instrumented execution. This frequency distribution is compared to the exponential *Probability Density Function* (*pdf*) by drawing the histogram such that area under the histogram is one. It can be observed that exponential inter-arrival times is a reasonable assumption.

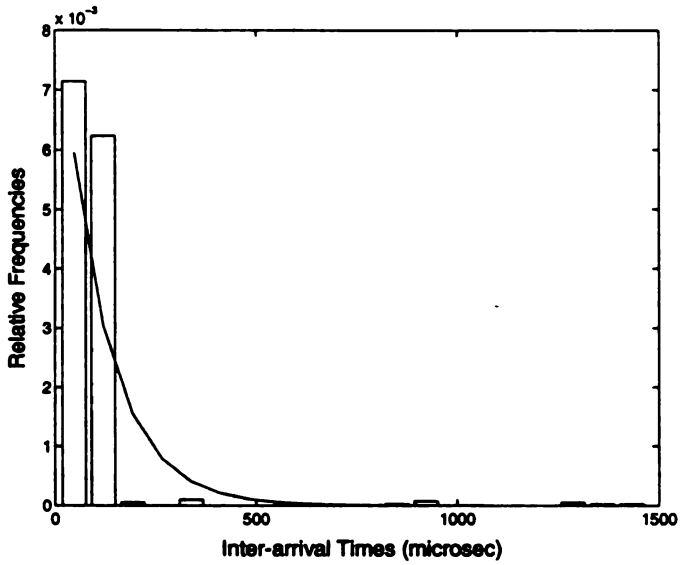


Figure 5-7. Histogram of inter-arrival times for PICL trace records at a particular nCUBE-2 node.

For the purpose of trace data transfer, we assume that the direct network connecting all the processors is worm-hole routed. Therefore, the latency of data transfer between any two nodes is independent of the distance between them [142]. Moreover, sufficient number of communication channels is available between each pair of neighboring nodes, so that the possibility of any channel contentions is negligible. We also assume that the trace data are transferred to the host system through specific I/O processors that are accessible from processing nodes through the direct network.

5.5.1.5 Performance Metrics

With the queuing model for PICL IS, we can perform a wide range of experiments using different parameters and calculate various metrics in order to evaluate IS performance

under alternative management policies. Two of the metrics selected for comparing the FOF and FAOF policies are:

1. length of the time interval after which a local buffer becomes full and needs to be flushed (*trace stopping time*); and
2. ratio of the number of flushes to the number of arrivals for a local buffer during a program's execution (*frequency of buffer flushes*).

These metrics represent the lower level quantitative considerations of IS behavior. Each metric, its method of calculation, and its interpretation are summarized in Table 5-1. Further evaluation using these metrics yields higher level feedback to aid in design decisions.

Table 5-1. Metrics for evaluating the PICL IS management policies.

Metric	Calculation	Interpretation
Trace stopping time	Stochastic analysis of arrivals to local buffers	A higher value is desirable
Flushing frequency	Regenerative nature of buffer filling stochastic process	A higher value indicates greater overhead to the user program

5.5.2 Paradyn IS

Modeling of Paradyn IS was carried out in conjunction with the Paradyn group at University of Wisconsin and University of Maryland [218]. The idea of ROCC modeling emerged as a consequence of this case study. In this subsection, we present modeling and management issues, the ROCC model for Paradyn IS, workload characterization for the ROCC model, and IS performance metrics of interest to this study.

5.5.2.1 IS Modeling Issues

We apply the structured IS development approach that we presented in Chapter 4 to the instrumentation system of the Paradyn parallel performance measurement tool. The

objectives of Paradyn IS modeling include: comparing alternatives for IS management policies and configurations; evaluating IS overheads due to resource sharing; identifying any IS-induced performance bottlenecks; and determining desirable operating conditions for the IS.

The Paradyn IS can be represented by a queuing network model to capture system-level details, as shown in Figure 5-8. It consists of several sets of identical subnetworks representing a local Paradyn daemon and application processes. We assume that the subnetworks at every node in the concurrent system show identical behavior in terms of sharing local resources during the execution of an SPMD program. Figure 5-8 highlights the performance data collection and forwarding activities of a Paradyn daemon on a node. These IS activities are central to Paradyn's support for on-line analysis of performance bottlenecks in long-running application programs. However, they may adversely affect application program performance, since they compete with application processes for shared system resources.

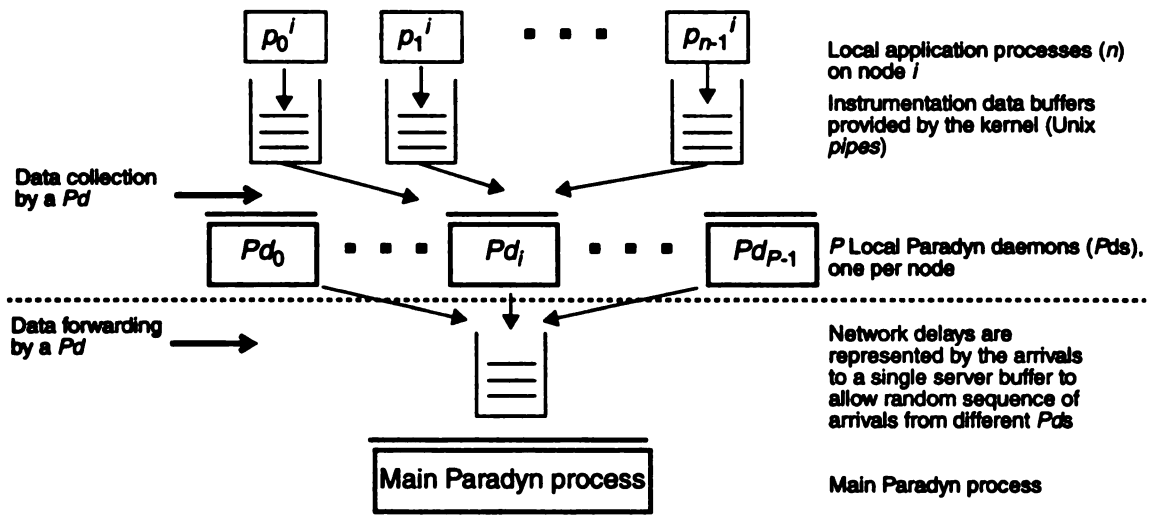


Figure 5-8. A model for the Paradyn IS with considerations of overall, system-level details. The distributed system consists of *P* nodes and each node may have up to *n* instrumented application processes.

Although Figure 5-8 adequately represents the operation of the Paradyn IS, the level of details captured by it are not sufficient for evaluating alternative configuration options and

management policies. In particular, the data flow is correlated with the instrumented application behavior and resource management enforced by the operating system. Therefore, the ROCC model for Paradyn IS (presented in Section 5.5.2.3) adequately captures the required level of detail to meet the objectives of this study.

5.5.2.2 IS Management Issues

Two possible options for a Paradyn daemon to schedule data collection and data forwarding at a node are *collect-and-forward* (CF) and *batch-and-forward* (BF). As illustrated in Figure 5-9, under the CF scheduling policy, the Paradyn daemon (Pd) collects a sample from an instrumented application process and immediately forwards it to the main Paradyn process. Under the BF policy, the Pd collects a sample from the application process and stores it in a buffer until a batch of an appropriate number of samples is accumulated and then forwarded to the main Paradyn process.

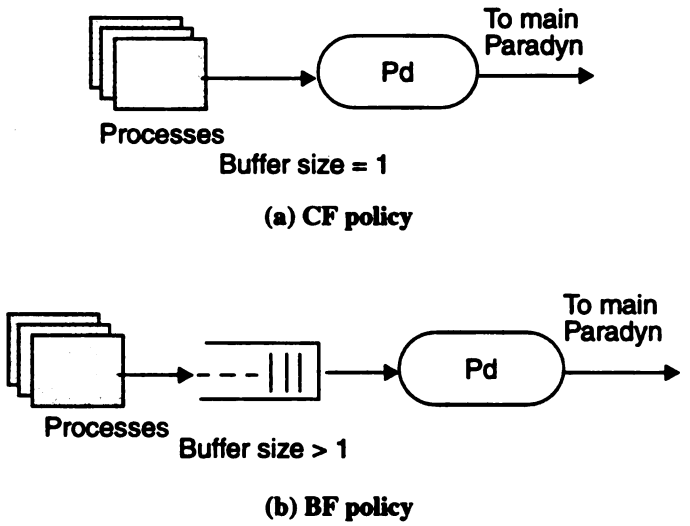
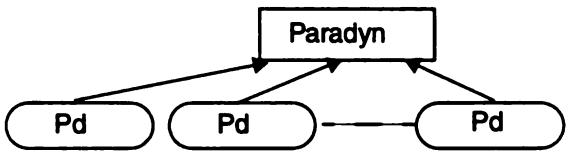


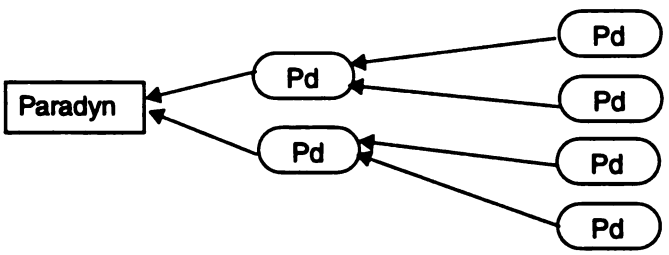
Figure 5-9. Two policies for scheduling data collection and forwarding: (a) collect-and-forward (CF) and (b) batch-and-forward (BF).

In case of using Paradyn IS on a Massively Parallel Processing (MPP) system, we consider two options for forwarding the instrumentation data from the Paradyn daemon to the main Paradyn process: *direct forwarding* and *binary tree forwarding*. Under the configuration

for direct forwarding, a Paradyn daemon directly forwards one or multiple samples (under the CF and BF policies, respectively) to the main Paradyn process. Under the binary tree forwarding scheme, the system nodes are logically arranged as a binary tree; every Paradyn daemon running on a non-leaf node receives, processes, and merges the samples or batches from Paradyn daemon running on its two children nodes. Figure 5-10 illustrates the two configurations.



(a) Direct forwarding

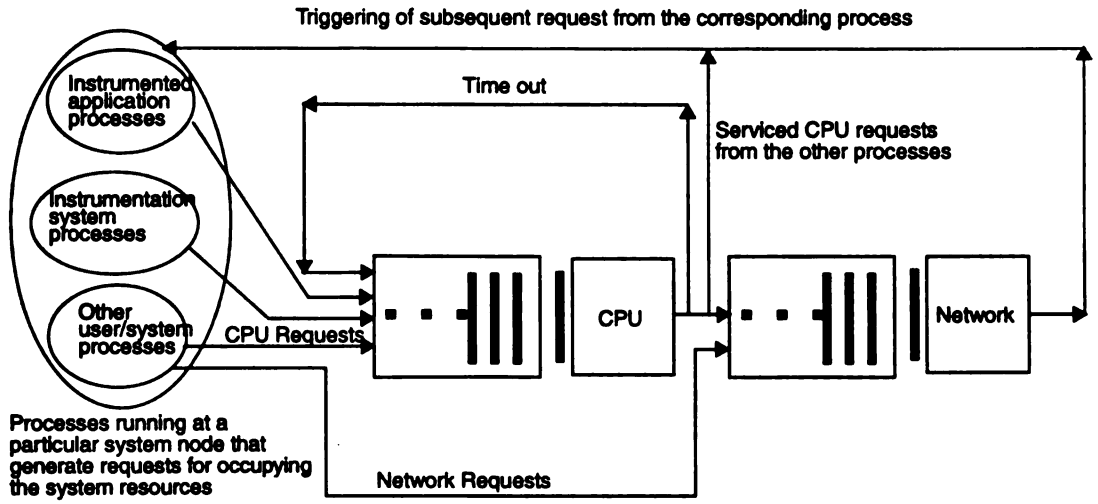


(b) Binary tree forwarding

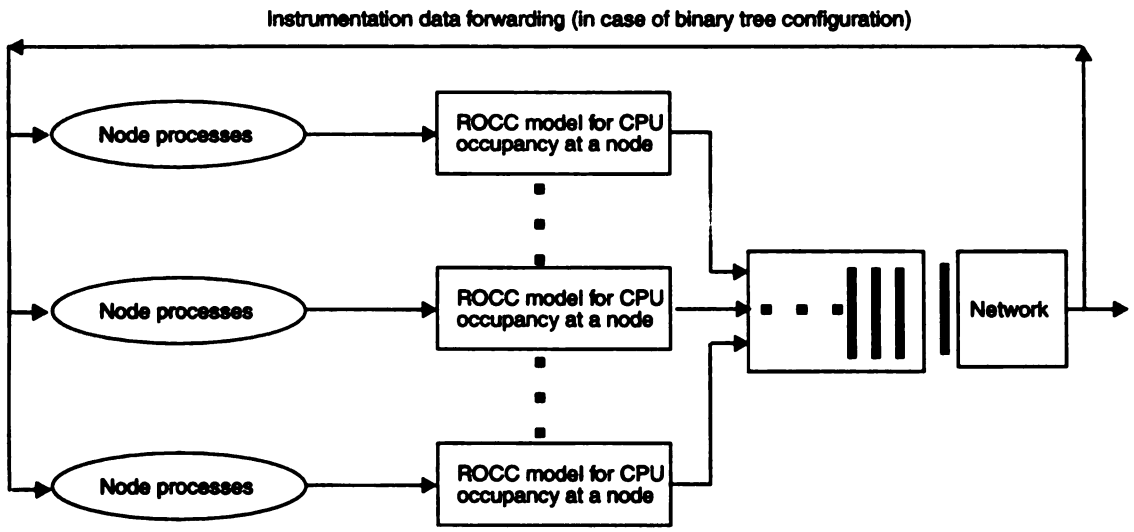
Figure 5-10. Two configurations for data forwarding for an MPP implementation of the Paradyn IS: (a) direct forwarding and (b) binary tree forwarding.

5.5.2.3 IS Model

This subsection introduces the application of the ROCC model to isolating the overheads due to non-deterministic sharing of resources between the Paradyn IS and application processes [214]. Figure 5-11 depicts the ROCC model for Paradyn IS with local and global levels of detail. The local level of detail considers only one system node and the global level of detail considers all of the system nodes. The ROCC model includes two types of resources of interest at a node for the Paradyn IS: CPU and network. Each CPU is being shared by three types of processes on every node: application, IS, and other user processes.



(a) ROCC model for a particular system node



(b) ROCC model for the entire system

Figure 5-11. The resource occupancy model for the Paradyn IS with (a) local and (b) global levels of detail.

Due to the interactions among different types of processes at the same node and IS processes at multiple nodes, it is impractical to solve the ROCC model analytically. Therefore, simulation is a natural choice. The execution of the ROCC model for the Paradyn IS relies on a workload characterization of the target system, which in turn, relies on measurement-based information from the specific system.

5.5.2.4 Workload Characterization

The workload characterization for this study has two objectives: (1) to determine representative behavior of each process of interest (i.e., application, IS, and other user/ system processes) at a system node (see Section 5.5.2.4.1); and (2) to fit appropriate theoretical probability distributions to the lengths of resource occupancy requests corresponding to the states of each of these processes (see Section 5.5.2.4.2).

5.5.2.4.1 Process Model

We consider the states of an instrumented process running on a node, as illustrated by Figure 5-12, which is an extension of the Unix process behavior model. After the process is admitted, it can be in one of the following states: *Ready*, *Running*, *Communication*, or *Blocked* (for I/O). The process can be preempted by the operating system to ensure fair scheduling of multiple processes sharing the CPU. After specified intervals of time (in case of *sampling*) or after occurrence of an event of interest (in case of *tracing*), such as spawning a new process, instrumentation data are collected from the process and forwarded over the network to the main Paradyn process via a Paradyn daemon.

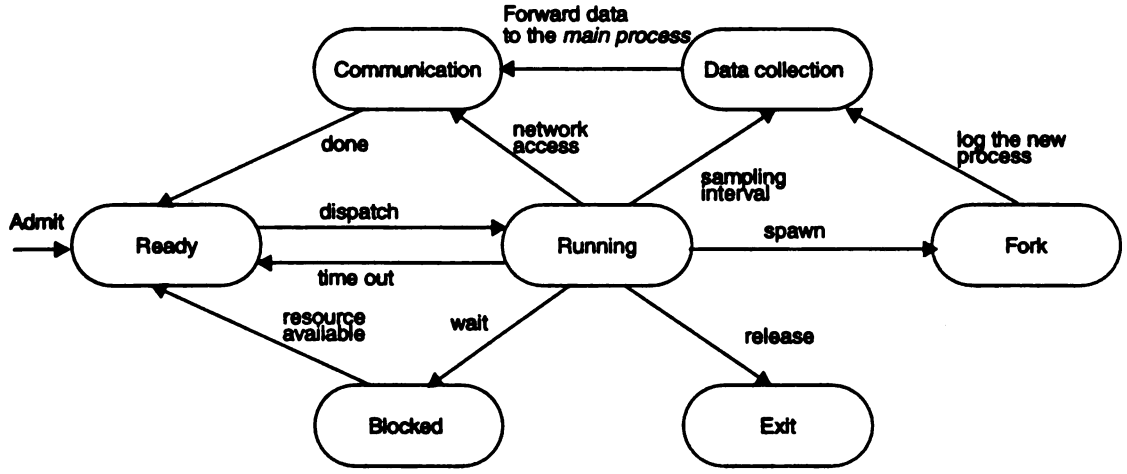


Figure 5-12. Detailed process behavior model in an environment using an instrumentation system.

In order to reduce the number of states in the process behavior model and hence the level of complexity, we group several states into a representative state. The simplified model, shown in Figure 5-13, considers only two states of process activity: *Computation* and *Communication*. This simplification facilitates obtaining measurements without any special operating system instrumentation. This characterization also considers the interactions among states across different processes. For instance, an instrumented application process interacts with the local Paradyn daemon process either by forwarding a sample to it or by blocking (via the operating system) if the pipe is full. The *Computation* and *Communication* states require the use of the CPU and network resources, respectively. The *Computation* state is associated with the *Running* state of the detailed model of Figure 5-12. Similarly, the *Communication* state is associated with Figure 5-12's *Communication* state, representing the data collection, network file service (NFS), and communication activities with other system nodes. Measurements regarding these two states of the simplified model are conveniently obtained by tracing the application programs. The model provides sufficient information to characterize the workload when applied in conjunction with the resource occupancy model.

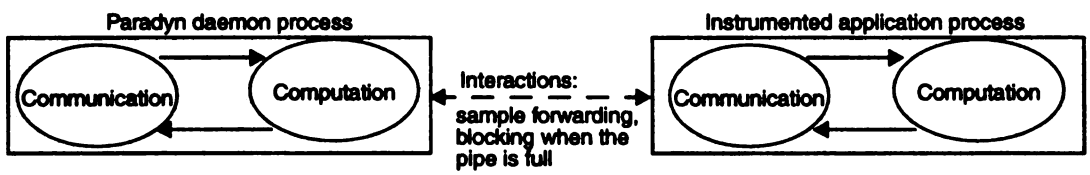


Figure 5-13. A process model based on alternating computation and communication states of two types of interacting workloads.

Figure 5-14 shows the ROCC simulation model for the process model presented in Figure 5-13. Although the interactions among processes as well as processes and resources are handled by message-passing queues, the occupancy requests for the system resources are treated separately from the interactions among the processes. The figure shows only the instrumented application process generating resource occupancy requests for the clarity of presentation; in fact, both application processes and a Paradyn daemon can concurrently send the occupancy requests during a ROCC model simulation.

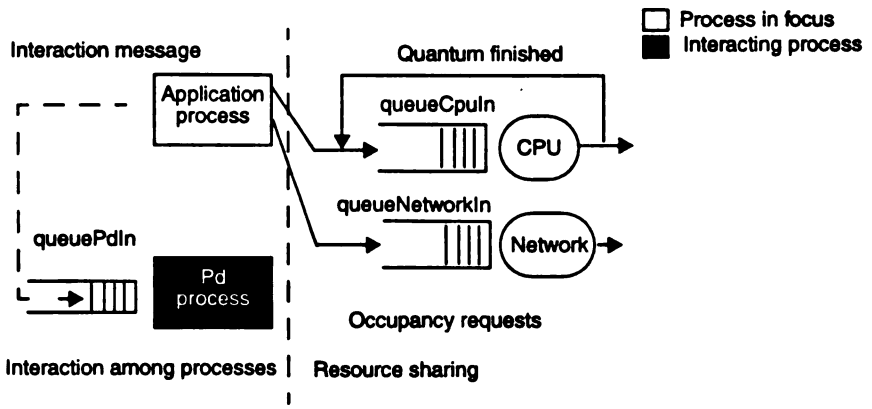


Figure 5-14. The ROCC simulation model corresponding to the alternating process model, shown in Figure 5-13.

5.5.2.4.2 Distribution of Resource Occupancy Requests

Trace data generated by the IBM SP-2's AIX operating system tracing facility are the basis for the workload characterization. We used the trace data obtained by executing the NAS benchmark, *pvmbt*, on the SP-2 system [178]. Table 5-2 presents a summary of the statistics for CPU and network occupancy by various processes.

Table 5-2. Summary of statistics obtained from measurements of NAS benchmark *pvmbt* on an SP-2.

Process Type	CPU Occupancy (microseconds)				Network Occupancy (microseconds)			
	Mean	St. Dev.	Min.	Max.	Mean	St. Dev.	Min.	Max.
Application process	2,213	3,034	9	10,718	223	95	48	5,241
Paradyn daemon	267	197	11	6,923	71	109	31	816
PVM daemon	294	206	9	1,662	58	59	36	5,169
Other processes	367	819	8	9,746	92	80	8	198
Main Paradyn process	3,208	3,287	11	10,661	214	451	46	4,776

We apply standard distribution fitting techniques to determine theoretical probability density functions that match the lengths of resource occupancy requests corresponding to

the states of the processes [41,46]. Figure 5-15 shows the histograms and *probability density functions* (*pdf*s) for the lengths of CPU and network occupancy requests (in (a) and (b), respectively) by an application (NAS benchmark) process.

Quantile-quantile (Q-Q) plots are often used to visually depict differences between observed and theoretical *pdf*s (see Law and Kelton [116]). For CPU requests (Figure 5-15a), the Q-Q plot of the observed and lognormal quantiles approximately follows the ideal linear curve, exhibiting differences at both tails, which correspond to very small and very large CPU occupancy requests relative to the CPU scheduling quantum. Despite these differences, the lognormal *pdf* is the best match. For network requests by application processes (Figure 5-15b), an exponential distribution yields the best fit. Table 2 summarizes the distribution fitting results for various processes; the inter-arrival time of requests to individual resources is approximated by an exponential distribution.

5.5.2.5 Model Parameterization and Validation

The workload characterization presented in the preceding section yields parameters for the ROCC model for the Paradyn IS, as listed in Table 5-3. Note that *exponential(m)* means an exponential random variable with mean inter-arrival time of m microseconds, and *lognormal(a, b)* means a lognormal random variable with mean a and variance b. These parameters were calculated using maximum likelihood estimators given by Law and Kelton [116].

In order to validate the ROCC simulation model and its parameterization, we simulated the same case that was measured empirically on an IBM SP-2 system. Table 5-4 compares the CPU time for the NAS benchmark and Paradyn daemon during the execution of the program using measurement and simulation. It is clear that the simulation model-based results follows the measurement-based results. Therefore, using the estimated parameters, the model can be simulated to answer "what if" questions.

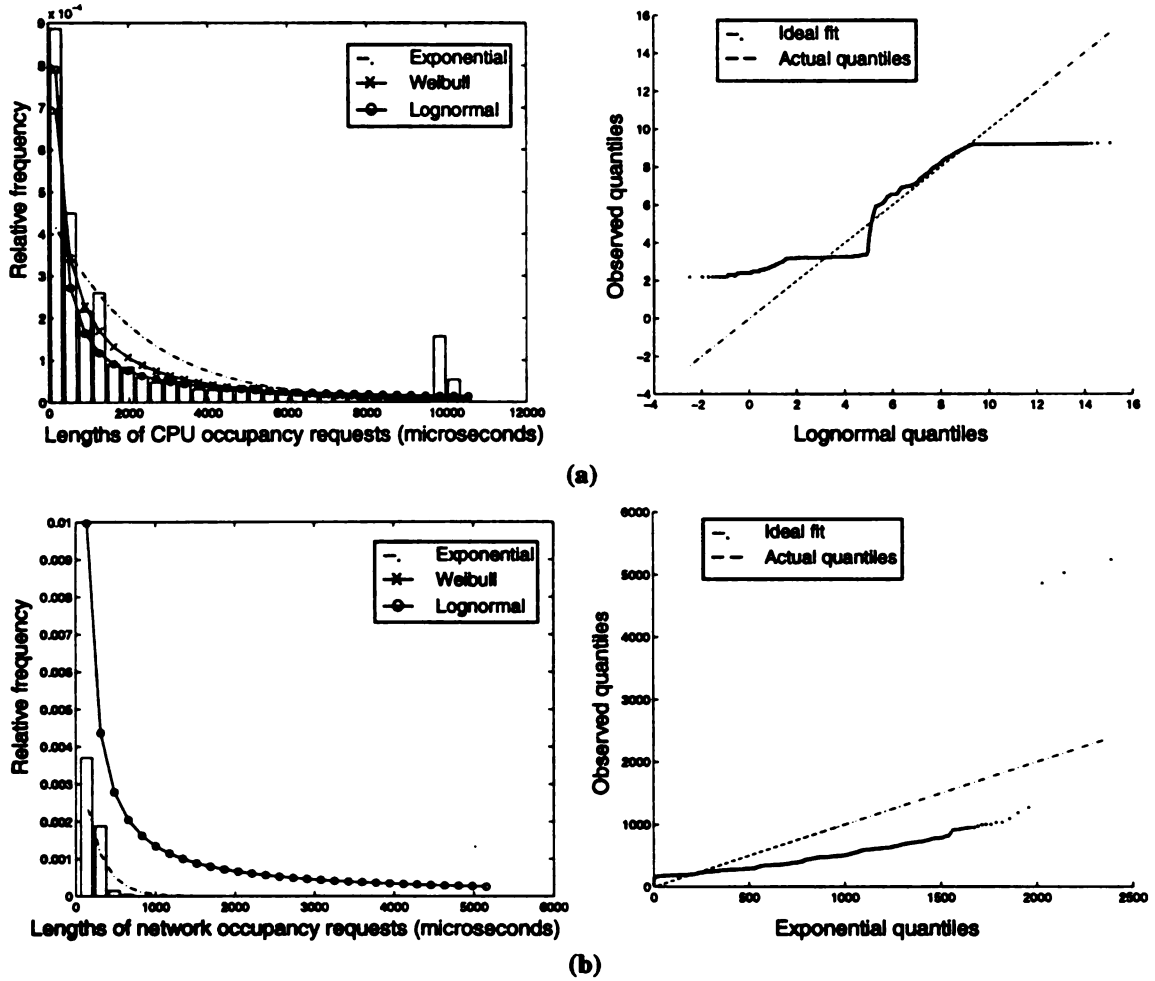


Figure 5-15. Histograms and theoretical *pdf*s of the lengths of (a) CPU and (b) network occupancy requests from the application process. Q-Q plots represent the closest theoretical distributions.

Table 5-4. Comparison of measurements of *NAS* benchmark *pvmbt* on an SP-2 with the simulation results of the same case.

Type of experiment	Application CPU time (sec)	Pd CPU time (sec)
Measurement based	85.71	0.74
Simulation model based	87.96	0.59

Table 5-3. Summary of parameters used in simulation of the *ROCC* model. All time parameters are in microseconds. The range of inter-arrival times for the Paradyn daemon corresponds to varying the rate of sampling (and forwarding) performance data by the application process.

Parameter Type	Parameter	Range of Values
Configuration	Number of application processes per node	1–32 (typical 1)
	Number of *Pd* processes per node	1–4 (typical 1)
	Number of CPUs per node	1
	Number of nodes	1–256 (typical 8)
	CPU scheduling quantum (microseconds)	10,000
Application Process	Length of CPU occupancy request	Lognormal (2213, 3034)
	Length of network occupancy request	Exponential (223)
Paradyn Daemon	Length of CPU request	Exponential (267)
	Length of network request	Exponential (71)
	Inter-arrival time	5,000–50,000 (typical 40,000)
PVM Daemon	Length of CPU request	Lognormal (294, 206)
	Length of network request	Exponential (58)
	Inter-arrival time	Exponential (6485)
Other Processes	Length of CPU request	Lognormal (367, 819)
	Length of network request	Exponential (92)
	Inter-arrival time of CPU requests	Exponential (31485)
	Inter-arrival time of network requests	Exponential (5598903)

5.5.2.6 Performance Metrics

Two performance metrics are of interest for this study: *average direct overhead* due to IS modules and *monitoring latency* of data forwarding. Average direct overhead represents the *occupancy time of a shared system resource by the IS modules, which is averaged over all the system nodes.* A lower value of the direct overhead is desirable. Direct overhead quantifies the contention between application and IS processes for the shared resources on a particular node of the system. Monitoring latency has been defined by Schwan et al. as *the amount of time between the generation of instrumentation data and its receipt at a logically central collection facility* (in our case, the main Paradyn process) [71]. Monitoring latency impacts the main Paradyn process, since a steady flow of data samples from individual system nodes is needed to allow the bottleneck searching algorithm to work properly. In order to quantify the IS intrusion to the application, we calculate the

application CPU utilization per node, with and without instrumentation. Malony et al. refer to this metric as *time perturbation* [125]. Performance metrics are summarized in Table 5-5.

Table 5-5. Metrics for evaluating the Paradyn IS evaluation.

Metric	Calculation	Interpretation
Average direct CPU overhead due to IS modules	Simulation of the ROCC model	A lower value means lower instrumentation overhead
Monitoring latency per received sample	Simulation of the ROCC model	A lower value is desirable for a steady flow of runtime information to the main Paradyn process
Average application CPU utilization	Simulation of the ROCC model	A lower value is desirable as it reflects lower intrusion to the application

5.5.3 JEWEL IS

Modeling of JEWEL IS is performed in the context of resource management for a real-time video conferencing application [219]. One of the motivations behind this study was to explore the potential application of the ROCC modeling technique in the area of real-time adaptive control. The results of this study show the promise of the technique for dynamic resource management problems for distributed real-time systems.

5.5.3.1 IS Modeling Issues

We intend to accomplish two objectives through modeling and evaluation of the JEWEL IS for video conferencing application:

1. provide early feedback to the system developers about the intrusion and overhead of alternative IS configuration options under different operating conditions; and
2. suggest policies for adaptively controlling the IS to meet domain-specific requirements.

Selection of a particular IS configuration based on modeling and evaluation process can ensure minimum overhead to the target application but cannot guarantee meeting domain-

specific requirements. The system can be adaptively controlled to meet local or global requirements that ensure desired operation.

The level of detail considered for this evaluation includes the behavior of the individual processes running on a system node, their interactions with one another, and their interaction with the IS processes. As in case of Paradyn IS, we account for interactions among workloads by representing each type of process as a series of inter-dependent coarse-grain states.

5.5.3.2 IS Management Issues

In this subsection, we introduce configuration options and adaptive control policies for the JEWEL IS, which are of interest from the perspective of our application. Two alternative configuration options related to data forwarding are: *collect-and-forward* and *batch-and-forward*. The external sensor can be configured to operate in one of two possible modes: *busy-waiting* and *polling*. The JEWEL IS can be adaptively controlled using one of two policies: *static polling period adaptation* or *dynamic polling period adaptation*. These policies can be scheduled in one of two possible manners: *centralized* or *distributed*. These configuration and adaptation alternatives are elaborated in the following:

1. **Collect-and-Forward vs. Batch-and-Forward**: The current version of the Jewel external sensor for Unix platforms collects one event record from the shared memory segment and forwards it to the collector and continues to repeat this procedure. We refer to this procedure as *collect-and-forward* (CF) policy. However, it is possible to customize Jewel components for a specific application; therefore, we model and evaluate a slightly different MDR forwarding policy, called *batch-and-forward* (BF) policy. Under the BF policy, the external sensor collects all the outstanding event records from the shared memory and forwards them as a batch to the collector.
2. **Busy-Waiting vs. Polling**: The external sensor of the Jewel IS remains in a busy-wait state to continuously check the ring buffer in a shared memory segment for the arrival of new event records. We propose a configuration of the external sensor that periodically polls the shared memory to collect any outstanding event data. As opposed to the busy-waiting approach, the polling-driven approach does not require CPU time during the period between two successive searches for outstanding event records.

3. **Static Polling Period vs. Dynamic Polling Period Adaptation Policies**: We consider two policies to adapt the Jewel IS behavior according the requirements of video conferencing application. Under the *static polling period* (SPP) adaptation, the period of polling the shared memory to collect event data is statically specified before the execution of the instrumented application. Adaptive control system tries to meet the system constraints while keeping the polling period fixed or completely turns off the data collection and forwarding when it cannot meet the constraints. However, it resumes collecting the event data as soon as it can meet the constraints. Under the *dynamic polling period* (DPP) adaptation policy, the polling period increases to the double of its current value at an adaptive controller sampling instant of time where the constraints are not met. Thus the polling period gradually increases until the data collection is (temporarily) turned off. As soon as the control system can again meet the constraints, the data collection and polling is turned on. The polling period continues to reduce by half at observation instants as long as the adaptive controller can meet the constraints. The controller checks the IS and application states only at discrete instants of time after each controller sampling period (observation instant). Choice of dynamically changing the polling period by a factor of two is somewhat arbitrary; our intention is to make only incremental changes to the polling periods to avoid sudden changes in the application and system response. We know one example where IS data collection rate dynamically changes by a factor of two; Paradyn instrumentation system continues to reduce the volume of collected data per unit of time by half with the passage of time [136].
4. **Centralized vs. Distributed Controller:** We model and evaluate two configurations of the adaptive controller: centralized and distributed. In case of a centralized control system, all the control decisions are made centrally by the resource manager, which is located at a logically centralized location in the system. Control decisions are based on the system state as a whole and implemented at all system nodes as a gang schedule with the help of resource manager agents. On the other hand, distributed control is based on making localized decisions by the resource manager agents using the system state at each node. In this case, the implementation of these decisions is scheduled only at the local node.

In addition to the above IS configuration, adaptation, and control options, we can consider the adaptive control and resource management strategies by directly using the controllable parameters of the application itself (i.e., algorithmically steering it [50]). Although important, such considerations are beyond the scope of this study.

Real-time video application has timing constraints for the tasks involved in sending and receiving video frames. It is a requirement of the client to receive and display 30 frame per second to represent a dynamic scene in real-time. However, the quality (smoothness of changes) will be lost if the frame rate reduces from this value. Since Jewel IS components

share system resources (such as, CPU, network, and I/O) with the clients and the server, the quality of service problem may get aggravated if the real-time tasks do not have adequate laxity. In this scenario, adaptive control of the IS addresses two domain-specific requirements:

- IS intrusion to the real-time characteristics of the application (in terms of the frames processed by a client per second) remains within the specified limit of 30 frames per second; and
- direct overhead of the IS to the application (in terms of Jewel sensor CPU utilization) remains within the user-specified limit.

In order to provide this adaptation, we have to identify a *controllable parameter* in this setup of the application, IS, and the resource manager subsystems. We use the *polling period* (definition follows) of Jewel sensor as the controllable parameter. Due to the design of a Jewel external sensor, it is always in a busy-wait state for the arrival of event records in the ring buffer. We introduce a polling scheme due to which the sensor gives up the CPU (i.e., sleeps) for a variable polling period. After this period, it (wakes up and) polls the ring buffer for any newly arrived event records.

An important consideration for the design of adaptive controller components (i.e., resource manager and its agents) is the *sampling period*, which is the time between discrete observation instants of the current system state derived from the information delivered by the IS. If the sampling period is too large, several system state changes may not be "observable" to the controller and it may not be very responsive. On the other hand, if this period is too small, adaptive controller components can cause excessive overhead. Our evaluation in Chapter 6 considers the choice of sampling period as a part of the controller design and provides feedback regarding the choice of this parameter.

In addition to considering potential intrusion to the real-time behavior of the video conferencing application, impact of the Jewel IS to the resource manager tasks should also be considered. If the IS cannot deliver runtime information to the resource manager within

a pre-calculated limit of time after it was generated by a client or the server (to be defined as *monitoring latency* in Section 5.5.3.6), it can cause "oscillations" of the video application response as the adaptive controller may continue to steer the system from one nominal point of operation to another in the available space of operating conditions. Jewel IS should either guarantee delivering the runtime information before this "hysteresis" time limit expires or discard it. Resource manager design that incorporates certain degree of hysteresis can make it less sensitive to the transient conditions. However, adaptation of the IS behavior to maintain a desired monitoring latency is beyond the scope of this paper and we use the adaptive controller sampling period as available hysteresis limit. Possible operating conditions for the adaptive controller depend on the types of adaptation (considered in the following subsection).

5.5.3.3 IS Model

There are three shared resources of interest at each system node: CPU, local X server (i.e., graphical display device), and network. Camera that is used by the application server process to capture the scene is not shared with any other process; therefore, it is not considered as a part of the ROCC model. System resources included in the ROCC model are shared among four types of processes at every node: video application client/server process, resource manger agent, jewel sensor, and system load visualizer at nodes running a client or the server. We assume that other user or system processes do not significantly load the system nodes. In cases where a node is shared among multiple users, we can easily incorporate the behavior of the user processes through adequate workload characterization. Lengths of occupancy requests for shared resources and interactions among processes is addressed in Section 5.5.3.4.

Figure 5-16 presents the ROCC model for the distributed system under study. At every node, local processes send occupancy requests to the shared system resources according to the workload characterization (to be presented in Section 5.5.3.4). When a resource finishes servicing a request, it can trigger the generating process to generate next request

for the cases where a process remains blocked until its request is fully serviced. In some cases, such as requests to the network for sending a message, the requesting process is not blocked (due to asynchronous sends) and can continue generating subsequent requests according to its workload model.

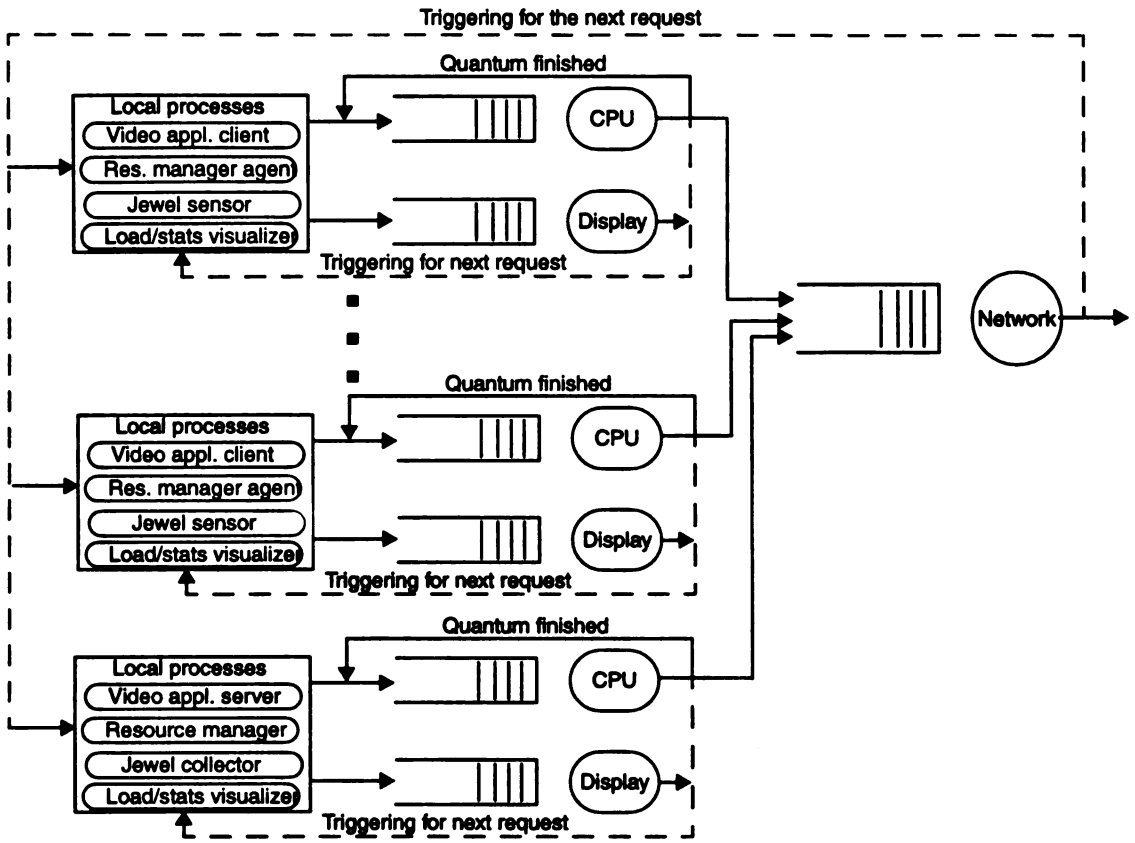


Figure 5-16. Resource occupancy model for the video application with real-time adaptive control and instrumentation.

5.5.3.4 Workload Characterization

There are three types of application processes in the video conferencing application: server and client processes, a visualizer process local to each client and the server, and resource manager agent processes on each node of the system. The central resource manager process runs as an independent process on a separate workstation with the JEWEL collector process.

Server Process: Figure 5-17(a) shows the behavior of the application server process and Figure 5-17(b) shows the corresponding components in the ROCC model. Application server process captures a frame from the camera, displays it locally, compresses the frame, and multicasts it to all the clients. At each of these stages, it triggers the Jewel external sensor process to collect and event record and forward it as an MDR to the Jewel collector. It also triggers a visualizer process that displays the data transfer statistics at the host workstation. Note that the actual data collection and forwarding is replaced by the following sequence of occupancy requests and interaction messages under the ROCC model: (1) an application server process sends a CPU occupancy requests (to charge the CPU time) for forwarding the event data to shared memory ring buffer; (2) an application process notifies the external sensor of the arrival of new data by putting a message in its input queue; (3) when the sensor polls the ring buffer after the current polling period, it retrieves the message; (4) the sensor charges CPU time through an occupancy request corresponding the system call overhead to forward an MDR to the collector; and (5) the sensor requests the network occupancy to (asynchronously) sending the MDR to the collector. These activities are not an exact replica of the actual process behavior but capture all the resource usage demands and dependences among the processes. Therefore, this workload model is a trade-off between high accuracy and simplicity.

Using Solaris OS-supplied high-resolution timing functions, we measure the occupancy of system resources corresponding to the server process states (illustrated in the Figure 5-17(a)) by running the server process on a Sun Ultra-1 platform for several hours. Our objective is to fit an appropriate theoretical distribution that best describes occupancy requirement of the system resources in each state. Figure 5-18 presents the normalized histograms (such that the area under every histogram is one) and exponential, weibull, and normal *probability density functions* (*pdf*). We use the *Kolmogorov-Smirnov (K-S) goodness of fit test statistic* to quantitatively measure the differences between a theoretical *pdf* and an observed *pdf*. The K-S test is based on a *statistic*, called the *K-S statistic*, which is equal to the absolute value of the largest difference between an observed and a

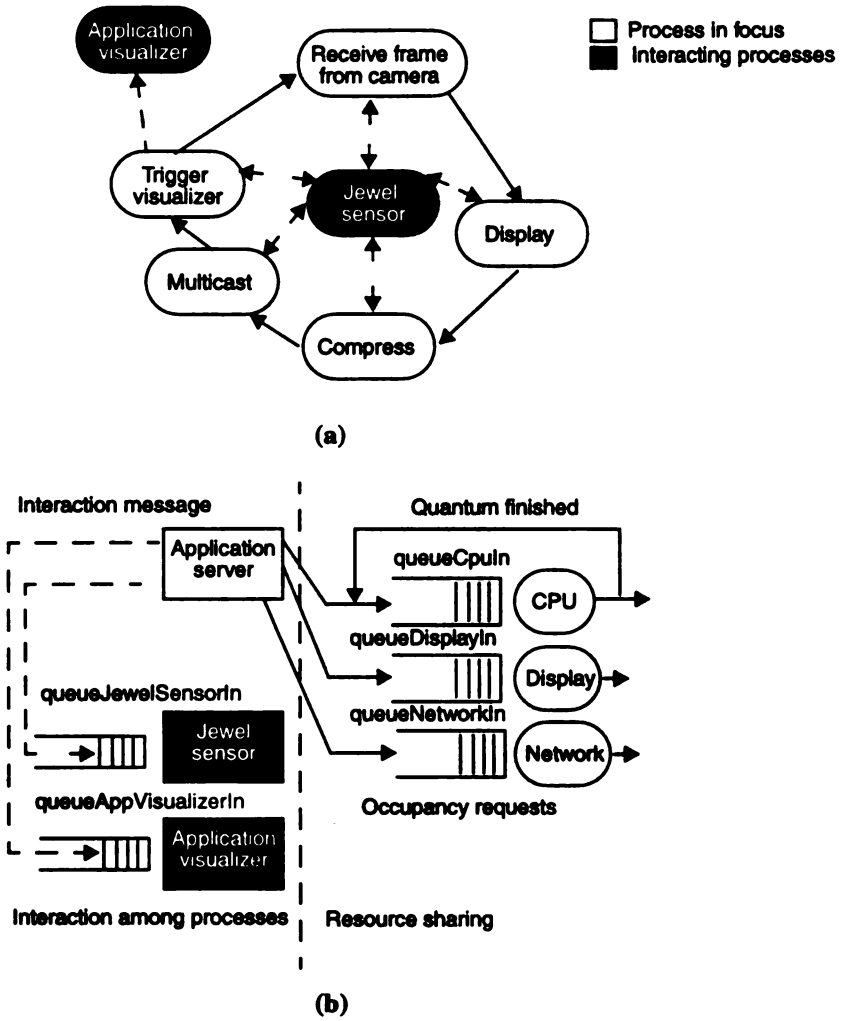


Figure 5-17. Characterization of the server process of the application. (a) Process behavior and (b) ROCC model of the server and other interacting processes.

theoretical *pdf*. Thus, a larger value of the K-S statistic for a particular distribution indicates a poor fit. See Law and Kelton [116] for further details about K-S goodness of fit test.

The K-S statistic has minimum value for the normal *pdf*; thus the normal pdf is a relatively closer match to the measured data compared to the weibull and exponential *pdf*s. Therefore, we characterize the resource occupancy requirements of the server process states using a normal distribution whose parameters are determined from the measured data using *maximum-likelihood estimators*.

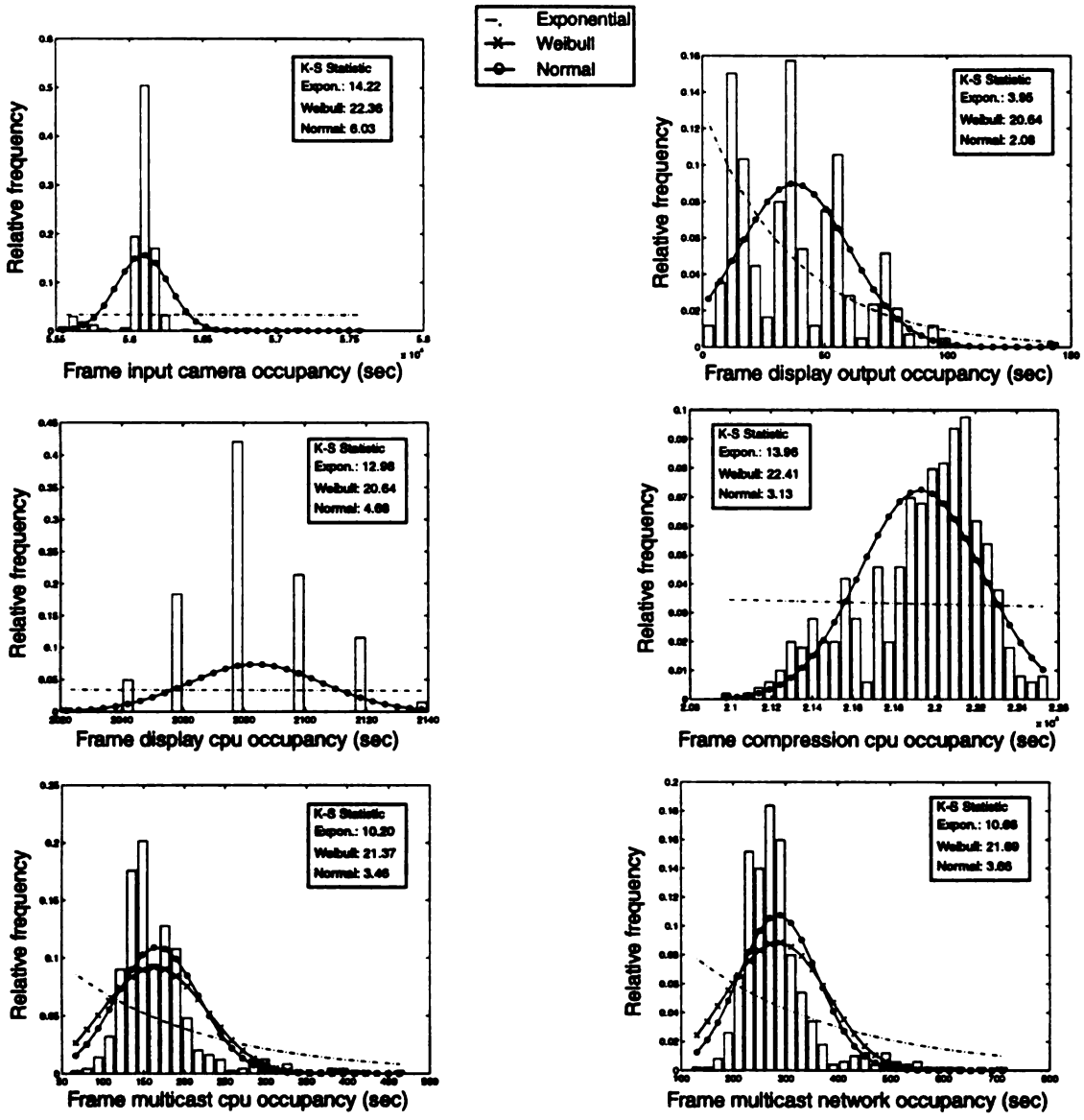


Figure 5-18. Histograms and theoretical probability distribution function for CPU, network, and I/O occupancy time for frame input, frame display, frame compression, and frame multicast states of the server process.

Client Processes: The behavior of the client processes differ from the server. A client process waits for the arrival of a multicast frame from the network, uncompresses the frame, and displays it. Figure 5-19 shows the behavior of a client of the video application as well as its ROCC model that illustrates its interactions with the local Jewel sensor and visualizer processes. We again fit theoretical *pdf*s to the measured resource occupancy requests from various states of the client. The display state has the same resource occupancy behavior as in the case of server process. Histograms and *pdf*s corresponding to

the occupancy requests in frame receive and uncompress states are shown in Figure 5-20. As in the case of the server process, the normal distribution closely fits the measured occupancy requests.

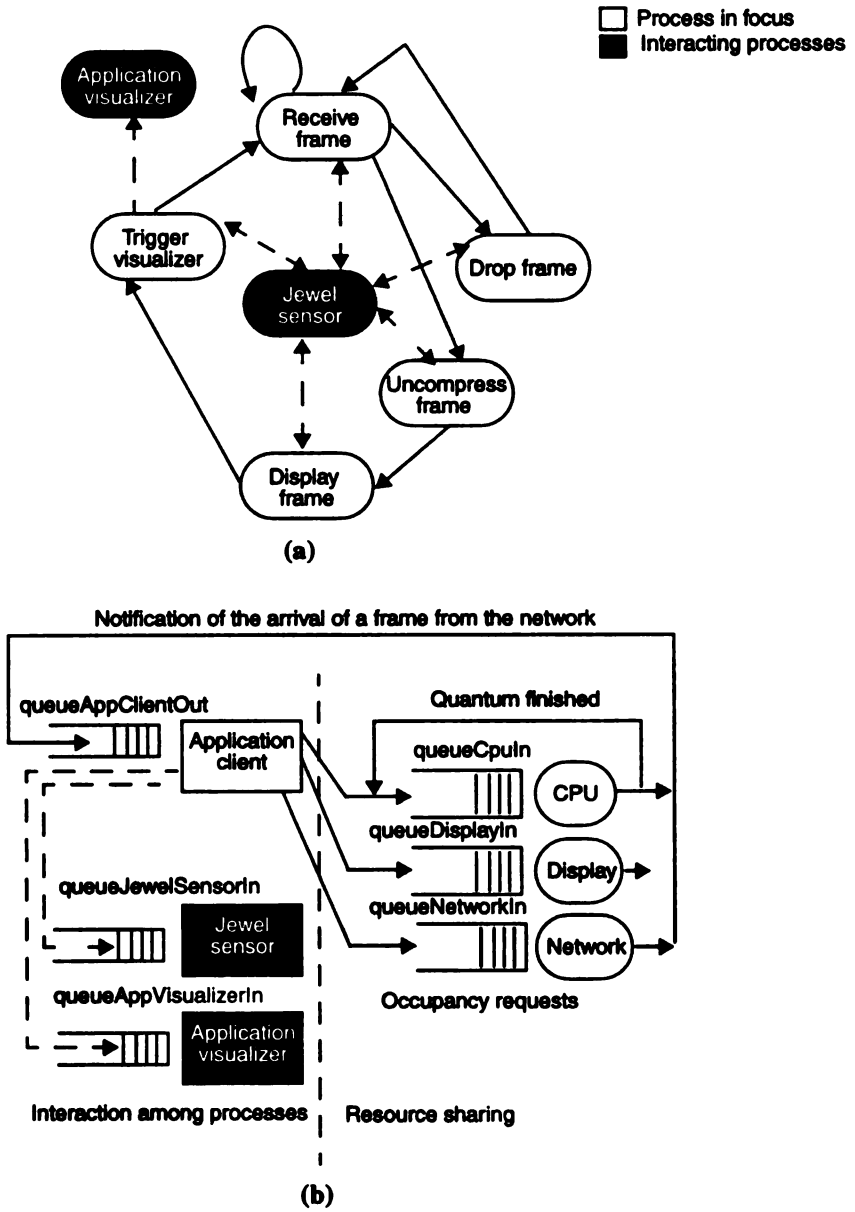


Figure 5-19. Characterization of a client process. (a) Behavior of a client process and (b) ROCC model for the client and Jewel sensor and visualizer processes that interact with it.

Behavior of other application and IS processes is similarly characterized. Distributions of the lengths of resource occupancy requests from these processes are used to parameterize the ROCC model for the JEWEL IS.

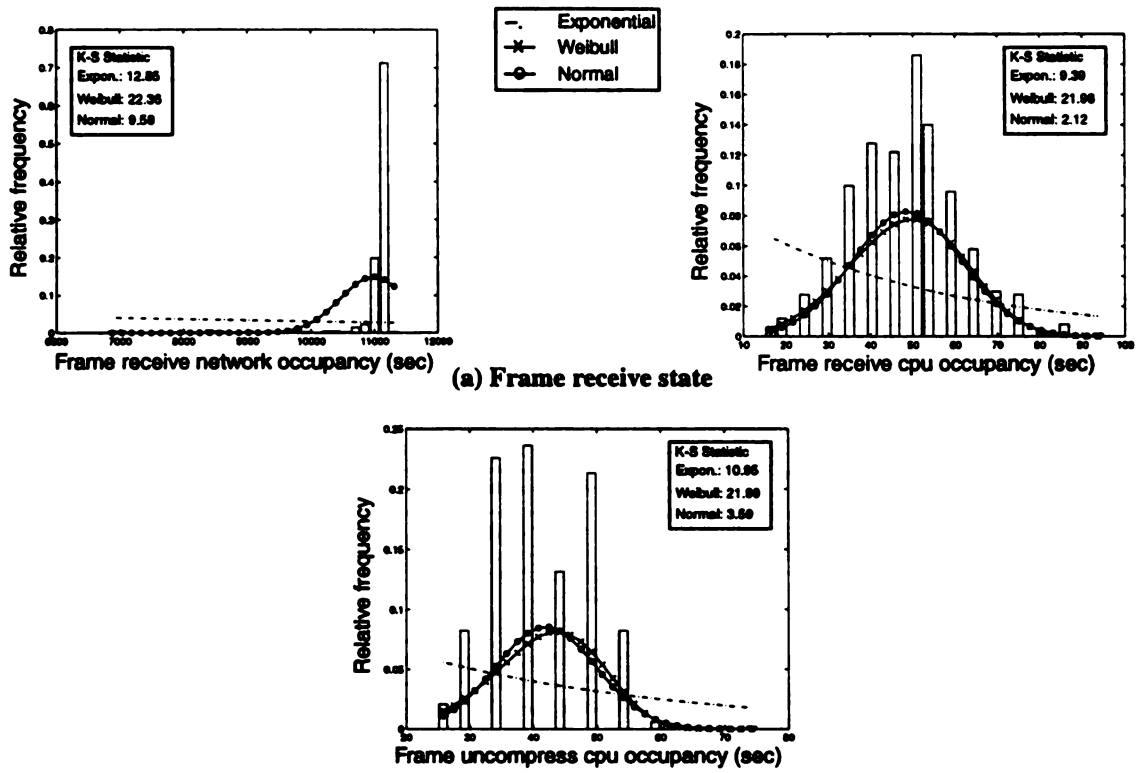


Figure 5-20. Histograms and theoretical probability distribution function for CPU and network occupancy times for (a) frame receive and (b) frame uncompress states of a client process.

5.5.3.5 Model Parameterization

The workload characterization presented in the preceding subsection yields parameters for the ROCC model for the Jewel IS, as shown in Table 5-3. These parameters were calculated using maximum likelihood estimators given by Law and Kelton [116]. Note that the standard deviation for parameters that have normal distribution is actually less than 1 μsec in most cases; however, we use a value of 1 μsec as it is the smallest time unit for this model.

5.5.3.6 Performance Metrics

Three types of metrics are of interest for the evaluation purposes of this study: (1) quality of service (QoS) metrics of the real-time video conferencing application; (2) IS metrics

Table 5-6. Summary of parameters used in simulation of the ROCC model for a Sun Ultra-1 platform. All time parameters, including resource occupancy requirements, are in microseconds.

Parameter Type	Parameter	Range of Values
System configuration	Number of nodes	1–6 (typical 6)
	CPU scheduling quantum (microseconds)	10,000
	Network (ethernet) bandwidth	100 Mbits/sec
	Ring buffer polling period	1-1,000,000 (typical 1000)
	Resource manager and agent sampling periods	1—1,000,000 (typical 1000)
	Hysteresis time	Same as sampling period
Application client process	CPU occupancy requirement for receiving a frame	Normal (49,1)
	CPU occupancy for uncompressing a frame	Normal (42,1)
	CPU occupancy for displaying a frame	Normal (2100,1)
	CPU occupancy for triggering the visualizer	Normal (100,1)
	CPU occupancy for dropping a frame	Normal (10,1)
	X server (display) occupancy for displaying a frame	Normal (38,1)
Application server process	Video hardware occupancy for capturing a frame through the camera	Normal (178,1)
	CPU occupancy for compressing a frame	Normal(21900,1)
	CPU occupancy for multicasting a frame	Normal (167,1)
Jewel sensor and collector processes	CPU occupancy for sampling shared memory by the external sensor	Normal (267,1)
	CPU occupancy for receiving a trace record by the collector	Lognormal (7,1)
	CPU occupancy by Jewel sensor and collector to forward data to collector or resource manager, respectively	Lognormal (200,20)

related to its performance and direct overhead to the application; and (3) adaptive control system performance metrics. These metrics are summarized and interpreted in Table 5-7.

The QoS metrics are related to the real-time behavior of the application. A high quality of video requires smoothness of changes in a dynamic scene being multicast to multiple clients from the server. For the video application, QoS depends on the rate at which a client processes a frame. Processing of a frame involves receiving the frame by a client, uncompressing it, and finally displaying it. We term this metric as *client frame rate*. Frame rate is required to be fixed at 30 frames per second for desired QoS. A related metric is the *client CPU utilization*, which indirectly affects the frame rate; the frame rate can decrease if the application client cannot get enough CPU time due to contention with IS processes.

Table 5-7. Metrics for evaluation of JEWEL IS and its adaptive control system.

Metric Type	Metric	Interpretation
Real-time application QoS metrics	Client frame rate (frames/sec)	Application frame rate is required to be fixed at 30 frames/sec for desired quality of the video.
	Client CPU utilization (percent of total CPU time)	A desirable value is one that results in 30 frames/sec rate.
IS overhead and performance metrics	Jewel sensor CPU utilization (%)	A lower value is desirable.
	Monitoring latency (sec)	A lower value is desirable for an MDR to be useful for real-time control.
	Hold-back ratio	A lower value is desirable and indicates small congestion in the IS.
	Number of lost trace records	A lower value (typically zero) indicates the reliability of the IS.
Adaptive control system performance metrics	Mean-squared error of adaptation with respect to required frame rate	A lower value means better tracking of the frame rate by the controller.
	Mean-squared error of adaptation with respect to sensor CPU utilization limit	A lower value implies that controller can keep the IS overhead close to the desired value.

We define four metrics related to the IS overhead and performance: JEWEL sensor CPU utilization, monitoring latency, hold-back ration, and the number of lost records. A higher value of *JEWEL sensor CPU utilization* implies higher overhead to the application processes; therefore, a lower value of this metric is desirable. *Monitoring latency*, as defined for Paradyn IS, is the period of time between generating an event record by the internal sensor until it is received by the JEWEL collector. A lower value is desirable because a longer latency means that the observed information may no longer be useful for the resource manager for control purposes. *Hold-back ratio* is defined as the number of MDRs enroute to the collector to the sum of event records generated by all the distributed internal sensors. This metric was originally defined and used by Gu et al. in the context of out-of-order arrivals at the ISM of Falcon IS [71]; however, we use this metric to quantify the number of queued-up MDRs at different locations in the JEWEL IS. We also explicitly use the number of received MDRs. The n*umber of lost trace records* indicates the event records that an internal sensor could not generate due to a full shared memory ring buffer.

In addition to QoS and IS performance metrics, we also define two performance metrics for the adaptive controller based on mean-squared error of adaptation. Mean-square error is a well-known metric used in systems theory for quantifying the adaptation errors for different types systems [156,175,180,201,220,222,232]. *Mean-squared error of adaptation with respect to the required frame rate* is defined as the sum of squared errors between desired and actual frame rates at observation epochs (determined by adaptive controller sampling period), averaged over the entire execution time. Similarly, *mean-squared error of adaptation with respect to sensor CPU utilization limit* is defined as the sum of squared errors between the upper bound on allowable sensor CPU utilization and its actual values at observation epochs, averaged over the entire execution time.

In this chapter, we presented a model, workload characterization, and a set of appropriate metrics for each of the three reference ISs. In the next chapter, we use these models and metrics to evaluate the reference ISs to address specific objectives in each case.

Chapter 6

Instrumentation System Evaluation

We present the evaluation results of three reference instrumentation system in this chapter. Our focus is more on the individual case studies than the overall IS modeling-based evaluation methodology. Given the application-specific nature of computer system performance evaluation, it is necessary to evaluate the PICL, Paradyn, and JEWEL ISs with all the relevant domain-specific details. Nevertheless, we adopt a consistent evaluation process (with differences in the techniques used) for the three case studies based on the reference ISs.

Section 6.1 presents a general perspective on evaluating an instrumentation system model. We present the PICL, Paradyn, and JEWEL ISs in Sections 6.2, 6.3, and 6.4, respectively. We conclude with a summary of IS evaluation results.

6.1 Evaluating a System Model

After modeling an instrumentation system, the next step is to use that model for determining values of the metrics of interest under a given set of operating conditions. In general, we can select one of two possible approaches of evaluating a model: analytical and simulation-based. An analytical approach extensively uses mathematical and statistical tools such as queuing theory, Markov processes, renewal theory, operations analysis, and so on to present the metrics as closed-form mathematical expressions; these expressions are functions of the system parameters. A simulator uses statistical input and algorithmically exhibits the same behavior that the system is supposed to exhibit under that input; the metric values can be calculated by examining the current and past (simulated) system state information.

Both analytical and simulation-based model evaluation techniques have their advantages and drawbacks. Typically, analytical models can provide accurate results under a number of simplifying assumptions to remain mathematically tractable. However, the simplifying assumptions may make the model far removed from the actual system for practical systems of even moderate complexity. Therefore, the analytical evaluation approaches provide only "back-of-the-envelope" calculations of the metrics that the system should exhibit under restrictive operating conditions. These calculations are useful when the actual system is not developed and any feedback about its behavior (even under restrictive assumptions) is helpful for the developers. A simulation-based study can incorporate minute details of system structure and behavior to calculate accurate results under more realistic operating conditions. Moreover, interdependence of different processes and their contention for the shared system resources can be captured adequately, using simulation techniques. Due to the use of a random number generators, simulation results usually exhibit high variance and require large number of independent experiments to bring the mean value of the metrics within a "tight" confidence interval. In this chapter, we use analytical results as approximate, back-of-the-envelope calculations to discern the gross behavior of the instrumentation system under study; simulation-based evaluation will be used for relatively more accurate results.

The primary goal of evaluating an IS model is to answer "what-if" questions regarding the system behavior. These answers can guide developers to choose appropriate configurations and management policies before actually implementing them. Not only this early feedback to the developers avoids later upgrades of the IS, it also helps developing ISs that can meet their domain-specific requirements. Meeting of domain-specific requirements is critical for a number of systems, such as distributed real-time systems.

In order to effectively use simulation-based evaluation of the models of reference ISs, we emphasize on careful experimental design. One useful approach is $2^k r$ experimental design approach, where k is number of IS factors (variable system parameters) and r is the

number of repetitions of an experiment to calculate a metric [100]. For the simulation results presented in this chapter, we used a value of r=50, i.e., each metric value in a simulation-based evaluation is a mean of the results of fifty independent experiments and falls within 90% confidence interval of the mean.

Before analyzing the IS behavior with respect to various factors, it is essential to know the sensitivity of selected IS performance- or intrusion-related metrics to these factors. Moreover, it is not correct to assume that all of the factors act independently on the system under test (i.e., the IS). We use *principal component analysis* technique to provide insight into the relative importance of individual factors, as well as their interactions to affect the performance metrics of interest.

6.2 Evaluation of the PICL IS

In this section, we present analytical calculations of the metrics of interest for the PICL IS, based on the model developed in Section 5.5.1. We also present simulation-based results and a discussion of the trade-off between the FOF and FAOF management policies for the PICL IS.

6.2.1 Analytic Calculations

Analytical calculations of PICL IS metrics are based on probabilistic calculations. Although, trace data buffers at every node can be treated as single-server queues, we do not commit to any particular queuing system. Instead, our approach is more generic as it uses well-known results from the renewal theory [166].

6.2.1.1 Definitions and Preliminary Results

In this subsection, we present notations and derive preliminary mathematical results based on the IS model presented in Figure 5-6. Preliminary results will be used to characterize

and compare the FOF and FAOF management policies. We begin with the definition of the repetitive nature of filling and flushing of the local trace data buffers. This process repeats after every flush of one or all the buffers under either of the two management policies.

Definition (Flushing Cycle)

Let t_n and t_{n+1} be the instants of time immediately after a trace data buffer is flushed for the n-th and n+1-st times, respectively. Then $t_{n+1}-t_n$ is the duration of time that constitutes n+1-st *flushing cycle*. ◆

We observe that the time until a local buffer fills up in a particular flushing cycle is a *stopping time* which is defined as *trace stopping time* under the FOF policy in the following.

Definition Definition (Trace Stopping Time)

We define the *trace stopping time* for the i-th local buffer during an n-th flushing cycle when it fills to its capacity l as:

$$\tau_l^{(i)} = inf\{t:\ Q_i(t)=l\} \tag{6.1}$$

where *inf* operator specifies the minimum time $t \in (t_{n-1}, t_n]$ until the local buffer i becomes full for $i \in [0, P\text{-}1]$. ◆

The buffer filling process that starts with an empty buffer and ends with a completely filled buffer at the trace stopping time is represented in Figure 6-1. In the following, we establish that the trace stopping time for any buffer i has an l-Erlang distribution.

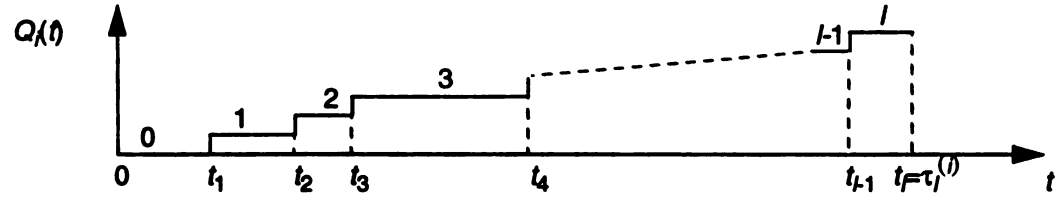


Figure 6-1. Arrivals of trace records at a local buffer in the concurrent system.

Let $Q_i(t)$ be the number of records in the i-th buffer at the i-th node in the concurrent system. Figure 6-1 shows the arrival epochs of each new record and the number of records

in the buffer at those times. Let $N_i(0,t]$ be a point process that counts the number of trace record arrivals in the local buffer from the start of the tracing at time t=0 to any other time t during the traced execution of a program. Since the inter-arrival times $\{t_n - t_{n-1}: n \geq 1\}$ are assumed exponential and the number of arrivals in disjoint intervals are independent, $N_i(0,t]$ is a homogeneous poisson process with mean measure αt. Therefore, the distribution of the trace stopping time is given by:

$$P[\tau_l^{(i)} \leq t] = P[N_i(0,t]=l] = e^{-\alpha t}\frac{(\alpha t)^l}{l!} \quad .$$

This is an l-Erlang distribution because $\tau_l^{(i)}$ is the sum of exponential inter-arrival times of individual trace records in the i-th buffer, and thus

$$P[\tau_l^{(i)} \leq t] = e^{-\alpha t}\frac{(\alpha t)^l}{\Gamma(l+1)} \quad . \tag{6.2}$$

It is useful to calculate the expected trace stopping time. Since the inter-arrival times are independent and exponentially distributed, the expected time till the arrival of any subsequent record is $1/\alpha$. The capacity of a local buffer can reach l records, which is a constant. Then the expected time until an empty buffer fills up can be determined by applying Wald's identity [resnick] and is given by:

$$E[\tau_l^{(i)}] = E[N_i(0,t_l]] \cdot E[t_1 - t_0] = l \cdot \frac{1}{\alpha}. \tag{6.3}$$

Additionally, we consider the trace stopping time from the perspective of the concurrent IS on all P nodes. This is needed to evaluate the FAOF policy. Therefore, we define the global trace stopping time in the following.

Definition (Global Trace Stopping Time)

The *global trace stopping time* is the time during the n-th flushing cycle at which any one of the P buffers becomes full and is given by:

$$\tau_l = min\ \{\tau_l^{(0)}, \tau_l^{(1)}, \ldots\ldots, \tau_l^{(P-1)}\}\ .\blacklozenge \tag{6.4}$$

According to this definition, the global trace stopping time is the minimum gamma order statistics of P gamma (l-Erlang) random variables. The following probabilistic calculations provides insight into the nature of the distribution of the global trace stopping time.

Consider an IS consisting of P trace buffers where trace records arrive with independent and exponentially distributed inter-arrival times, and an arrival process at one buffer can be considered independent of that at another buffer. Using the definition of the global trace stopping time, it can be noted that

$$P[\tau_l > t] = P[\tau_l^{(0)} > t, \tau_l^{(1)} > t, \ldots, \tau_l^{(P-1)} > t]$$

and using the independence of the buffer sizes at all the local buffers

$$P[\tau_l > t] = P[\tau_l^{(0)} > t]P[\tau_l^{(1)} > t]\ldots P[\tau_l^{(P-1)} > t] = [1 - P[\tau_l^{(i)} \le t]]^P$$

$$P[\tau_l > t] = \left[1 - e^{-\alpha t}\frac{(\alpha t)^l}{l!}\right]^P\ .$$

Generally, after determining the distribution of the global trace stopping time, its expected value can be calculated as:

$$E[\tau_l] = \int_0^\infty (P[\tau_l > t])dt = \int_0^\infty \left[1 - e^{-\alpha t}\frac{(\alpha t)^l}{l!}\right]^P dt\ , \tag{6.5}$$

which is not easily solved for explicitly. There is another possible approach to calculate the expected global stopping time. Global trace stopping time is the minimum of P of the gamma random variables for P local buffers. Gupta [72] has derived and tabulated the first four moments of this type of gamma order statistics. In this case, we are interested in the

first moment of the smallest order gamma statistics, which is specified by the recursion relations given in [11]. Calculations of these have been tabulated in [19]. However, there are restrictions on l and P to be small, which prohibits calculating the expected global trace stopping times for long buffers on massively parallel processing (MPP) systems where P could be large. Therefore, we decided to evaluate the bounds on the global trace stopping time.

Let P be the total number of trace buffers in an IS, each having a capacity of holding l records, where trace records arrive with exponential (α) inter-arrival times. Let $\alpha_0=\alpha_1=\ldots=\alpha_{P-1}=\alpha$ be the parameters of the exponential inter-arrival times at P local trace buffers. Let α_{min} be the parameter associated with the first arrival in any of the P buffers, which is given by:

$$\alpha_{min} = \alpha_0 + \alpha_1 + \ldots + \alpha_{P-1} = P\alpha$$

as shown by [164]. Then the time until there are l distinct trace record arrivals in the whole IS, regardless of the number of arrivals in individual trace buffers, is $l/P\alpha$ according to Wald's identity. Clearly, the global trace stopping time is at least equal to $l/P\alpha$, i.e.,

$$E[\tau_l] \geq \frac{l}{P\alpha},$$

which represents the case that all the trace records arrive at a single buffer. On the other hand, if there is only one node in the system (i.e., P=1) then the global trace stopping time cannot be larger than the local trace stopping time, i.e.,

$$E[\tau_l] \leq \frac{l}{\alpha}$$

Therefore, it can be concluded that:

$$\frac{l}{P\alpha} \le E[\tau_l] \le \frac{l}{\alpha}. \tag{6.6}$$

The result shown by equation (6.6) is useful in practice because it shows that the global trace stopping time can not be more the trace stopping time observed at an individual node of the system.

In both the FOF and FAOF policies whenever one or all the buffers have been flushed, the IS experiences the same buffer filling and flushing *cycle*. We consider the successive flushes of a local buffer (under either of the two policies) as a renewal process, as shown in the following.

Dynamic flushing of the trace data buffer constitutes a *regenerative stochastic process* $\{Q_i(t)\}$ for the length of any buffer in the IS. In order to justify this, we consider $\{S_n: n \ge 0\}$ to be the starting times of new service periods at the i-th buffer, i.e., the buffer starts to refill from $Q_i(S_n)=0$ to $Q_i(S_{n+1})=l$, as shown by Figure 6-2. Clearly, $N_i(S_n,S_{n+1}]=l$. Further, it can be noticed that: (1) $\{S_n\}$ is a renewal process as inter-arrival times are independent and identically (exponentially) distributed; (2) for $0<t_1<t_2<...<t_k$, $(Q_i(S_n+t_1),Q_i(S_n+t_2),...,$ $Q_i(S_n+t_k)) \sim (Q_i(t_1), Q_i(t_2),..., Q_i(t_k))$ for $n \ge 0$ and $k \ge 1$, where the processes on either side of $\sim$ symbol are equal in their joint distributions; and (3) $\{Q_i(S_n+t)\}$ is independent of $\{S_1,S_2,..., S_n\}$. Hence, $\{Q_i(t): t \in (S_n,S_{n+1}]\}$ is a regenerative process.

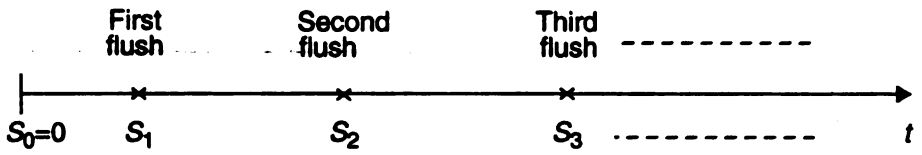


Figure 6-2. Regenerative process of buffer fillings and flushings.

The regenerative process of the buffer filling and flushing repeats in *cycles*. After the buffer is filled, it gets flushed. Note that the time for buffer flushing has a different distribution than the exponentially distributed inter-arrival times of the trace records. The

buffer flushing time distribution does not affect the regenerative nature of the overall process, as $\{Q_i(t)\}$ alternates between two regenerative processes. Therefore, the regenerative process of buffer filling and flushing repeats with the same probabilistic structure in each subsequent cycle. These cycles continue till the end of the instrumented execution of the program.

6.2.1.2 Comparison of the Management Policies

In this subsection, we compare the FOF and FAOF policies in terms of flushing frequency under each of the two policies. We begin with the definition of the *flushing frequency.*

Definition (Flushing Frequency)

Let $N_{flushes}$ be the number of flushes recorded at a local buffer (under FOF policy) or all buffers (under FAOF policy) during program execution. Let N_{total} be the count of all the trace records that arrived into the local buffer(s) during the entire execution of the program. Flushing frequency of a particular local trace data buffer (under the FOF policy) or all the buffers (under the FAOF policy), denoted ω_o and ω_a, respectively, is defined as:

$$\omega_o \text{ or } \omega_a = \frac{N_{flushes}}{N_{total}}. \blacklozenge \tag{6.7}$$

In the following, we calculate the values of flushing frequency under the FOF and FAOF policies. The flushing process starts immediately after the buffer has been filled to its capacity. and is responsible for transferring all the trace records to the main buffer at the host of the concurrent system. This transfer is accomplished through the (wormhole-routed) direct network. An I/O node transfers the data to an appropriate host system entity, for instance, a larger buffer in the main memory of the host or a disk that instantiates the main buffer. We assume that the time for flushing the local buffer is time required to transfer all the trace records in that buffer to an I/O node. The subsequent transfer from the I/O node to the host system does not add any latency to the local buffer flushing facility.

Let $f(l)$ be a function of the capacity of the buffer that represents the time of transferring the local buffer to the I/O node through the wormhole-routed direct network. It has been shown by [ni] that $f(l)$ can be approximated by Cl/B where B is the bandwidth of the channels of the direct network and C is the average number of bytes per trace record. Therefore, Cl is the amount of data to be transferred. Clearly, the average service time after the arrival of l trace records at the local buffer is equal to the latency of transferring this trace data. Therefore, the mean service time is given by: $b = f(l) = C_1 l + C_2$, where $C_1 = C/B$ and C_2 accounts for the start-up latencies (fixed overhead of calling a system function for sending and receiving a message) and the path establishment time between the local node and the I/O node.

The FOF Policy: We prove that the long-term buffer flushing frequency ω_o under an FOF management policy of a trace data buffer is the ratio of mean inter-arrival time of a trace record to the expected time to complete one flushing cycle of that buffer. Let $[Q_i(t)=0]$ be the event that the i-th buffer has been flushed and the process has been regenerated to proceed through this *cycle* once again, where $\{Q_i(t): t \in (S_n, S_{n+1}]\}$ is the $n+1$-st flushing cycle. The buffer filling process $\{Q_i(t): t \geq 0\}$ is a regenerative process with renewal times $\{S_n: n \geq 0\}$. We can define a state j of the buffer corresponding to j records in the buffer, such that: $q_j(t) = P[Q_i(t)=j, S_{n+1} > t]$. If $\mu_o = E$[cycle length], it can be given by:

$$\mu_o = E[\tau_l^{(i)}] + f(l) = \frac{l}{\alpha} + C_1 \cdot l + C_2$$

and clearly $\mu < \infty$. Then under mild regularity conditions, *Smith's theorem* [164] shows that:

$$\lim_{t \to \infty} P[Q_i(t)=j] = \frac{E[\text{occupation time of state } j \text{ in a cycle}]}{\mu_o}.$$

The long-term number of visits to state $j=l$ are equal to the number of visits to state $j=0$ (in the next cycle). However, the latter will be used here as the distribution of occupation time

of state $j=l$ is different from the (exponential) distribution of the occupation time of the rest of the states. When $j=0$, the buffer has been flushed and the process has been regenerated, and the expected time of occupancy of this state is equal to the expected time until the first trace record arrives to the flushed buffer. Therefore, E[occupation time of $j=0$ state]$=1/\alpha$. Hence, the long-term flushing frequency is

$$\omega_o = \lim_{t \to \infty} P[Q_i(t)=0] = \frac{1}{\alpha \cdot \mu_o} = \frac{1}{l + \alpha C_1 l + \alpha C_2} = \frac{1}{l + \alpha f(l)} \tag{6.8}$$

where $1/\alpha$ is the mean inter-arrival time of a trace record. When we let $l \to \infty$, then equation (6.8) implies that $\omega_o = \lim_{t \to \infty} P[Q_i(t)=0] = 0$, which is merely a proof of the intuitive fact that a buffer having an infinitely large capacity will never have to be flushed.

The FAOF Policy: Under the FAOF policy, whenever one of the P local buffers is filled to its capacity, all the buffers are flushed to the main buffer. This policy is implemented by some ISs, such as Pablo [161] and TAM [165]. Implementation of this policy requires synchronization of individual processes and gang scheduling of the flushing operation on all nodes, which is not trivial on a loosely coupled distributed-memory parallel system.

The process $\{Q(t)\}$ under the FAOF policy is a regenerative process, which is analogous to the regenerative process $\{Q_i(t)\}$ under the FOF policy. After flushing all the local buffers, the service process waits until one of the buffers becomes full. Then it again flushes all the local buffers. This sequence of events constitutes one *flushing cycle* under the FAOF policy that repeats until the program execution ends. Under the FAOF policy, the maximum long-term frequency of flushing all the local buffers, ω_a, is less than or equal to the long-term flushing frequency under the FOF policy. This can be proved by applying Smith's theorem as in the case of FOF policy, except that the exact value of μ_a can not be calculated. Since $\mu_a \leq l/(P\alpha)$, therefore:

$$\omega_a \leq \lim_{t \to \infty} P[Q(t)=0] = \frac{1}{P\alpha\mu_a} = \frac{1}{l + P\alpha f(l)}, \tag{6.9}$$

which is an upper bound on the long-term frequency of flushing all the buffers. When P=1, $\mu_a=\mu_o$ and right hand side of the equation (6.9) becomes equal to ω_o. Therefore $\omega_a \leq \omega_o$. When we let $l \to \infty$, then equation (6.9) implies that $\omega_a \leq \lim_{t \to \infty} P[Q(t)=0] = 0$. In the worst case, the flushing frequency under the FAOF policy can become equal to the flushing frequency of an individual buffer under the FOF policy. It proves that the FAOF policy is desirable based on the flushing frequency metric.

6.2.1.3 Summary of IS Management Policies

Table 6-1 summarizes the analytical results that were derived in Section 6.2.1.2. These results compare the FOF and FAOF policies using expected buffer filling time (trace stopping time) and flushing frequency for a given arrival rate. Note that the analytical results for the FAOF policy (except for its distribution) can be obtained from the corresponding results for the FOF policy by replacing α with $P\alpha$. In order to compare the FOF and FAOF policies on the basis of analytical results for flushing frequency and perturbation index metrics, it is important to identify the dominant factor between α (or $P\alpha$) and the time for flushing ($f(l)$).

Table 6-1. Summary of management policies.

Performance Metric	FOF Policy	FAOF Policy
Distribution	$P[\tau_l^{(i)} \leq t] = e^{-\alpha t}\frac{(\alpha t)^l}{\Gamma(l+1)}$	$P[\tau_l > t] = \left[1 - e^{-\alpha t}\frac{(\alpha t)^l}{l!}\right]^P$
Expected trace stopping time	$E[\tau_l^{(i)}] = l \cdot \frac{1}{\alpha}$	$\frac{l}{P\alpha} \leq E[\tau_l] \leq \frac{l}{\alpha}$
Time for flushing	$f(l) = C_1 l + C_2$	$f(l) = C_1 l + C_2$
Long-term flushing frequency	$\omega_o = \frac{1}{l + \alpha f(l)}$	$\omega_a \leq \frac{1}{l + P\alpha f(l)}$

6.2.2 Simulation-Based Experiments

In this subsection, we compare the FOF and FAOF management policies based on simulating the PICL IS model. We begin with the description of the experimental setup for simulations, followed by the principal component analysis of selected factors and investigation of the PICL IS behavior under the two management policies.

6.2.2.1 Experimental Setup

We developed a simulator for the nCUBE-2 multicomputer that uses exponentially distributed inter-arrival times. The FAOF policy is modeled by using P independent streams of the random number generator to ensure the independence assumption among the local buffer filling processes. All simulations presented here were executed for one second of simulation time with a timer resolution of one microsecond; this allows the buffer filling and flushing processes to run for a sufficient length of time. The communication latency on an nCUBE-2 multicomputer for flushing a buffer of capacity l records was estimated as:

$$f(l) = 2l + 187$$

where start-up latency is estimated to be 187 microseconds [mckinley].

6.2.2.2 Principal Component Analysis

In order to use 2^k factorial design technique, we need to conduct 8 experiments to obtain the required data needed for the *principal component analysis* (PCA). Table 6-2 shows the results of these experiments. We use the technique outlined by Jain to perform the principal component analysis [100].

Table 6-3 shows the results of the PCA. Number in the parenthesis in the second and third columns indicates the relative rank of the corresponding factor (or a combination of them)

Table 6-2. Results of initial experiments to use principal component analysis for Paradyn IS evaluation.

Buffer length (records)	Inter-arrival Time (microsec)	Flushing policy	Trace stopping time (microsec)	Flushing Frequency
10	100	FOF	997	0.1
1000	100	FOF	99,307	0.001
10	2000	FOF	20,083	0.12
1000	2000	FOF	2,022,762	0.001
10	100	FAOF	526	0.011874
1000	100	FAOF	94,727	0.000066
10	2000	FAOF	10,930	0.011522
1000	2000	FAOF	1,891,962	0.000066

FOF—Flush One when it Fills

FAOF—Flush All when One Fills

in terms of its importance in explaining the variation of the metric. Clearly, a combination of buffer lengths (A) and flushing policy (C) explain most of the variation of flushing frequency metric. On the other hand, a combination of buffer lengths and inter-arrival times explain most of the variability of trace stopping time. Therefore, a further investigation of the behavior of the IS with respect to buffer lengths, inter-arrival times, and flushing policy is justified.

Table 6-3. Results of principal component analysis for PICL IS.

Factors or combination of factors	Variation explained for trace stopping time (%)	Variation explained for flushing frequency (%)
A (buffer length)	37.45 (1)	42.45 (1)
B (inter-arrival time)	31.69 (2)	0.28
C (flushing policy)	0.05	28.72 (2)
AB	30.71 (3)	0.28
AC	0.04	27.66 (3)
BC	0.04	0.3
ABC	0.03	0.3

6.2.2.3 Investigation of Management Policies

In practice, different programs or even different phases of the same program can have variable arrival rates. In order to test the validity of the conclusions drawn from analytical and trace-driven simulation statistics, we carried out several simulations using different arrival rates and buffer lengths to analyze their effects on trace stopping time, flushing frequency, and average number of lost arrivals per cycle. Trace stopping times directly impact the flushing frequency and number of lost arrivals per cycle, therefore, we discuss these results separately.

Trace Stopping Time: Figure 6-3 compares the trace stopping times for the FOF and FAOF policies using three different arrival rates and twenty different buffer sizes.

- The trace stopping times behave consistently with different values of arrival rates under both policies. Stopping times are longer under the FOF policy because the trace data buffer is flushed only when it fills up, whereas under the FAOF policy, a trace data buffer can be flushed even if it is not completely filled (due to the filling of some non-local buffer).
- These results also support the analytical result for trace stopping time, which predicts a linear relationship between buffer length and the trace stopping time.
- Another notable observation is the proximity of trace stopping times under both policies.

If the inter-arrival time of trace records at individual local buffers is exponentially distributed, the trace stopping time for one local buffer under the FOF policy is approximately equal to the trace stopping time for all the buffers under the FAOF policy. This favors the FAOF policy because it implies that if one buffer fills up, then it is highly likely that others will also fill up soon, and flushing all the buffers simultaneously can be justified.

Flushing Frequency: Buffer flushing frequencies under the two management policies are compared in Figure 6-4 for the three arrival rates.

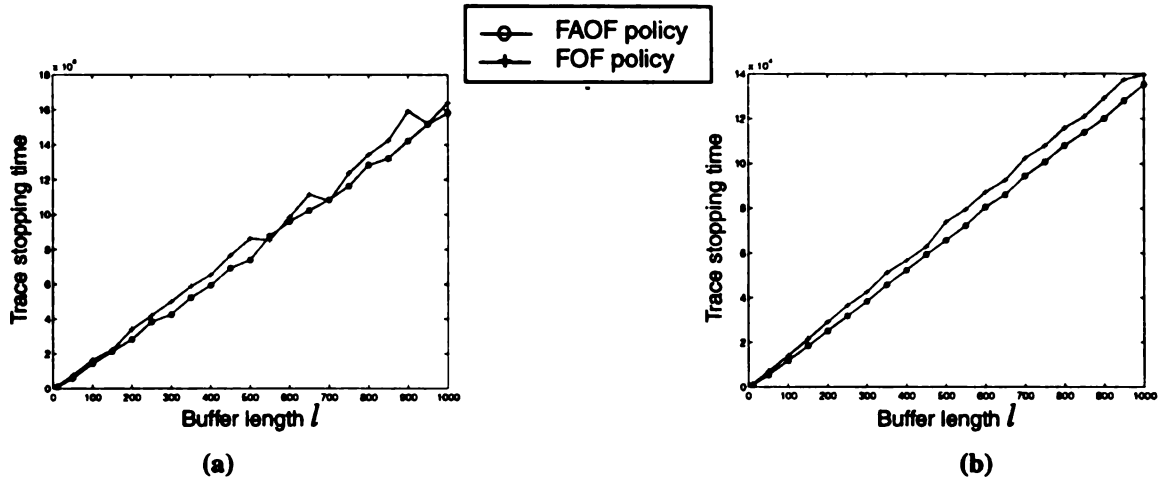


Figure 6-3. Comparison of trace stopping times for the FOF and FAOF policies. Trace stopping time is in microseconds for three arrival rates, (a) α_1=0.00006 and (b) α_2=0.007.

- The flushing frequency also behaves consistently for different arrival rates. Flushing frequency is higher under the FOF policy regardless of arrival rate or buffer length.
- Flushing frequency diminishes very rapidly with small increases in the buffer size at small buffer lengths under either policy.
- The lower flushing frequency under the FAOF translates into less time spent overall on flushing during program execution. Intuitively, the lower flushing frequency can be attributed to the proximity of the trace stopping times.

Hence, flushing all the buffers simultaneously reduces the number of flushes during program execution.

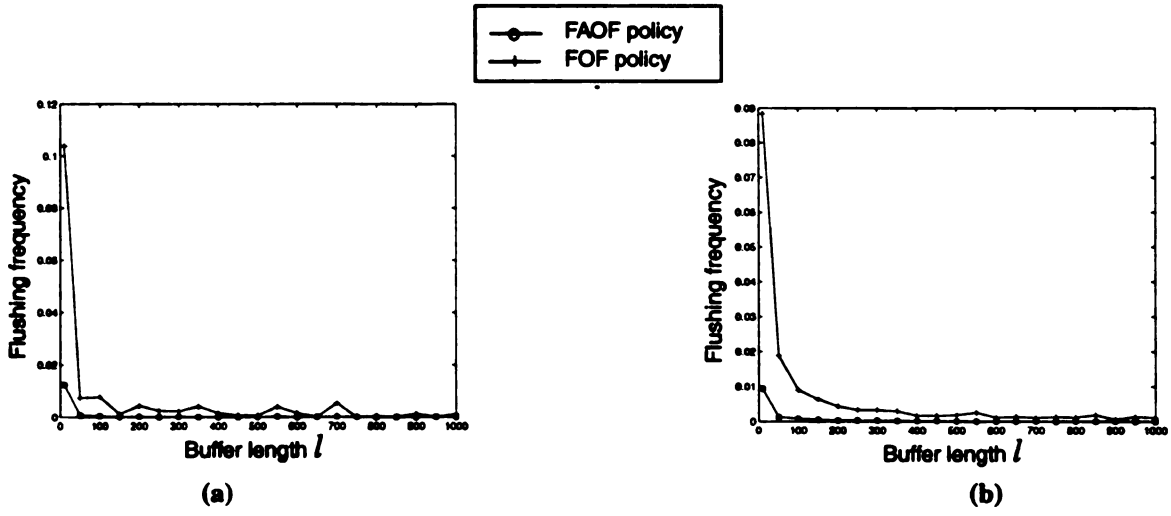


Figure 6-4. Comparison of buffer flushing frequencies of the FOF and FAOF policies. Buffer flushing frequencies are given for three arrival rates, (a) α_1=0.00006 and (b) α_2=0.007.

6.2.3 Feedback to the Developers

Analysis of the two management policies reveal that the FAOF buffer flushing policy has a smaller buffer flushing frequency than the FOF policy. Under the worst case, the flushing frequency for the FAOF policy can become equal to that for the FOF policy. Therefore, the FAOF policy should be considered optimal on the basis of flushing frequency. Although this conclusion is obvious from the analytical and simulation results, the caveat is in terms of the practical implementation of this policy on a multicomputer system. The FAOF policy can be implemented if the IS has the capability of gang scheduling the flushing operation after synchronizing all the nodes. This essentially mean two requirements: (1) the multicomputer operating system needs to be able to run multiple processes per node; and (2) the operating system needs to support synchronization operation and be able to gang schedule the IS processes at all nodes by saving the context of application processes on each node as well as any pending messages over the network. Some multicomputer architectures and operating systems support such facilities (e.g., CM-5) whereas others may not support them (e.g., nCUBE-2).

While selecting a particular buffer flushing policy, the IS developers should consider the perturbation effects of implementing a particular policy. The FOF policy may incur relatively lower overheads because flushing one buffer independently from the others is similar to sending a message from one processor to another processor (an I/O processor, in this case). However, under the FOF, flushing frequency is relatively larger. Independently flushing local trace data buffers during program execution may result in variable delays among processors. The overall effect of these delays can also change the application program behavior in an unexpected manner. However, such perturbation effects are difficult to be specified quantitatively.

6.3 Evaluation of the Paradyn IS

We evaluate the Paradyn IS with respect to three target architectures: Network of Workstations (NOW), Symmetric MultiProcessors (SMP), and Massively Parallel Processing (MPP) systems. For the Paradyn IS, we apply ROCC modeling technique for evaluating alternative design and configuration options. In this section, we begin with analytic calculations for the Paradyn IS model using operations analysis techniques. Then, we present simulation-based evaluation of the ROCC model for the Paradyn IS followed by the recommendations to the developers on the choice of IS design and configuration.

6.3.1 Analytic Calculations

In this section, we present approximate analytical calculations using operations analysis on the ROCC model of the Paradyn IS. The ROCC model is a queueing network that has two workloads of interest to this study at each node of the system: Paradyn daemon's resource occupancy requests to collect and forward samples, and user application requests to execute the application program. On one hand, the ROCC model may be considered an open queuing network for the Paradyn daemon's workload because its requests actually leave the system when a sample is received by the main Paradyn process. Thus, the total number of Paradyn daemon requests in the system can vary with time. On the other hand, the ROCC model may be considered a closed queueing network for the application workload. An application process generates a request and waits for its completion before initiating a new occupancy request for the same or a different resource. Thus, the total number of application requests at a given time is always constant. This scenario is typical of a closed queueing network with a batch workload [100]. Therefore, the overall ROCC model for the Paradyn IS is a mixed queueing network with two workloads, assumed to be independent for analytic calculations. We derive the analytical results for Network of Workstations (NOW), Symmetric MultiProcessors (SMP), and Massively Parallel Processing (MPP) architectures in this subsection.

6.3.1.1 The NOW Architecture

Using (transaction) workload due to Paradyn daemon requests at each node of the system, we first calculate the arrival rate λ of Paradyn daemon requests at each node. It is given as:

$$\lambda = \frac{1}{\text{Sampling period}} \times \frac{1}{\text{Batch size}} \times \#\text{ of application processes per node}. \quad (6.10)$$

This definition of arrival rate makes it sensitive to three of the four system parameters that can vary for this study. The CPU utilization per node due to Paradyn daemon requests follow from the *utilization law* and *forced flow law* [100] as:

$$\mu_{Pd,\,CPU}(\lambda) = \lambda D_{Pd,\,CPU} \quad (6.11)$$

where $D_{Pd,CPU}$ represents the average length of a CPU occupancy request from the Paradyn daemon. In order to calculate the monitoring latency and Paradyn CPU utilization, we calculate the overall Paradyn daemon CPU request throughput of P concurrent nodes. Using flow balance assumption, the throughput of each node is equal to λ. Therefore, overall Paradyn daemon CPU request throughput is given by:

$$X_{Pd}(\lambda) = P\lambda,$$

which is the arrival rate of Paradyn network requests. The network utilization by Paradyn daemon requests is given by:

$$\mu_{Pd,\,Network}(\lambda) = P\lambda D_{Pd,\,Network}. \quad (6.12)$$

The monitoring latency of a sample that reached the main Paradyn process in the form of a CPU request followed by a network request can be defined as a sum of residence times (resource occupancy and queueing time) in two resources. Thus, monitoring latency for a

sample is calculated by using *utilization law* and *Little's law* under the assumption of *flow balance*, to yield:

$$R(\lambda) = \frac{D_{Pd,\,CPU}}{1-\mu_{Pd,\,CPU}(\lambda)} + \frac{D_{Pd,\,Network}}{1-\mu_{Pd,\,Network}(\lambda)}. \tag{6.13}$$

Since we know the overall arrival rate of Paradyn daemon requests to the main Paradyn process (under flow balance assumption), we can calculate the CPU utilization of the main Paradyn process as:

$$\mu_{Paradyn,\,CPU}(\lambda) = P\lambda D_{Paradyn,\,CPU}. \tag{6.14}$$

In order to calculate the application CPU utilization per node, we use (batch) workload with a closed queueing network. We can use mean value analysis (MVA) to solve this model to calculate the throughput of application CPU requests at each node and use this throughput to calculate the CPU utilization (as product of throughput and average CPU occupancy time for an application request). However, there are two problems in using this approach: the resulting application CPU utilization does not vary with any of the system parameters and the calculation does not account for the contention for CPU between Paradyn daemon and application process. Therefore, an MVA based evaluation of application CPU utilization is not useful in this case. We can calculate the application CPU utilization in an indirect way as:

$$\mu_{Application,\,CPU}(\lambda) = 1-\mu_{Pd,\,CPU}(\lambda). \tag{6.15}$$

This approximate calculation for the application CPU utilization per node does not account for the time that the application process spends waiting for its network occupancy request to be serviced. Therefore, the resulting values of the application CPU utilization are expected to be higher than the actual values. Nevertheless, this technique serves the

purpose of providing "back-of-the-envelope" calculations to be used as an intuitive check on the simulation results.

Figure 6-5 plots the results of analytical calculations of the metrics of interest with respect to number of system nodes and sampling rate. The results indicate that the Paradyn daemon CPU overhead does not change with respect to the number of system nodes. However, under the BF forwarding policy, the overhead is significantly lower. This difference between the CF and BF cases is due to the dependence of arrival rate λ on the batch size, as given in equation (6.10). The batch sizes are 1 and 32, respectively, for the CF and BF policies. The larger batch size for the BF policy results in lower overhead. Analytical results with respect to variable number of nodes and sampling periods predict that the BF policy is more desirable as it yields lower CPU overhead and monitoring latency.

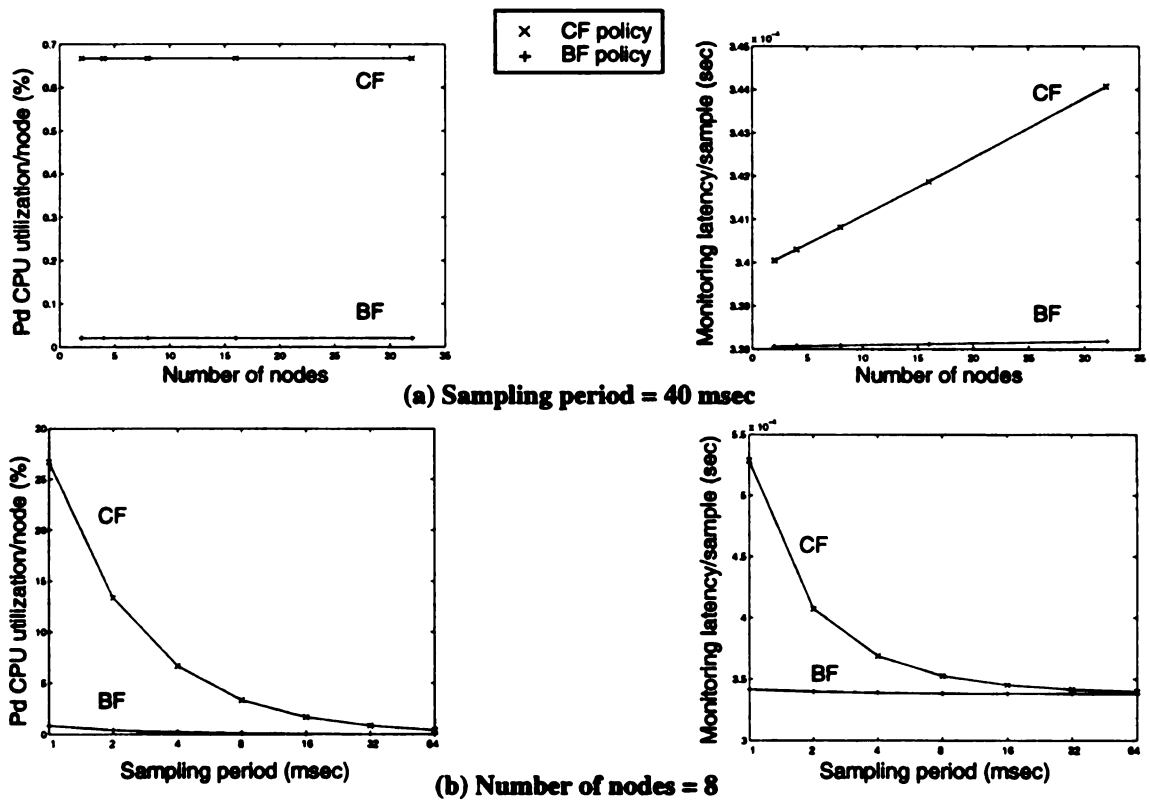


Figure 6-5. Analytic calculations of the effects of varying number of nodes and sampling periods on metrics with respect to CF and BF data forwarding policies (logarithmic horizontal scale in (b)).

6.3.1.2 The SMP Architecture

In the SMP case, the ROCC model is slightly different from the model for the NOW architecture. Instead of multiple nodes having their own CPUs and workloads, the system consists of multiple CPUs that are shared by a set of application processes, one or more Paradyn daemons, and a main Paradyn process. Therefore, the ROCC model has one queue for all the CPUs and one queue corresponding to the system bus.

For the SMP case, we include the factor of multiple Paradyn daemons that may be sharing the system resources into the definition of arrival rate, as given by equation (6.10). Thus, the definition of arrival rate for SMP is:

$$\lambda = \frac{1}{\text{Sampling period}} \times \frac{1}{\text{Batch size}} \times \#\text{ of application processes per node} \times \#\text{ of Pds}$$

The Paradyn daemon, main Paradyn process, overall IS processes, and application process CPU utilization are given by:

$$\mu_{Pd,\,CPU}(\lambda) = \lambda \frac{D_{Pd,\,CPU}}{P}, \tag{6.16}$$

$$\mu_{Paradyn,\,CPU}(\lambda) = \lambda \frac{D_{Paradyn,\,CPU}}{P}, \tag{6.17}$$

$$\mu_{IS,\,CPU}(\lambda) = \frac{(\#\text{ of Pds} \cdot \mu_{Pd,\,CPU}(\lambda)) + \mu_{Paradyn,\,CPU}(\lambda)}{\#\text{ of Pds} + 1}, \text{ and} \tag{6.18}$$

$$\mu_{Application,\,CPU}(\lambda) = 1 - \mu_{IS,\,CPU}(\lambda). \tag{6.19}$$

The bus utilization and monitoring latency are given as:

$$\mu_{Pd,\,Bus}(\lambda) = \lambda D_{Pd,\,Bus} \text{ and} \tag{6.20}$$

$$R(\lambda) = \frac{D_{Pd,CPU}/n}{1-\mu_{Pd,CPU}(\lambda)} + \frac{D_{Pd,Bus}}{1-\mu_{Pd,Bus}(\lambda)}. \quad (6.21)$$

Figure 6-6 plots the results of analytical calculations under the BF policy with respect to sampling periods and multiple Paradyn daemons. Equations (6.18) and (6.21) indicate that IS (i.e., Pd and Paradyn) CPU utilization and monitoring latency metrics depend on the number of Paradyn daemon processes because the arrival rate λ is proportional to the number of Paradyn daemons. Therefore, the analytical results predict that the use of multiple daemons may result in a higher monitoring latency and CPU overhead compared to the single daemon case, but the effects appear to be negligible, especially at larger sampling periods.

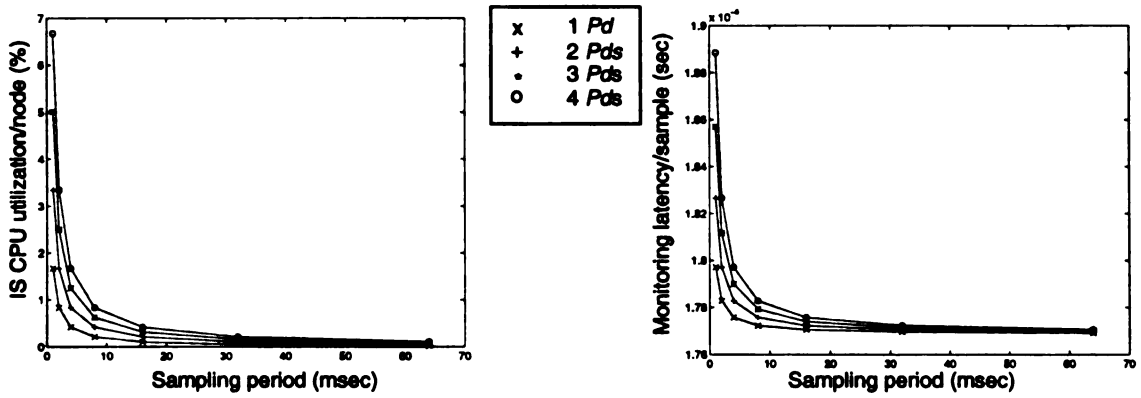


Figure 6-6. Analytical calculations of the effects of multiple Paradyn daemons on two metrics (number of nodes = 16, number of application processes = 32, BF policy). IS CPU utilization represents the combined CPU utilization due to Paradyn daemons and the main Paradyn process.

6.3.1.3 The MPP Architecture

For an MPP system, the ROCC model is the same as depicted in Figure 5-11 with the exception that the shared network is replaced by a direct network. For this architecture, we model and evaluate direct and binary tree forwarding approaches.

The analytical results for the direct forwarding are same as in the case of NOW system, presented by equations (6.10)—(6.15). In case of binary tree forwarding approach, the

Paradyn daemons running at non-leaf nodes perform extra work of collecting the instrumentation data samples from their two children nodes, merging them into single samples, and forwarding them to their parent node. We define the arrival rate of enroute samples that are to be merged as λ_m. If we assume that the total number system nodes P is equal to a multiple of 2, then three are $P/2$ leaf nodes that have $\lambda_m=0$; one less than $n/2$ nodes that have two children and $\lambda_m=2\lambda$; and one node that has only one child and $\lambda_m=\lambda$. The CPU utilizations due to the Paradyn daemon and main Paradyn process under tree forwarding are given by:

$$\mu_{Pd,\,CPU}(\lambda) = \frac{\frac{n}{2}\lambda D_{Pd,\,CPU} + \left(\frac{n}{2}-1\right)(\lambda D_{Pd,\,CPU} + 2\lambda D_{Pdm,\,CPU}) + \lambda D_{Pdm,\,CPU}}{P} \quad (6.22)$$

$$\mu_{Paradyn,\,CPU}(\lambda) = 2\lambda D_{Paradyn,\,CPU}. \quad (6.23)$$

The network utilization and monitoring latency are given as:

$$\mu_{Pd,\,Net}(\lambda) = \frac{\frac{n}{2}\lambda D_{Pd,\,Net} + \left(\frac{n}{2}-1\right)(\lambda D_{Pd,\,CPU} + 2\lambda D_{Pd,\,Net}) + \lambda D_{Pd,\,Net}}{P}, \quad (6.24)$$

$$R(\lambda) = \frac{D_{Pd,\,CPU} + D_{Pdm,\,CPU}}{1-\mu_{Pd,\,CPU}(\lambda)} + \frac{D_{Pd,\,Net}}{1-\mu_{Pd,\,Net}(\lambda)}. \quad (6.25)$$

Note that the network occupancy needed for forwarding a merged sample is the same as for forwarding a local sample.

Figure 6-7 presents the analytical results under the BF policy with respect to the number of nodes in the MPP system. The graph in the middle indicates that tree forwarding has a clear advantage over direct forwarding in terms of lower CPU overhead for the main Paradyn process. Analytical results show that under tree forwarding the CPU overhead due to Paradyn remains unchanged as long as the arrival rate at a node (i.e., λ) is constant. Conversely, this overhead increases linearly with the number of nodes under direct forwarding. The monitoring latency is higher for tree forwarding due to additional arrivals

at non-leaf nodes corresponding to the "enroute" samples. The differences in Paradyn daemon CPU overhead between the two forwarding policies are insignificant (hundredths of a percent).

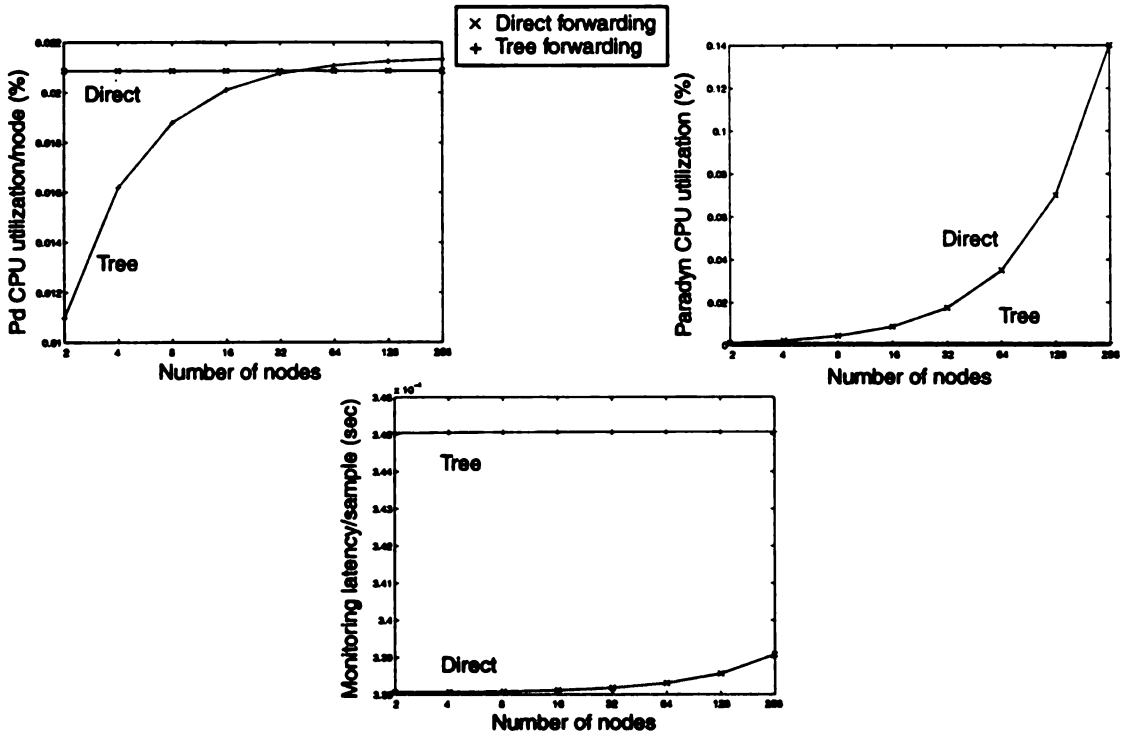


Figure 6-7. Analytical calculations of the effects of varying number of nodes with respect to direct and tree forwarding policies. (sampling period = 40 msec, BF policy, logarithmic horizontal scale)

As we noted earlier, the application and IS workloads actually interact. Because the analytical calculations have a limited scope of providing "back-of-the-envelope" calculations, we did not consider inter-dependences between the workloads in this section. We consider these inter-dependences in the following section on simulation-based experiments.

6.3.1.4 Summary of Analytic Calculations for Paradyn IS

Table 6-4 presents the analytical results for the NOW, SMP, and MPP systems. Based on these approximate calculations, we can predict the following about the IS behavior:

Table 6-4. Summary of analytic results for the ROCC model of Paradyn IS.

System Type	Performance Metric	Analytic Results
NOW	Arrival rate of Pd requests	$\lambda = \frac{1}{\text{Sampling period}} \times \frac{1}{\text{Batch size}} \times \#\text{ of application processes per node}$
	Pd CPU utilization	$\mu_{Pd,\,CPU}(\lambda) = \lambda D_{Pd,\,CPU}$
	Pd network utilization	$\mu_{Pd,\,Network}(\lambda) = n\lambda D_{Pd,\,Network}$
	Monitoring latency	$R(\lambda) = \frac{D_{Pd,\,CPU}}{1-\mu_{Pd,\,CPU}(\lambda)} + \frac{D_{Pd,\,Network}}{1-\mu_{Pd,\,Network}(\lambda)}$
	Paradyn CPU utilization	$\mu_{Paradyn,\,CPU}(\lambda) = n\lambda D_{Paradyn,\,CPU}$
SMP	Arrival rate of Pd requests	$\lambda = \frac{1}{\text{Sampling period}} \times \frac{1}{\text{Batch size}} \times n \times \#\text{ of Pds}$
	Pd CPU utilization	$\mu_{Pd,\,CPU}(\lambda) = \lambda \frac{D_{Pd,\,CPU}}{n}$
	Pd bus utilization	$\mu_{Pd,\,Bus}(\lambda) = \lambda D_{Pd,\,Bus}$
	Monitoring latency	$R(\lambda) = \frac{D_{Pd,\,CPU}/n}{1-\mu_{Pd,\,CPU}(\lambda)} + \frac{D_{Pd,\,Bus}}{1-\mu_{Pd,\,Bus}(\lambda)}$
	Paradyn CPU utilization	$\mu_{Paradyn,\,CPU}(\lambda) = \lambda \frac{D_{Paradyn,\,CPU}}{n}$
	IS CPU utilization	$\mu_{IS,\,CPU}(\lambda) = \frac{(\#\text{ of Pds} \cdot \mu_{Pd,\,CPU}(\lambda)) + \mu_{Paradyn,\,CPU}(\lambda)}{\#\text{ of Pds} + 1}$
MPP (binary tree)	Arrival rate of Pd requests	$\lambda = \frac{1}{\text{Sampling period}} \times \frac{1}{\text{Batch size}} \times n$
	Pd CPU utilization	$\mu_{Pd,\,CPU}(\lambda) = \frac{\frac{n}{2}\lambda D_{Pd,\,CPU} + \left(\frac{n}{2}-1\right)(\lambda D_{Pd,\,CPU} + 2\lambda D_{Pdm,\,CPU}) + \lambda D_{Pdm,\,CPU}}{n}$
	Pd network utilization	$\mu_{Pd,\,Net}(\lambda) = \frac{\frac{n}{2}\lambda D_{Pd,\,Net} + \left(\frac{n}{2}-1\right)(\lambda D_{Pd,\,CPU} + 2\lambda D_{Pd,\,Net}) + \lambda D_{Pd,\,Net}}{n}$
	Monitoring latency	$R(\lambda) = \frac{D_{Pd,\,CPU} + D_{Pdm,\,CPU}}{1-\mu_{Pd,\,CPU}(\lambda)} + \frac{D_{Pd,\,Network}}{1-\mu_{Pd,\,Network}(\lambda)}$
	Paradyn CPU utilization	$\mu_{Paradyn,\,CPU}(\lambda) = 2\lambda D_{Paradyn,\,CPU}$

- In the case of NOW system, analytical results indicate that the Paradyn daemon CPU overhead does not change with respect to the number of system nodes. However, under the BF forwarding policy, the overhead is significantly low. This difference between the CF and BF cases is due to the dependence of arrival rate λ on the batch size, as depicted in equation (6.10). In case of the CF and BF forwarding policies, the batch sizes are 1 and 32, respectively. This is the main reason for differences between the CF and BF

results for same number of nodes. Analytical results with respect to variable number of nodes and sampling periods indicate that the BF policy is desirable as it results in lower CPU overhead and monitoring latency.

- In case of the SMP system, IS CPU utilization and monitoring latency metrics depend on the number of Paradyn processes. Additionally, the arrival rate λ is also proportional to the number of Paradyn daemons. Therefore, the analytical results indicate that the use of multiple daemons results in a lower monitoring latency on the cost of relatively higher CPU overhead compared to the single daemon case. Use of multiple daemons is particularly justified at lower values of the sampling periods.
- In case of the MPP system, results indicate that the tree forwarding has clear advantage over the direct forwarding in terms of lower CPU overhead for the main Paradyn process. Equation (6.22) shows that under the tree forwarding the CPU overhead due to Paradyn remains unchanged as long as the arrival rate at a node (i.e., λ) is constant. On the other hand, this overhead increases with the number of nodes under the direct forwarding policy. Monitoring latency is higher for the tree forwarding due to the merge operations at each subsequent level of the tree before a sample reaches the Paradyn process at the root of the tree. The differences between Paradyn daemon CPU overhead under the two forwarding policies are insignificant.

6.3.2 Simulation-Based Evaluation

In this subsection, we compare possible configurations and management policies for the Paradyn IS using simulation-based evaluation of the ROCC model. Simulation-based evaluation is more accurate than analytical approaches because we account for the inter-dependences between the application and IS workloads and details of system functionality. We keep this evaluation process focused by posing specific "what-if" questions that are of interest to the developers and users of the IS. This focus is further refined by using the *principal component analysis* (PCA) technique to determine the system parameters and their combinations that can significantly affect the selected IS performance metrics.

6.3.2.1 Experimental Setup

As with analytical calculations, simulation-based experiments also consider three types of parallel or distributed system architectures: NOW, SMP, and MPP. We make minor modifications in the ROCC model to accommodate the specific characteristics of each of these configurations. In the case of a NOW system, each node has one CPU and the nodes

are interconnected via a switch-based or a shared network. For an SMP, multiple CPUs are connected through a bus. An MPP system is similar to the NOW system but has a multi-stage switched network and typically consists of a larger number of nodes. We parameterized the ROCC model for an IBM SP-2 system, which is closer to the NOW configuration. Nevertheless we use the modified ROCC models to extend the scope of the Paradyn IS evaluation to the SMP and MPP systems.

In answering various "what-if" questions regarding the Paradyn IS management and configuration, our simulation experiments are designed to analyze the effects of six parameters (factors):

1. *number of concurrent system nodes*: the number of NOW, SMP, or MPP system nodes that execute the instrumented application as well as the IS processes;
2. *sampling period*: length of wall-clock time between two successive collections of performance data samples from an instrumented application process;
3. *number of local application processes*: number of application processes running on one node of the parallel/distributed system;
4. *forwarding policy*: the policy implemented by the Paradyn daemon at each node to forward instrumentation data samples to the main Paradyn process;
5. *application type*: compute- or communication-intensive (determined by the network occupancy requirement and frequency of synchronization barrier operations); and
6. *network configuration*: direct or binary tree (logical) configuration of the nodes to forward the instrumentation data from a Paradyn daemon to the main Paradyn process.

For each system architecture type, we use a subset of these factors for simulation-based experiments. We use a $2^k r$ *factorial design* technique for the simulation-based experiments, where k is the number of factors of interest for a given case, r is the number of repetitions of each experiment, and each factor can assume one of two possible values. For these experiments, we select k=4 factors and r=50 repetitions, and the mean values of the performance metrics of interest are derived within 90% confidence intervals from a sample of fifty values. This approach helps reduce the variance (or "noise") in the results; thus any differences among the performance metrics under varying IS configurations and management policies are clarified.

6.3.2.2 Principal Component Analysis

For each of the three system architecture types, we supplement the $2^k r$ factorial experiment design technique with *principal component analysis* (PCA) to assess the sensitivity of the performance metrics to selected model parameters (factors) [100]. With multiple factors, we cannot assume that each acts independently on the system under test (i.e., the IS). PCA helps determine the relative importance of individual factors, as well as their inter-dependences. Instead of evaluating the metrics for all possible combinations of the factors for each "what-if" question, we use a subset of combinations that are deemed important by the PCA.

For the NOW architecture, we assume that the system nodes are connected through a shared network (Ethernet). Each node runs an application process and a Paradyn daemon. One of the nodes also executes the main Paradyn process. Paradyn daemons on individual nodes directly forward the instrumentation data to the main process. We arbitrarily select a batch size of 32 samples for the BF policy. For compute-intensive applications, the mean network occupancy requirement is arbitrarily set at 200 μsec; and for communication-intensive applications, 2000 μsec. Four factors of interest in this case are: number of nodes, sampling period, forwarding policy, and application type. Applying the $2^k r$ factorial design technique, we conduct sixteen simulation experiments, obtaining the results shown in Table 6-5. Four factors produce fifteen combinations that affect a metric: four for individual factors; six for the combinations of two factors; four for the combinations of three factors; and one for the combination of four factors.

The bar graphs in Figure 6-8 present the results of the principal component analysis. Clearly, the sampling period (labeled as B) is the single most important factor that affects the direct overhead of the Paradyn daemon, followed by the data forwarding policy (C), and the combination of the two (BC). The data forwarding policy (C) and number of nodes (A) are the most important factors affecting monitoring latency. Thus, a further investigation of the IS behavior with respect to the sampling period (B), the number of nodes (A), and the data forwarding policy (C) is justified.

Table 6-5. Results of simulation experiments for the NOW system.

Parameters			Compute-Intensive Application		Communication-Intensive Application	
Number of nodes	Sampling period (msec)	Forwarding Policy	Pd CPU Time per Node (sec)	Monitoring Latency per Received Sample (msec)	Pd CPU Time per Node (sec)	Monitoring Latency per Received Sample (msec)
2	5	CF	5.33	3.52	5.34	2.83
2	50	CF	0.54	0.07	0.54	0.07
32	5	CF	5.34	0.07	5.34	2.92
32	50	CF	0.53	5.42	0.53	4.63
2	5	BF	2.48	0.08	2.48	0.08
2	50	BF	0.21	0.07	0.21	0.07
32	5	BF	1.78	0.68	1.78	0.70
32	50	BF	0.22	0.94	0.20	0.88

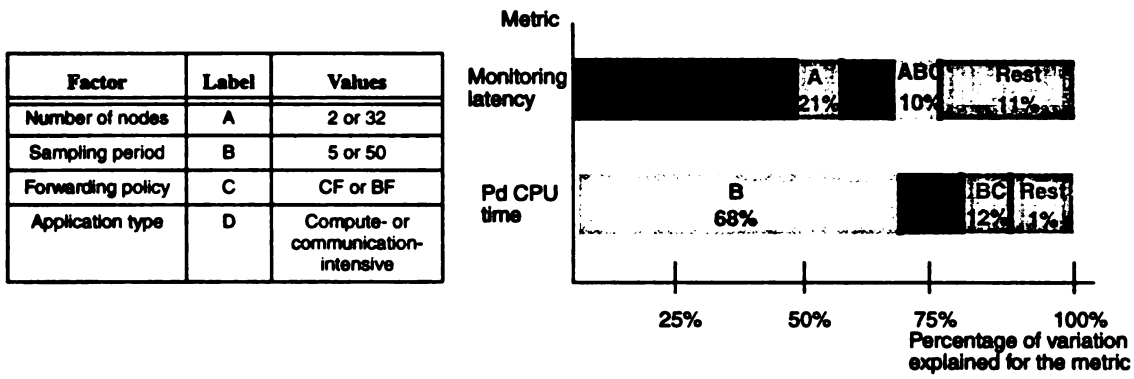


Figure 6-8. Results of principal component analysis of four factors and their combinations for the NOW system.

PCA for the SMP and MPP architectures is conducted in a similar manner but with slightly different sets of factors. Tables 6-6 and 6-7 present the results of sixteen experiments for the SMP and MPP architectures, respectively.

The final results of the PCA for SMP and MPP architectures are depicted in Figure 6-9. Figure 6-9(a) shows the results of the PCA for an SMP architecture. The number of nodes (labeled as A) is the most important factor that affects the direct overhead of the Paradyn IS (i.e., daemon and the main process), followed by the forwarding policy (C) and the sampling period (B). The data forwarding policy (C), the number of nodes (A), and the

Table 6-6. Results of simulation experiments for the SMP system. (number of application processes = number of nodes)

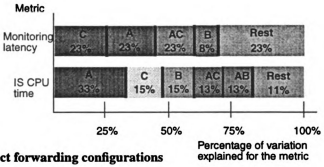
Parameters			Compute-Intensive Application		Communication-Intensive Application	
# of Nodes	Sampling Period (msec)	Forwarding Policy	IS CPU Time per Node (sec)	Monitoring Latency per Received Sample (msec)	IS CPU Time per Node (sec)	Monitoring Latency per Received Sample (msec)
1	5	CF	11.16	0.93	11.16	0.93
1	50	CF	2.69	3.57	2.69	3.57
32	5	CF	0.52	0.001	0.52	0.001
32	50	CF	0.17	0.001	0.17	0.001
1	5	BF	2.60	0.001	2.60	0.001
1	50	BF	0.72	0.001	0.72	0.001
32	5	BF	0.11	0.001	0.11	0.001
32	50	BF	0.11	0.001	0.11	0.001

Table 6-7. Results of simulation experiments for the MPP system.

Parameters			Direct Forwarding		Tree Forwarding	
# of Nodes	Sampling Period (msec)	Forwarding Policy	Pd CPU Time per Node (sec)	Monitoring Latency per Received Sample (msec)	Pd CPU Time per Node (sec)	Monitoring Latency per Received Sample (msec)
2	5	CF	0.54	3.76	0.54	3.76
256	5	CF	0.35	3.84	0.02	10.00
2	50	CF	0.54	0.28	0.05	0.30
256	50	CF	0.05	5.60	0.02	4.07
2	5	BF	0.21	0.12	0.21	0.12
256	5	BF	0.14	0.16	0.16	0.19
2	50	BF	0.01	0.12	0.01	0.12
256	50	BF	0.02	0.20	0.02	0.07

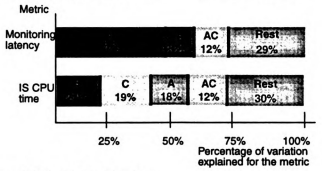
combinations of the two (AC) are the most important factors affecting monitoring latency. Figure 6-9(b) shows the results of PCA for an MPP architecture. The sampling period (B), the forwarding policy (C), and number of system nodes (A) are equally important factors affecting the direct overhead of the Paradyn IS, followed by the forwarding policy (C). The forwarding policy (C) and the number of nodes (A) are the most important factors affecting monitoring latency.

Factor	Label	Values
Number of nodes	A	2 or 32
Sampling period	B	5 or 50
Forwarding policy	C	CF or BF
Application type	D	Compute- or communication-intensive



(a) SMP with direct forwarding configurations

Factor	Label	Values
Number of nodes	A	2 or 32
Sampling period	B	5 or 50
Forwarding policy	C	CF or BF
Network configuration	D	Direct or binary tree



(b) MPP with compute-intensive applications

Figure 6-9. Results of PCA for (a) SMP and (b) MPP architectures for four factors and their combinations.

In summary, PCA directs us to focus on the following features, in order of importance: B, C, A for NOW; A, C for SMP; and C, A, B for MPP architecture. PCA indicates that monitoring latency is most affected by forwarding policy and number of nodes; and IS overhead by sampling period and forwarding policy as well as number of nodes in SMP and MPP cases.

6.3.2.3 Investigation of "what-if" Questions

In this subsection, we present simulation-based results that answer specific "what-if" questions that are posed to the ROCC model. These questions are explored in order on the NOW, SMP, and MPP architectures. These questions are related to the forwarding policy, use of multiple Paradyn daemons, and logical network configurations for data forwarding.

NOW Architecture: What are the effects of forwarding policy with varying the number of nodes and sampling periods?

The PCA in Section 6.3.2.2 shows that the choice of data forwarding policy significantly impacts the IS overhead. In this subsection, we compare the CF and BF policies by varying the number of system nodes and sampling periods.

The simulation results with respect to varying the number of system nodes, depicted in Figure 6-10, lead to following observations:

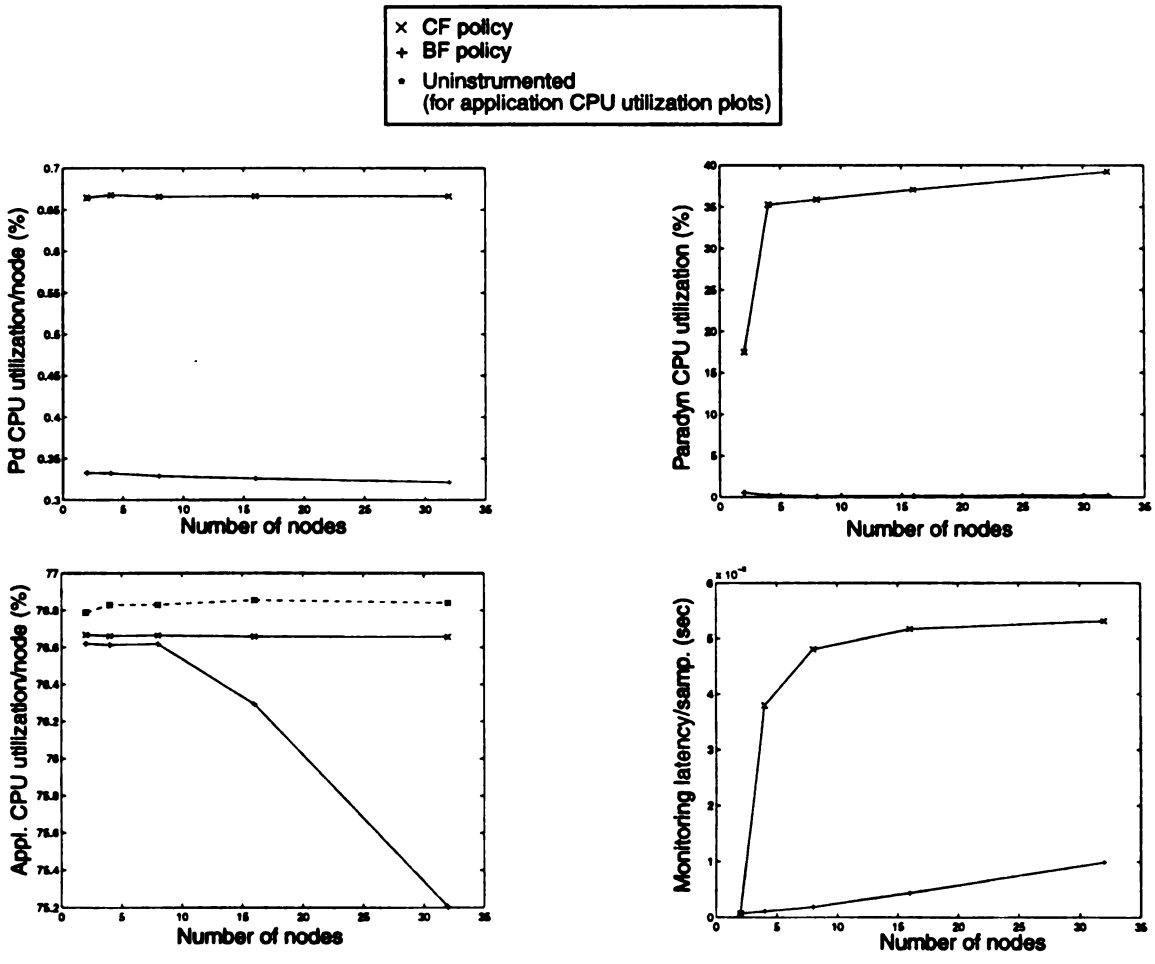


Figure 6-10. Effects of varying number of system nodes on the metrics with respect to the CF and BF policies (sampling period = 40 msec).

- Although the direct overhead of Paradyn daemon CPU utilization does not vary with the number of system nodes due to its localized nature, Figure 6-10 shows that the BF policy incurs lower overhead. The CPU overhead by the main Paradyn process under the CF policy increases with the number of nodes due to more data samples forwarded

to it. However, this overhead is significantly smaller under the BF policy since fewer batches provide the same number of data samples as under the CF policy. Monitoring latency is also lower under the BF policy because on the aggregate more data can be transferred in a shorter time.

- The behavior depicted by the simulation results in Figure 6-10 is generally consistent with the analytic results in Figure 6-5(a). Differences are due to the approximate nature of analytic calculations that do not accurately consider inter-dependences and resource contentions among the workloads.

The simulation results with respect to varying the sampling period, presented in Figure 6-11, lead to following observations:

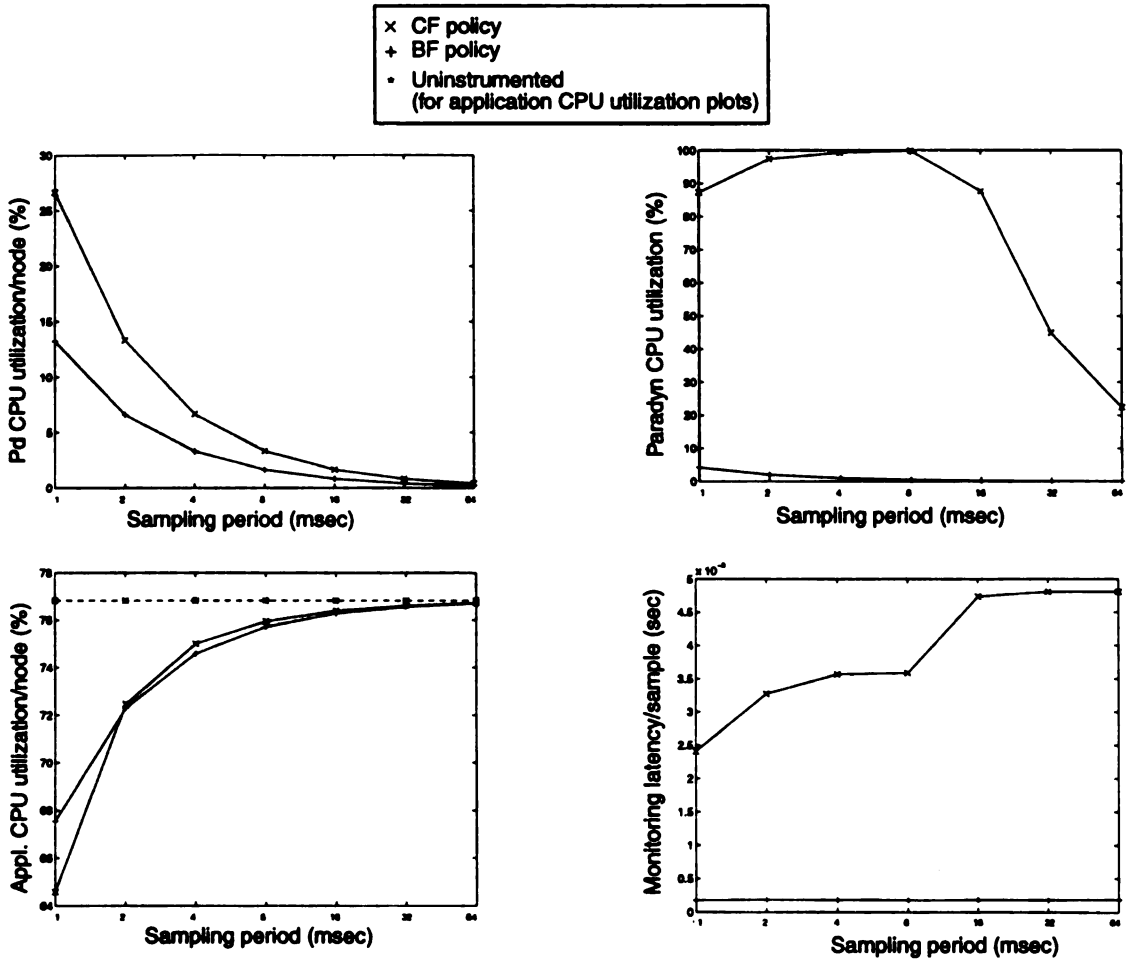


Figure 6-11. Effects of varying the sampling periods on the metrics with respect to the CF and BF data forwarding policies (number of nodes = 8, contention-free network).

- Figure 6-11 shows that the monitoring latency is not significantly affected by variations in the sampling period. The direct IS overhead and intrusion to the application decrease with increasing sampling period. As sampling period increases, the application CPU utilization approaches the uninstrumented level.

- The application CPU utilization significantly decreases at sampling periods less than 4 msec (see the lower left plot). Therefore, neither CF nor BF policy can support more than 250 samples per second.
- The behavior depicted by the simulation results in Figure 6-11 is generally consistent with the analytic results in Figure 6-5(b). The differences are elaborated in third "what-if" question related to the SMP architecture. Simulation of the ROCC model accurately accounts for the resource contentions according to the scheduling policies used by the operating system.

These results indicate that the BF data forwarding policy outperforms the CF policy with respect to both direct overhead and monitoring latency. This was also found true for the SMP and MPP architecture. Therefore, we consider only the BF policy in the following subsections.

NOW Architecture: What should be the size of the batch?

After determining that the BF policy is better with respect of our metrics of interest, we investigate the effect of the batch size on the overall system performance. Since the PCA for the NOW system indicates that sampling period is the most important factor for the CPU overhead of Paradyn daemon, we investigate this question by varying the batch sizes for three levels of the sampling period: a short value of 1 msec, an intermediate value of 40 msec, and a longer value of 64 msec. Simulation results presented in Figure 6-12 indicate that:

- Typically, significant changes can be observed when batch size is increased from one, i.e., at the transition from the CF to BF policy. In addition, these changes are more relevant for shorter sampling periods. This is consistent with the results of the PCA that verifies the importance of forwarding policy and sampling period.
- Monitoring latency exhibits a sharp decrease with increasing batch sizes after the change over point from the CF to the BF policy. However, this sharp initial decrease levels off at larger batch sizes. A batch size of greater than 2 samples reduces the CPU occupancy requirements for forwarding individual samples. An excessively large batch size also takes a longer time to accumulate, especially at lower sampling periods; thus it does not result in any significant improvement in monitoring latency.

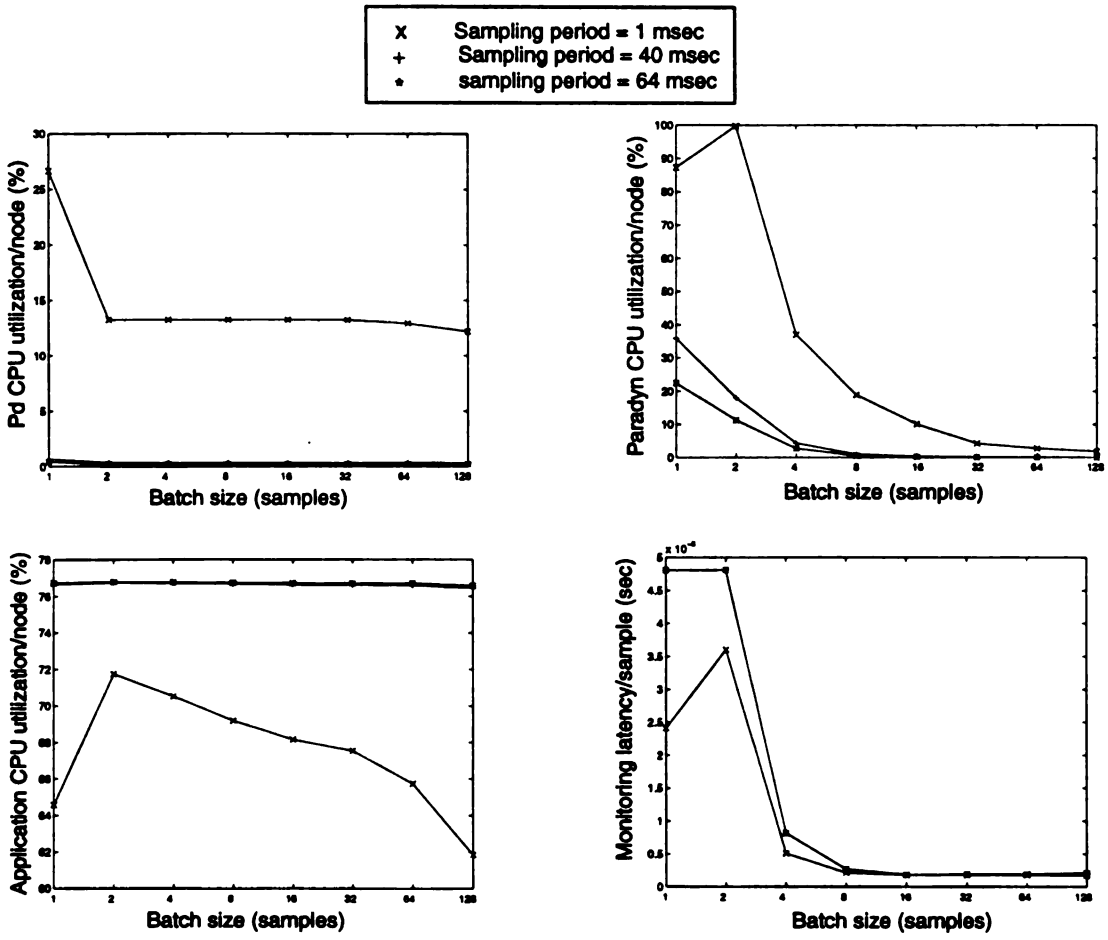


Figure 6-12. Effects of varying the size of batch of samples to be forwarded from Paradyn daemon to the main Paradyn process on IS performance metrics (number of nodes = 8, contention-free network).

Based on the above observations, a value of batch size close to the "knee" of the monitoring latency curve is desirable. We selected a batch size of 32 for the BF cases presented here.

SMP Architecture: What is the effect of multiple Paradyn daemons on the monitoring latency?

Simulation results in Figure 6-10 show that the monitoring latency increases with the number of nodes. In order to maintain a lower monitoring latency, we investigate the potential effects of using multiple Paradyn daemons (up to four) on an SMP. An important

factor with respect to the use of multiple daemons is the sampling period. Figure 6-13 evaluates the use of multiple Paradyn daemons in terms of direct Paradyn daemon and main process (IS) overhead, monitoring latency, and intrusion to the application processes under the BF policy. We conclude the following from Figure 6-13:

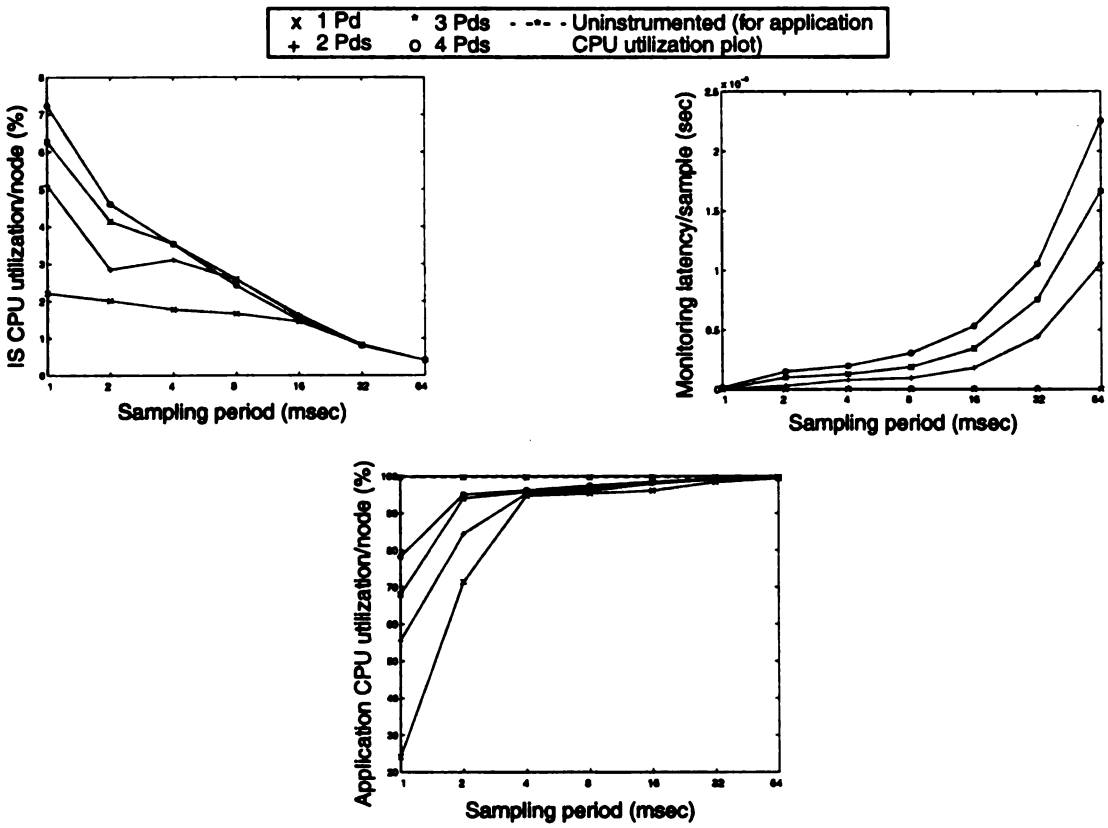


Figure 6-13. Effects of multiple Paradyn daemons on two metrics (number of nodes = 16, application processes = 32, BF policy, duration of simulation = 100 sec, logarithmic horizontal scale).

- The number of Paradyn daemons does not have any intrusive impact on the application except at sampling periods of less than 10 msec. At shorter sampling periods, the application CPU time significantly decreases, particularly for one Paradyn daemon. This is not a consequence of high CPU utilization by Paradyn daemons at lower sampling periods (see the left plot). Rather, at a lower sampling period, the pipe that holds data samples for a Paradyn daemon fills to its capacity more often. When the pipe is full, the application process that generates a sample is blocked until the daemon is able to forward outstanding data samples. The effect of this blocking is reduced if the number of Paradyn daemons is increased for smaller sampling periods.
- The monitoring latency increases with the number of Paradyn daemons. This small increase is a consequence of additional CPU contention due to multiple Paradyn daemon processes.

- It is interesting to note that the behavior of the monitoring latency shown in Figure 6-13 is opposite to that predicted by analytical calculations in Figure 6-6. Analytical calculation of monitoring latency for the SMP architecture (presented in Table 6-4) is a function of only the arrival rate (λ). Arrival rate is inversely proportional to the sampling period and directly proportional to the number of Paradyn daemons. Since the ratio of the number of Paradyn daemons to the sampling period decreases with increases in sampling period, the arrival rate and hence analytical monitoring latency also decrease. However, the analytical model does not account for the fact that a longer sampling period means longer periods of time between successive samples being forwarded from a node, which translates to longer latency in the end. On the other hand, simulation of the ROCC model accurately accounts for the time between successive samples in calculating monitoring latency. It also accounts for CPU and bus contention of multiple daemons.

SMP Architecture: What are the effects of multiple Paradyn daemons under CF and BF policies with varying number of application processes?

With multiple Paradyn daemons, we investigate the effect of varying the number of application processes while keeping the number of nodes and sampling period constant. The objective is to evaluate the use of multiple daemons when varying amount of work is being generated, depending on the number of application processes. Figure 6-14 shows the results of this case.

- The effect of multiple Paradyn daemons on the IS CPU overhead is insignificant until the number of application processes becomes greater than the number of CPUs (i.e., system nodes). Similarly, the intrusion to the application is also unaffected by the number of Paradyn daemons.
- Monitoring latency for multiple Paradyn daemons is greater than the latency for one daemon, especially for larger number of application processes. This increase is due to more contention for shared resources due to larger number of application and daemon processes.

These results show that the use of multiple Paradyn daemons per node may not result in improved monitoring latency on an SMP. In fact, it may increase the latency due to additional resource contentions.

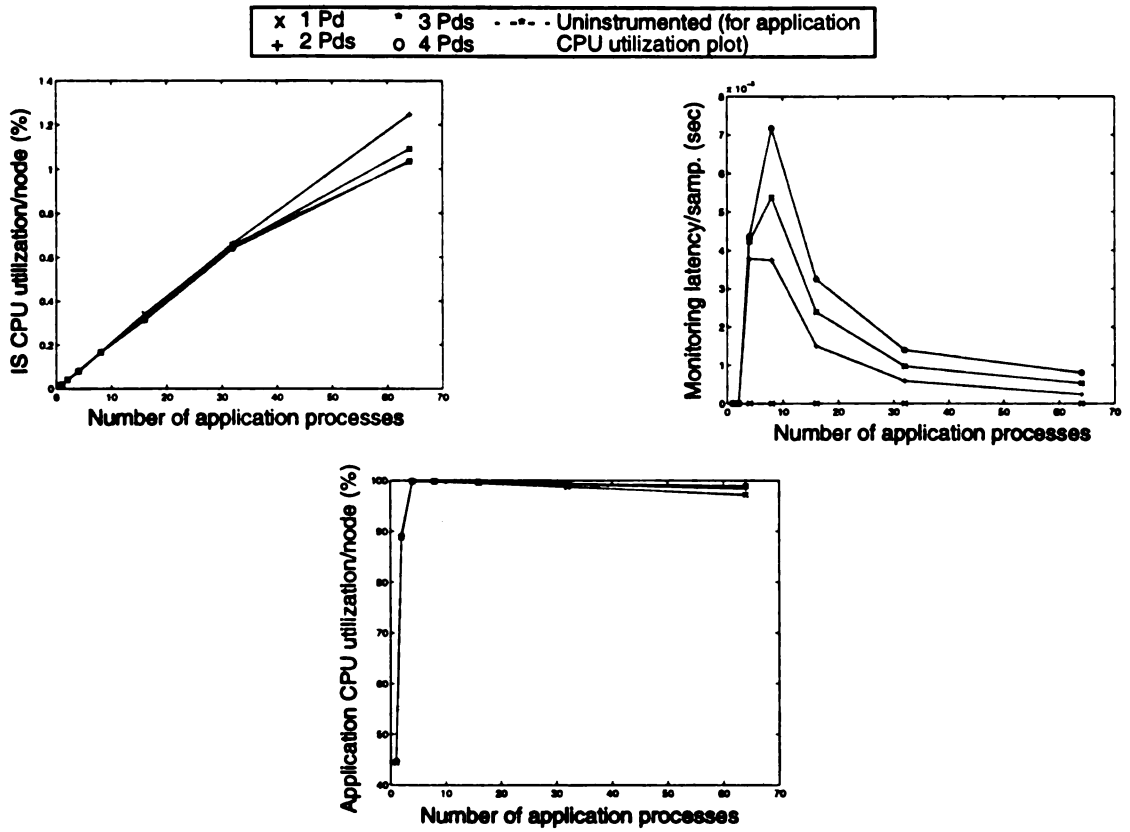


Figure 6-14. Effects of multiple Paradyn daemons on the metrics with respect to CF and BF data forwarding policies (sampling period = 40 msec, number of nodes = 16, BF policy, duration of simulation = 100 sec).

MPP Architecture: What is the effect of direct vs. tree forwarding on scalability?

A typical MPP system may consist of hundreds of nodes. Our objective is to study the scalability of data collection when hundreds of nodes forward instrumentation data samples through their local Paradyn daemons. In this cases, a single data collection and reduction node that hosts the main Paradyn process is likely to become a bottleneck. We proposed the use of a binary tree configuration in Section 5.5.2.2 (Figure 5-10) for intermediate reduction and forwarding of instrumentation data samples. In this subsection, we compare the scalability of the Paradyn IS under direct and tree configurations.

Principal component analysis in Section 6.3.2.2 indicates that the effect of varying the sampling period on the direct IS overhead should be significant. Figure 6-15 represents the effects of varying sampling periods under the direct and binary tree data forwarding

configurations. The results are again shown under the BF policy only. Analyzing direct forwarding versus tree forwarding, we make the following comparisons:

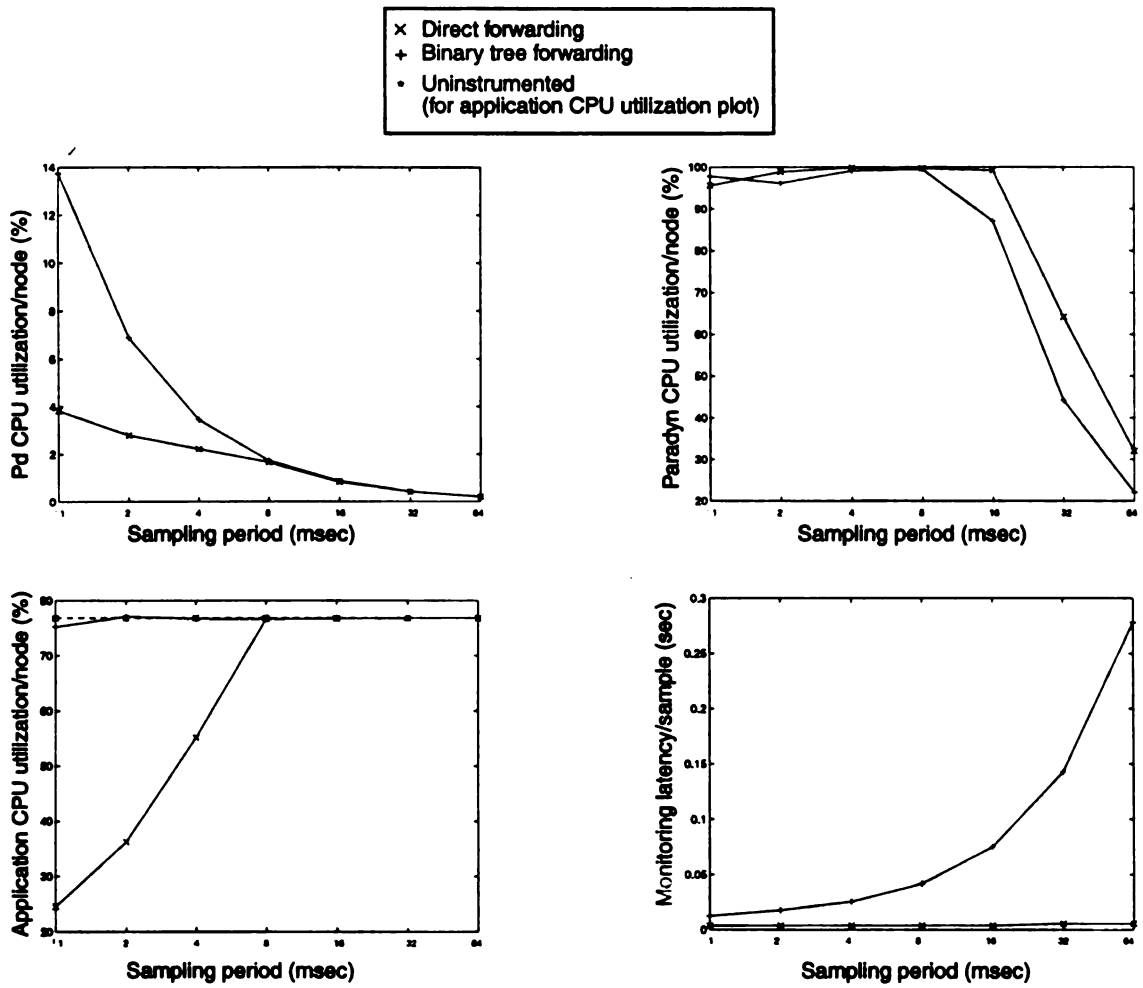


Figure 6-15. Effects of varying sampling periods with respect to direct or tree forwarding on the IS performance metrics (number of nodes = 256, BF policy, logarithmic horizontal scale).

- Per node Paradyn daemon CPU overhead is higher under the binary tree configuration at shorter sampling periods due to the increased volume of samples being generated. CPU utilization of the main Paradyn process reaches nearly 100% because it is swamped by sample arrivals from 256 nodes. With direct forwarding, a swamped main Paradyn process blocks all the Paradyn daemons that try to forward further samples to it. Blocking results in lower Paradyn daemon CPU utilization even though samples are pending. Since most of the data reduction and merging are handled by intermediate Paradyn daemons, blocking is less likely under tree forwarding because the main Paradyn process has less work to do.
- The same phenomena that lead to the performance of the IS processes impacts the application processes as well. When a Paradyn daemon blocks, waiting to forward additional samples, it forces the application process generating samples to block. Thus

the application CPU utilization at a node is reduced to 25% instead of an uninstrumented 78%. Tree forwarding greatly reduces this intrusion to the application processes.

- Although the CPU utilization values for the main Paradyn process are almost the same under the direct and tree forwarding cases, the number of samples collected under the tree forwarding is larger.
- Monitoring latency is higher for the tree configuration because a set of samples originating from the leaf nodes undergoes a logarithmic number of forwarding operations instead of one for direct forwarding. Additionally, the monitoring latency increases with sampling period for the tree configuration because intermediate Paradyn daemons do not merge and forward the "enroute" samples asynchronously; these samples are forwarded after the expiration of the current sampling period. We do not process the enroute samples asynchronously because doing so significantly reduces the CPU time available for the local application process.

Simulation results presented in this subsection suggest that the use of binary tree forwarding is beneficial to improve the scalability of the Paradyn IS as the number of system nodes increases to several hundreds. For 256 nodes and sampling periods less than 8 msec (i.e., more than 25 samples per second), the Paradyn IS should switch from direct to tree forwarding.

MPP Architecture: What is the effect of varying the frequency of barrier operations in a program on IS overhead and intrusion?

Barrier synchronization is frequently used to explicitly implement a lock-step execution of parts of a program on an MPP system. Since barrier synchronization causes global coordination among application processes, it is of interest to consider the impact of on-line data collection on programs with different rates of barrier synchronization. In particular, we want to verify that the additional Paradyn daemon overhead for tree forwarding does not unduly perturb application execution time. Figure 6-16 presents the results of our investigation of the effect of varying the frequency of barrier synchronization operations on the Paradyn IS.

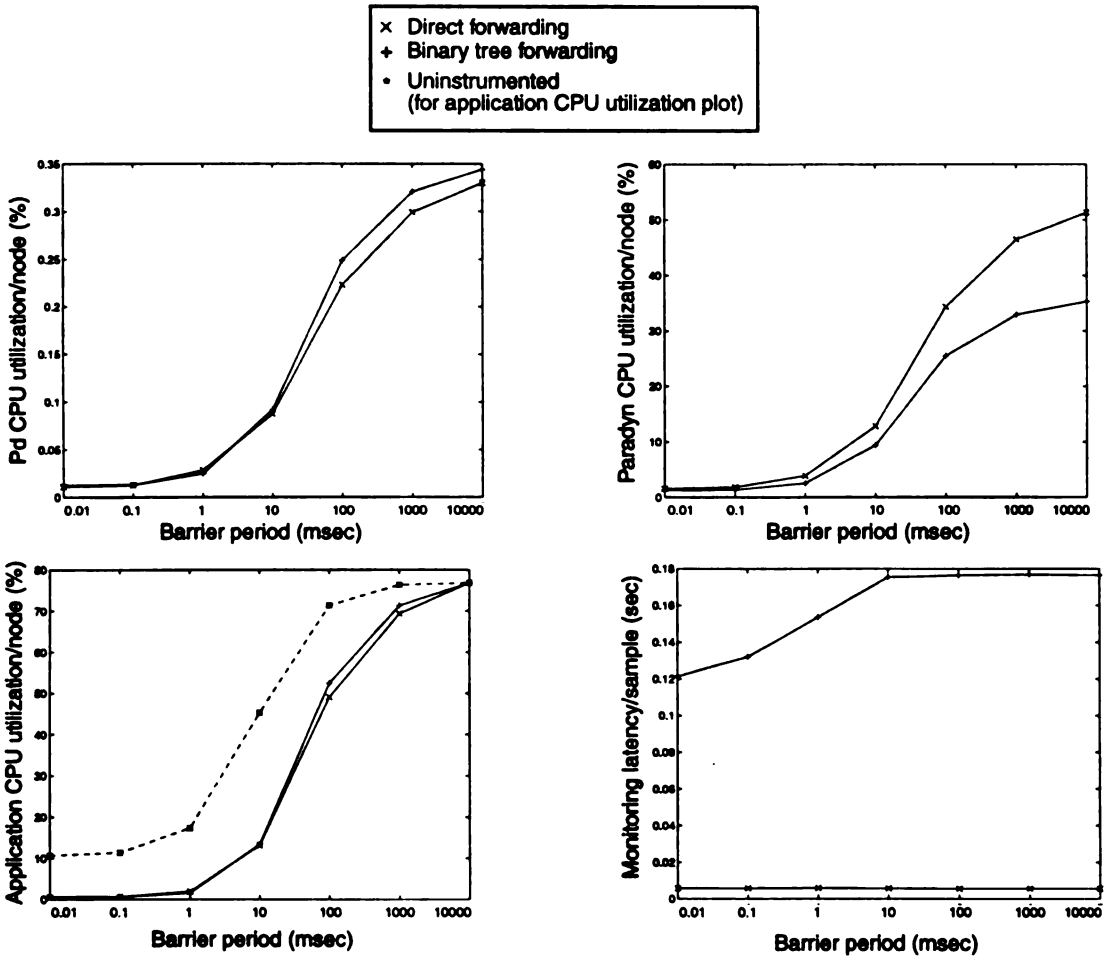


Figure 6-16. Effects of varying frequency of barrier operations (number of nodes = 256, sampling period = 40 msec, BF policy, logarithmic scales for barrier periods).

- CPU overhead of both Paradyn daemons and main process decreases at higher barrier frequencies (and lower barrier periods as shown in the figure). While an application process waits to exit from the barrier, the Paradyn daemon does not compete for CPU time with the application process. Note that the CPU overhead for Paradyn daemon is only a fraction of a percent for the entire range of barrier period.
- Tree forwarding does not result in a lower application CPU utilization compared to the direct forwarding at any barrier period value. Thus, tree forwarding does not cause any additional intrusion to the application.
- The penalty of having instrumentation in the application varies from 10%–35% for a barrier period range of 1 to 100 msec. It appears that any delay in dispatching an instrumented application process on a node significantly reduces the amount of useful work done by that process, especially when coupled with the synchronization operations. This behavior identifies a potential bottleneck in the Paradyn IS for an MPP system.
- The monitoring latency is unaffected due to barrier operations but exhibits differences due to the direct or tree configurations.

These results indicate that barrier synchronization operations result in greater intrusion, essentially independent of the choice of data forwarding configuration. As a result, we are able to use tree forwarding without introducing additional application perturbation.

6.3.3 Feedback to the Developers

The investigation of the "what-if" questions presented in the preceding subsections evaluated the Paradyn IS with several low-level details. However, such low-level details are typically less beneficial for tool developers or users. In order to provide them with useful feedback, we summarize the simulation-based evaluation results in this subsection.

Simulation-based evaluation results can be divided into two categories: results directly relevant to the actual implementation of the Paradyn IS on an IBM SP-2 (NOW architecture) platform; and results projecting the performance of the Paradyn IS to the SMP and MPP architectures under different operating conditions. The first category of results is useful for improving the IS; and the second category, for porting the IS to other platforms without compromising scalability or performance. We presented several conclusions from the individual "what-if" simulation-based analyses in the preceding subsection. The important results are summarized as follows:

1. the BF policy should be implemented as a default policy to schedule data forwarding operations because it outperforms the CF policy;
2. in the case of an SMP, use of multiple daemons per node represents a trade-off between more samples received by the main process and additional contention for system resources;
3. binary tree forwarding should be used on an MPP system due to its superior scalability characteristics compared to direct forwarding; and
4. specific application characteristics, such as frequency of barrier operations on an MPP system, may affect IS performance, which may in turn impact the instrumented application.

This feedback was well-received by the Paradyn IS developers and the BF policy was implemented in addition to the CF policy for the IBM SP-2 platform. Thus, we can experimentally validate these simulation results via testing of the actual IS.

6.3.4 Experimental Validation

We use measurement-based experiments to test the actual IS and validate the simulation-based results. Our objective is to experimentally verify that the performance of the real system with actual application programs matches the predictions of the simulator. Measurement-based tests generate large volumes of trace data. Investigating a number of "what-if" questions is less feasible than with simulation. Time is also required to implement and debug new policies. Therefore, testing necessarily focuses on specific aspects of performance under carefully controlled experimental conditions.

6.3.4.1 Experimental Setup

Figure 6-17 depicts the experimental setup for measuring the Paradyn IS performance on an IBM SP-2 system. We initially use the NAS benchmark *pvmbt* as the application process; and we use the AIX tracing facility on one of the SP-2 nodes executing the application process. The main Paradyn process executes on a separate node, which is also traced. Therefore, one experiment with a particular sampling period and data forwarding policy results in two AIX trace files. These trace files are then processed to determine execution statistics relevant to the test.

We conduct a set of four experiments based on two factors, *sampling period* and *scheduling policy*, each having two possible values. As in the simulation, the forwarding policy options are CF and BF. The sampling period is assigned a relatively low value (10 msec) or a higher value (30 msec). Experiments using Paradyn on SMP and MPP architectures are left to future work with Paradyn. Consistent with the simulation, network

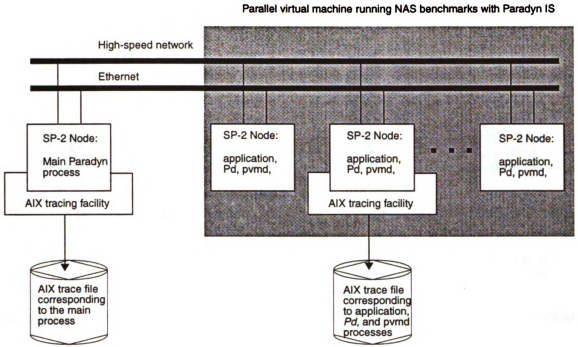


Figure 6-17. Measurement-based experiment setup for Paradyn IS on an SP-2.

occupancy is not considered (which means that communication events are not traced); this also reduces the disk space needed for AIX traces.

6.3.4.2 Evaluation

Figure 6-18 summarizes the Paradyn IS testing results related to the CPU overhead of the Paradyn daemon (a) and the main Paradyn process (b). The CPU utilization of the Paradyn daemon under the BF policy is about one-third of its value under the CF policy. This indicates a more than 60% reduction in overhead when Paradyn daemons send batches of samples rather than making system calls to send each sample individually. Similar analysis of the trace data obtained from the node running the main Paradyn process indicates that the overhead is reduced by almost 80% under the BF policy.

In order to determine the relative contribution of these two factors to the direct CPU overhead, we use principal component analysis. The results of this analysis for the Paradyn daemon and main Paradyn processes are shown in Table 6-8. Clearly, the

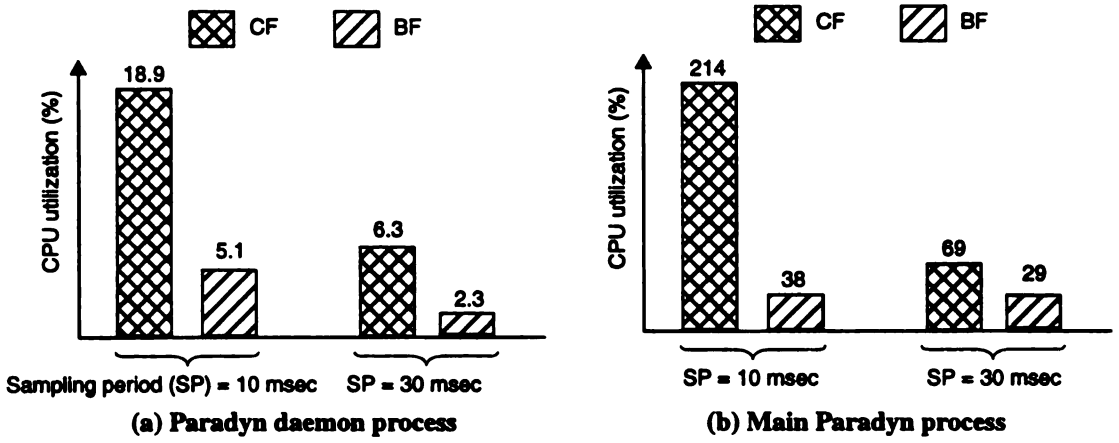


Figure 6-18. Comparison of CPU overhead measurements under the CF and BF policies using two sampling period values for (a) Paradyn daemon and (b) main Paradyn process.

scheduling policy to forward data is primarily responsible for variations in IS overhead. Thus, within the scope of our testing, the results verify that the performance of the real system matches the predictions of the simulation.

Table 6-8. Results of principal component analysis of scheduling policy vs. sampling period for the tests in Figure 6-18.

Factors or combination of factors	Variation explained for Paradyn daemon CPU time (%)	Variation explained for main Paradyn process CPU time (%)
A (scheduling policy for data forwarding)	47.6	52.9
B (sampling period)	35.9	26.5
AB	16.5	20.7

We conduct another set of measurement experiments to isolate the effect of a particular application on the Paradyn IS overheads. To do this, we experiment with two forwarding policies, CF and BF, and two NAS benchmark programs, *pvmbt* and *pvmis*. Benchmark *pvmbt* solves three sets of uncoupled systems of equations, first in the x, then in the y, and finally in the z direction. The systems are *block tridiagonal* with 5×5 blocks. Benchmark *pvmis* is an integer sort kernel. All experiments use a sampling period of 10 msec. The results are summarized in Figure 6-19. The key observation is that the reduction in IS

overheads under the BF policy is not significantly affected by the choice of application program.

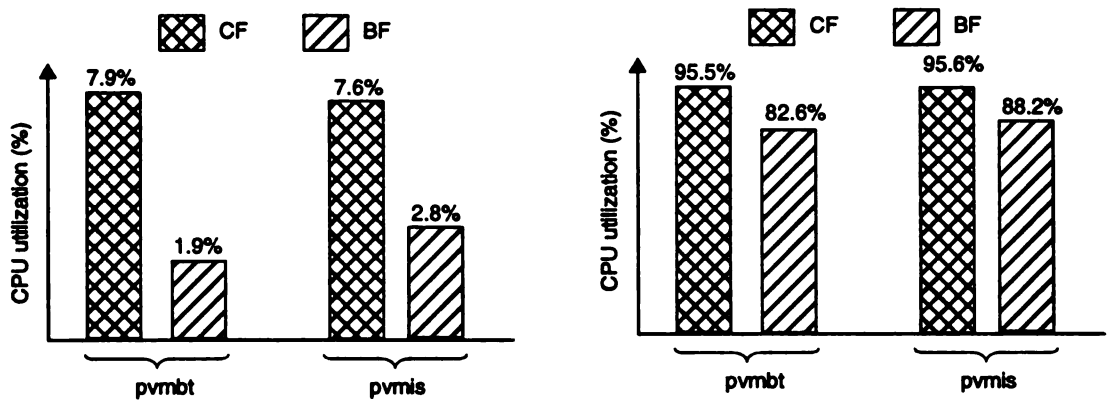


Figure 6-19. Paradyn IS testing results related to (a) Paradyn daemon and (b) main Paradyn process.

We again use principal component analysis to quantify the dependence of *IS* overheads on the choice of application program. The results of this analysis are shown in Table 6-9. Not surprisingly, the effect of the application program is negligible. Once again, the dominant factor under the current experimental setup is the scheduling policy.

Table 6-9. Results of principal component analysis of scheduling policy vs. application program for the tests in Figure 6-19.

Factors or combination of factors	Variation explained for Paradyn daemon's normalized CPU time (%)	Variation explained for main Paradyn process's normalized CPU time (%)
A (scheduling policy for data forwarding)	98.5	86.8
B (application program)	0.3	6.8
AB	1.2	6.4

6.4 Evaluation of the JEWEL IS

In this subsection, we evaluate the JEWEL IS model based on workload characterization presented in Section 5.5.3.4. We present analytical calculations using operations analysis

in subsection 6.4.1, simulation results in subsection 6.4.2, feedback to the developers in subsection 6.4.3, and experimental validation of selected simulation results in subsection 6.4.4.

6.4.1 Analytic Calculations

Similar to the Paradyn IS case, the ROCC model for the JEWEL IS is a mixed queueing network. It is an open network for the JEWEL sensor requests and a closed network for the application, visualizer, and resource manager agent requests. Since the operations analysis techniques cannot capture the interactions among these types of workloads, we focus on the JEWEL sensor requests and calculate only the IS-related metrics. This calculation is similar to the one for Paradyn IS in a NOW system.

6.4.1.1 Calculation of IS-Related Metrics

We first calculate the arrival rate λ of JEWEL sensor requests at each node. It is given as:

$$\lambda = \frac{1}{\text{Polling period}}. \tag{6.26}$$

The JEWEL sensor CPU utilization per node follows from the *utilization law* and *forced flow law* as:

$$\mu_{Sensor,\,CPU}(\lambda) = \lambda D_{Sensor,\,CPU}. \tag{6.27}$$

Using flow balance assumption, the throughput of each node is equal to λ. Therefore, overall JEWEL sensor CPU request throughput is equal to $P\lambda$, which is the overall arrival rate of JEWEL sensor network requests. The network utilization by JEWEL sensor requests is given by:

$$\mu_{Sensor,\,Network}(\lambda) = P\lambda D_{Sensor,\,Network}. \tag{6.28}$$

The monitoring latency of a sample that reaches the JEWEL collector in the form of an MDR is given by:

$$R(\lambda) = \frac{D_{Sensor,\,CPU}}{1-\mu_{Sensor,\,CPU}(\lambda)} + \frac{D_{Sensor,\,Network}}{1-\mu_{Sensor,\,Network}(\lambda)}. \tag{6.29}$$

In order to calculate the hold-back ratio, we first need to know the number of outstanding sensor requests in the system and an estimate of the number of received requests in observation time T. The maximum possible number of MDRs received by the collector in time T is equal to $n\lambda T$. The number of outstanding sensor requests in the system is given by:

$$Q_{Sensor}(\lambda) = \frac{P\mu_{Sensor,\,CPU}}{1-\mu_{Sensor,\,CPU}} + \frac{\mu_{Sensor,\,Net}}{1-\mu_{Sensor,\,Net}}.$$

Therefore, the hold-back ratio (HBR) is give by:

$$\mathrm{HBR}(\lambda) = \frac{1}{P\lambda T}\left[\frac{P\mu_{Sensor,\,CPU}}{1-\mu_{Sensor,\,CPU}} + \frac{\mu_{Sensor,\,Net}}{1-\mu_{Sensor,\,Net}}\right]. \tag{6.30}$$

As in the case of Paradyn IS, the analytical results presented by equations (6.26)–(6.30) are only approximate calculations. Simulation-based approach will be used to capture the interesting details such as sharing of system resources by multiple workloads.

6.4.1.2 Summary of Analytic Results

Table 6-10 summarized the analytical results from the ROCC model of JEWEL IS. These results are expressed as functions of arrival rate only, which depends on the polling period of the shared-memory ring buffer of the JEWEL IS. It is obvious from these results that sensor CPU utilization decreases and monitoring latency increases as the polling period

increases. However, the more accurate simulation-based results may deviate from this behavior as they take the contention of shared system resources into account.

Table 6-10. Summary of analytic results for the ROCC model of JEWEL IS.

Performance Metric	Analytic Results
Arrival rate of Pd requests	$\lambda = \frac{1}{\text{Polling period}}$
Sensor CPU utilization	$\mu_{Sensor,\,CPU}(\lambda) = \lambda D_{Sensor,\,CPU}$
Sensor network utilization	$\mu_{Sensor,\,Network}(\lambda) = P\lambda D_{Sensor,\,Network}$
Monitoring latency	$R(\lambda) = \frac{D_{Sensor,\,CPU}}{1-\mu_{Sensor,\,CPU}(\lambda)} + \frac{D_{Sensor,\,Network}}{1-\mu_{Sensor,\,Network}(\lambda)}$
Hold-back ratio	$\text{HBR}(\lambda) = \frac{1}{P\lambda T}\left[\frac{P\mu_{Sensor,\,CPU}}{1-\mu_{Sensor,\,CPU}} + \frac{\mu_{Sensor,\,Net}}{1-\mu_{Sensor,\,Net}}\right]$

6.4.2 Simulation-Based Evaluation

In this subsection, we present the results of simulating the ROCC model for the JEWEL IS to evaluate the performance of the IS and the controller. As in case of Paradyn IS, we present experimental design, principal component analysis, investigation of the "what-if" questions, and feedback to the developers.

6.4.2.1 Experimental Design

We again design the simulation experiments to calculate the value of a metric based on fifty independent repetitions. The mean values of the six metrics (defined in Section 5.5.3.6) are derived within 90% confidence intervals from this sample of fifty values at each operating point of interest. We use four variable model parameters (factors): ring buffer polling period, controller sampling period, adaptation policy, and control system scheduling policy.

6.4.2.2 Principal Component Analysis

Applying the 2^k factorial design technique, we conduct sixteen simulation experiments, obtaining the results shown in Table 6-2. For this analysis, each factor can assume one of two possible values.

Table 6-11. Results of simulation experiments for adaptive control of the JEWEL IS for the video application (number of nodes = 8, ring buffer size = 4000 MDRs, simulation time = 100 sec).

Parameters			Metrics				
Ring buffer polling period (msec)	Sampling period (msec)	Adaptation policy, control system scheduling	Client frame rate (frames/ sec)	Sensor CPU utilization per node (%)	Monitoring latency (msec)	Hold-back ratio (%)	Percent of lost MDRs (%)
0.001	0.001	SPP, Dis	29.99	8.00	1.00	23.08	17.95
100	0.001	SPP, Dis	29.99	3.40	1.52	23.13	17.95
0.001	100	SPP, Dis	29.99	8.02	0.81	23.64	17.93
100	100	SPP, Dis	29.99	3.40	1.36	23.11	17.93
0.001	0.001	DPP, Dis	29.99	0.01	0.81	99.96	74.31
100	0.001	DPP, Dis	29.99	5.75	0.60	45.43	31.98
0.001	100	DPP, Dis	29.99	8.52	2.14	33.28	24.35
100	100	DPP, Dis	29.99	8.00	1.61	23.01	17.88
0.001	0.001	SPP, Cen	29.99	10.01	1.25	0.03	0
100	0.001	SPP, Cen	29.99	4.40	1.27	0.07	0
0.001	100	SPP, Cen	29.30	10.00	0.83	0.92	0
100	100	SPP, Cen	29.99	4.40	1.27	0.04	0
0.001	0.001	DPP, Cen	24.15	41.16	2.16	14.68	14.68
100	0.001	DPP, Cen	29.98	15.82	2.33	20.19	15.56
0.001	100	DPP, Cen	29.96	11.48	1.21	3.92	0
100	100	DPP, Cen	29.97	24.52	1.86	0.001	0

SPP—Static Polling Period adaptation policy

DPP—Dynamic Polling Period adaptation policy

Dis—Distributed scheduling of the control system

Cen—Centralized scheduling of the control system

Figure 6-20 shows the results of the PCA. Clearly, scheduling policy for the control system (labeled as D) is the most important factor that affects all six metrics of interest. In case of monitoring latency, the most important factor is the combination of controller sampling period and the scheduling policy (labeled as BD). Also note that the client frame

rate and CPU utilization are sensitive to the same factors because one metric is dependent on the other. The adaptation policy is the second most important factor after control system scheduling policy that affects the sensor CPU utilization. Thus, a further investigation of the behavior of the IS with respect to control system scheduling policies, adaptation policies, and sampling periods is justified.

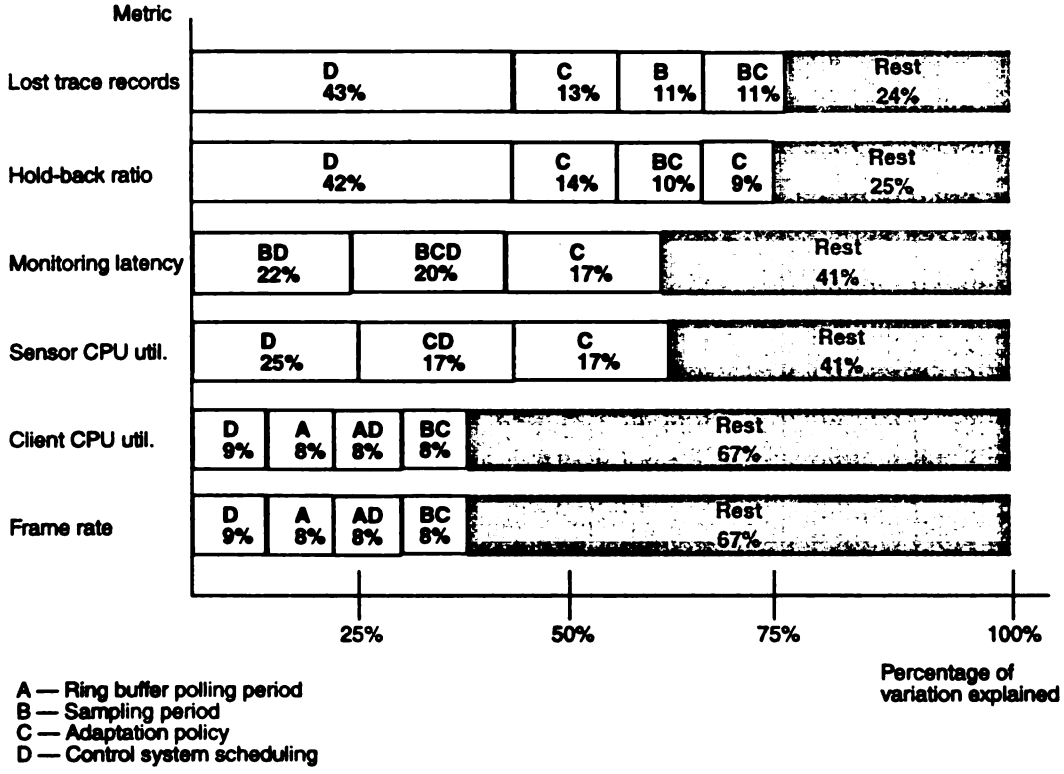


Figure 6-20. Results of principal component analysis of four factors and their combinations for the metrics of interest for JEWEL IS case study.

6.4.2.3 Investigation of "what-if" Questions

Simulation-based evaluation of the ROCC model for JEWEL IS explores the answers to three questions:

1. What is a desirable IS configuration and operating conditions with respect to the requirements of real-time video conferencing application?;
2. Which one of the two adaptation policies should be selected for actual implementation? Should this policy be scheduled in a centralized or a distributed fashion?; and

3. What is the performance of the adaptive controller designed on the basis of the answers to the above two questions?

The above questions are posed in a logical order such that the results of investigating one question are directly useful for the subsequent question. Results of investigating these questions are presented in the rest of this subsection.

What is a desirable IS configuration and a set of operating conditions?

We begin with an investigation of the effects of selecting different values for ring buffer polling period and ring buffer size under the CF and BF instrumentation data forwarding policies. Our primary goal is to select one of the two forwarding policies and size of the ring buffer. Ring buffer size should be suitable for any value of polling period, which is to be varied for adaptive control. For the cases presented in this subsection, we keep the adaptation turned off while the instrumentation system continues to perform its functions throughout the simulation experiments. Moreover, in order to compare with these cases, we use a base-line case that involves disabling the instrumentation in the application processes.

Figure 6-21 presents the behavior of the metrics of interest with respect to variable ring buffer polling periods under the CF and BF forwarding policies.

- It is important to note that under the CF policy and shorter polling periods, the frame rate drops well below the 30 frames/sec requirement. In comparison, intrusion to this real-time characteristic of the application is significantly reduced under the BF policy. Under the BF policy, the JEWEL external sensor can forward several MDRs to the collector as a batch by charging the same amount of CPU time (system call overhead) that is required to forward a single MDR. An application process can use this additional CPU time to process the incoming frames, resulting in higher frame processing rate.
- The BF policy also helps maintain a steady flow of instrumentation data (i.e., MDRs) from the distributed sensors to the collector. This is evident from the comparatively low monitoring latency and hold-back percentage and larger number of MDRs received by

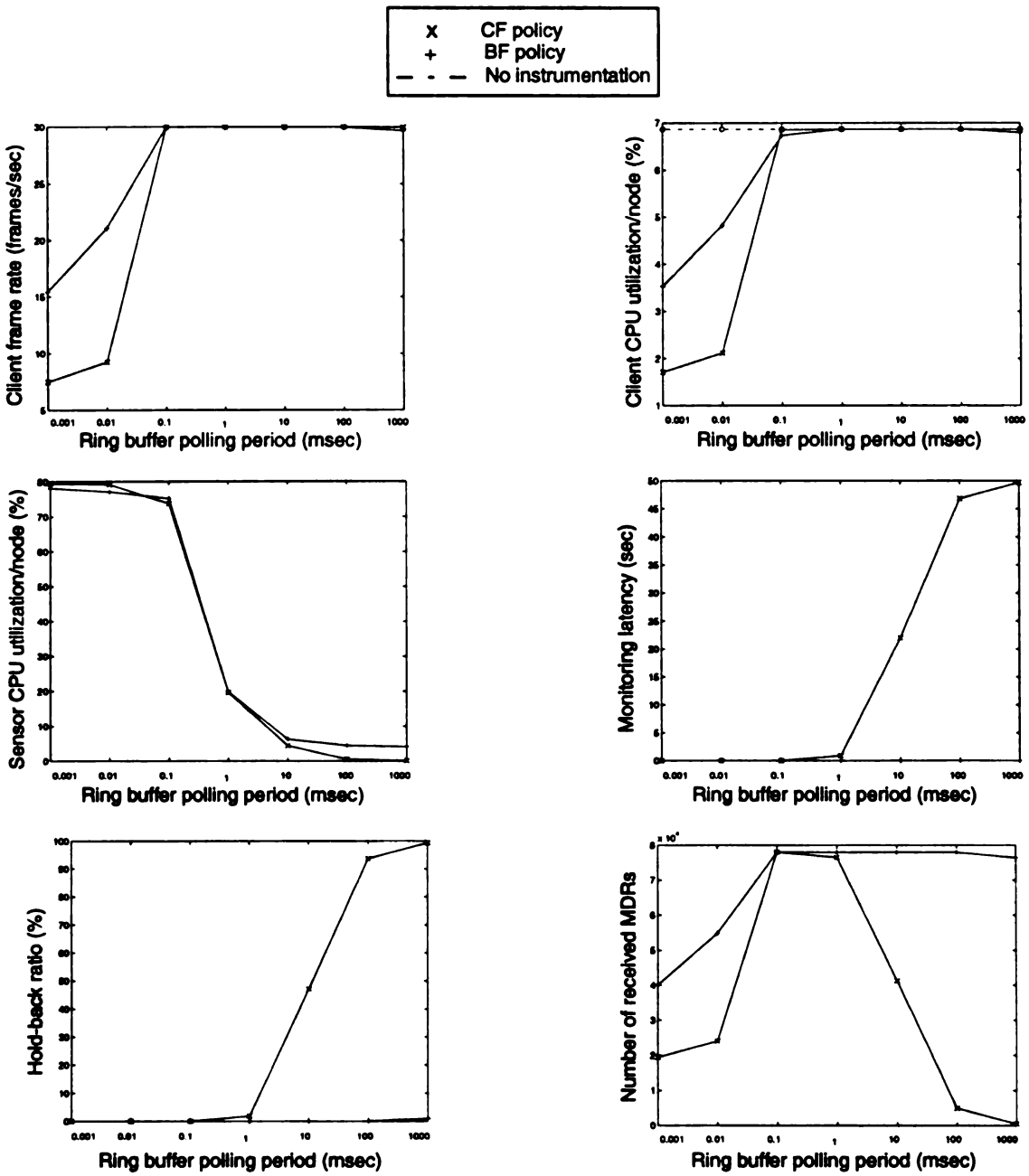


Figure 6-21. QoS and IS metrics for variable ring buffer polling periods under the CF and BF policies of forwarding instrumentation data to the JEWEL collector (number of nodes = 8, rings buffer size = 4000, simulation time = 100 sec, logarithmic scale for ring buffer sampling period).

the collector, at all ring buffer polling periods. The default technique of JEWEL external sensor to busy-wait for any event records to arrive in the shared memory correspond to very small polling periods.

- Although the BF policy outperforms JEWEL's default CF forwarding policy at shorter polling periods, it is advisable to poll the shared memory segment after relatively longer polling periods to avoid intrusion to the real-time behavior of the application.

From the results presented in Figure 6-21, it appears that a suitable polling period that maintains a 30 frames/sec rate keeps the application client CPU usage close to its base-line (no instrumentation) value, keeps sensor CPU overhead low, and maintains steady instrumentation data flow is at least 1 msec.

Next, we select a suitable size for the shared memory segment that temporarily holds the event data arriving from the internal sensor. Figure 6-22 presents only the data flow related metrics of interest under variable ring buffer sizes and CF and BF forwarding policies.

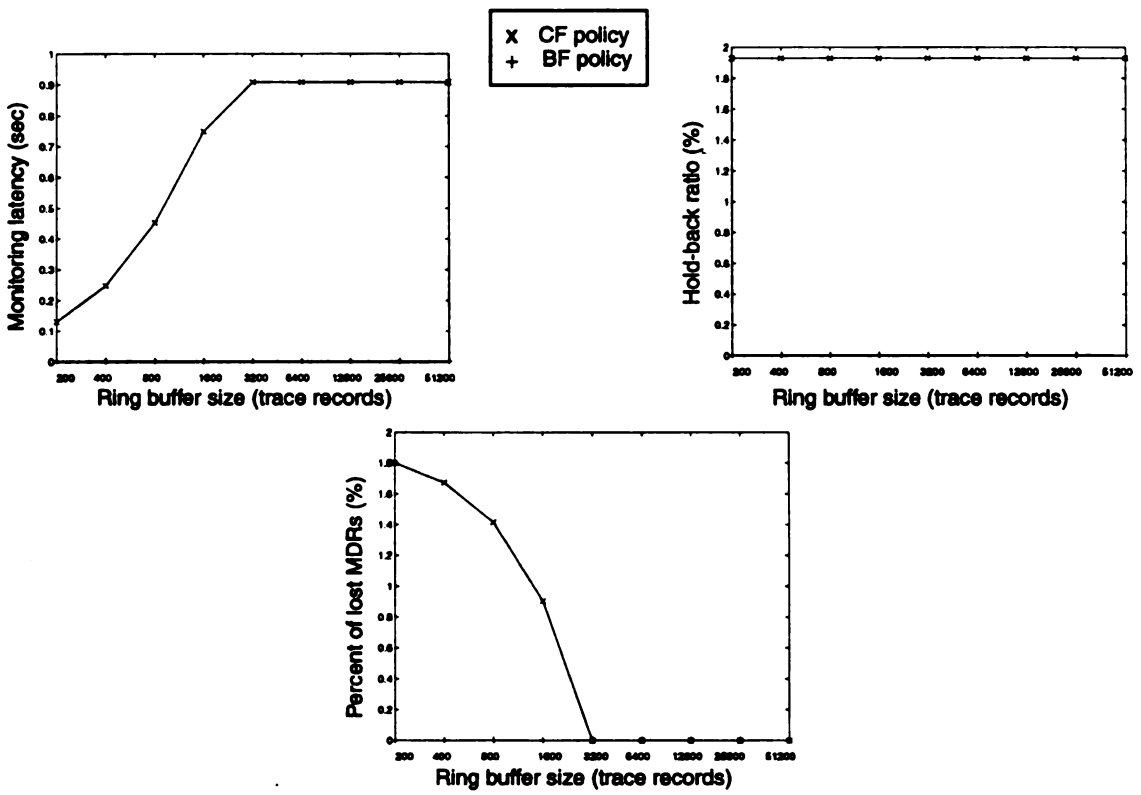


Figure 6-22. IS metrics for variable ring buffer sizes under the CF and BF policies of forwarding instrumentation data to the JEWEL collector (ring buffer sampling period = 1000 msec, number of nodes = 6, simulation time = 100 sec, logarithmic horizontal scale).

- None of the metrics, other than data flow metrics, changes with the ring buffer size, because the application does not block if this buffer becomes full; the internal sensor simply drops the event data.
- The default shared memory segments size of 128K bytes allows about 4000 event records, which appears to be an appropriate size under either of the two forwarding policies. Within each cycle of client process execution, it does not generate more than six

event records. There are at the most 30 cycles that a client process completes per seconds; therefore, the average number of events waiting in the shared memory ring buffer do not grow more than 180 records.

- Monitoring latency is low at small buffer sizes because a number of event records are not generated by the internal sensor in the first place.

Based on the results presented in this section, we conclude that the BF policy is desirable at all ring buffer polling periods and sizes. Additionally, the default ring buffer size of 128K bytes (or 4000 event records) is sufficient for the purposes of this application. Therefore, for the rest of the cases presented in subsequent sections, we use the BF forwarding policy and a shared memory segment of size 4000 records.

What is a suitable adaptation policy and how it should be scheduled?

We compare two adaptation policies: *static polling period* (SPP) and *dynamic polling period* (DPP). JEWEL IS parameter changes due to these policies can be implemented using one of the two possible types of scheduling: *centralized* or *distributed* scheduling. Adaptation is made possible through the resource management component that examines the state of the entire system after each controller sampling interval and makes decisions that are applied to all the nodes in the system using centralized scheduling approach. Under distributed scheduling options, the resource manager agents collect and analyze the states of the local system using JEWEL sensor data and schedule any needed actions for their node. The goal of the simulation-based evaluation presented in this section is to select a combination of an adaptation policy and scheduling policy that can better meet the QoS requirement of a constant frame rate (30 frames/sec) and JEWEL sensor CPU overhead constraint (to be less than or equal to 10% of CPU usage). Additionally, we have to consider that the selected policy also maintains a steady flow of MDRs to the collector.

We begin with an evaluation of the effect of controller sampling period on the performance metrics of interest. The results are presented in Figure 6-23.

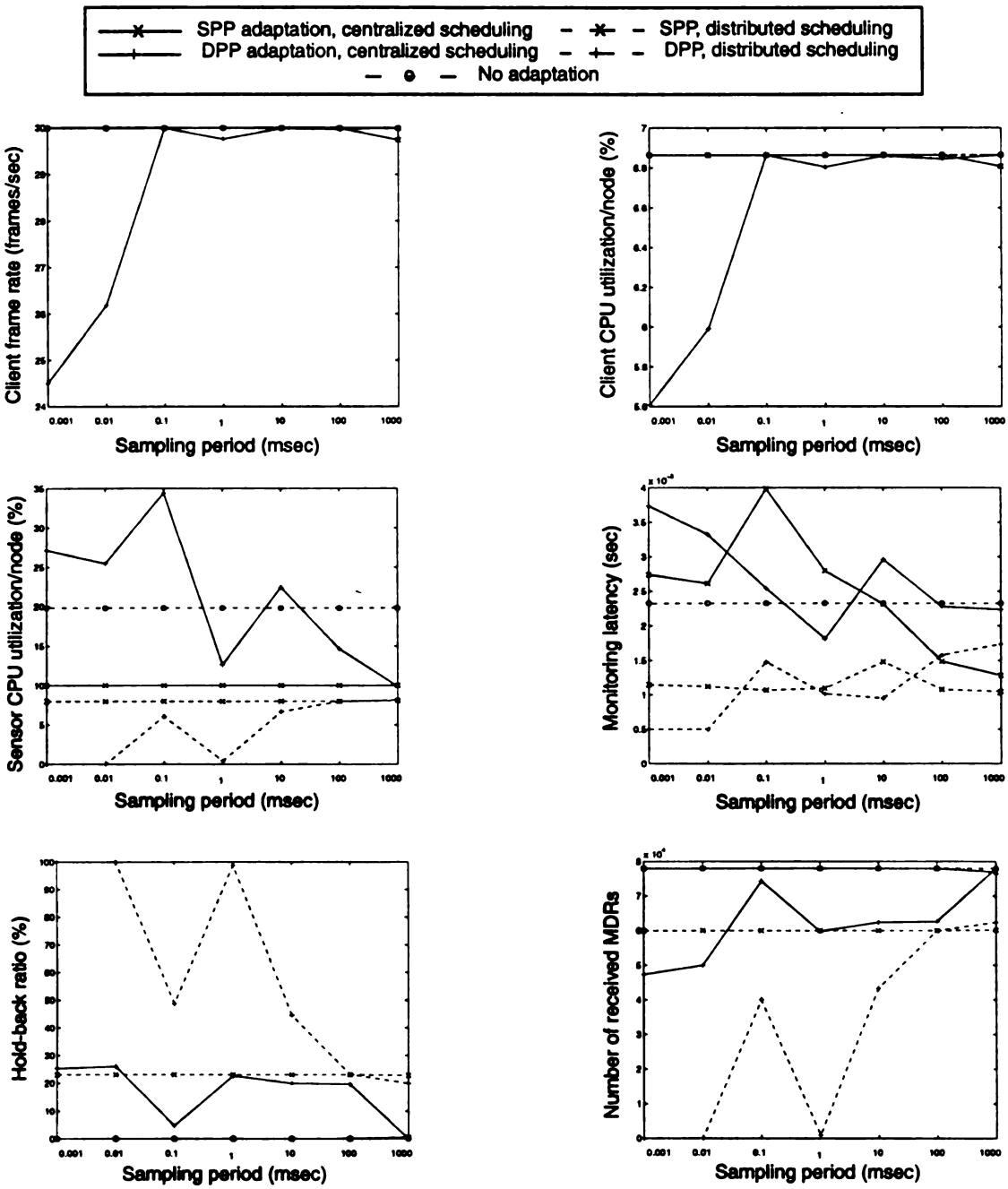


Figure 6-23. QoS and IS metrics for variable controller sampling periods under the BF policy of forwarding instrumentation data to the JEWEL collector (number of nodes = 8, ring buffer size = 4000, simulation time = 100 sec, logarithmic scale for sampling period).

- Under both distributed and centralized scheduling policies, the SPP adaptation policy outperforms the DPP adaptation policy in terms of almost no intrusion to the real-time behavior of the application and meeting constraints on sensor CPU utilization. The reason is the incremental changes in the ring buffer polling period under the DPP adaptation. Particularly at shorter controller sampling periods under the centralized scheduling, the DPP cannot attain a "steady state" as resource manager has to change the polling period at every sampling instant (up or down by a factor of two) resulting in

low frame rate and high sensor CPU overhead. These changes result in only small improvements and the system continues to require further adjustments resulting in larger variability in the metrics. On the other hand, static adaptation makes only one of the two changes: (1) turn the instrumentation off at all nodes (under centralized scheduling) or at local node (under distributed scheduling) if the current system state shows that the constraints are not being met; or (2) turn the instrumentation on if the system starts meeting the constraints. This policy results in more predictable (in terms of lesser variability) and steady behavior of the real-time application and instrumentation system.

- The benefit of using adaptive control is clear from the plot of sensor CPU utilization, which shows a reduction in CPU overhead, under the static adaptation, to 50% of its value under no adaptive control.
- In case of distributed scheduling, both frame rate and sensor CPU overhead requirements are met by both adaptation policies because unlike resource manager's operation under centralized scheduling local resource manager agent's decisions are not influenced by the state changes at any other node. The centralized scheduling approach works better for greater data flow (low monitoring latency and hold-back ratio and larger number of received MDRs). In fact, the centralized scheduling may be the only option if the goal of the controller were to maintain the monitoring rate at a desired level. However, the SPP adaptation with distributed scheduling meets the QoS and IS overhead requirements set forth in this study.

Adaptation of the instrumentation system starts with the initial ring buffer polling period specified at the beginning of the execution. Working in a closed loop, the adaptive controller (centralized or local) tries to adjust this parameter based on the runtime measurements. The adaptive controller is effective only if its performance is not overly sensitive to the initial ring buffer polling period value. Figure 6-24 shows the effects of different initial ring buffer polling periods under SPP and DPP adaptation policies using centralized and distributed scheduling schemes.

- The overall benefit of using adaptation, regardless of the controller design, is clearly demonstrated by the frame rate and sensor CPU overhead values that are within the required ranges. The SPP adaptation under distributed scheduling can better meet the real-time application QoS and IS overhead requirements due to the same reasons described above for the case shown in Figure 6-23.
- Adaptation is not sensitive to the initial ring buffer polling period values except using DPP adaptation, which shows unpredictable behavior in any case.

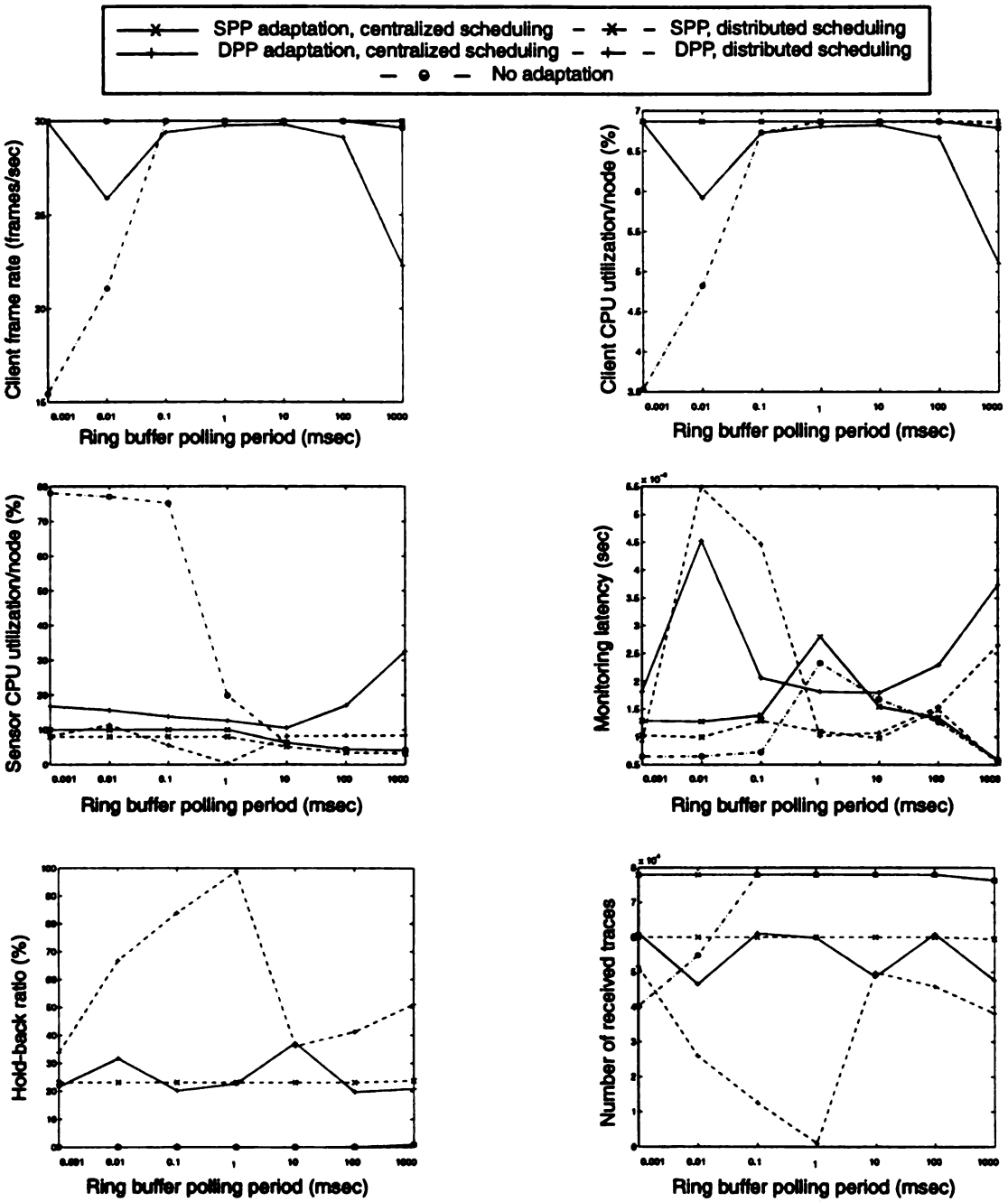


Figure 6-24. QoS and IS metrics for variable initial ring buffer polling periods using static and dynamic adaptation policies under centralized and distributed scheduling (number of nodes = 8, controller sampling period = 1 msec, ring buffer size = 4000, BF policy, simulation time = 100 sec, logarithmic scale for ring buffer polling period).

- Note that the adaptation is achieved on the cost of data flow. In case of no adaptation, the monitoring latency and hold-back ratio are very small, particularly at shorter polling periods (at the cost of excessively high sensor CPU utilization resulting in a drop in frame rate). The SPP adaptation with centralized scheduling is the closest match with the data flow characteristics of the case with no adaptation.

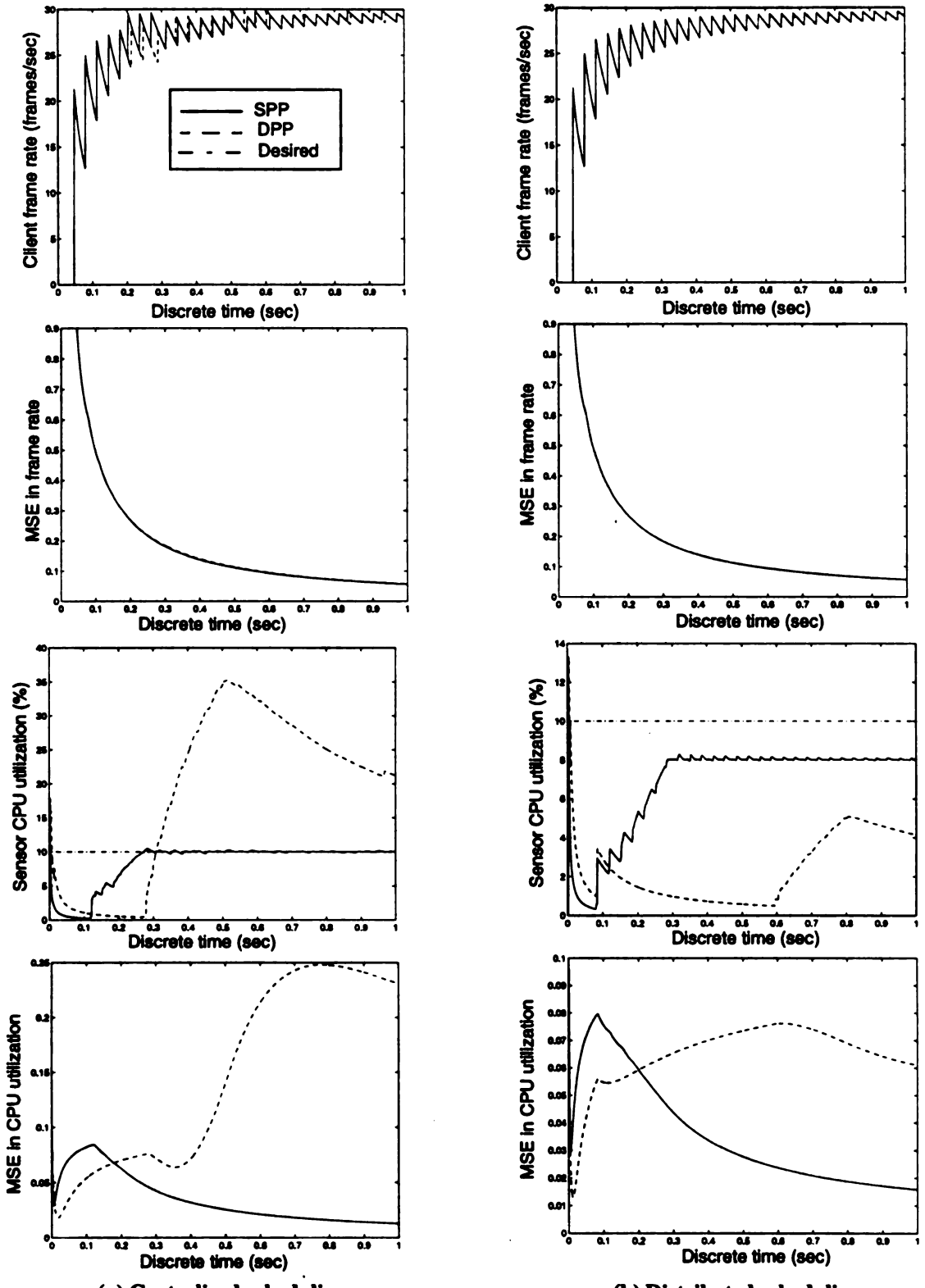
The results reported in this subsection indicate that SPP adaptation outperforms DPP adaptation under both distributed and centralized scheduling policies by consistently meeting the frame rate and sensor CPU utilization constraints (due to lesser variability) while maintaining a steady flow of MDRs to the collector. Due to the localized nature of constraints, they are efficiently met using distributed resource manager agents (i.e., distributed scheduling). If the nature of the constraints were global (such as a limit on monitoring latency), centralized control through a resource manager (using centralized scheduling) would be preferred. To meet the requirements identified in this paper, SPP adaptation of the instrumentation system with an initial polling period of 1 msec or longer and a controller sampling period of about 1 msec are suitable for the video application.

How does the adaptive controller perform?

The evaluation of the adaptive control for the JEWEL instrumentation system in the preceding parts of this subsection was based on meeting the application-specific QoS and IS overhead constraints. However, there are well-known performance metrics to directly evaluate the performance of an adaptive controller, such as the difference between the desired and actual response of the system [222]. One such metric is *mean square error* (MSE). Variation of the MSE with time lends insight into the temporal characteristics of the adaptive behavior of the controller and its usability for a particular application.

We monitor the frame rate and sensor CPU utilization during a simulation using a polling period equal to the sampling period used by the resource manger and its agents to observer the system state. We also measure the MSE values at these points. The results are presented in Figure 6-25.

- The results corresponding to the desired frame rate value do not show any clear differences between the SPP and DPP adaptation or centralized vs. distributed control.
- In all the cases, the frame rate reaches close to the required 30 frames/sec within about 0.5 sec after the start of simulation and mean square error drops to about 10% of its initial value. This behavior is in part due to the dependence of frame rate on the CPU time



(a) Centralized scheduling (b) Distributed scheduling

Figure 6-25. Performance of the adaptive control system using the SPP and DPP adaptation policies under (a) centralized scheduling and (b) distributed scheduling (number of nodes = 8, ring buffer polling period = 1 msec, controller sampling period = 1 msec, simulation time = 1 sec).

that the client process can get. If the CPU usage by the local external sensor does not causes contention for the client's CPU usage, the rate of frames processed by the client process should remain close to the required value.

- For adapting to the required value of the JEWEL sensor CPU utilization, the adaptive controller exhibits different behavior for different adaptation and scheduling policies. For both types of scheduling policies, the controller does not attain a "steady state" under the DPP adaptation policy as the CPU utilization remains different from the required value. Thus the error is also larger for the DPP case. However, under the distributed scheduling the CPU utilization is lesser than the required value and error is smaller. It means that distributed scheduling for the adaptive control of frame rate and sensor CPU utilization is desirable to maintain a smaller error of adaptation with respect to the desired system response.

6.4.3 Feedback to the Developers

The modeling and evaluation process for the JEWEL IS and adaptive controller is being conducted at a time when a prototype version of the video application with initial JEWEL instrumentation exists. Any feedback to the developers regarding the application of JEWEL IS and adaptively controlling its overhead can be useful. Based on the simulation-based evaluation, the following specific results can be provided to the developers:

1. Use of BF forwarding policy is better to keep the intrusion to the real-time behavior of the application low and maintain a steady flow of data to the collector. Moreover, use of polling with periods of 1 msec or longer rather than busy-waiting for the event records to arrive in the shared memory ring buffer is desirable for low sensor CPU overhead.
2. A ring buffer size to hold about 4000 event records is enough for all the practical cases of IS usage for this application.
3. A resource manager sampling period of close to 1 msec is a reasonable compromise between responsiveness of the controller and its "unsteady" behavior due to high variability.
4. For all practical purposes, SPP adaptation policy should be used because it meets the criterion of low intrusion to the real-time behavior of the application, low sensor CPU overhead, low variability (i.e., predictable), and smaller adaptation errors.
5. Distributed scheduling is a better choice over centralized scheduling to meet the QoS requirements of this application and IS overhead constraints of JEWEL IS.

These recommendations are well-received by the developers who are in the process of implementing the resource manager to control the IS as well as the application. Some of

the measurement-based results obtained from an early prototype of JEWEL customization for the video application are presented in the following section.

6.4.4 Experimental Validation

Customization of JEWEL IS for the video application is at its initial development stages. We have a prototype version of the JEWEL external sensor that is being used with the application for collecting runtime data. The initial version of this instrumentation used the default scheme of JEWEL IS: it used a busy-wait technique to collect the event data from shared memory ring buffer. Based on the feedback of the modeling and evaluation presented in this paper, the BF policy with polling scheme was implemented in the external sensor.

6.4.4.1 Experimental Setup

In order to measure the improvement, we transferred trace records at a rate of 180 records per second (corresponding to generating 6 records per cycle of application client function and 30 such cycles per second). Corresponding to the 100 second time limit used in simulations, we ran these measurement-based experiments until 18,000 MDRs are transferred. The application and JEWEL IS were run on a network of Sun Ultra-1 workstations connected through a high-speed Ethernet. We use two polling period values: 1 μsec and 1 msec. These values correspond to the time that external sensor spends polling the ring buffer in the shared memory. The first value is close to the "busy-wait" case while the second value is the value of polling period recommended to the developers as a result of evaluation presented in this paper.

6.4.4.2 Evaluation

Figure 6-26 compares the CF and BF policies using two polling period values. It is clear that the use of polling scheme has its advantage over the default "busy-wait" scheme as the sensor CPU overhead considerably reduces in the former case. At a polling period of 1

μsec, the CPU overhead under the BF policy is higher than the CF policy because the external sensor requires greater CPU time to collect multiple event records under the BF policy. Under the CF policy, the external sensor collects only one event record at a time even though others may be waiting. However, at a polling rate of 1 msec, the CPU overhead under the BF becomes equal to that of the CF and balances its capability to collect and forward multiple MDRs with its CPU overhead. Therefore, the measurement results indicate that use of 1 msec polling period under the BF policy is desirable for low CPU overhead and maintaining steady data flow to the collector, as predicted by the simulation results.

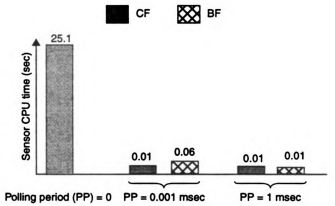


Figure 6-26. Comparison of JEWEL sensor CPU overhead measurements under the CF and BF policy using two polling period values. (total measurement time = 100 sec)

6.5 Summary of IS Evaluation Results and Discussion of Methodology

We applied the model-based evaluation methodology to three reference ISs in the preceding sections. In this section, we first summarize the important results of these evaluation efforts. Subsequently, we discuss the methodology itself in suitability for IS evaluation. In particular, we present a systematic approach of developing a ROCC model and evaluating it for evaluating the design and configuration alternatives for a particular IS.

6.5.1 Summary

A summary of the key results of evaluating three reference ISs is given in Table 6-12. These ISs were modeled to address their domain-specific objectives. In this chapter, we evaluated these models to quantitatively determine the metrics of interest under specific "what-if" scenarios. However, the summary of results listed in Table 6-12 is of qualitative nature to be directly useful for the tool developers.

Table 6-12. Summary of key results of evaluating selected *ISs*.

IS	Key Evaluation Results
PICL	• The FAOF flushing policy outperforms the FOF policy in terms of reducing the frequency of flushes and perturbation; and • compared to the FOF policy, it is not trivial to implement the FAOF policy on a loosely-coupled, distributed-memory parallel system.
Paradyn	• The BF policy should be implemented as a default policy to schedule data forwarding operations because it outperforms the CF policy; • in the case of an SMP, use of multiple daemons per node represents a trade-off between more samples received by the main process and additional contention for system resources; • binary tree forwarding should be used on an MPP system due to its superior scalability characteristics compared to direct forwarding; and • specific application characteristics, such as frequency of barrier operations on an MPP system, may affect IS performance, which may in turn impact the instrumented application.
JEWEL	• Use of BF forwarding policy is better to keep the intrusion to the real-time behavior of the application low and maintain a steady flow of data to the collector; • for all practical purposes, SPP adaptation policy is desirable because it meets the criterion of low intrusion to the real-time behavior of the application, low sensor CPU overhead, low variability (i.e., predictable), and smaller adaptation errors; and • distributed scheduling is a better choice over centralized scheduling to meet the QoS requirements of the video conferencing application and IS overhead constraints of Jewel IS.

6.5.2 Discussion

In this subsection, we evaluate our IS design, modeling, and evaluation methodology in general instead of the specific results. Our objective is to highlight the strengths and

weaknesses of this approach and provide a set of guidelines to conduct a performance evaluation study of an IS.

Results related to the use of ROCC modeling approach indicate that its accuracy depends on the level of details captured in the model. If we capture only the coarse-grain details of the IS behavior, the measurement process to collect model parameterization information becomes a simple task. However, this simplicity results in reduced accuracy of the model predictions. On the other hand, it is possible to capture minute details of the IS in the simulation of the ROCC model. The major problem with this approach is the availability of the low-level system measurements that are necessary to parameterize a detailed model. Obtaining these measurements is a difficult task especially when the IS is not yet developed or at an early prototype stage. Therefore, the analyst has to make a compromise between the two extremes of modeling coarse-grain IS details and exact details according the objectives of a particular study.

The notion of an early feedback to the IS and tool developers is another important aspect of our modeling and evaluation efforts. Timely feedback to the developers at an early prototype stage helps them to choose suitable system configuration. If the IS does not undergo such an evaluation, it is possible that performance problems are discovered for certain cases that were not taken into the consideration by the developers. Using a model of an IS, it is simpler to exercise the functionality of different modules of an IS and improve the design based on this feedback. However, it is not practical to subject an IS (or its prototype) using a variety of practical workloads. Additionally, testing and benchmarking of an IS is still a relatively unexplored area.

Our experience shows that a formal performance study of an IS usually results in a better understanding of the actual system. Several of the policies that we proposed for the reference ISs are a result of this modeling and evaluation process. It is difficult for a tool developer to explore a number of possible IS management policies. Collaboration between

the developers and performance analysts can result in useful feedback to the developers to explain the strengths and weaknesses of different management policies.

Based on the modeling and evaluation experiences of the reference systems, we recommend the following set of guidelines to conduct an IS performance study at the time of its design and development:

1. As a first step, it is essential to determine the objectives and scope of the study. If the IS is being designed for an HPC tool, its overhead is an important consideration. On the other hand, if it is being designed for an embedded real-time system, its intrusion to the real-time behavior is of prime importance. Similarly, ISs for other systems may have their own domain-specific requirements that should determine the objectives of an IS modeling and evaluation study.
2. There should be an initial design of the IS that depicts all of its modules and their functionalities.
3. Based on the information about the SUT and IS modules, identify the system resources that are shared between the two types of modules (processes).
4. Characterization of the workload. This should initially be based on the coarse-grain functions of the SUT and IS processes. Workload characterization process also includes collecting relevant measurements from an initial prototype of the IS, analyzing these data, and fitting appropriate distributions. If a prototype of the IS does not exist, we can use empirical workload characterization for the IS modules to allow the modeling and evaluation process to proceed.
5. Steps 3 and 4 should result in a ROCC model for the SUT and IS combination.
6. We can optionally derive analytical results for the ROCC model, especially in the case of ISs that have not yet reached an early prototype stage of their development.
7. Detailed simulation-based evaluation of the ROCC model. The results of the simulation study should be validated in an intuitive manner. For instance, variations of a specific operating condition may be expected to affect a particular metric. If the simulation results follow that pattern, it is likely that simulator is functionally accurate (at least to certain extent). Analytical results can also help validate the simulator.
8. IS development according to the evaluation results.
9. Selective measurement-based testing of the IS to validate the predictions of the modeling and evaluation study.

Some of the above guidelines such as determining the objective of the study are generic while others are noted as a result of modeling and evaluation experiences with the reference ISs.

In this chapter, we concluded the performance studies of the PICL, Paradyn, and JEWEL ISs. Additionally, we elaborated on the choice of analytic and simulation-based evaluation techniques for IS model. Results of evaluating three reference ISs were presented with all appropriate details and feedback to the developers, in each case, was summarized. Finally, we listed a number of steps to develop a ROCC model for an IS and use it for investigating specific performance-related questions.

Chapter 7

Deliverables of the Research

In this chapter, we present and discuss three outcomes of this research:

1. evaluation of extant, well-known ISs and feedback to the developers and users;
2. design and implementation of a simulator to analyze the ROCC models of ISs; and
3. design and implementation of the Vista IS.

The latter two outcomes are "deliverables" of this research because they are readily usable for extending this work. This chapter can be helpful for practically using the ROCC simulator and the Vista IS. The ROCC simulator can be used for modeling and evaluating an IS while the Vista IS can be used for collecting runtime information from a distributed system.

We presented specific results of evaluating three reference ISs in Chapter 6. In this chapter, our objective is not to consider the specific details about each case study but to recognize the outcome of the overall effort and its impact on the state-of-the-art. Two different simulators were developed for the ROCC models of Paradyn and JEWEL ISs. However, there are number of similarities between the designs of the two. Based on the same design principles, we consider the possibility of extending these simulators as a tool for exploratory analysis of different ISs. Finally, we consider the design and implementation of the Vista IS, which can be considered a link between the design and synthesis of an IS envisioned by this research and its future directions, discussed in Chapter 8.

7.1 IS Evaluation Methodology

Realization of an IS for a target system is a non-trivial process requiring many person-hours of software development effort. Moreover, evaluation of an IS by users upon its release may lead to requests for corrections, changes, or enhancements in its functions. In contrast, preliminary evaluation of an IS using the modeling-based approach can be applied to ensure that specific requirements of a target system are met prior to the investment in programming effort. This process is likely to deliver better performance and be less costly for the target system.

Performance evaluation studies presented in this dissertation focused on three ISs. We applied our model-based evaluation methodology to an existing IS (PICL) and two ISs at different stages of development: Paradyn IS and JEWEL IS.

Evaluation of an IS requires low-level information about the design and implementation of an IS from tool developers. However, such a study during the early stages of tool development can lead to a better design of the IS, which can ensure lower overhead to meet its specifications. This is a worthwhile effort because various instrumentation data consumers, such as visualization, modeling and prediction, debugging, steering, etc. tools, will be successful only if proper framework exits for developing an IS to support them.

The purpose of the initial feedback provided by a modeling- and simulation-based study is to answer generic, performance-related "what if" questions. It is both advisable and practical to relax the accuracy requirements at this stage. Achieving a high degree of accuracy is costly due to the complexity of an instrumentation system. One lesson that we learned by modeling the Paradyn IS is that an approximate simulation model, following the gross behavior of the actual instrumentation system, is sufficient to provide useful feedback. At an early stage of modeling the Paradyn IS, we arbitrarily parameterized the model based on information provided by the developers [214]. The case study presented in this dissertation used a more detailed workload characterization based on measurement

data. Although we enhanced the scope of "what-if" questions in this study, e.g., to include the SMP and MPP architectures and factors such as forwarding policy, this more detailed study does not contradict the earlier study that used an approximate model [218]. Obviously, with an approximate model, the analyst relies on correlating the simulation results with some intuitive explanation of the system behavior. Unfortunately, approximate modeling results are open to speculation without extensive workload study based on actual data.

Instrumentation system design and maintenance are difficult and costly since supported IDCs may undergo frequent modifications for new platforms and applications. The HPCC community, for instance, has recognized the high cost of software tool development [151]. As with any large software system, a software tool environment should be partitioned into components and services that can be developed as off-the-shelf, retargettable software products. Due to the generic nature of an IS, which consists of components and services for runtime data collection and management, it is an excellent candidate for modular development [216]. Off-the-shelf IS components will need to meet a number of functional as well as non-functional requirements. The modeling and evaluation methodology of this dissertation research is a necessary steps toward implementing high-performance, well-specified off-the-shelf IS components.

7.2 The ROCC Simulator

ROCC simulators for Paradyn and JEWEL IS studies, presented in this dissertation, are developed as C++ programs using *task* library. The simulation framework of the task library provides four types of base classes: tasks, objects, queues, and a scheduler. The task library allows the simulation of concurrent activity by deriving the corresponding object from its base class *task*. The object thus derived becomes a (simulated) thread, which can be controlled either through the *scheduler* using its time-keeping facility or its explicit controls for executing, suspending, or deleting the threads. Concurrent tasks (i.e.,

simulated threads) can interact using message-passing, first-in, first-out *queues*. Messages can be derived from the base class *object*.

Figure 7-1 presents the design of a ROCC simulator based on the task library. The timing and controlling functions are handled by the task library itself and user does not have to explicitly implement them. We model the shared system resources as well as multiple processes on a node as classed derived from the task base class. Using a convenient mechanism of instantiating the objects, multiple nodes can be modeled by replicating the resources and processes of one node for all other nodes. These nodes can be identified by their unique identifiers and different seeds for the purposes of (approximate) statistical independence among nodes. Processes send occupancy requests to the resources; a *request* is a class derived from the *object* base class. Different processes interact with one another using objects of another class called *message*, which is also derived from the *object* base class. Messages and requests go to different *queues*, which are supported by the task library. In addition to these classes, we define additional classes for generating random numbers according to a number of distributions and collecting results. This is a flexible framework for simulating a ROCC model that can easily be extended for different ISs. The major differences are due to different workload characterizations. Therefore, the workload classes that are derived from the task base class has to be modified for every IS.

In order to use the above setup for simulating the ROCC models for Paradyn and JEWEL ISs, we had to customize it for each of these two cases. However, it is possible to extend this setup as a tool with a convenient GUI-based mechanism to configure different components according the needs of a particular IS. This extension is left for the future work on the ROCC simulator.

7.3 The Vista IS

Vista is a part of an integrated tool environment being developed at Michigan State University, called PG^{RT}, for instrumenting and testing distributed, real-time systems [145].

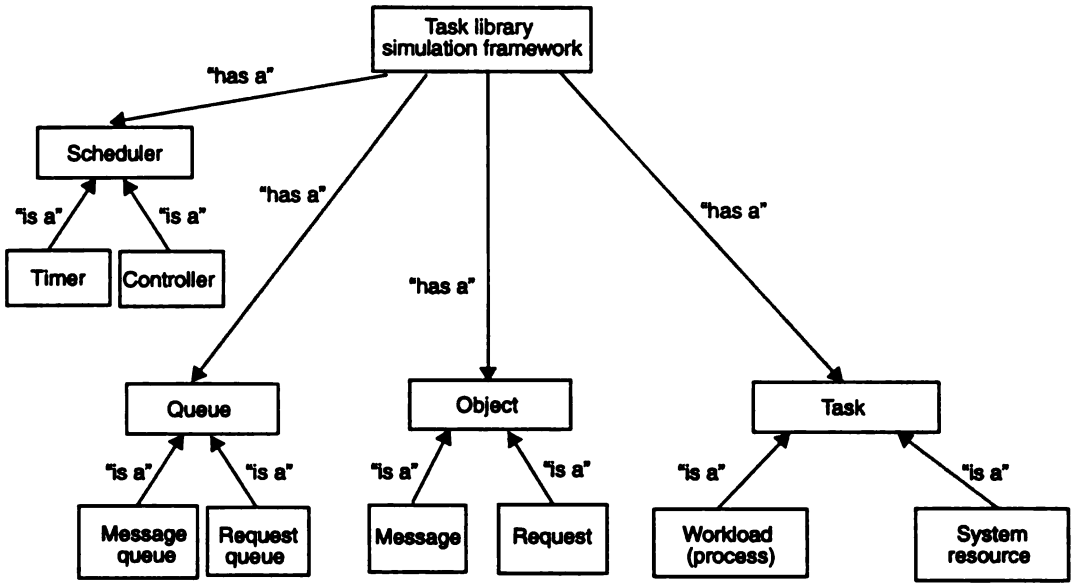


Figure 7-1. Design of a ROCC simulator using the *task* library.

We previously used a similar environment to integrate specialized performance analysis tools with generic data analysis and visualization tools to perform off-line performance analysis [210]. Presently, we are using the Vista IS for supporting on-line as well as off-line performance analysis and visualization of real-time tasks running on a cluster of workstations based testbed for real-time systems. Vista IS controls data collection, forwarding, processing, and dispatching to the environment. Integrated environment consists of a set of custom and off-the-shelf tools for visualization, performance analysis, and real-time task scheduling. User interacts with the environment as well as the Vista IS through a front-end supported by the PG^{RT} environment. Two types of applications are instrumented using the Vista IS: one that simulates general-purpose real-time systems and tasks and another that emulates a real-time system using PVM message-passing library [63] on a cluster of workstations.

7.3.1 Overview of Vista IS

Figure 7-2 illustrates the functionality of the Vista IS as a part of the PG^{RT} environment. In order to collect runtime information, Vista library is linked with the distributed application

program. Before using the Vista library interface to collect and forward any data, local modules of the IS are initialized by every distributed process. Subsequently, the events of interest can be captured by calling appropriate library functions that forward them to the tools in PG^{RT} environment. Instrumentation is event-driven, and instrumentation data related to an event of interest are forwarded without local buffering. The data may arrive out of order at the Vista instrumentation system manager module, which is the (logically) central part of the distributed IS. To avoid problems due to the lack of a global clock, we use the technique of assigning logical time-stamps, as implemented by VIZIR [77]. This ordered and time-stamped trace record is consumed by the tools in the PG^{RT} environment for analysis purposes. Vista IS generates PICL-formatted trace records to be able to visualize them with ParaGraph tool. In addition to event-driven tracing, Vista IS also allows the users to generate application-specific trace records in PICL format. These trace records are forwarded to the environment without any modifications. Such data collection and forwarding is implemented in Vista to support user-defined events that may have special significance for a particular real-time system under test.

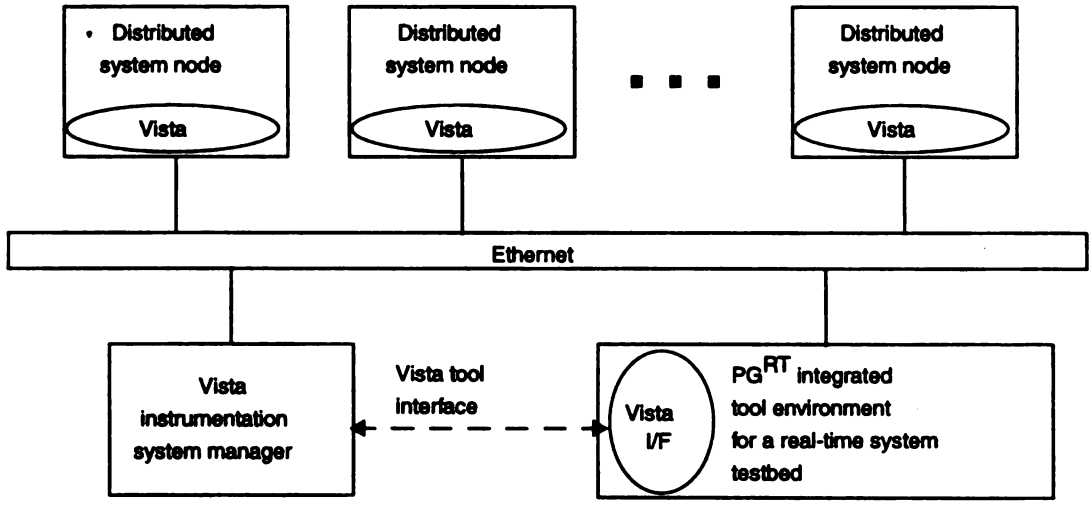


Figure 7-2. Overview of Vista IS functionality to support data collection needs of an integrated tool environment for testing distributed, real-time systems.

7.3.2 Domain-Specific Requirements of the Vista IS

Vista IS supports a testbed for a distributed, real-time system, which is not a real system. Although actual real-time systems operate under strict timing constraints, a simulator or an emulator often uses simulated time to schedule the tasks. Therefore, the overhead of IS modules to the application do not affect the measurements that are based on simulation time instead of real time. A more important requirement of this setup is to ensure a steady flow of instrumentation data to the integrated environment to support their analyses. Thus, the behavior and performance of the Vista instrumentation system manager should be a focus of a modeling and evaluation based study of the IS. Additionally, system resources are shared between the ISM and tool environment (i.e., IDC), therefore, the Vista IS modules should have minimum intrusion to the tool environment. As the environment is responsible to interact with the user and graphically present the data, it is required to be highly responsive. Therefore, ISM tasks should be scheduled such that they do not block a shared system resource while waiting for instrumentation data to arrive from the distributed system nodes.

7.3.3 Design of the Vista IS

The Vista framework is developed in C++ and utilizes the typical features of object-oriented languages to enable the users to develop domain-specific ISs. This framework consists of four *abstract* and several *base* classes for customized configuration of various IS modules. The Vista *IS* framework has four abstract classes to define instrumentation data, timers and clocks, transfer protocols, and data structures for buffering of instrumentation data. There are two classes derived from the *instrumentation data* abstract class: *event data* and *program data* classes. A user can derive further classes from these base classes to represent application-specific data. The *timer* abstract class has two base classes, one to define clocks to measure elapsed real time and the other to define the sampling intervals. The *transfer protocol* abstract class has three classes derived from it using various application-level transport facilities based on available operating system

support. These classes use X library calls, remote procedure calls (RPC), and PVM library functions to implement various types of communication among IS modules. An important constituent of any IS module is a data structure that temporarily stores the instrumentation data. The *instrumentation data structure* abstract class supports three types of data structures: first-in, first-out (FIFO) queues, priority queues, and doubly linked lists. These classes of the Vista framework are summarized in Figure 7-3.

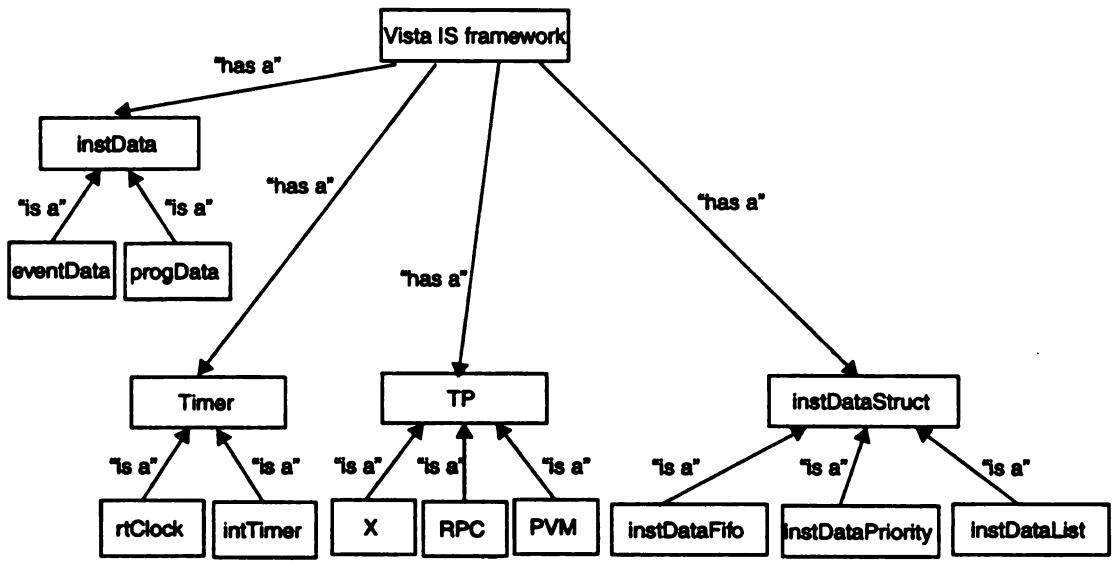


Figure 7-3. Abstract and base classes in the Vista framework.

The Vista framework supports a layered approach to develop an IS. At the lowest level, it provides abstract classes to configure the structure of an IS using appropriate TPs. At the next higher level, it provides various convenience functions supported by Vista library for rapid prototyping of the design. The next higher level is the Vista toolkit. It supports several pre-implemented and -configured modules of various ISs that can be used as "plug-and-play" components in different integrated environments. Figure 7-4 depicts the details of these layers and their relationship with applications development and available tool technology.

The specific features of this framework include:

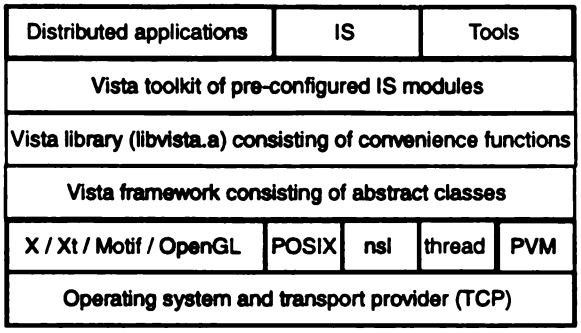


Figure 7-4. Tool development using Vista framework and class library.

1. object-oriented design that ensures domain-specific and application-specific development;
2. a rapid prototyping technique to support design of performance-critical ISs;
3. QoS at application level using separate threads for IS-related communication via the TP;
4. compliance with the POSIX standard for system-level tasks, such as multi-threading, scheduling, synchronizing, timing, etc. to ensure portability; and
5. C++ based application programming interface (API), which is independent of the implementation.

It should be noted that certain features are supported individually by certain technologies, for example: object-oriented design, rapid prototyping, and API, by languages and compilers; QoS, by operating systems and network architectures; and portability across operating systems, by standardization efforts. However, the integration of these features as a part of the Vista framework makes them transparent to the user, who can then concentrate on the specific issues related to a particular application domain.

7.3.4 Vista IS Modeling and Evaluation

We present an initial effort of modeling the Vista IS, which is focused primarily on the design of an event ordering part of its ISM. Modeling and evaluation of the Vista IS modules is an on-going project and a part of the future directions of the work presented in this dissertation.

7.3.4.1 IS Modeling Issues

The Vista LIS captures instrumentation data from an application process by invoking its instrumentation library functions. Instrumentation is event-driven, and data related to an event of interest are forwarded to the ISM without local buffering. The size of this data structure is kept very small to avoid excessive communication delays. The data are received and ordered by the ISM. To avoid problems due to the lack of a global clock, we use the technique of assigning logical time-stamps, as implemented by VIZIR. If an arriving event is in correct causal order, it is assigned a logical time-stamp and stored in an output buffer. When a tool selected by the user is ready, the processed event information is dispatched to the tool from the output buffer. If the arriving event is not in causal order, it is added in one (or multiple) input buffer(s) to reconstruct the causal order of the data before dispatch to a tool. For this type of ISM, it is desirable that input buffer management and event ordering are efficient, so that the (monitoring) latency between the arrival of data to the input buffer and the dispatch of data to the output buffer is minimized. Otherwise, the logical time-stamp will become less accurate and may even perturb the visualizations presented by the tools.

7.3.4.2 IS Management Issues

The ISM is modeled because its performance is deemed critical to obtaining correct and efficient presentation of program behavior from tools in an integrated environment. The specific objective of this modeling effort is to guide the developers in selecting one of two possible configurations of the ISM that will guarantee regular receipt of instrumentation data with minimum delays. The two possible configurations are: *Single Input buffer, Single Output buffer* (*SISO*) and *Multiple Input buffers, Single Output buffer* (*MISO*). As the names suggest, the *SISO* configuration uses one input buffer to store out-of-order instrumentation data from all the processes, whereas the *MISO* configuration has one buffer per each application process. These configurations are commonly used in on-line ISs, for example, Falcon uses the *MISO* approach [71].

7.3.4.3 IS Model

The *ISM* is modeled as a network of two single-server queues. Queuing models for the *SISO* and *MISO* systems are shown in Figure 7-5.

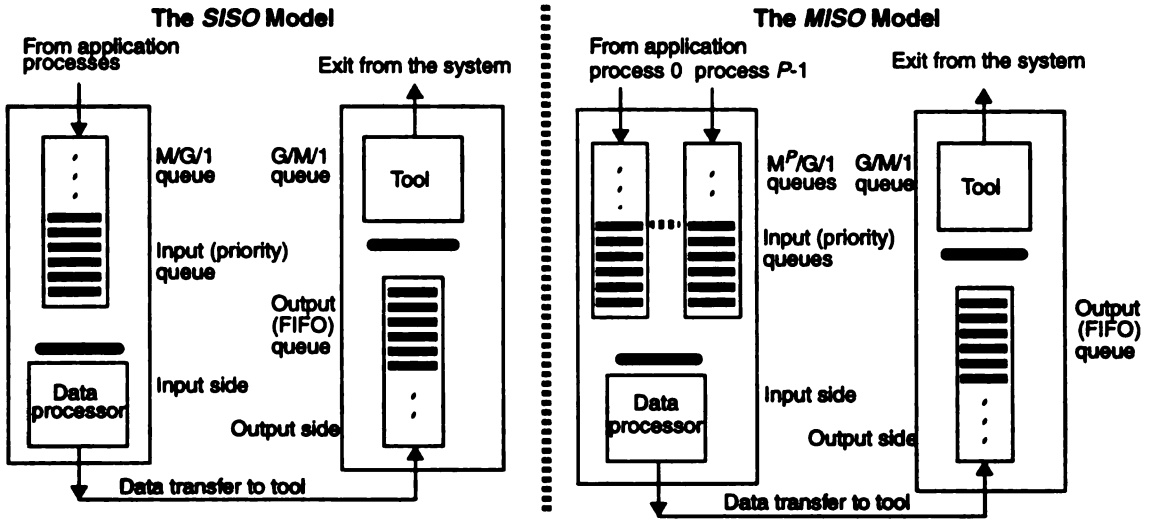


Figure 7-5. Models for the SISO and MISO configurations of the Vista *ISM*.

7.3.4.4 Workload Characterization

As in the case of PICL IS study, we focus on the low-level instrumentation data arrivals regardless of the number of LISs or the instrumented SUT that generates these data. Instrumentation data are assumed to arrive at the input buffer(s) with exponentially distributed inter-arrival times. The data processor of the ISM processes and dispatches this data according to a normal distribution. The processed instrumentation data are consumed by a tool in a first come, first served fashion.

7.3.4.5 Performance Metrics

We have selected two metrics to compare the performance of the *ISM* configurations: data processing latency and average length of buffer(s). *Data processing latency* is defined as the amount of time between the arrival of instrumentation data at the *ISM* and its arrival (after processing) at the output buffer. Lower is better, since a high latency may result in

inaccurate presentation of program behavior by tools. *Average buffer length* is defined as the ratio of the total number of instrumentation data records that arrive out of order (and hence need to be buffered) to the total observation time. A larger value of average buffer length indicates that many arrivals are out of order due to the management policies implemented by the LIS. A similar metric, called *hold back ratio* has been used by Gu et al. to evaluate the performance of the Falcon *ISM* [71]. This metric is defined as the ratio of the number of out-of-order arrivals to the total number of arrivals (rather than to total time). However, the two metrics provide the same qualitative measure of ISM performance. Each metric, its calculation, and its interpretation are summarized in Table 7-1.

Table 7-1. Metrics for evaluating the Vista *IS* management policies.

Metric	Calculation	Interpretation
Data processing latency	Queuing model evaluation and simulation	Longer latency may be undesirable for the tools
Average buffer length (hold back ratio)	Queuing model evaluation and simulation	Higher value indicates a potential bottleneck in the IS

7.3.4.6 IS Evaluation

In order to evaluate two configurations of the Vista IS, we present two approaches. Initially, during the Vista planning and design stages, we used a simulation model to evaluate the performance impact of the two configurations. Presently, as it is realized, we obtain actual measurements by running real programs to compare them with the simulation results.

Simulation Results

The simulation experiments are set up to analyze the effects of the *SISO* or *MISO* configuration on the two performance metrics. Two factors are varied for these experiments: the ISM configuration (*SISO* or *MISO*) and the mean inter-arrival time

between successive instrumentation data arrivals to the *ISM*. We use a $2^k r$ factorial design technique for these experiments. For these experiments, k=2 factors and r=50 repetitions, and the mean values of the two metrics are derived within 90% confidence intervals.

Data processing latency and average buffer length statistics for the two configurations and various arrival rates are shown in Figure 7-6. The data processing latency exhibits higher variance at longer inter-arrival times (lower arrival rates) for both *SISO* and *MISO* configurations, making them less distinguishable. For shorter inter-arrival times (higher arrival rates), the *SISO* ISM has relatively lower latency. Intuitively, maintenance of multiple buffers should incur more overhead, especially in accessing memory (including virtual memory), under high arrival rate conditions. The average buffer length follows a similar pattern. At lower arrival rates, the average buffer lengths are almost the same, but at higher rates, *SISO* is better than *MISO*. We analyzed these results using *principal component analysis* techniques and found that the inter-arrival rate is the dominant factor that affects data processing latency and average buffer length.

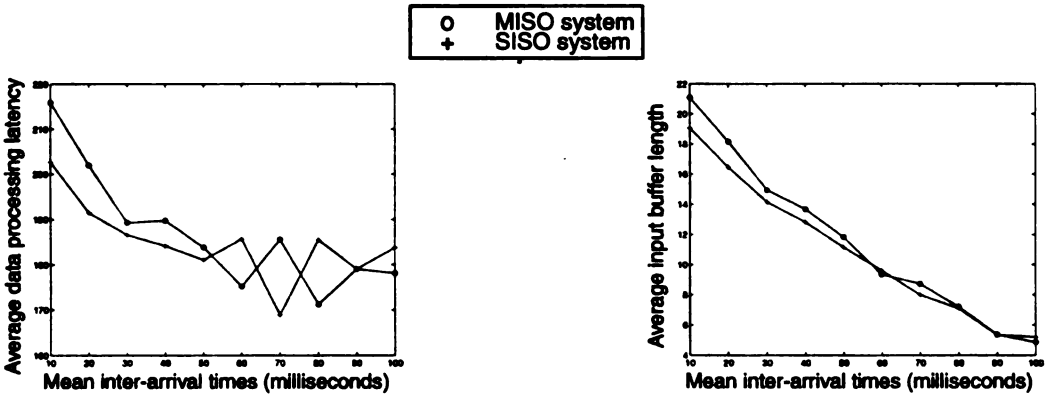


Figure 7-6. Comparison between the *SISO* and *MISO* ISMs in terms of average data processing latencies and input buffer lengths.

The simulation results do not indicate that one configuration is clearly superior to another. Some researchers favor the *MISO* configuration, and tools such as Falcon have implemented it. However, the models and simulation-based evaluation presented here suggest that the *SISO* configuration performs equally well at moderate arrival rates and

marginally better at higher arrival rates. In event-driven monitoring, it is not uncommon for the rate of arrivals to surge during certain intervals, yielding unstable ISM behavior. Since the Vista IS uses an event-driven approach, a design decision was made to incorporate both the *SISO* and *MISO* configurations based on this modeling and evaluation feedback, so that user could dynamically configure the ISM based on the requirements of the application. In general, assessing and validating design decisions with measurements of the operating IS (i.e., with testing and benchmarking) is an essential step of the development process and one that we are currently addressing.

Measurement Results

After undertaking the simulation-based evaluation study for designing Vista ISM, we developed its prototype version that supports both *SISO* and *MISO* systems as options. Results presented in this section are based on the measurements obtained from this prototype. We use two example programs for the measurement experiments: one that is communication-intensive (a linear solver) and another that is compute-intensive with a comparatively smaller number of messages, both using a master/slave computing paradigm supported by PVM.

We collected the inter-arrival times of the successive instrumentation data samples at the ISM. These inter-arrival times for the two programs are presented as frequency distribution histograms in Figure 7-7. In both cases, the number of arrivals that have large inter-arrival times becomes exponentially small. This is a typical scenario where assumption of exponentially distributed inter-arrival times can be used. However, we did not prove the exponential nature of arrivals using statistical tests, because the purpose of simulation-based studies was only to furnish only back-of-the-envelope type of calculations. A rigorous workload evaluation was beyond the scope and details necessary to gain an initial insight into the design options when the system was not yet prototyped.

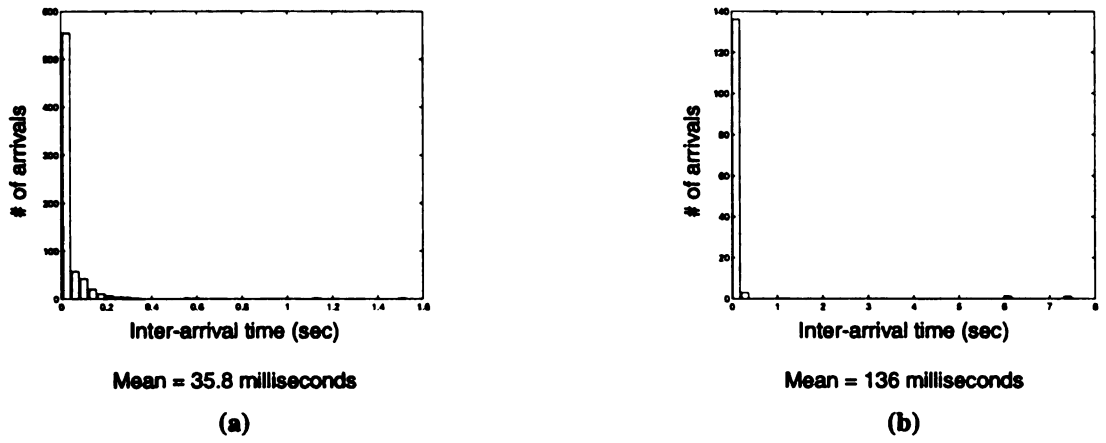


Figure 7-7. Frequency distribution of two arrival processes to the Vista ISM from (a) communication-intensive and (b) compute-intensive master/slave PVM programs.

The simulation study assumed that the service times for each incoming instrumentation data sample were not Markovian. In order to determine the validity of this assumption, we collected service times for each instrumentation data sample for the two example programs. The frequency distribution histograms for the communication intensive program using *SISO* and *MISO* systems are presented in Figure 7-8. For the *SISO* system, the service time distribution resembles the shape of an exponential distribution. However, the shape of the service time distribution for the *MISO* system is not close to the exponential distribution. This is apparent if the outliers are removed from the range of the service times for the *MISO* system. In that case the shape of the distribution curve is close to that of a normal distribution, as assumed in the simulation-based experiments.

Figure 7-9 shows the frequency distribution for the service times for the compute-intensive master/slave example program. As in the case of communication-intensive example, the service time distribution is different from exponential. One possible explanation of this behavior is the involvement of a number of factors that influence the service time of an instrumentation data sample, such as the computation load due to other user and system processes running on the workstation, number of memory references and page faults for servicing a sample, pipelining effects, and so forth. When there are several factors influencing the nature of a stochastic process, a normal distribution is often an

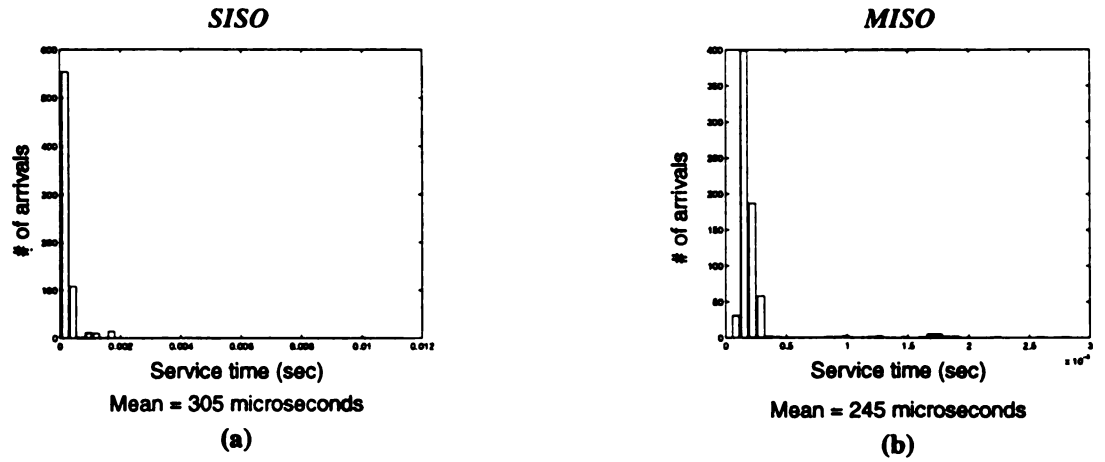


Figure 7-8. Frequency distribution of the service processes for the communication-intensive program at the Vista ISM using (a) *SISO* and (b) *MISO* configurations.

appropriate assumption. In the absence of a precise workload study, service times are assumed to be normally distributed for the simulation experiments.

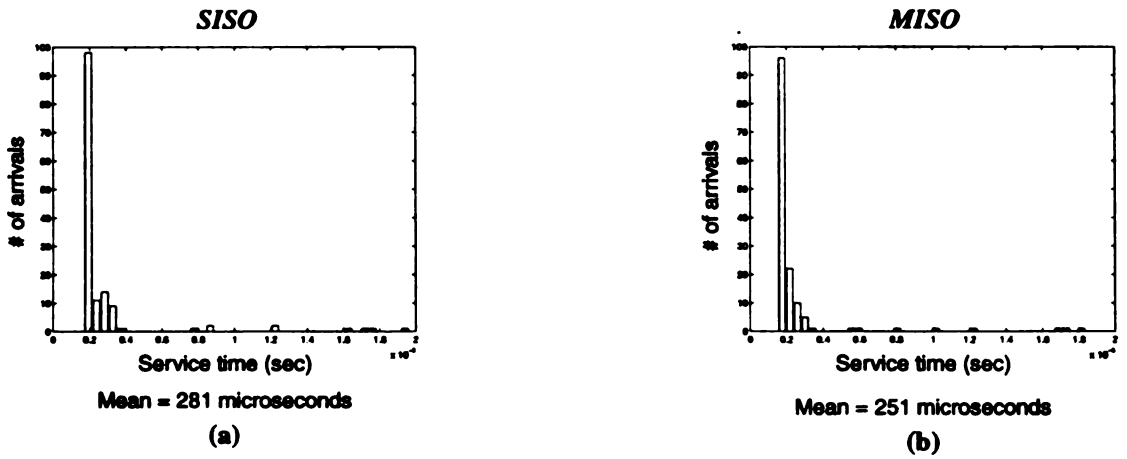


Figure 7-9. Frequency distribution of the service processes for the compute-intensive example program at the Vista ISM using (a) *SISO* and (b) *MISO* configurations.

Table 7-2 summarizes the results of this measurement study. It presents the measurements for two metrics of interest, i.e., data processing latency and the mean buffer length (hold-back ratio) over the entire execution of a program. As in the case of simulation-based results, the measurement results also do not show that one configuration outperforms the other given the information flows in the studies. Therefore, the choice between the two depends largely on the preference of the tool developers and the nature of specific

applications that need to be supported by the IS. Further investigations and modeling of event flows in other application domains may reveal distinctive ISM performance.

Table 7-2. Summary of measurement results for evaluating the Vista ISM.

System	Program	Mean inter-arrival time (milliseconds)	Mean service time (μseconds)	Data processing latency (milliseconds)	Mean input buffer length
SISO	Communication-intensive	35. 8	305	2.10	26.9
	Compute-intensive	136	281	2.13	6.6
MISO	Communication-intensive	35.8	245	2.15	26.4
	Compute-intensive	136	251	2.24	6.28

7.3.4.7 Summary

Development of software tools to assist parallel and distributed computing is considered a formidable task, involving multidisciplinary efforts. Unlike a typical software development project, the development of software tools for concurrent systems requires the accomplishment of the following three tasks:

1. determining ways to present a consistent "ordered" picture of parallel or distributed computation, which is easily comprehensible by a human user;
2. determining ways to present a "synchronous" picture of inherently asynchronous computing activities, such as computations local to a node and message-passing among various nodes of a concurrent system; and
3. determining ways to achieve the former two objectives in a manner that is appealing to the users.

Several paradigms have been proposed to develop tools that are technically sound and are successful to varying degrees in addressing user requirements. Based on state-of-the-art in parallel and distributed computing tools, one may conclude that: (1) tools that have been developed to address specific user requirements for specific classes of applications,

utilizing available tool development technology, are considered useful; and (2) expanding application areas of parallel and distributed processing necessitate the tool development process that adheres to well-known techniques of designing software as well as other complex systems.

An instrumentation system is a vital component of the middleware of an integrated tool environment [14]. We applied several aspects of the structured IS design, modeling, evaluation, and development approach to the Vista IS. This application is driven by the domain-specific requirements and focuses on designing and evaluating the IS based on these requirements. This process provided initial insight to the developers of PG^{RT} environment, so that performance impacts of IS design alternatives were appreciated at an early stage of development. Considerable effort is needed to extend the Vista framework, so that it can be used as a library of configurable, retargettable, plug-and-play IS modules for multidisciplinary applications. This is one of the future directions of this work.

Chapter 8

Conclusions, Contributions, and Future Work

In this concluding chapter, we evaluate the contributions of the research presented in this dissertation and suggest possible ways of extending the research. An important measure of success for a research effort is the achievement of projected goals to address specific problems in an area. We specified a number of goals of this research, including: modeling of off-line and on-line ISs; evaluation of IS management policies based on a set of generic metrics; and implementation of an IS based on the proposed (at the time of starting with this research) modeling and evaluation methodology. Through this research, we have demonstrated the feasibility of a structured IS design, modeling, and evaluation approach by applying it to PICL, Paradyn, JEWEL, and Vista ISs. We were able to identify a set of generic performance metrics to evaluate an IS; however, we advocate that performance metrics be defined in the context of domain-specific requirements to be more useful in practice. We proposed, modeled, and evaluated the management policies for Paradyn and JEWEL ISs. We applied the structured approach to develop an object-oriented framework to configure the Vista IS for multidisciplinary applications. We have met our goals to the extent that the validity and applicability of the proposed research is demonstrated.

In section 8.1, we present the specific contributions of this work. Some of the possible future directions of this work are presented in Section 8.2. We conclude with a discussion of the impact of this research on state-of-the-art in instrumentation system design, modeling, evaluation, development, and usage in Section 8.3.

8.1 Contributions

There are four main contributions of the work presented in this dissertation: development of a taxonomy for multidisciplinary ISs; development and application of ROCC modeling

technique; modeling-based evaluation of a number of ISs; and proposition and evaluation of novel management policies and alternative configurations for real ISs. These contributions are further elaborated in the following subsections.

8.1.1 A Taxonomy for ISs

ISs are used with diverse parallel and distributed tool environments, applications, and systems. Tool environments consisting of debugging, performance analysis, bottleneck searching, modeling, and prediction tools rely on runtime measurements supplied by an IS. Multidisciplinary applications, such as administration of commercial transaction processing systems [49], measurement-based testing of complex military systems [9], and resource management for distributed real-time systems [16,187,199,202] consume the runtime information supplied by an IS to perform their specified functions. A variety of distributed systems, such as a pattern recognition system [99] or an embedded real-time controller [122] require continuous data collection for either measuring the features of an object for its appropriate representation or adaptively controlling a device or process, respectively. Based on the available information about the common practices of IS design and usage in each of the above three entities, we were able to synthesize a taxonomy of an IS. This taxonomy identifies a number of modules and services that were found common (explicitly or implicitly) in ISs across diverse disciplines. Initially, we used this taxonomy to develop a framework consisting of generic implementations of the modules identified in the taxonomy. We applied this framework to develop the Vista IS, which is being used for collecting runtime information from two types of applications: message-passing PVM programs and simulated complex distributed real-time systems.

8.1.2 The ROCC Modeling Technique

The above discussion outlines two major contributions of this work: development of a unified IS taxonomy and a well-defined methodology to obtain early evaluation of the IS performance and intrusion. It is appropriate to mention a third contribution of this work: the Resource OCCupancy (ROCC) modeling technique. ROCC models were developed and used for evaluating the contention for system resources shared among IS and

application processes. This technique is distinguished from a number of other computer system modeling approaches in terms of capturing inter-dependences among different processes. A majority of the existing models rely on simplifying assumptions to factor out these dependences [38,80,93,177]. However, ROCC modeling combined with a coarse-grain workload characterization can represent application- and system-level task scheduling and dependences. Although we have used this technique for evaluating different IS configurations and management policies, we expect to apply it to a broader range of system resource management problems.

8.1.3 Modeling and Evaluation of Real ISs

This research is mainly focused at applying the IS taxonomy to better understand the design and domain-specific requirements of three real ISs: PICL, Paradyn, and JEWEL ISs. Based on this understanding, we were able to model them; propose policies for IS runtime management and reduction of intrusion to the target system; and evaluate them. The results indicate that the modeling and evaluation approach is effective to provide early feedback to the IS developers as well as users about the performance and intrusion of IS modules, available management policies, and alternative configurations. With early feedback, it is possible for the IS or application developers to make informed decisions about the selection of IS components, management policies, and configurations that are suitable for a given application.

8.1.4 IS Management Policies

In addition to modeling and evaluating the existing runtime management policies of the ISs, we proposed alternative policies and configurations. In case of the PICL IS, the flush-all when one-fills (FAOF) and flush-one when it fills (FOF) policies were proposed by the developers. However, the modeling-based evaluation of these two policies exposed their trade-offs. Based on this evaluation, Haake et al. developed a *flush-on-barrier* policy that implements the FAOF policy at barrier synchronization points in Split-C programs [74]. We proposed and evaluated the Batch Forwarding (BF) policies for the Paradyn and

JEWEL ISs that resulted in considerable reduction of overhead and intrusion over the default management policies for these ISs. Similarly, alternative configuration options, such as tree forwarding for the Paradyn IS and adaptive control scheme for the JEWEL IS, addressed the domain-specific requirements of ISs.

These four contributions identify the original work in a steadily maturing area. In order for this work to be beneficial to advance the state-of-the-art in instrumentation systems, it should be extended in several ways. We identify a number of avenues for the future work in the following section.

8.2 Future Work

The research presented in this dissertation has contributed mainly to the area of software tools and environments for parallel and distributed systems. It can be extended to further explore a number of related applications and systems. Although this research work falls in the general area of computer system performance modeling, it is directly linked to the software development and implementation of those systems. Thus it can be extended to four broad areas:

1. design and evaluation of ISs for emerging parallel and distributed computing applications;
2. design and implementation of suitable IS testing approaches;
3. development of ISs from a set of configurable, possibly commercial off-the-shelf, plug-and-play modules; and
4. extension of the ROCC modeling approach, which is applied to the study of ISs and trade-offs among its management policies, to other parallel and distributed systems and applications.

The rest of this section explores the above areas with considerable details to motivate the application and extension of this research.

8.2.1 Design and Evaluation of ISs for Emerging Applications

There are opportunities to extend the IS design, modeling, management, and evaluation work presented in this dissertation to several applications that consume instrumentation data collected from parallel and distributed systems. We discuss a number of emerging applications of parallel and distributed computing systems that benefit from runtime information to fulfil their domain-specific needs. These applications include: distributed real-time adaptive control systems, commercial on-line transaction processing systems, pattern recognition systems, and complex distributed systems.

8.2.1.1 Distributed Real-Time Adaptive Control Systems

Adaptive control is commonly applied to the distributed, real-time embedded systems. Typical examples of such systems include military combat systems, safety-critical systems, switching and routing in telecommunication systems, and aircraft and automobile control subsystems [66,121,202,224]. An IS accomplishes three tasks for this class of applications:

- collects runtime data for on-line monitoring of "health" of the target system;
- observes the internal states and system response to help decide any resource management "actions" to maintain the system at desired operating points; and
- collects data from the distributed sensors to help an adaptive controller make decisions in real-time, according to the mission of the system.

Clearly, for adaptive control applications, the scope of an instrumentation system is extended beyond its role as a set of data collection, processing, and consumption components. An IS is a part of the closed-loop control system and works as a *system observer.* A consequence of this extended scope is the need for meeting stringent design and operation specifications to ensure that functional and non-functional requirements of the target system are not impacted. The IS design, modeling, and evaluation approach

presented in this dissertation did consider such systems. However, several case studies are needed to practically apply this approach to real-time adaptive control applications.

In addition to designing and using an IS for distributed adaptive control applications, an IS itself can be designed as an adaptive system. The case study of JEWEL IS, presented in this dissertation, explored the trade-offs involved in implementing different policies. This area can be further investigated by considering the application of real-time dynamic scheduling techniques to IS tasks using real-time features available in a number of operating systems.

8.2.1.2 Commercial Transaction Processing Systems

Transaction processing is one of the most important commercial applications of distributed computing. Transaction processing systems consist of a large number of sources of data and services distributed throughout some geographical region with a consistent set of management policies across the system. In such systems, the data and control flow mechanisms play an important role in integrating and managing the enterprise-wide distributed resources. Instrumentation systems are used to collect runtime information to accomplish two types of system management strategies:

- IS can support a single point-of-control to manage the entire distributed system from a logically centralized location; or
- IS can support a hierarchical, distributed control of the local resource.

Modeling and evaluation of an IS for either type of management strategy is relevant to provide an early feedback to the developers. For a large transaction processing system, it is desirable to evaluate the design of the IS to make sure that it can meet domain-specific requirements.

8.2.1.3 Distributed Embedded Systems

Distributed embedded systems are distinguished due to their constraints on space, weight, power consumption, and real-time behavior. Such systems are widely used for real-time control, signal processing, and pattern recognition applications. An embedded system can be developed as a parallel or distributed system, depending on the requirements on its performance and functionality. An IS design that ensures accuracy of measurements is desirable to achieve the domain-specific performance goals of the embedded system, such as small *error rate*.

Distributed embedded systems are often complex systems. The term *complex system* refers to a distributed computing system, possibly including any system within which it is embedded [174]. Computation and/or communication are critical aspects of system behavior. A complex system is often encountered in an application that needs to accomplish a large number of interdependent tasks to satisfy a given set of requirements that may conflict with one another in multiple ways [198]. There is a growing number of software tools and environment that address the needs of distributed complex systems [221]. A sizeable subset of these tools, such as monitoring, visualization, performance tuning, real-time steering, dynamic assertion checking, and debugging tools depend on runtime data collection. An IS that is implemented according to the structured design, modeling, and evaluation approach presented in this dissertation is desirable for these systems.

This list of emerging applications of ISs is expected to grow with time. However, the purpose of this subsection is to identify some important potential areas of application rather than enumerating all possible areas.

8.2.2 IS Testing

With increasing diversity of their areas of usage, ISs are being developed as *off-the-shelf* software systems that can easily be integrated in a target system where runtime data collection and processing is needed. These ISs are developed as a set of "middleware" modules and services that interface application processes with operating system services to allow an application to collect data from the system under test.

At the time of writing this dissertation, a great deal of emphasis is being placed on the testing of parallel tools [228]. Software reliability models [24,225] and tests [140] are often used by software vendors to analyze the suitability of a software product for a particular application. However, there are very few examples of applying quantitative approaches to test and improve ISs. This situation motivates the need of extending IS modeling and evaluation work to the development of quantitative testing methodologies.

Due to the emerging applications discussed in Section 8.2.1, stringent performability constraints may be imposed on an IS, in addition to non-functional requirements such as fault-tolerance and reliability. There is a growing need for these ISs to be tested by their developers with results of these tests reported as IS *specifications*. Testing is an essential aspect of system design in many areas including VLSI devices and circuits, *fly-by-wire* aircraft control systems, telecommunication systems, pattern recognition systems, and so on. However, use of software instrumentation systems for collecting data from physically distributed, large *complex systems* is a relatively recent phenomenon. Therefore, IS testing and a number of issues related to it are yet to be explored. We identify some of these issues in the following subsections.

8.2.2.1 Models for IS Testing

Testing of parallel tool components, especially an instrumentation system can greatly benefit from the state-of-the-art testing approaches in other areas. For instance, IC design

and manufacturing industry has well-developed fault models and test-generation methods, such as *built-in self-test* (BIST) [34,139]. BIST, which utilizes scanning technology, provides the stimulus-generation and response processing functionality to test complex logic structures and embedded components or cores [109]. Due to intense competition in microprocessor industry, it is desirable that a newly designed microprocessor does not require an excessively long testing period before marketing. Marr et al. describe the use of an innovative testing methodology that was used to validate the multiprocessor functionality of Intel's Pentium Pro microprocessor [128].

Developing models to test software systems such as ISs, is a more complicated process compared to the testing of VLSI systems; lack of physical entities with well-defined behaviors in a software system contributes to the complexity in representing them using simple modeling tools. Software systems are made up of *abstractions* such as processes, threads, semaphores, locks, etc. that have intricate behavior and dependences on one another. Exact analytic models of such systems are of little practical value because they are often mathematically intractable. A Markov model is considered a reasonable compromise between real-world dependences and independence needed for mathematical tractability. Chen and Mills suggest a Markov process model for random testing of software [35]. Miller reports the results of testing real parallel programs using random inputs [137]. It is necessary to study a number of existing ISs from the perspective of developing appropriate test models. Such studies can yield a valuable body of knowledge about the suitability of different models for testing existing ISs.

8.2.2.2 Synthetic Workload Generation

According to the IS development and usage envisioned in this dissertation, an IS undergoes at least three phases in its life-cycle: (1) evaluation of alternative configurations of modules and task scheduling options at an early development stage; (2) testing IS features for correctness and reliability by the developers before the IS goes into production; and (3) usage of the IS for real applications (production phase). For

accomplishing the first two tasks, we need a model for the IS and an adequate workload characterization that drives the model. This dissertation specifically focused at the first phase where the model was evaluated analytically or through simulations to provide feedback to the IS developers. Testing phase can also benefit from the model and workload characterization by focusing on specific aspects, found critical to the performability of the IS during the evaluation phase. While workload characterization is used for simulation-based studies of computer systems, testing phase is conducted empirically.

Figure 8-1 illustrates an approach to use the initial characterization results for generating synthetic workload for testing the ISs. This setup has some similarities with a typical pattern recognition system. A typical pattern recognition system consists of two phases: *training* and *classification*. Workload characterization is similar to the training phase. However, workload generation requires the synthesis of test patterns according to the workload characterization rather than classifying the patterns as in case of a typical classifier. Thus, test pattern generation phase is in effect inverse of the pattern classification phase of a typical pattern recognition system. Test patterns undergo a *postprocessing* phase to invoke appropriate executable instructions that can be used for testing an actual instrumentation system. Some initial efforts are reported in the literature to use workload characterization to design test suites. For example, Nanda and Ni emphasize the importance of characterizing the workload to generate synthetic workload kernels that can expose weaknesses and strengths of multiprocessor memory subsystem behavior [141].

Although the IS test workload generation scheme depicted in Figure 8-1 appears simple, the challenging part is the designing of actual code to implement a particular synthetic test pattern. For instance, if a test pattern requires a specific value for the system bus bandwidth utilization during a test, there is no simple way to implement it on most of the Unix-like operating systems that are based on fair scheduling of system resources among

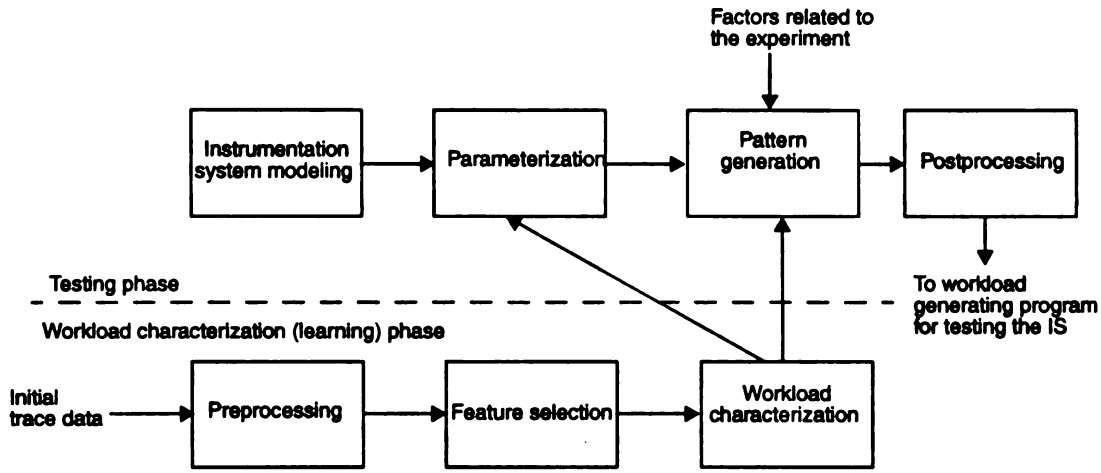


Figure 8-1. Approach adopted for workload characterization and testing of an instrumentation system.

user processes. Therefore, the approach needs additional effort to undergo appropriate modifications to be useful in practice.

Instrumentation system testing can be considered a part of a much broader initiative for measurement-based tool testing [228]. Although we restricted our attention to the quantitative aspects of testing, there is a growing need to address the qualitative aspects also.

8.2.3 IS Development

We presented a taxonomy to characterize an IS in terms of a number of modules and services. We used this taxonomy to understand, model, and evaluate the behavior of a number of existing ISs. This characterization is also useful for actual software development to implement an IS beyond its design and evaluation stages. Some of the possible future directions for IS development are presented in the following subsections.

8.2.3.1 Plug-and-Play IS Modules

Software tool environments including those being used for parallel and distributed systems, are increasingly using commercial, off-the-shelf (COTS) software products. This is emerging as a cost-effective and efficient approach as opposed to relying on a single software developer to custom design every single module of an environment for a particular application. Many safety-critical [157] and high performance, distributed combat systems [79] are also relying on plug-and-play COTS software. ISs can also benefit from this approach. We have done preliminary work to define a framework for developing ISs for multidisciplinary applications [216]. However, considerable effort is needed to develop a set of IS modules according to this framework that can be conveniently ported to various platforms and integrated into different tool environments.

8.2.3.2 Configurable IS Kernels

Embedded system design is distinguished due to the constraints imposed on their size, weight, requirements for computing resources, and performance. Runtime measurements of system behavior are useful to tune their performance and test their functional correctness. Thus, ISs are needed for such systems. In Section 8.2.1.1, we discussed the need for using the structured IS design and evaluation approach for embedded and real-time systems. However, the actual development of such ISs requires the use of novel software techniques and tools. One possible approach is to design an IS "kernel" instead of fully functional IS that can be retargeted to new embedded applications. In order to develop a fully functional IS for an embedded system, customized software layers can be added on top of the kernel.

8.2.3.3 IS Interfaces

As ISs developed as COTS software products are being applied to a broad range of applications, the need for standardizing the interface between an IS and the SUT as well as an IS and a tool (or environment) is becoming obvious. A number of standardization

efforts, such as MPI [133], HPF [84], and POSIX [97] are driven by the needs of a diverse user community for portable communication functions, language constructs for HPCC applications, and operating system functions, respectively. Due to a large size of parallel tool developer community, development of a standard IS interface will be a worthwhile effort. Our research as well as several related efforts can contribute toward the development of a standard for IS interfaces.

8.2.4 Resource Management Using ROCC Modeling Technique

Resource management problems in a parallel or distributed system are not restricted to the instrumentation systems. Performance of most of the computer systems depends on evaluating the trade-offs among available design options. We applied the resource occupancy (ROCC) modeling technique for evaluating the resource contention between the SUT and IS tasks. A notable benefit of using ROCC modeling technique is its ability to model the software system at a high level of abstraction considering dependences among different processes. This modeling approach has the potential of extending to parallel or distributed systems that need dynamic resource management. Examples of such systems include: embedded real-time systems, distributed high-performance computing systems (connected through a high-speed wide-area network), computer networks for multimedia traffic, and parallel file systems. We intend to apply ROCC modeling approach to evaluate the dynamic resource management policies for these systems.

As a final note related to the future investigations, we consider the state-of-the-art in ISs to be in a transition phase. A number of IS-related projects in relatively new application areas of parallel and distributed computing indicate the thrust for future changes. Therefore, we view the work presented in this dissertation as a static snapshot of this area at the time of writing it. We are already making progress on some of the future directions described here.

8.3 Concluding Remarks

The impact of a research effort on the state-of-the-art in its area is an essential measure of success. While the application of this research proceeds and its ideas continue to mature, outcomes of this research—publications, collaborations, funded proposals, and citations—indicate its relevance to the field.

Bibliography

[1] M. Abrams, N. Doraswamy, and A. Mathur, "Chitra: Visual Analysis of Parallel and Distributed Programs in the Time, Event and Frequency Domains," *IEEE Transactions on Parallel and Distributed Systems*, 3(6), November 1992.

[2] Richard M. Adler, "Distributed Coordination Models for Client/Server Computing," *IEEE Computer*, 28(4), April 1995, pp. 14–22.

[3] A. K. Ahluwalia and M. Singhal, "Performance Analysis of the Communication Architecture of the Connection Machine," *IEEE Transactions on Parallel and Distributed Systems*, 3(6), November 1992.

[4] Arnold O. Allen, *Probability, Statistics, and Queuing Theory with Computer Science Applications*, Second Edition, Academic Press, 1990.

[5] George S. Almasi and Allan Gottlieb, *Highly Parallel Computing*, The Benjamin/ Cummings Publishing Company, Inc., 1989.

[6] Howard Anton and Chris Rorres, *Elementary Linear Algebra — Applications Version*, John Wiley & Sons Inc., 1991. *IEEE Transactions on Parallel and Distributed Systems*, 3(6), November 1992.

[7] Masanao Aoki, *State Space Modeling of Time Series*, Springer-Verlag, 1990.

[8] W. Appelbe and C. McDowell, "Integrating Tools for Debugging and Developing Multitasking Programs," *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, Madison*, Wisconsin, May 5-6, 1988.

[9] *ARPA Integrated Demonstration Report on* "High Performance Distributed Computing Program (HiPer-D)," September 6, 1994.

[10] Touraj Assefi, *Stochastic Processes and Estimation Theory with Applications*, John Wiley & Sons, Inc., 1979.

[11] N. Balakrishnan and A. C. Cohen, *Order Statistics and Inference—Estimation Methods*, Academic Press, Inc., 1991.

[12] P. Beadle, C. Pommerell, and M. Annaratone, "K9: A Simulator of Distributed-Memory Parallel Processors," *Proc. of Supercomputing '89*, ACM Press, 1989.

[13] David G. Belanger, Yih-Farn Chen, Neal R. Fildes, Balachander Krishnamurthy, Paul H. Rank Jr., Kiem-Phong Vo, and Terry E. Walker, "Architecture Styles and Services: An Experiment Involving Signal Operations Platforms-Provisioning Operations Systems," *AT&T Technical Journal*, January/February 1996, pp. 54–60.

[14] Philip A. Bernstein, "Middleware: A Model for Distributed System Services," *Communications of the ACM*, 39(2), Feb. 1996, pp. 86–98.

[15] Devesh Bhatt, Rakesh Jha, Todd Steeves, Rashmi Bhatt, and David Wills, "SPI: An Instrumentation Development Environment for Parallel/Distributed Systems," *Proc. of Int. Parallel Processing Symposium*, April 1995.

[16] Laxmi N. Bhuyan and Xiadong Zhang, *Multiprocessor Performance Measurement and Evaluation, IEEE Computer Society Press*, 1995.

[17] Robert J. Block, Pankaj Mehra, and Sekhar Sarukkai, "Automated Performance Prediction of Message-Passing Parallel Programs," *Proceedings of Supercomputing '95*, San Diego, California, Dec. 4–8, 1995. Available on-lin from: http:// scxy.tc.cornell.edu/sc95/proceedings/450_SSAR/SC95.HTM.

[18] George P. Box and Gwilym M. Jenkins, *Time Series Analysis — Forecasting and Control*, Holden-Day Inc., 1976.

[19] M. C. Breiter and P. R. Krishnaiah, "Tables for the Moments of Gamma Order Statistics," *Sankhya*, Series B, Volume 30, 1968, pp. 59–72.

[20] D. Brown, S. Hackstadt, A. Malony, B. Mohr, "Program Analysis Environments for Parallel Language Systems: The TAU Environment," *Proc. of the Second Workshop on Environments and Tools For Parallel Scientific Computing*, Townsend, Tennessee, May 1994, pp. 162–171.

[21] Marc Brown and John Hershberger, "Color and Sound in Algorithm Animation," *IEEE Computer*, December 1992.

[22] Andreas Buja et al., "Interactive Data Visualization using Focusing and Linking," *Proceedings of Visualization '91*, 1991.

[23] Peter Burger, Duncan Gillies, *Interactive Computer Graphics*, Addison-Wesley Publishing Company, Inc., 1989.

[24] Ricky W. Butler and Finelli, George B., "The Infeasibility of Quantifying the Reliability of Life-Critical Real-Time Software," *IEEE Transactions on Software Engineering*, 19(1), Jan. 1993, pp. 3-12.

[25] D. Callahan and J. Subhlok, "Static Analysis of Low-level Synchronization," *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, Madison*, Wisconsin, May 5-6, 1988.

[26] B. M. Carlson, T. D. Wagner, L. W. Dowdy, and P. H. Worley, "Speedup Properties of Phases in the Execution Profile of Distributed Parallel Programs," Technical Report ORNL/TM-11900, Oak Ridge National Laboratory, August 1992.

[27] Gordon E. Carlson, *Signal and Linear System Analysis*, Houghton Mifflin Company, 1992.

[28] Thomas L. Casavant, "Tutorial: Software Tools for Visualization of Parallel and Distributed Programs and Systems," Department of Electrical and Computer Engineering, University of Iowa, September 1991.

[29] Thomas L. Casavant, "Tools and Methods for Visualization of Parallel Systems and Computations," *Journal of Parallel and Distributed Computing*, 18(2), June 1993.

[30] K. M. Chandy and Jayadev Misra, *Parallel Program Design—A Foundation*, Addison-Wesley Publishing Company, Inc., 1988.

[31] Carl K. Chang, Young-Fu Chang, Lin Yang, Ching-Roung Chou, and Jong-Jeng Chen, "Modeling a Real-Time Multitasking System in Timed PQ Net," *IEEE Software*, March 1989.

[32] John Chapin, Stephen A. Herrod, Mendel Rosenblum, and Anoop Gupta, "Memory System Performance of Unix on CC-NUMA Multiprocessors," *Proceedings of Sigmetrics '95*, Ottawa, Canada, May 15-19, 1995, pp. 1–13.

[33] M. L. Chaudhry and J. G. C. Templeton, *A First Course in Bulk Queues*, John Wiley, 1983.

[34] Chih-Ang Chen and Sandeep K. Gupta, "BIST Test Pattern Generators for Two-Pattern Testing—Theory and Design Algorithms," *IEEE Transactions on Computers*, 45(3), March 1996.

[35] Sanping Chen and Shirley Mills, "A Binary Markov Process Model for Random Testing," *IEEE Transactions on Software Engineering*, 22(3), March 1996.

[36] Doreen Y. Cheng, "A Survey of Parallel Programming Languages and Tools," Report RND-93-005, NASA Ames Research Center, March 1993.

[37] M. J. Clement and M. J. Quinn, "Multivariate Statistical Techniques for Parallel Performance Prediction," *Proceedings of the Twenty Eighth Hawaii International Conference on System Sciences*, Maui, Hawaii, Jan. 3-6, 1995, pp. 446–455.

[38] Mark J. Clement, Michael R. Steed, and Phyllis E. Crandall, "Network Performance Modeling for PVM Clusters," *Proceedings of Supercomputing '96*, Pittsburgh, Pennsylvania, Nov. 17–22, 1996. Available on-line from http://scxy.tc.cornell.edu/sc96/proceedings/SC96PROC/CLEMENT/INDEX.HTM.

[39] John R. Clymer, *Systems Analysis Using Simulation and Markov Models*, Prentice-Hall, Inc., 1990.

[40] Richard Comerford, "Software on the Brink," *IEEE Spectrum*, 29(9), September 1992.

[41] R. A. Cooper and A. J. Weekes, *Data, Models and Statistical Analysis*, Barens and Noble Books, 1983.

[42] Alva Couch, "Graphical Representation of Program Performance on Hypercube Message-Passing Multiprocessors." *Ph.D. dissertation*, Department of Computer Science, Tufts University, April 1988.

[43] Alva Couch and David W. Krumme, "Projection, Pursuit, and the Triplex Tool Set for the NCUBE Multiprocessor," Department of Computer Science, Tufts University, November 1989.

[44] Alva Couch, "Categories and Context in Scalable Execution Visualization," *Journal of Parallel and Distributed Computing*, 18(2), June 1993.

[45] Mark E. Crovella and Thomas J. LeBlanc, "Parallel Performance Prediction Using Lost Cycles Analysis," *Proceedings of Supercomputing '94*, Washington, DC, Nov. 14–18, 1994, pp. 600–609.

[46] Morris H. DeGroot, *Probability and Statistics*, Addison-Wesley Publishing Company, 1987.

[47] Robert T. Dimpsey and Ravishankar K. Iyer, "A Measurement-Based Model to Predict the Performance Impact of System Modifications: A Case Study," *IEEE Transactions on Parallel and Distributed Systems*, 6(1), January 1995, pp. 28–40.

[48] J. Dongarra, R. van de Geijn, and D. Walker, "A Look at Scalable Dense Linear Algebra Libraries," *Proceeding of Scalable High Performance Computing Conference*, April 1992.

[49] Stephen G. Eick and Daniel E. Fyock, "Visualizing Corporate Data," *AT&T Technical Journal*, January/February 1996, pp. 74–85.

[50] Greg Eisenhauer, Weiming Gu, Thomans Kindler, Karsten Schwan, Dilma Silva, and Jeffrey Vetter, "Opportunities and Tools for Highly Interactive Distributed and Parallel Computing," in *Debugging and Performance Tuning for Parallel Computer Systems*, M. L. Simmons, A. H. Hayes, D. A. Reed, and J. Brown, Editors, IEEE Computer Society Press, Dec. 1995, pp. 245–277.

[51] Loretta S. Ellwood and Michael T. Heath, "A Tracing Environment for MPI," *Proc. of MPI Developers Conference*, June 22–23, 1995. Proceedings available on-line from http://www.cse.nd.edu/mpidc95/proceedings.

[52] P. Emrath and D. Padua, "Automatic Detection of Nondeterminancy in Parallel Programs," *ACM SIGPLAN and SIGOPS Workshop on Parallel and Distributed Debugging, Madison*, Wisconsin, May 5-6, 1988.

[53] Domenico Ferrari, *Computer Systems Performance Evaluation*, Prentice-Hall, Inc., 1978.

[54] Domenico Ferrari, "Considerations on the Insularity of Performance Evaluation," *IEEE Transactions on Software Engineering*, June 1986.

[55] D. Ferrari and D. Verma, "A Scheme for Real-Time Channel Establishment in Wide-Area Networks," *IEEE Transactions on lected Area in Communications*, 8(3), 1990, pp.368–379.

[56] C. E. Fineman and P. J. Hontalas, "Selective Monitoring Using Performance Metric Predicates," *Proc. Scalable High-Perf. Comp. Conf.*, IEEE Comp. Soc., 1992.

[57] Ian Foster, *Designing and Building Parallel Programs*, Addison-Wesley, 1995.

[58] Ian Foster, "Tools for Network-Based Supercomputing: Lessons from the I-WAY Experience," presented at *Workshop on Software Tools for High Performance Computing Systems*, Cape Cod, Massachusetts, Oct. 15–18, 1996.

[59] G. Fox, M. Johnson, G. Lyzenga, S. Otto, J. Salmon, and D. Walker, *Solving Problems on Concurrent Processors: Volume 1—General Techniques and Regular Problems*, Prentice-Hall, Inc., 1988.

[60] D. Gannon, "Predicting Performance: Spreadsheets and What-If Questions," presentation at Workshop on Parallel Computer Systems: Software Performance Tools, Santa Fe, October 2–4, 1991.

[61] G. Geist, M. Heath, B. Peyton, and P. Worley, "A Machine-Independent Communication Library," *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, Los Altos: Golden Gate Enterprises, 1990.

[62] G. Geist, M. Heath, B. Peyton, and P. Worley, "A User's Guide to PICL", Technical Report ORNL/TM-11616, Oak Ridge National Laboratory, March 1991.

[63] G. Geist, A. Beguelin, J. Dongarra, W. Jiang, R. Manchek, and V. Sunderam, *PVM*, MIT Press, 1994.

[64] G. Geist, J. Kohl, and P. Papadopoulos, "Visualization, Debugging, and Performance in PVM," in *Debugging and Performance Tuning for Parallel Computer Systems*, edited by M. L. Simmons, A. H. Hayes, D. A. Reed, and J. Brown, IEEE Computer Society Press, Dec. 1995, pp. 65–77.

[65] G. Geist, J. Kohl, and P. Papadopoulos, "Providing Fault-Tolerance, Visualization, and Steering of Parallel Applications," *Proc. of Workshop on Environments and Tools for Parallel Scientific Computing*, August 1996.

[66] Martin Gergeleit, J. Kaiser, and H. Streich, "DIRECT: Towards a Distributed Object-Oriented Real-Time Control System," Technical Report, 1996. Available from http://borneo.gmd.de:80/RS/Papers/direct/direct.html.

[67] R. Glenn, and D. Pryor, "Instrumentation for a Massively Parallel MIMD Application," *Journal of Parallel and Distributed Computing*, 12(3), July 1991.

[68] E. Gelenbe, G. Pujolle, and J. C. C. Nelson, *Introduction to Queuing Networks*, John Wiley, 1987.

[69] W. Gropp, E. Lusk, and A. Skjellum, *Using MPI: Portable Parallel Programming with the Message-Passing Interface*, MIT Press, 1994.

[70] Morris Grossman, "Modeling Reality," *IEEE Spectrum*, 29(9), Sep. 1992, pp. 56–60.

[71] Weiming Gu, Greg Eisenhauer, Eileen Kraemer, Karsten Schwan, John Stasko, and Jeffrey Vetter, "Falcon: On-line Monitoring and Steering of Large-Scale Parallel Programs," Technical Report GIT–CC–94–21, 1994.

[72] Shanti S. Gupta, "Order Statistics from the Gamma Distribution," *Technometrics*, 2(2), May 1960, pp. 243–262.

[73] John Gustafson, Diane Rover, Stephen Elbert, and Michael Carter. "The Design of a Scalable, Fixed-time Computer Benchmark," *Journal of Parallel and Distributed Computing*, 12(4), August 1991.

[74] Bjoern Haake, Klaus E. Schauser, and Chris Scheiman, "Profiling a Parallel Language Based on Fine-Grained Communication," *Proc. of Supercomputing '96*, Pittsburgh, Pennsylvania, Nov. 17–22, 1996. Available on-line from http:// www.supercomp.org/sc96/proceedings/SC96PROC/SCHAUSER/INDEX.HTM.

[75] S. Hackstadt, A. Malony, B. Mohr, "Scalable Performance Visualization for Data-Parallel Programs," *Proceedings of the Scalable High Performance Computing Conference (SHPCC)*, Knoxville, Tennessee, May 1994.

[76] Ming C. Hao, Alan H. Karp, Milon Mackey, Vineet Singh, and Jane Chien, "On-the-Fly Visualization and Debugging of Parallel Programs," *Proc. of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94) Tools Fair*, Durham, North Carolina, Jan. 31– Feb. 2, 1994.

[77] Ming C. Hao, Alan H. Karp, Abdul Waheed, and Mehdi Jazayeri, "VIZIR: An Integrated Environment for Distributed Program Visualization," *Proc. of Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '95) Tools Fair*, Durham, North Carolina, Jan. 1995, pp. 288–292.

[78] Ming C. Hao, Abdul Waheed, Alan Karp, and Mehdi Jazayeri, "Multiple Views of Parallel Application Execution," in *Debugging and Performance Tuning for Parallel Computer Systems*, edited by M. L. Simmons, A. H. Hayes, D. A. Reed, and J. Brown, IEEE Computer Society Press, Dec. 1995, pp. 199–206.

[79] R. Harrison, L. Zitzman, G. Yoritomo, "High Performance Distributed Computing Program (HiPer-D)—Engineering Testbed One (T1) Report," Technical Report, Naval Surface Warfare Center, Dahlgren, Virginia, Nov. 1995.

[80] T. Hasegawa, H. Takagi, and Y. Takahashi, editors, *Performance of Distributed and Parallel Systems*, Elsevier Science Publishers B.V., 1989.

[81] Michael T. Heath and Jennifer A. Etheridge, "Visualizing the Performance of Parallel Programs," *IEEE Software*, 8(5), September 1991, pp. 29–39.

[82] M. Heath, A. Malony, and D. Rover, "The Visual Display of Parallel Performance Data," *IEEE Computer/IEEE Parallel and Distributed Technology* special theme issues on Performance Evaluation Tools for Parallel and Distributed Computer Systems, November 1995.

[83] B. R. Helm and A. D. Malony, "Automating Performance Diagnosis: a Theory and Architecture," *Proceedings of International Workshop on Computer Performance Measurement and Analysis (PERMEAN '95)*, Beppu, Japan, Aug. 20--23, 1995, pp. 84–91.

[84] High Performance Fortran Forum, "High Performance Fortran Language Specifications: Version 1.0," Technical Report CRPC-TR92225, Center for Research on Parallel Computation, Rice University, Houston, Texas, 1993.

[85] J. K. Hollingsworth and B. P. Miller, "Dynamic Control of Performance Monitoring on Large Scale Parallel Systems," *Proc. of Int. Con. on Supercomputing*, Tokyo, Japan, July 19–23, 1993, pp. 185–194.

[86] J. K. Hollingsworth, B. P. Miller, Jon Cargille, "Dynamic Program Instrumentation for Scalable Performance Tools," *Proc. of Scalable High-Performance Computing Conference*, Knoxville, Tenn., 1994, pp. 841–850.

[87] J. K. Hollingsworth and B. P. Miller, "An Adaptive Cost Model for Parallel Program Instrumentation," *Proc. of EuroPar '96*, Lyon, France, August 1996, Volume 1, pp. 88–98.

[88] J. K. Hollingsworth, James E. Lump, Jr., and Barton P. Miller, "Techniques for Performance Measurement of Parallel Programs," in *Parallel Computers: Theory and Practice, IEEE Press*, 1995.

[89] J. K. Hollingsworth and B. P. Miller, Marcelo J. R. Goncalves, Oscar Naim, Zhichen Xu and Ling Zheng, "MDL: A Language and Compiler for Dynamic Program Instrumentation," Technical Report, 1996.

[90] Alfred A. Hough and Janice E. Cunny, "Belvedere: Prototype of a Pattern-Oriented Debugger for Highly Parallel Computation," *Proceedings of the 1987 International Conference on Parallel Processing*, pp. 735–738, 1987.

[91] Alfred A. Hough and Janice E. Cuny, "Perspective Views: A Technique for Enhancing Parallel Program Visualization," *Proc. 1990 Int. Conf. on Par. Proc.*, IEEE Comp. Soc., 1990.

[92] J. V. Huber, C. L. Elford, D. A. Reed, A. A. Chien, and D. S. Blumenthal, "PPFS: A High-Performance Portable Parallel File System," *Proceedings of the Ninth ACM International Conference on Supercomputing*, July 1995.

[93] Jau-Hsiung Huang and Leonard Kleinrock, "Performance Evaluation of Dynamic Sharing of Processors in Two-Stage Parallel Processing Systems," *IEEE Transactions on Parallel and Distributed Systems*, 4(3), March 1993.

[94] Herman D. Hughes, "Generating a Drive Workload from Clustered Data," *Computer Performance*, 5(1), March 1984, pp. 31–37.

[95] Watts S. Humphrey and Nozer D. Singpurwalla, "Predicting (Individual) Software Productivity," *IEEE Transactions on Software Engineering*, 17(2), February 1991.

[96] Giuseppe Iazeolla and Francesco Marinuzzi, "LISPACK—A Methodology and Tool for the Performance Analysis of Parallel Systems and Algorithms," *IEEE Transactions on Software Engineering*, 19(5), May 1993.

[97] *IEEE Standard for Information Technology (Std 1003.1b-1993), Portable Operating System Interface (POSIX)—Part 1: System Application Program Interface (API)*, IEEE, 1994.

[98] *IEEE Transactions on Parallel and Distributed Systems*, special issue on measurement and evaluation, 3(2), November 1992.

[99] Anil K. Jain, *Fundamentals of Digital Image Processing*, Prentice-Hall, Inc., 1989.

[100] Raj Jain, *The Art of Computer Systems Performance Analysis—Techniques for Experimental Design, Measurement, Simulation, and Modeling*, John Wiley & Sons, Inc., 1991.

[101] Jeffrey Joyce, Greg Lomow, Konrad Slind, and Brian Unger, "Monitoring Distributed Systems," *ACM Transactions on Computer Systems*, 5(2), May 1987, pp. 121–150.

[102] Kai Hwang, *Advanced Computer Architecture*, McGraw-Hill, 1993.

[103] Maurice Kendall and Keith Ord, *Time Series*, Edward Arnold, 1990.

[104] Carol Kilpatrick and Karsten Schwan, "ChaosMON—Application-Specific Monitoring and Display of Performance Information for Parallel and Distributed Systems," *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, Santa Cruz, California, May 20–21, 1991.

[105] Doug Kimmelman and Dror Zernik, "On-the-Fly Topological Sort—A Basis for Interactive Debugging and Live Visualization of Parallel Programs," *Proceedings of the ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, California, May 17–18, 1993.

[106] Leonard Kleinrock, *Queuing Systems—Volume II: Computer Applications*, John Wiley, 1976.

[107] Leonard Kleinrock and Willard Korfhage, "Collecting Unused Processing Capacity: An Analysis of Transient Distributed Systems," *IEEE Transactions on Parallel and Distributed Systems*, 4(4), May 1993, pp. 535–546.

[108] D. Knuth, *The Art of Computer Programming*, Addison-Wesley, 1981.

[109] Bernd Konemann, Ben Bennetts, Najmi Jarwala, and Benoit Nadeau-Dostie, "Built-In Self-Test: Assuring System Integrity," *IEEE Computer*, 29(11), November 1996, pp. 39–45.

[110] Eileen Kraemer and John T. Stasko, "The Visualization of Parallel Systems: An Overview," *Journal of Parallel and Distributed Computing*, 18(2), June 1993.

[111] K. Kunchithapadam and B. P., Miller, "Integrating a Debugger and a Performance Tool for Steering," in *Debugging and Performance Tuning for Parallel Computer Systems*, M. L. Simmons, A. H. Hayes, D. A. Reed, and J. Brown, Editors, IEEE Computer Society Press, Dec. 1995, pp. 53–63.

[112] L. Lamport, L., "Time, Clocks, and the Ordering of Events in a Distributed System," *Communications of the ACM*, 21(7), July 1978, pp. 558–565.

[113] F. H. Lange, *Correlation Techniques*, London Iliffe Books Ltd., 1967.

[114] F. Lange, Reinhold Kroger, and Martin Gergeleit, "JEWEL: Design and Implementation of a Distributed Measurement System.". *IEEE Transactions on Parallel and Distributed Systems*, 3(6), November 1992, pp. 657-671. Also available on-line from http://borneo.gmd.de:80/RS/Papers/JEWEL/JEWEL.html.

[115] Stephen S. Lavenberg, editor, *Computer Performance Modeling Handbook*, Academic Press, 1983.

[116] Averill M. Law and W. D. Kelton, *Simulation Modeling and Analysis*, McGraw-Hill, Inc., 1991.

[117] Edward D. Lawzowska, John Zahorjan, G. Scott Graham, and Kenneth C. Sevcik, *Quantitative System Performance—Computer System Analysis Using Queuing Network Models*, Prentice-Hall, 1984.

[118] Thomas J. Leblanc, John M. Mellor-Crummey, and Robert J. Fowler, "Analyzing Parallel Program Executions Using Multiple Views," *Journal of Parallel and Distributed Computing*, 9(2), June 1990.

[119] Hwa-Chun Lin and C. S. Raghvendra, "An Approximate Analysis of the Join the Shortest Queue (JSQ) Policy," *IEEE Transactions on Parallel and Distributed Systems*, 7(3), March 1996.

[120] C. Liu and J. Layland, "Scheduling Algorithms for Multiprogramming in a Hard Real-Time Environment," Journal of the ACM, 20(1), 1973, pp. 46–61.

[121] C. Locke, D. Vogel, and T. Mesler, "Building a Predictable Avionics Platform in Ada: A Case Study," *Proc. of the IEEE Real-Time Systems Symposium*, 1991, pp. 181–189.

[122] C. B. Lynch and G. A. Dumont, "Control Loop Performance Monitoring," *IEEE Transactions on Control Systems Technology*, 4(2), March 1996.

[123] M. H. MacDougall, *Simulating Computer Systems—Techniques and Tools*, The MIT Press, 1987.

[124] Michael R. Macedonia and Donald P. Brutzman, "MBone Provides Audio and Video Across the Internet," *IEEE Computer*, 27(4), April 1994, pp. 30–36.

[125] A. D. Malony, D. A. Reed, and H. A. G. Wijshoff, "Performance Measurement Intrusion and Perturbation Analysis," *IEEE Transactions on Parallel and Distributed Systems*, 3(4), July 1992, pp. 433–450.

[126] A. Malony, B. Mohr, P. Beckman, D. Gannon, S. Yang, F. Bodin, and S. Kesavan, "Implementing a Parallel C++ Runtime System for Scalable Parallel Systems," *Proceedings of Supercomputing '93*, Portland, Oregon, November 15–19, 1993.

[127] A. D. Malony, "Measurement and Monitoring of Parallel Programs," *Tutorial, Sigmetrics '1994*, Nashville, Tennessee, May 16–20, 1994.

[128] Deborah T. Marr, Subramanian Natarajan, Shreekant Thakkar, and Richard Zucker, "Multiprocessor Validation of the Pentium Pro," *IEEE Computer*, 29(11), November 1996, pp. 47–53.

[129] M. A. Marsan, G. Balbo, and G. Conte, *Performance Models of Multiprocessor Systems*, The MIT Press, 1986.

[130] Margaret Martonosi, Douglas W. Clark, and Ganesh Lakshminarayanan, "The SHRIMP Performance Monitoring System: Design and Application," Princeton University, 1995.

[131] Philip K. McKinley and Christian Trefftz, "Multisim: A Simulation Tool for the Study of Large-Scale Multiprocessors," *Proceedings of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '93)*, San Diego, California, January 1993.

[132] Clifford W. Mercer and Ragunathan Rajkumar, "Interactive Interface and RT-Mach Support for Monitoring and Controlling Resource Management," *Proceedings of Real-Time Technology and Applications Symposium*, Chicago, Illinois, May 15-17, 1995, pp. 134–139.

[133] Message Passing Interface Forum, "MPI: A Message-Passing Interface Standard," *International Journal of Supercomputer Applications*, 8(3), Fall/Winter 1994, pp. 159–416.

[134] Barton P. Miller, Morgan Clark, Jeff Hollingsworth, Steven Kierstead, Sek-See Lim, and Timothy Torzewski, "IPS-2: The Second Generation of a Parallel Program Measurement System," *IEEE Transactions on Parallel and Distributed Systems*, 1(2), April 1990, pp. 206–217.

[135] Barton P. Miller, "What to Draw? When to Draw? An Essay on Parallel Program Visualization," *Journal of Parallel and Distributed Computing*, 18(2), June 1993.

[136] Barton P. Miller, Jonathan M. Cargille, R. Bruce Irvin, Krishna Kunchithapadam, Mark D. Callaghan, Jeffrey K. Hollingsworth, Karen L. Karavanic, and Tia Newhall, "The Paradyn Parallel Performance Measurement Tool," *IEEE Computer*, 28(11), November 1995, pp. 37–46.

[137] Barton P. Miller, "Making Real Programs Explode: A Simple Application of Random Testing," Technical Report, University of Wisconsin at Madison, 1995. Available on-line from http://www.cs.wisc.edu/~bart/fuzz/fuzz.html.

[138] A. Mink, R. Carpenter, G. Nacht, and J. Roberts, "Multiprocessor Performance Measurement Instrumentation," *IEEE Computer*, 23(9), September 1990, pp. 63–75.

[139] Brian T. Murrary and John P. Hayes, "Testing ICs: Getting to the Core of the Problem," *IEEE Computer*, 29(11), November 1996, pp. 32–38.

[140] John D. Musa, "Software-Reliability-Engineering Testing," *IEEE Computer*, 29(11), November 1996, pp. 61–68.

[141] Arun K Nanda and Lionel M. Ni, "MAD Kernels: An Experimental Testbed to Study Multiprocessor Memory System Behavior," *IEEE Transactions on Parallel and Distributed Systems*, 7(2), February 1996.

[142] Lionel M. Ni and Philip K. McKinley, "A Survey of Wormhole Routing Techniques in Direct Networks," *IEEE Computer*, February 1993.

[143] Kathleen Nicholas and Paul W. Oman, "Navigating Complexity to Achieve High Performance," *IEEE Software*, 8(5), September 1991.

[144] Ahmed K. Noor and Samuel L. Venneri, "A Perspective on Computational Structures Technology," *IEEE Computer*, 26(10), October 1993.

[145] Gary J. Nutt and Adam J. Griff, "Extensible Parallel Program Performance Visualization," *Proc. of Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '95)*, Durham, North Carolina, Jan. 1995.

[146] David M. Ogle, Karsten Schwan, and Richard Snodgrass, "Application-Dependent Dynamic Monitoring of Distributed and Parallel Systems," *IEEE Transactions on Parallel and Distributed Systems*, 4(7), July 1993, pp. 762–778.

[147] Karen F. O'Donoghue and Timothy R. Plunkett, "Development and Validation of Network Clock Measurement Techniques," *Proc. of the Fourth International Workshop on Parallel and Distributed Real-Time Systems*, Honolulu, Hawaii, April 15–16, 1996, pp. 65–68.

[148] *OMIS—On-Line Monitoring Interface Specifications*. Accessible from http://www-bode.informatik.tu-muenchen.de/~omis.

[149] Cherri M. Pancake and Sue Utter, "Models for Visualization in Parallel Debuggers," *Supercomputing '89*, November 1989.

[150] Cherri M. Pancake, "Software Support for Parallel Computing: Where Are We Headed?", *Comm. ACM*, Nov. 1991.

[151] Cherri M. Pancake, "The Emperor Has No Clothes: What HPC Users Need to Say and HPC Vendors Need to Hear,", *Supercomputing '95, invited talk*, San Diego, Dec. 3–8, 1995.

[152] Athanasios Papoulis, *Signal Analysis*, McGraw-Hill, Inc., 1977.

[153] Peyton Z. Peebles, *Probability, Random Variables, and Random Signal Principles*, McGraw-Hill, Inc., 1993.

[154] Paul E. Pfeiffer, *Concepts of Probability Theory*, Dover Publications, Inc., 1978.

[155] Sol D. Prensky and Richard L. Castellucis, *Electronic Instrumentation*, Prentice-Hall, 1982.

[156] John G. Proakis and Dimitris G. Manolakis, *Digital Signal Processing--Principles, Algorithms, and Applications*, Macmillan Publishing Company, 1992.

[157] Joseph A. Profeta III, Nikos P. Andrianos, Bing Yu, Barry W. Johnson, Todd A. DeLong, David Guaspari, and Damir Jamsek, "Safety-Critical Systems Built with COTS," *IEEE Computer*, 29(11), November 1996, pp. 54–60.

[158] L. R. Rabiner and R. W. Schafer, *Digital Processing of Speech Signals*, Prentice-Hall, 1978.

[159] Stephen A. Rago, *Unix System V Network Programming*, Addison-Wesley, 1993.

[160] Ragunathan Rajkumar, Mike Gagliardi, and Lui Sha, "The Real-Time Publisher/ Subscriber Inter-Process Communication Model for Distributed Real-Time Systems: Design and Implementation," *Proceedings of Real-Time Technology and Applications Symposium*, Chicago, Illinois, May 15–17, 1995, pp. 66–75.

[161] Daniel A. Reed, Ruth A. Aydt, Tara M. Madhyastha, Roger J. Noe, Keith A. Shields, Bradley W. Schwartz, "The Pablo Performance Analysis Environment," Dept. of Comp. Sci., Univ. of Ill., 1992.

[162] Daniel A. Reed, "Building Successful Performance Tools," Presented in ARPA PI Meeting, July 1995. Available on-line from http://www-pablo.cs.uiuc.edu/June95-ARPA/index.html.

[163] Daniel A. Reed, Jeffrey S. Brown, Ann H. Hayes, and Margaret L. Simmons, "Performance and Debugging Tools: A Research and Development Checkpoint," in *Debugging and Performance Tuning for Parallel Computer Systems*, M. L. Simmons, A. H. Hayes, D. A. Reed, and J. Brown, Editors, IEEE Computer Society Press, Dec. 1995, pp. 1–22.

[164] Sidney I. Resnick, *Adventures in Stochastic Processes*, Birkhauser, 1992.

[165] B. Ries, R. Anderson, D. Breazeal, K. Callaghan, E. Richards, and W. Smith, "The Paragon Performance Monitoring Environment," *Proceedings of Supercomputing '93*, Portland, Oregon, Nov. 15–19, 1993, pp. 850–859.

[166] Sheldon M. Ross, *Introduction To Probability Models—Fourth Edition*, Academic Press, 1989.

[167] Diane T. Rover, "Visualization of Program Performance on Concurrent Computers," *Ph.D. Dissertation*, Dept. of Electrical Engineering and Computer Engineering, Iowa State University, December 1989.

[168] Diane T. Rover, "A Performance Visualization Paradigm for Data-Parallel Computing," *Proceedings of the 25th Hawaii International Conference on System Sciences*, New York: IEEE Computer Society, 1992, pp. 146–160.

[169] Diane T. Rover, A. Waheed, and M. Doetsch, "Advanced Methods of Performance Data Processing and Analysis," *Proceedings of the Seventh International Parallel Processing Symposium*, Newport Beach, April 13-16, 1993, pp. 609–613.

[170] Diane T. Rover and A. Waheed, "Multiple-Domain Analysis Methods," *Third ACM/ONR Workshop on Parallel and Distributed Debugging*, San Diego, May 17-18, 1993, pp. 53–63. Proceedings appeared in *ACM SIGPLAN Notices*, 28(12), December 1993.

[171] Diane T. Rover and Charles T. Wright, "Visualizing the performance of SPMD and Data-Parallel Programs," *Journal of Parallel and Distributed Computing*, 18(2), June 1993.

[172] Diane T. Rover, "Performance Evaluation: Integrating Techniques and Tools into Environments and Frameworks," *Roundtable, Supercomputing '94*, Washington DC, November 14–18, 1994.

[173] Diane T. Rover, Allen D. Malony, and Gary J. Nutt, "Summary of Working Group on Integrated Environments Vs. Toolkits," in *Debugging and Performance Tuning for Parallel Computing Systems*, edited by A. Hayes, M. Simmons, J. Brown, and D. Reed, IEEE Computer Society Press, May 1996.

[174] Diane T. Rover, Abdul Waheed, Matt W. Mutka, and Aleksandar Bakic, "The Application of Software Tools to Complex Systems: An Overview," to appear in *IEEE Parallel and Distributed Technology*, 1997.

[175] Wilson J. Rugh, *Linear System Theory*, Prentice-Hall, 1993.

[176] James Rumbaugh, Michael Blaha, William Premerlani, Frederick Eddy, and William Lorensen, *Object-Oriented Modeling and Design*, Prentice Hall, 1991.

[177] M. Ruschitzka, editor, *Computer Systems: Performance and Simulation*, Elsevier Science Publishers B.V., 1986.

[178] Subhash Saini and David Bailey, "NAS Parallel Benchmark Results," Report NAS-95-021, NASA Ames Research Center, December 1995. Available on-line from: http://www.nas.nasa.gov/NAS/TechReports/NASreports/NAS-95-021/NAS-95-021.html.

[179] Meera Sampath, Raja Sengupta, Stephane Lafortune, Kasim Sinnamohideen, and Demosthenis C. Teneketzis, "Failure Diagnosis Using Discrete-Event Models," *IEEE Transactions on Control Systems Technology*, 4(2), March 1996.

[180] Gary M. Sandquist, *Introduction to system science*, Prentice-Hall, 1985.

[181] Sekhar R. Sarukkai and Jerry C. Yan, "Event-Based Study of the Effect of Execution Environment on Parallel Program Performance," *Proc. of Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '96)*, San Jose, Feb. 1–3, 1996, pp. 257–261.

[182] Charles H. Sauer and K. M. Chandy, *Computer Systems Performance Modeling*, Prentice-Hall, Inc., 1981.

[183] Robert J. Schilling and Hua Lee, *Engineering Analysis—A Vector Space Approach*, John Wiley & Sons, 1988.

[184] L. Schnell, editor, *Technology of Electrical Measurements*, John Wiley & Sons, 1993.

[185] Beth A. Schroeder, "On-Line Monitoring: A Tutorial," *IEEE Computer*, 28(6), June 1995, pp. 72–78.

[186] S. K. Setia, M. S. Squillante, and S. K. Tripathi, "Analysis of Processor Allocation in Multiprogrammed, Distributed-Memory Parallel Processing Systems," *IEEE Transactions on Parallel and Distributed Systems*, 5(4), April 1994.

[187] Chia Shen, Krithi Ramamritham, and John A. Stankovic, "Resource Reclaiming in Multiprocessor Real-Time Systems," *IEEE Transactions on Parallel and Distributed Systems*, 4(4), April 1993.

[188] M. Simmons, R. Koskela, I. Bucher, editors, *Instrumentation for Future Parallel Computing Systems*, ACM & Addison-Wesley, 1989.

[189] M. Simmons and R. Koskela, editors, *Performance Instrumentation and Visualization*, ACM & Addison-Wesley, 1990.

[190] M. Simmons, A. Hayes, J. Brown, and D. Reed, editors, *Debugging and Performance Tuning for Parallel Computing Systems*, IEEE Computer Society Press, 1996.

[191] Evgenia Smirni and Daniel A. Reed, "Parallel I/O: Problems and Solutions," presented at *Workshop on Software Tools for High Performance Computing Systems*, Cape Cod, Massachussetts, Oct. 15–18, 1996.

[192] R. Snodgrass, "A Relational Approach to Monitoring Complex Systems," *ACM Transactions on Computer Systems*, 6(2), May 1988.

[193] William Stallings, *Data and Computer Communications*, Macmillan Publishing Company, 1991.

[194] William Stallings, *Operating Systems*, Macmillan Publishing Company, 1992.

[195] John Stasko and Eileen Kraemer, "A Methodology for Building Application-Specific Visualization of Parallel Programs," *Journal of Parallel and Distributed Computing*, 18(2), June 1993.

[196] W. R. Stevens, *UNIX Network Programming*, Prentice-Hall, Inc., 1990.

[197] H. Stone, *High-Performance Computer Architecture*, Addison-Wesley, 1987.

[198] Alexander D. Stoyenko, Phillip A. Laplante, Robert Harrison, and Thomas J. Marlowe, "Doubling the Engineer's Utility," *IEEE Spectrum*, 31(12), December 1994, pp. 32–39.

[199] J. Strosnider, T. Marchok, and J. Lehoczky, "Advanced Real Time Scheduling Using the IEEE 802.5 Token Ring," *Proc. of the IEEE Real-Time Systems Symposium*, 1988, pp. 42–52.

[200] Andrew S. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, 1995.

[201] LaMar K. Timothy, *State space analysis: an introduction*, McGraw-Hill, 1968.

[202] Jeffrey J.P. Tasi and Steve J.H. Yang, *Monitoring and Debugging of Distributed Real-Time Systems, IEEE Computer Society Press*, 1995.

[203] T. F. Tsuei and M. K. Vernon, "A Multiprocessor Bus Design Model Validated by System Measurement," *IEEE Transactions on Parallel and Distributed Systems*, 3(6), November 1992.

[204] Edward R. Tufte, *The Visual Display of Quantitative Information*, Graphics Press, Cheshire, Connecticut, 1983.

[205] Edward R. Tufte, *Envisioning Information*, Graphics Press, Cheshire, Connecticut, 1990.

[206] Sue Utter-Honig and Cherri M. Pancake, "Graphical Animation of Parallel Fortran Programs," *Supercomputing '91*, November 18 - 22, 1991.

[207] Andreas Vogel, Brigitte Kerherve, Gregor von Bochmann, and Jan Gecsei, "Distributed Multimedia and QOS: A Survey," *IEEE Multimedia*, Summer '1995, pp. 10–19.

[208] Abdul Waheed and D. T. Rover, "Performance Visualization of Parallel Programs," *Visualization '93*, San Jose, California, Oct. 25-29, 1993.

[209] Abdul Waheed, B. Kronmuller, and D. T. Rover, "A Matrix Approach to Performance Data Modeling, Analysis and Visualization," *Proc. of International Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94)*, Durham, North Carolina, Jan. 31-Feb. 2, 1994.

[210] Abdul Waheed, B. Kronmuller, Roomi Sinha, and D. T. Rover, "A Toolkit for Advanced Performance Analysis," *Proc. of Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '94) Tools Fair*, Durham, North Carolina, Jan. 31– Feb. 2, 1994, pp. 376–380.

[211] Abdul Waheed, Vincent Melfi, and Diane T. Rover, "A Model for Instrumentation System Management in Concurrent Systems," *Proceedings of the Twenty Eighth Hawaii International Conference on System Sciences*, Maui, Hawaii, Jan. 3-6, 1995, pp. 432–441.

[212] Abdul Waheed and Diane T. Rover, "A Schema for Specifying and Classifying Instrumentation Systems," *Proceedings of International Workshop on Computer Performance Measurement and Analysis (PERMEAN '95)*, Beppu, Japan, Aug. 20--23, 1995, pp. 42–51.

[213] Abdul Waheed, and Diane T. Rover, "A Structured Approach to Instrumentation System Development and Evaluation," *Proceedings of Supercomputing '95*, San Diego, California, Dec. 4–8, 1995.

[214] Abdul Waheed, Herman D. Hughes, and Diane T. Rover, "A Resource Occupancy Model for Evaluating Instrumentation System Overheads," *Proceedings of the 20th Annual International Conference of the Computer Measurement Group (CMG '95)*, Nashville, Tennessee, Dec. 3–8, 1995, pp. 1212–1223.

[215] Abdul Waheed and Diane T. Rover, "Performance Evaluation of an Integrated Instrumentation System," *Proc. of Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '96)*, San Jose, Feb. 1–3, 1996.

[216] Abdul Waheed, Diane T. Rover, Aleksandar Bakic, Matt W. Mutka, and David Pierce, "Vista: A Framework for Instrumentation System Design for Multidisciplinary Applications," *Proc. of Int. Workshop on Modeling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS '96) Tools Fair*, San Jose, Feb. 1–3, 1996.

[217] Abdul Waheed and Diane T. Rover, "Instrumentation Systems for Parallel Tools, "*Chapter in the Book State-of-the-Art in Performance Modeling and Simulation: Advanced Computer Systems*, edited by K. Bagchi, J. Walrand, and G. Zobrist, Gordon and Breach Publishers Inc., 1996.

[218] Abdul Waheed, Diane T. Rover, and Jeffrey K. Hollingsworth, "Modeling, Evaluation, and Testing of Paradyn Instrumentation System," *Proc. of Supercomputing '96*, PIttsburgh, Pennsylvania, Nov. 17–22, 1996. Available on-line from http:// www.supercomp.org/sc96/proceedings/SC96PROC/WAHEED/INDEX.HTM.

[219] Abdul Waheed, Diane T. Rover, Hough Smith, Matt W. Mutka, and Aleksandar Bakic, "Modeling, Evaluation, and Adaptive Control of an Instrumentation System," Technical Report, November, 1996.

[220] Edward J. Wegman and James G. Smith, editors, *Statistical Signal Processing*, Marcel Dekker, Inc., 1984.

[221] Lonnie R. Welch, Michael W. Masters, and Robert D. Harrison, "Toward a 21st Century Shipboard Computing Infrastructure," Technical Report, Naval Surface Warfare Center, Dahlgren, Virginia, Jan. 1996.

[222] Bernard Widrow and Samuel D. Stearns, *Adaptive Signal Processing*, Prentice-Hall, Inc., 1985.

[223] W. W. Wilcke et al., "The IBM Victor Multiprocessor Project," *Proceedings of the Fourth Conference on Hypercubes, Concurrent Computers, and Applications*, Los Altos: Golden Gate Enterprises, 1990.

[224] Tom Williams, "System-Visualization Tools Help Spot Real-Time Problems," *Computer Design*, August 1994, pp. 50–54.

[225] Alan Wood, "Predicting Software Reliability," *IEEE Computer*, 29(11), November 1996, pp. 69–77.

[226] Paul R. Woodward, "Perspective on Supercomputing: Three Decades of Change," *IEEE Computer*, 29(10), Oct. 1996, pp. 99–111.

[227] *Workshop on Debugging and Performance Tuning of Parallel Computing Systems*, Chatham, Mass., Oct. 3-5, 1994.

[228] *Workshop on Software Tools for High Performance Computing Systems*, Chatham, Mass., Oct 15–18, 1996.

[229] Jerry C. Yan and S. Listgarten, "Intrusion Compensation for Performance Evaluation of Parallel Programs on a Multicomputer," *Proceedings of the Sixth International Conference on Parallel and Distributed systems*, Louisville, KY, Oct. 14–16, 1993.

[230] Jerry C. Yan, "Performance Tuning with AIMS—An Automated Instrumentation and Monitoring System for Multicomputers," *Proc. of the Twenty-Seventh Hawaii Int. Conf. on System Sciences*, Hawaii, January 1994.

[231] Jerry C. Yan, S. R. Sarukkai, and P. Mehra, "Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit,", *Software Practice and Experience*, 25(4), April 1995, pp. 429–461.

[232] Lotfi Asker Zadeh, *Linear system theory; the state space approach*, McGraw-Hill, 1963.

[233] Marco Zagha, Brond Larson, Steve Turner, and Marty Itzkowitz, "Performance Analysis Using the MIPS R10000 Performance Counters," Proceedings of Supercomputing '96, Pittsburgh, Pennsylvania, November 17–22, 1996. Available online from http://www.supercomp.org/sc96/proceedings/SC96PROC/ZAGHA/INDEX.HTM.

[234] W. Zhao, J. Stankovic, and K. Ramamritham, "A Window Protocol for Time Constrained Messages," *IEEE Transactions on Computers*, 39(9), Sep. 1990, pp. 1186–1203.

[235] W. Zhao, J. Stankovic, and K. Ramamritham, "A Multiaccess Window Protocol for Time Constrained Communications," *Proceedings of the 8th International Conference on Distributed Computing Systems. IEEE*, June 1991.