Modeling, Evaluation, and Testing of Paradyn Instrumentation System

Abdul Waheed and Diane T. Rover *
Department of Electrical Engineering
Michigan State University
E-mail: {waheed,rover}@egr.msu.edu

Jeffrey K. Hollingsworth
Department of Computer Science
University of Maryland
E-mail: hollings@cs.umd.edu

Abstract

This paper presents a case study of modeling, evaluating, and testing the data collection services (called an instrumentation system) of the Paradyn parallel performance measurement tool using well-known performance evaluation and experiment design techniques. The overall objective of the study is to use modeling- and simulation-based evaluation to provide feedback to the tool developers to help them choose system configurations and task scheduling policies that can significantly reduce the data collection overheads. We develop and parameterize a resource occupancy model for the Paradyn instrumentation system (IS) for an IBM SP-2 platform. This model is parameterized with a measurement-based workload characterization and subsequently used to answer several "what if" questions regarding configuration options and two policies to schedule instrumentation system tasks: collect-and-forward (CF) and batch-and-forward (BF) policies. Simulation results indicate that the BF policy can significantly reduce the overheads. Based on this feedback, the BF policy was implemented in the Paradyn IS as an option to manage the data collection. Measurement-based testing results obtained from this enhanced version of the Paradyn IS are reported in this paper and indicate more than 60% reduction in the direct IS overheads when the BF policy is used.

1 Introduction

Application-level software *instrumentation systems* (*IS*s) collect runtime information from parallel and distributed systems. This information is collected to serve various purposes, for example, evaluation of program execution on *high performance computing and communication* (*HPCC*) systems [23], monitoring of distributed real-time control systems [3,10], resource management for real-time systems [18], and administration of enterprise-wide transaction processing systems [1]. In each of these application domains, different demands may be placed on

* This work was supported in part by DARPA contract No. DABT 63-95-C-0072 and .National Science Foundation grant ASC-9624149. Jeffrey K. Hollingsworth was supported in part by DOE grant DE-FG02-93ER25176 and NIST CRA award 70-NANB-5H0055.

the *IS* and it should be designed accordingly. In this paper, we present a case study on *IS* design; we apply a structured development approach to the instrumentation system of the Paradyn parallel performance measurement tool [20]. This structured approach is based on modeling and simulating the *IS* to answer several "what-if" questions regarding possible configurations and scheduling policies to collect and manage runtime data [28]. The Paradyn *IS* is enhanced based on the initial feedback provided by the modeling and simulation process. Measurement-based testing validates the simulation-based results and shows more than 60% reduction in the data collection overheads for two applications from the *NAS* benchmark suite executed on an IBM SP-2 system. Using *principal component analysis* on these measurements, we also show that the reduction of overheads is not affected by the choice of an application program.

A rigorous system development process typically involves evaluation and testing prior to system production or usage. In *IS* development, formal evaluation of options for configuring modules, scheduling tasks, and instituting policies should occur early. Testing then validates these evaluation results and qualifies other functional and non-functional properties. Finally, the *IS* is deployed on real applications. Evaluation and testing require a model for the *IS* and adequate characterization of the workload that drives the model. The model can be evaluated analytically or through simulations to provide feedback to *IS* developers. The model and workload characterization also facilitate testing by highlighting performance-critical aspects. In this paper, we focus on evaluation and testing of the Paradyn *IS*.

One may ask if such rigor is needed in *IS* development. The *IS* represents enabling technology of growing importance for effectively using parallel/distributed systems. The *IS* often supports tools accessed by end-users; the user typically sees the tool and not the *IS*. Consequently, tools are scrutinized, and the *IS* and its overheads receive little attention. Users may be unaware of the impact of the *IS*. Unfortunately, the *IS* can perturb the behavior of the application [17], degrading the performance of an instrumented application program from 10% to more than 50% according to various measurement-based studies [8,19]. 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 [24]. Toward this end, tool developers have implemented adaptive *IS* management approaches; for instance, Paradyn's dynamic *cost model*

[12] and Pablo's user-specified (static) *tracing levels* [23]. With these advancements come increased complexity and more design decisions. Dealing with these design decisions is the topic of this paper.

A *Resource OCCupancy* (*ROCC*) model for the Paradyn *IS* is developed and parameterized in Section 2. We simulate the *ROCC* model to answer a number of interesting "what-if" questions regarding the performance of the *IS* in Section 3. Modifications to the Paradyn *IS* design are tested in Section 4 to assess their impact on *IS* performance. We conclude with a discussion of the contributions of this work to the area of parallel tool development.

2 A Model for Paradyn IS

In this section, we introduce Paradyn and present a model for its IS. Paradyn is a tool 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 [20]. The Paradyn parallel performance measurement tool runs on TMC CM-5, IBM SP-2, and clusters of Unix workstations. We have modeled the Paradyn *IS* for an IBM SP-2 system. 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 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.

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 [11]. 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 1 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.

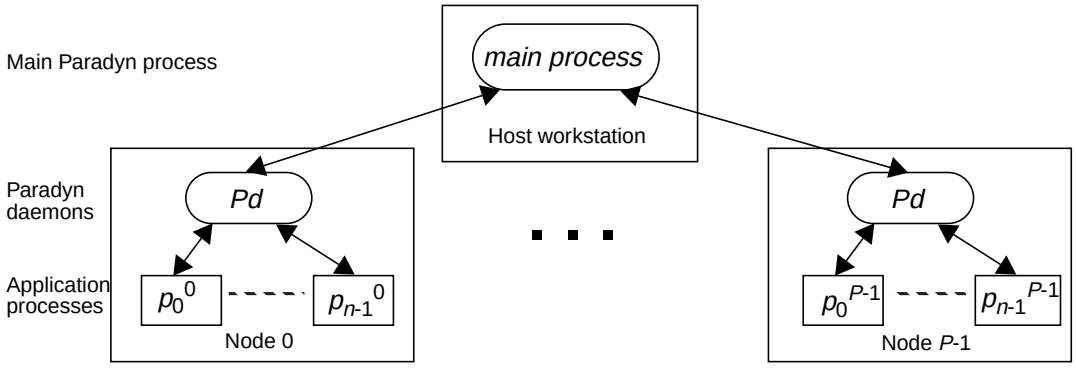


Figure 1. An overview of the Paradyn *IS* [20].

2.1 Queuing Network Model

The Paradyn *IS* can be represented by a queuing network model, as shown in Figure 2. 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 distributed system show identical behavior during the execution of an *SPMD* program. Since the focus of this study is resource sharing among processes at a node, we consider only one subnetwork at a given node and apply the *ROCC* model, introduced in the next section, for a detailed evaluation.

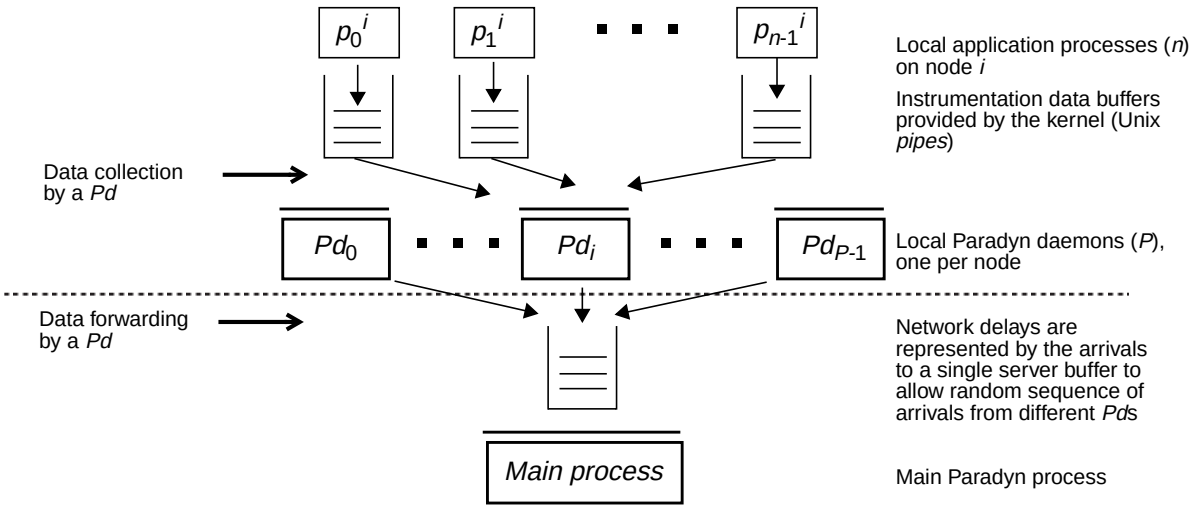


Figure 2. A model for the Paradyn instrumentation system.

Figure 2 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. Objectives of our modeling include evaluating *IS* overheads due to resource sharing, identifying any *IS*-induced performance bottlenecks, and determining desirable operating conditions for the *IS*.

Scheduling Policies for Data Forwarding. Two possible options for a Paradyn daemon to schedule data collection and data forwarding are *collect-and-forward* (*CF*) and *batch-and-forward* (*BF*). Under the *CF* scheduling policy, the *Pd* collects a sample from an instrumented application process and immediately forwards it to the main 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, which is then forwarded to the main Paradyn process.

Metrics. Two performance metrics are of interest for this study: *direct overhead* due to data collection and *throughput* of data forwarding. Direct overhead represents the *occupancy time by the IS of a shared system resource*. It quantifies the contention between application and *IS* processes for the shared resources on a particular node of the system. A lower value of the direct overhead is desirable. Throughput 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. High throughput is desirable particularly in cases where Paradyn assists real-time adaptive steering of the application. Throughput of data forwarding by a Paradyn daemon is directly related to the monitoring latency. Monitoring latency has been defined as the amount of time between the generation of instrumentation data and its receipt at a logically central collection facility [8]. A higher throughput means lower monitoring latency and vice versa.

Simulation-based experiments presented in Section 3 calculate these two metrics to help answer a number of "what-if" questions.

2.2 Resource Occupancy Model

This subsection introduces the *Resource OCCupancy* (*ROCC*) model and its application to isolating the overheads due to non-deterministic sharing of resources between the Paradyn *IS* and application processes [29]. The *ROCC* model, founded on traditional modeling techniques, consists of three components: *system resources*, *requests*, and *management policies*. *Resources* are shared among (instrumented) application processes, other user and system processes, and *IS* processes; for example, CPU, network, and I/O devices. *Requests* are demands from application, other user, and *IS* processes 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 computation, communication, or I/O step of a process. *IS management* involves scheduling of system resources to perform data collection and forwarding activities. Figure 3 depicts the *ROCC* model with two types of resources of interest for the Paradyn *IS*, CPU and network, being shared by three types of processes: application, *IS*, and other user processes.

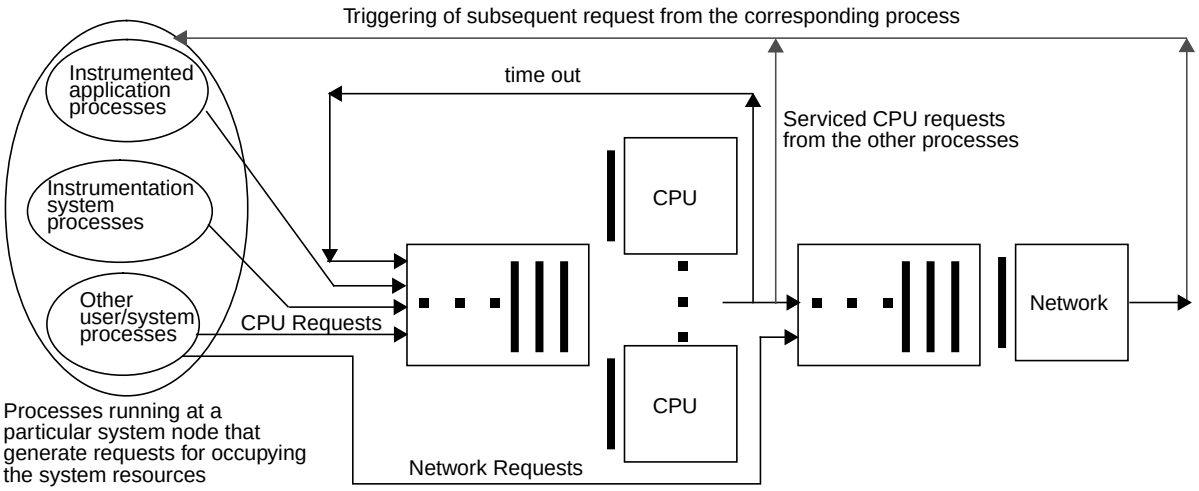


Figure 3. The resource occupancy model for the Paradyn *IS*.

Due to the interactions between different types of processes, 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 [5,13]. We present a brief discussion of the workload characterization for this study in the following subsections; a complete description is presented in a technical report [30].

2.3 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 2.3.1); and (2) to fit appropriate theoretical probability distributions to the lengths of resource occupancy requests from each of these processes (see section 2.3.2). The resulting workload model is both practical and realistic.

2.3.1 Process Model

We consider the states of an instrumented process running on a node, as illustrated by Figure 4, which is an extension of the Unix process behavior model. After the process has been 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.

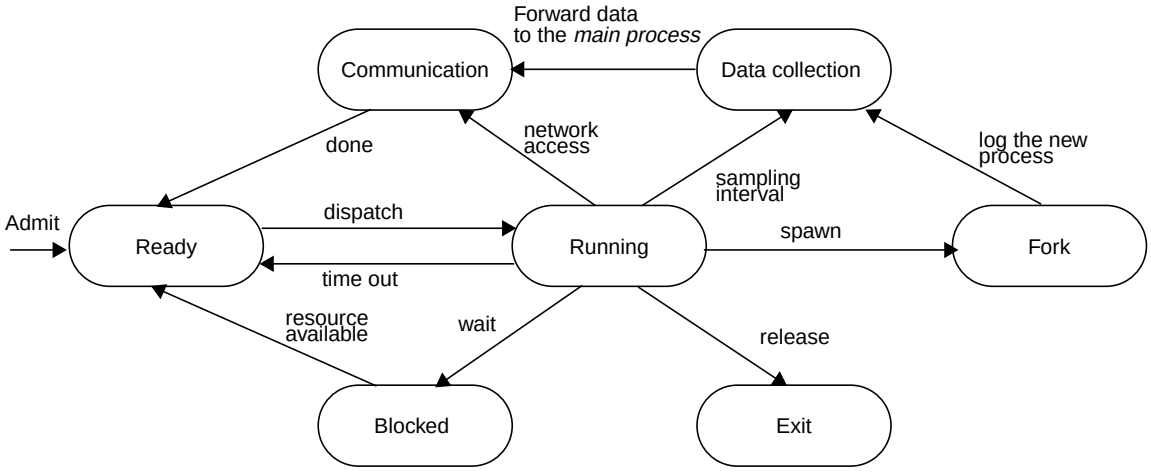


Figure 4. Detailed process behavior model in an environment using an instrumentation system.

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, considers only two states of process activity: *Computation* and *Communication*. This simplification facilitates obtaining measurements without any special operating system

instrumentation. The *Computation* and *Communication* states require the use of the CPU and network resources, respectively. The model provides sufficient information to characterize the workload when applied in conjunction with the resource occupancy model. The *Computation* state is associated with the *Running* state of the detailed model of Figure 4. Similarly, the *Communication* state is associated with Figure 4'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.

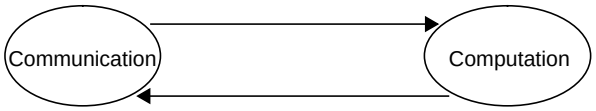


Figure 5. Alternating computation and communication states of a process for *ROCC* model.

2.3.2 Distribution of Resource Occupancy Requests

Trace data generated by the SP-2's AIX operating system tracing facility is the basis for the workload characterization. We used the trace data obtained by executing the *NAS* benchmark *pvmbt* on the SP-2 system [27]. Table 1 presents a summary of the statistics for CPU and network occupancy by various processes.

Table 1. 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 by the processes [16]. Figure 6, on the left, shows the histograms and *probability density functions* (*pdf*s) for the lengths of CPU

and network occupancy requests by the application (NAS benchmark) process (in (a) and (b), respectively).

Quantile-quantile (Q-Q) plots are often used to visually depict differences between observed and theoretical *pdf*s (see [16]). For CPU requests (Figure 6a), 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 6b), 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 (see [30]).

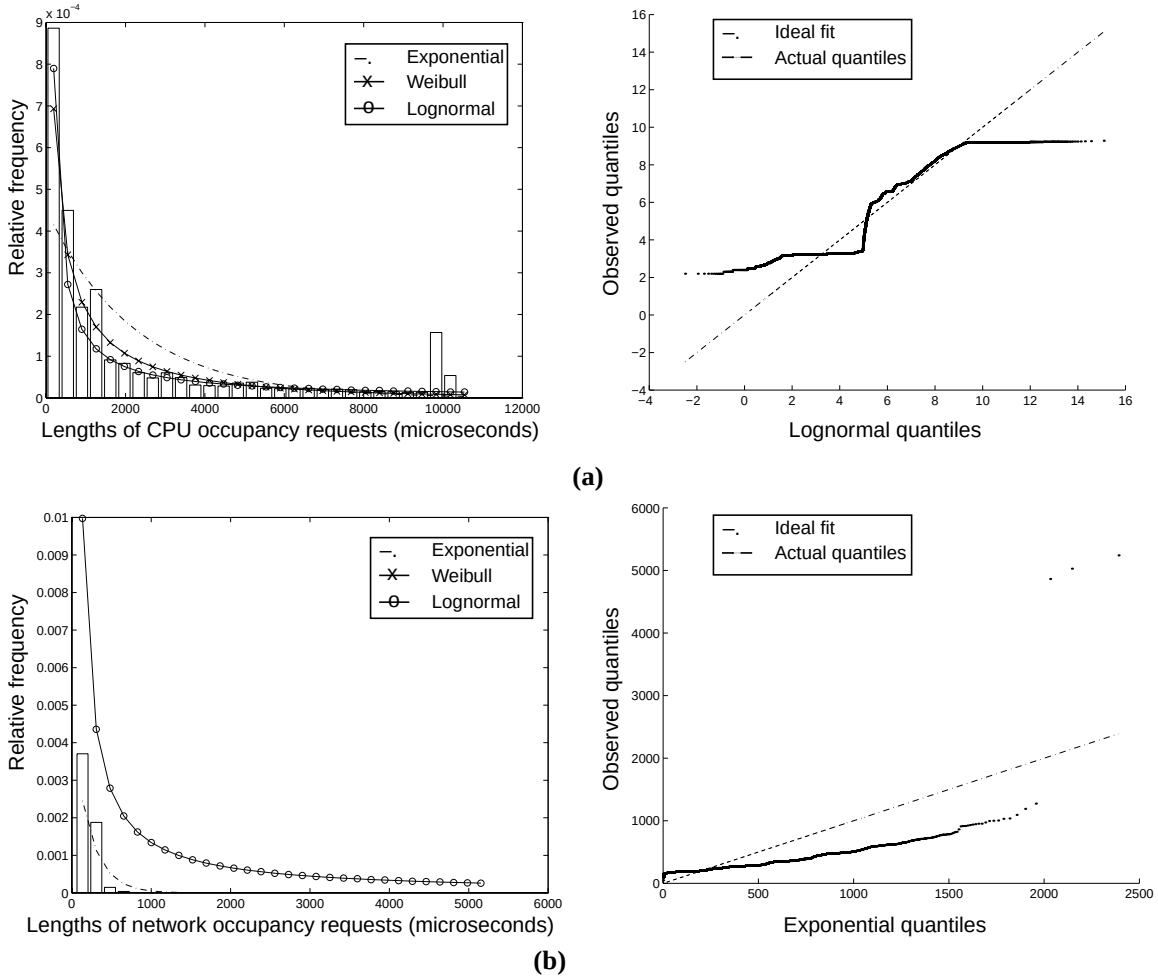


Figure 6. Histograms and theoretical *pdf*s of the lengths of (a) CPU and (b) network occupancy requests from the application process. Q-Q plots represent the closest theoretical distributions.

2.4 Model Parameterization

The workload characterization presented in the preceding section yields parameters for the *ROCC* model for the Paradyn *IS*, as shown in Table 2. 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 [16]. With these parameters, the model can be simulated and used to answer "what if" questions, which we consider next in Section 3.

Table 2. 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
	Number of *Pd* processes per node	1–4
	Number of CPUs per node	1
	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 47,344)
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)

3 Simulation-Based Experiments

In this section, we describe the simulation of the model and use it to answer several questions about configuration and scheduling policies for the Paradyn *IS*:

- What is the effect of varying the number of application processes on each node?
- What is the effect of varying the length of the sampling period?
- Does an increase in the number of Paradyn daemons on a node increase throughput, especially given a large number of application processes?

- What is the effect of a scheduling policy given multiple application processes and varying sampling periods?

3.1 Experimental Setup

In answering these questions, our simulation experiments are designed to analyze the effects of four parameters (factors):

- *sampling period*: length of time between two successive collections of performance data samples from an instrumented application process;
- *number of local application processes*: number of processes of the application running on one node of the parallel/distributed system;
- *rate of instrumentation insertion*: frequency of changing the instrumentation level in a process. The instrumentation increases or decreases when a Paradyn daemon (dynamically) inserts instrumentation code into the application process to collect additional or different data. The main Paradyn process requests the daemon to insert instrumentation in the binary image of the application process based on performance metrics being used. The amount of time between main process requests is the length of an *instrumentation interval or period* and is the reciprocal of this rate; and
- *scheduling policy* to forward instrumentation data samples to the main Paradyn process: the data forwarding policy determines the manner in which a Paradyn daemon sends data samples to the main Paradyn process.

Initial "back-of-the-envelope" analytical calculations of the *ROCC* model for the Paradyn *IS* indicated that the CPU is the bottleneck resource [29]. Therefore, we do not investigate the network occupancy time in this study. Our specific interest is in the analysis of the CPU time taken by the Paradyn daemon (i.e., its direct overhead) and the throughput of data forwarding (i.e., number of data samples forwarded per unit of time). We use a $2^k r$ *factorial design* technique for these experiments, where k is the number of factors of interest and r is the number of repetitions of each experiment [14]. For these experiments, k=4 factors and r=50 repetitions, and the mean values of the two metrics (direct overhead and throughput) are derived within 90% confidence intervals from a sample of fifty values.

Applying the $2^k r$ factorial design technique, we conduct sixteen simulation experiments, obtaining the results shown in Table 3. For this analysis, each factor can assume one of two possible values. For factors having numerical values, we use their maximum and minimum. The length of an instrumentation period (reciprocal of rate of instrumentation insertion) represents either no change or periodic change; a value of 100 seconds indicates that the instrumentation does not change during the execution (as simulation experiments have a time limit of 100 seconds), and a value of

10 seconds indicates the Paradyn daemon changes the instrumentation level of the local application processes (i.e., increases or decreases) after every 10 seconds.

Table 3. Results of simulation experiments.

Parameters			CF Policy		BF Policy	
Sampling period (msec)	Number of application processes	Length of instr. period (sec)	Pd CPU time (sec)	Pd throughput (samples/sec)	Pd CPU time (sec)	Pd throughput (samples/sec)
5	1	100	4.5	164.5	2.2	163.2
50	1	100	0.5	18.9	0.2	19.2
5	32	100	13.2	5.8	7.5	473.6
50	32	100	12.5	6.2	6.2	412.8
5	1	10	6.2	153.2	2.0	153.6
50	1	10	0.5	17.7	0.2	16
5	32	10	18.2	4.0	12.2	361.6
50	32	10	17.9	6.2	11.5	374.4

CF—Collect-and-Forward

BF—Batch-and-Forward

3.2 Principal Component Analysis

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). 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 interdependencies. We use the technique outlined by Jain to perform PCA on the results from Table 3 [14].

Figure 7 shows the results of the principal component analysis. Clearly, the number of application processes (labeled as B) is the single most important factor that affects the direct overhead of the Paradyn daemon, followed by the data forwarding policy (D) and length of an instrumentation period (C). The forwarding policy (D) and combination of the number of application processes and the forwarding policy (BD) are the most important factors affecting throughput, followed by the number of application processes (B), and the length of the sampling period (A). Thus, a further investigation of the behavior of the *IS* with respect to the number of application processes (B), the data forwarding policy (D), and their interaction (BD) is justified.

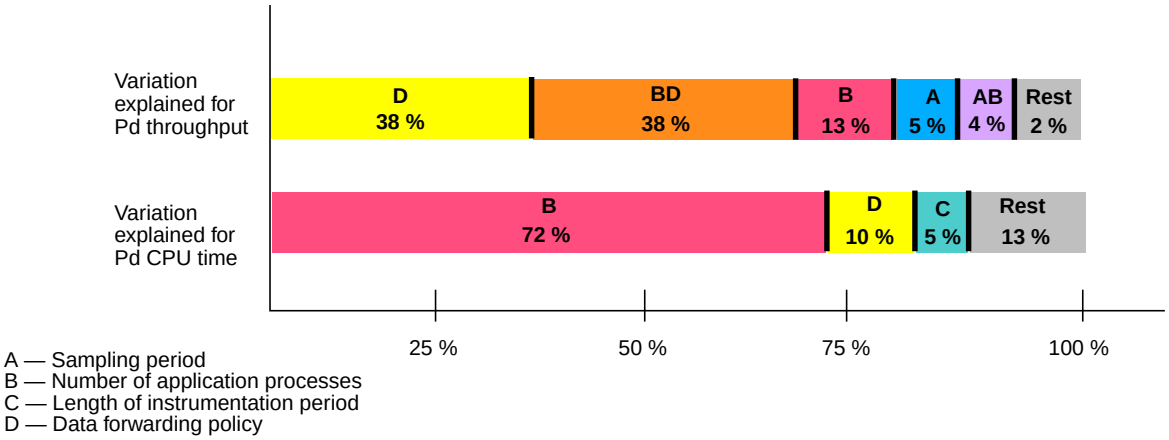


Figure 7. Results of principal component analysis of four factors and their combinations.

3.3 Evaluation of "what-if" Questions

The questions raised in the introduction to Section 3, repeated below, were formulated based on the principal component analysis presented in the preceding section.

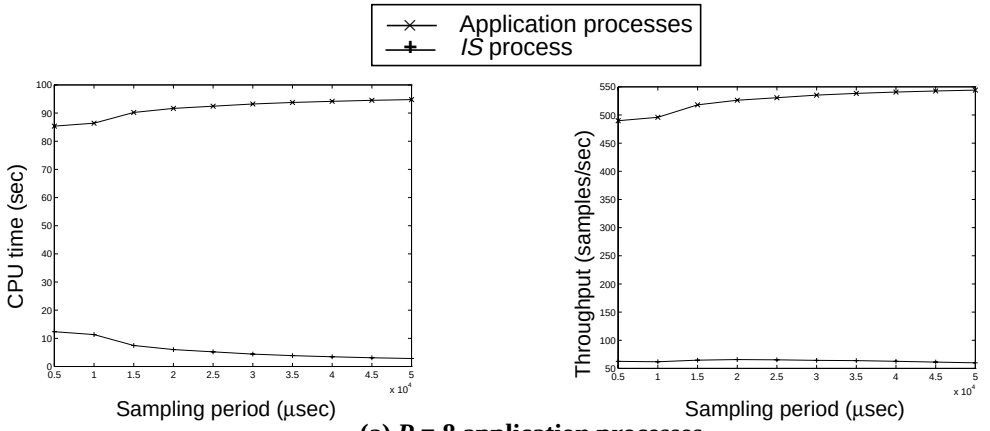
- What is the effect of varying the number of application processes on each node?
- What is the effect of varying the length of the sampling period?
- Does an increase in the number of Paradyn daemons on a node increase throughput, especially given a large number of application processes?
- What is the effect of a scheduling policy given multiple application processes and varying sampling periods?

The following subsections investigate these questions using the simulation configuration presented in Section 3.1. Via simulation, we vary the factors of interest and observe the effects on the overhead and throughput metrics.

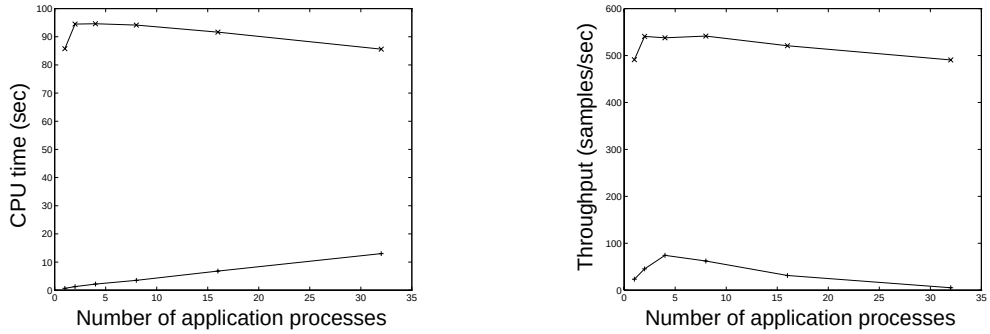
3.3.1 What are the effects of varying the number of application processes and the sampling period?

We run simulations in which we vary the sampling period and the number of application processes. Results are shown in Figure 8, where the performance is plotted for both the Paradyn daemon (instrumentation, *IS*, process) and the application processes: in (a), sampling period varies for a fixed number of application processes; in (b), number of application processes varies for a fixed sampling period; the graphs on the left show CPU time taken by the processes; and the graphs on the right, throughput.

Consider (a). The CPU time taken by the Paradyn daemon (*IS* process) decreases noticeably as the sampling period increases (i.e., as rate of sampling decreases) and then levels off. For smaller sampling periods, the Paradyn daemon makes a large number of CPU occupancy requests, resulting in higher overhead (i.e., more CPU time). On the other hand, the throughput of data forwarded by the *IS* remains relatively unaffected with increasing sampling periods. Next, consider (b). The CPU time taken by the Paradyn daemon increases linearly with the number of processes. The throughput increases up to a small number of processes and then decreases steadily. The initial increase in throughput is attributable to an increase in *IS* requests per unit time due to the additional application processes. However, with the growing number of application processes, contention for shared resources arises among *IS* and application requests, resulting in a lower throughput.



(a) *P* = 8 application processes



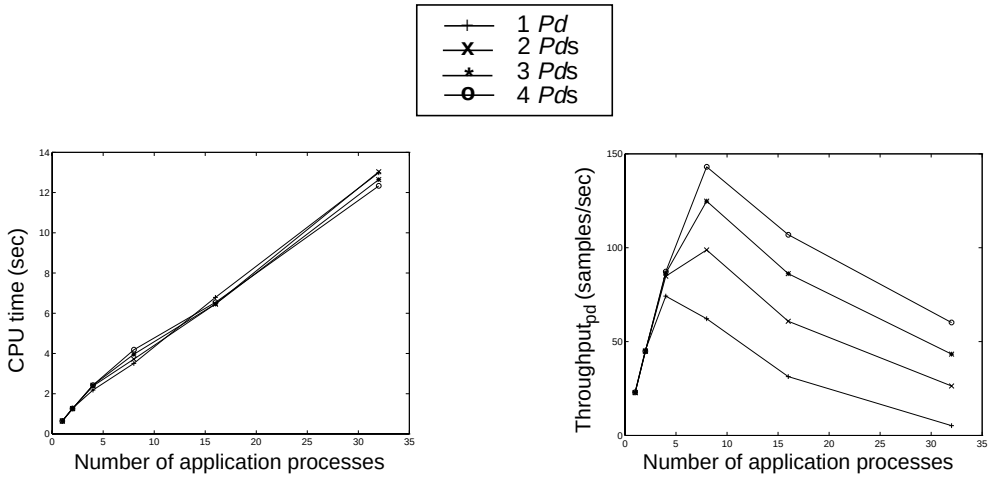
(b) Sampling period = 40 msec

Figure 8. CPU time and throughput metrics calculated with the *ROCC* simulation model.

Thus, there are interesting metric variations, especially with respect to number of application processes. Before arriving at any conclusions, we take a closer look at the Paradyn daemon performance in the next section, including varying the number of daemons.

3.3.2 Does an increase in the number of Paradyn daemons on a node increase throughput?

In Figure 8(b), we observed a decrease in the data forwarding throughput by the Paradyn daemon. One possible means to enhance the throughput, especially with many application processes, is the use of multiple daemon processes per node. Figure 9 is close-up of Figure 8(b), focusing on *IS* process performance for multiple Paradyn daemons on the same node. We use the same two metrics and vary the number of application processes while keeping the sampling period fixed. In the graph on the left, we see little variation in the CPU time taken by Paradyn daemons for different numbers of daemons. Therefore, it appears feasible to use multiple daemons without incurring excessive overhead. Moreover, the graph on the right shows that multiple daemons are useful in maintaining higher throughputs, especially when the number of application processes is above a "threshold" value (i.e., the knee in a curve). This is due to a lower overall blocking probability for the Paradyn daemon requests for CPU resources.



Sampling period = 40 msec

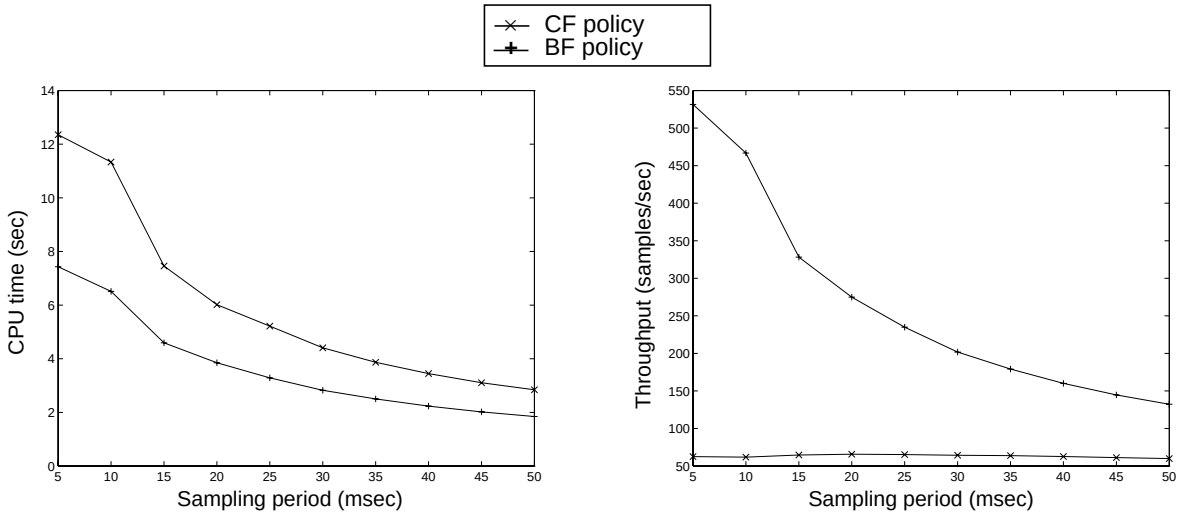
Figure 9. CPU time and throughput metrics calculated with the *ROCC* simulation model using multiple *Pd*s.

3.3.3 What is the effect of different scheduling policies for data forwarding?

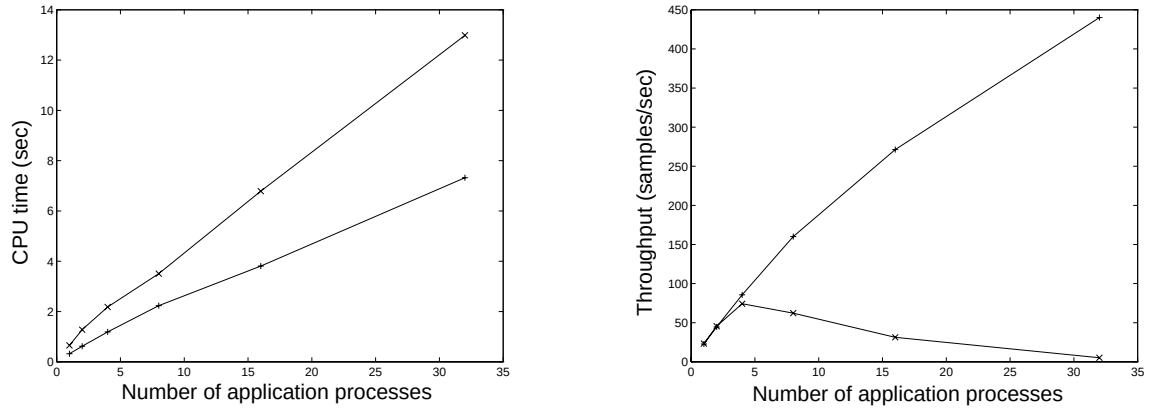
For the preceding simulation results, the *CF* policy was used to schedule the forwarding of samples from the Paradyn daemon to the main Paradyn process; *CF* is the initial policy implemented by Paradyn developers (pre-release). In this section, we compare the performance of the *CF* and *BF* policies and show that *BF* is a better choice; *BF* is implemented in the current release (1.0) of Paradyn. Figure 10 (similar in format to Figure 8) focuses on *IS* process

performance under the two policies; the *IS* process curve in Figure 8 is the *CF* policy curve in Figure 10, and it is plotted with the *BF* policy curve.

The CPU time taken by the Paradyn daemon (i.e., direct overhead) is significantly smaller using the *BF* policy, particularly with short sampling periods (see (a), left) or large numbers of application processes (see (b), left). In the *CF* policy, a system call is necessary to forward each data sample, whereas in the *BF* policy, a number of samples are forwarded per system call. Thus, system call overhead is incurred more frequently under the CF policy, and the magnitude of this overhead is depicted in Figure 10. The impact of the policy is more profound with respect to the data forwarding throughput, as shown in the graphs on the right. In the CF policy, noted earlier, there is considerable CPU contention between the Paradyn daemon and the application processes. Under the BF policy, however, the CPU time is utilized more efficiently.



(a) *P* = 8 application processes



(b) Sampling period = 40 msec

Figure 10. CPU time and throughput metrics calculated with the *ROCC* simulation model using *CF* and *BF* (with an arbitrarily selected batch size = 32 samples) policies.

3.4 Summary of Results and Initial Feedback to the Developers

We can draw several conclusions from the "what-if" simulation-based analysis. The intent was to provide Paradyn *IS* developers with useful high-level feedback for improving the *IS*. A summary of the results follows:

1. The number of application processes and the instrumentation data forwarding policy are the most important factors under consideration that affect the *IS* performance. While the choice of forwarding policy can be controlled by the developers, the number of application processes can not be.
2. Data forwarding throughput can be improved by using multiple Paradyn daemons, especially beyond a threshold in the number of application processes (e.g., eight or more processes, in this case). CPU overhead due to multiple daemons is not significantly higher than the overhead due to a single daemon. We continue to investigate the optimal number of daemon processes for a given number of application processes. However, since the number of application processes on a node of a high performance parallel/distributed system is usually very small (typically, one), there often is no need to run multiple daemons in practice.
3. The *BF* policy outperforms the *CF* policy both in terms of direct CPU overhead and data forwarding throughput.

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

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. In this study, we test the system under two sampling periods and two scheduling policies.

4.1 Experimental Setup

Figure 11 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.

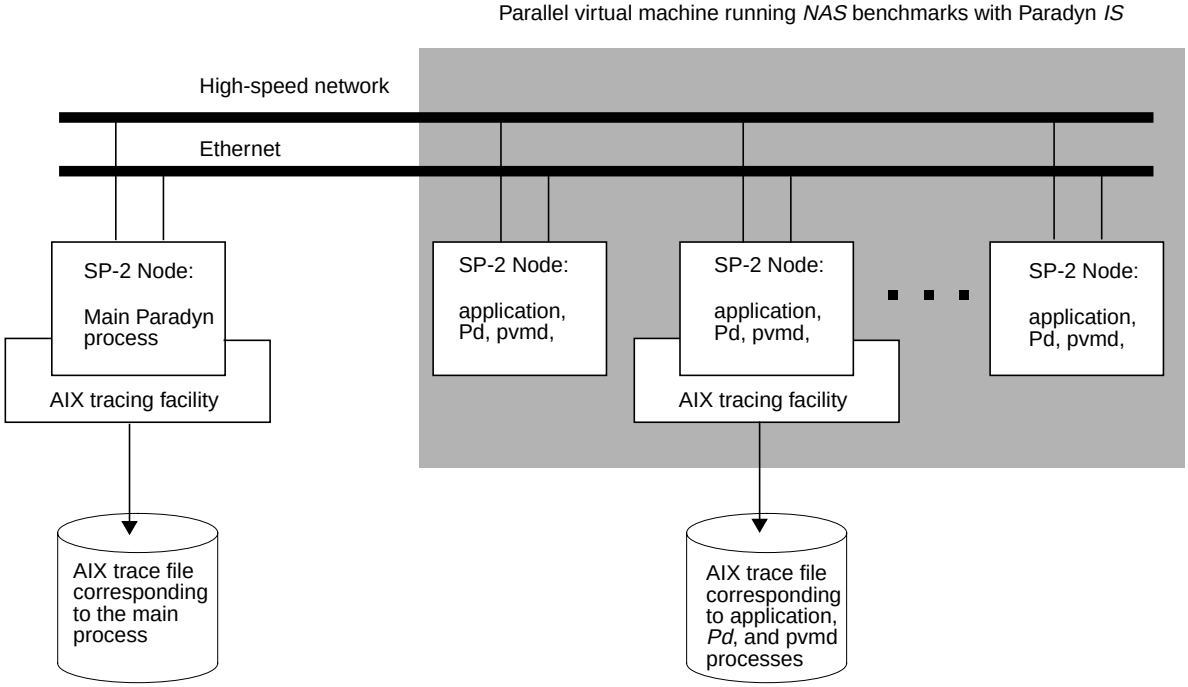


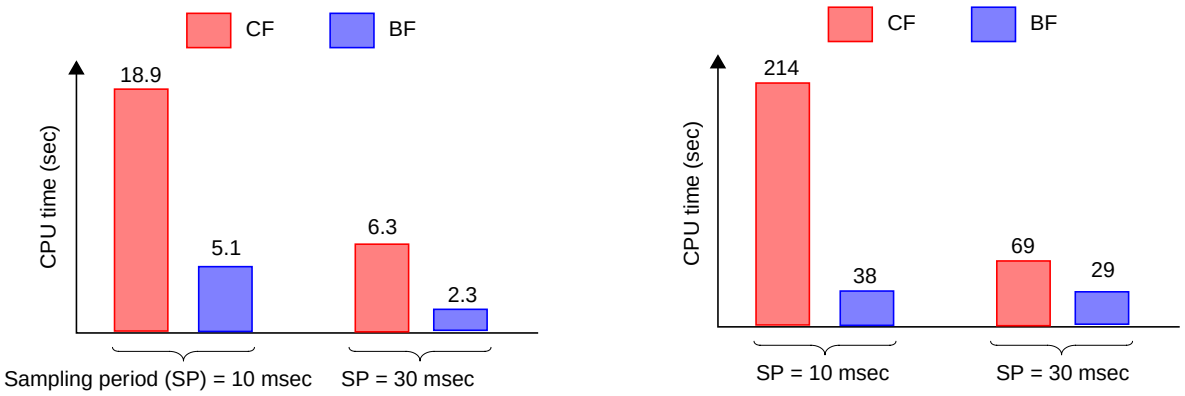
Figure 11. Measurement-based experiment setup for Paradyn *IS* on an SP-2.

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 scheduling policy options are *CF* and *BF.* The sampling period is assigned a relatively low value (10 msec) or a higher value (30 msec). Certain experiments were found interesting in the simulation-based study, such as the use of multiple application and Paradyn daemon processes per node. However, the scope of this paper is being limited to CPU-intensive, single-program multiple-data (SPMD) types of applications, for which multiple application processes per node are less likely. Additionally, the volume of the AIX trace data accumulated during each experiment can become unmanageable with multiple application processes per node. Therefore, testing experiments are conducted with only one application process per node; and, hence, there is no need for multiple Paradyn daemon processes per node. Experiments using multiple processes are left to future work with other types of

applications (e.g., real-time control). Consistent with the simulation, network occupancy is not considered; this also reduces the disk space needed for AIX traces.

4.2 Evaluation

Figure 12 summarizes the Paradyn IS testing results related to the CPU overhead of the Paradyn daemon (graph on the left) and the main Paradyn process (graph on the right). The CPU time taken by 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.



(a) Paradyn daemon process **(b) Main Paradyn process**

Figure 12. Comparison of CPU overhead measurements under the CF and BF policy using two sampling period values for (a) Paradyn daemon and (b) main Paradyn process.

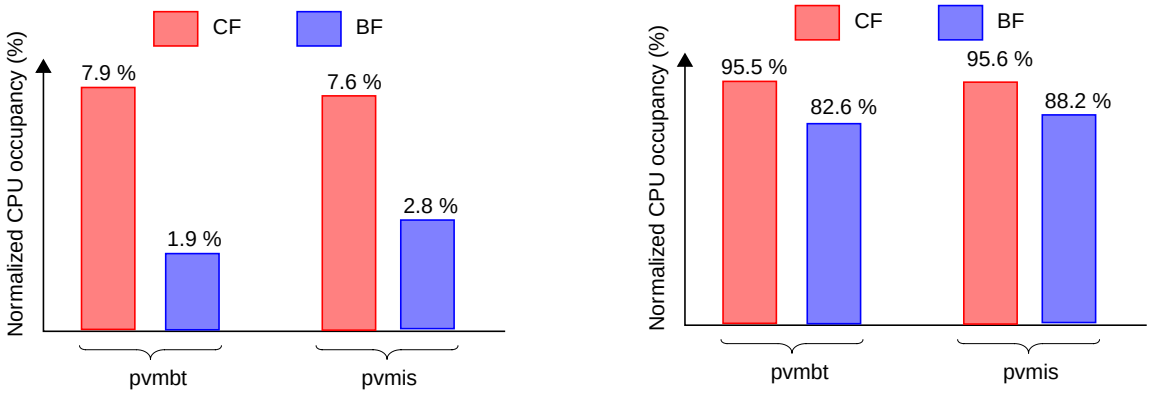
In order to determine the relative contribution of these two factors to the direct CPU overhead, we use principal component analysis. The results of this analysis for the Paradyn daemon and main Paradyn processes are shown in Table 4. Clearly, the scheduling policy to forward data is primarily responsible 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.

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 scheduling 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.

Table 4. Results of principal component analysis of scheduling policy vs. sampling period for the tests in Figure 12.

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

The systems are *block tridiagonal* with 5×5 blocks. Benchmark *pvmis* is an integer sort kernel. All experiments use a sampling period of 10 milliseconds. In order to compare the overheads due to different application programs having different CPU time requirements, we normalize the CPU time for each process with the total CPU time requirement at a node during the execution of the benchmark program. The results are summarized in Figure 13. The key observation is that the reduction in *IS* overheads under the *BF* policy is not significantly affected by the choice of application program.



(a) Paradyn daemon process **(b) Main Paradyn process**

Figure 13. 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 5. 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 5. Results of principal component analysis of scheduling policy vs. application program for the tests in Figure 13.

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

5 Discussion

In this paper, we presented a case study of applying a structured modeling, evaluation, and testing approach to the Paradyn instrumentation system. We investigated various performance questions using a model of the Paradyn *IS* and provided feedback to the developers. Specifically, a simulation-based study indicated the potential advantage of a proposed *batch-and-forward* policy over the *collect-and-forward* policy. The effects of implementing this policy were tested by using measurement-based experiments. Testing results indicate that use of the BF policy reduces the CPU overhead of the Paradyn daemon and main Paradyn process by about 60%. Perhaps more significantly, this study has shown the successful role of modeling and simulation to design more efficient instrumentation systems through appropriate feedback at an early development stage.

The purpose of the initial feedback provided by the 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 [29]. The case study presented in this paper uses a more detailed workload characterization based on measurement data. Although we enhanced the scope of the "what-if" questions in this study, e.g., to include factors such as scheduling policy and length of instrumentation period, this more detailed study does not contradict the earlier study that uses an approximate model [29]. 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 tools may undergo frequent modifications for new platforms and applications. The *HPCC* community has recognized the high cost of software tool development [22]. 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 [21]. Off-the-shelf *IS* components will need to meet a number of functional as well as non-functional requirements. The modeling, evaluation, and testing presented in this paper represent necessary steps to realize high-performance, well-specified off-the-shelf *IS* components.

6 Related Work

We conclude this paper by placing the work in perspective with related work. This paper focused on the Paradyn tool. However, a number of tools exist that provide a range of functionality and rely on instrumentation system services. Table 6 is a representative listing of tools, their functionality, and IS services. Thus, the potential impact of sound *IS* design practices and well-understood implementation alternatives is considerable.

More specific to IS modeling, while we have emphasized its use to tool developers, users can also take advantage of it. With an appropriate model for the *IS*, users can specify tolerable limits for *IS* overheads relative to the needs of their applications. The *IS* can use the model to adapt its behavior in order to regulate overheads. Some initial work has already been done in this direction for Paradyn [12]. Previous work related to *IS* modeling and overhead analysis has focused on analyzing the intrusion due to instrumenting parallel programs [17,32].

Several other researchers have given special attention to the monitoring overheads of their tools. Miller et al. present measurements of overheads of the IPS-2 tool and compare them with the

Table 6. *IS* services used by tools to support a range of functions.

Functionality	Representative Tools	Description of Key *IS* Services
Performance Evaluation	ParAide	ParAide is the integrated tool environment for Intel Paragon. Commands are sent to the distributed monitoring system, called Tools Application Monitor (TAM). TAM consists of a network of TAM processes arranged as a broadcast spanning tree with one TAM process (part of the *IS*) at each node. **Reference**: http://www.ssd.intel.com/paragon.html and [26]
Debugging	VIZIR	This debugger consists of an integrated set of commercial sequential debuggers. Its *IS* synchronizes and controls the activities of individual debuggers that run the concurrent processes. The *IS* also collects data from these processes to run multiple visualizations. **Reference**: [9]
Performance Modeling and Prediction	AIMS, Lost cycles analysis toolkit	These tools integrate monitoring and statistical modeling techniques. Measurements are used to parameterize the model, which is subsequently used for predicting performance. The *IS* performs the basic data collection tasks. **Reference**: http://www.nas.nasa.gov/NAS/Tools/Projects/AIMS/ and [4,33]
Performance and Program Visualization	ParaGraph and POLKA	The *IS* collects runtime data in the form of time-ordered *trace records*. These trace records are used to drive hard-coded (ParaGraph) or user-defined (POLKA) visualizations of system and program behavior. **References**: http://www.netlib.org/picl/ and http://www.cc.gatech.edu/gvu/softviz/parviz/polka.html
Correctness Checking	SPI	Scalable Parallel Instrumentation (SPI) is Honeywell's real-time *IS* for testing and correctness checking on heterogeneous computing systems. SPI supports a 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/ and [3]
Adaptive Real-Time Steering	DIRECT/JEWEL	Runtime information collected by the off-the-shelf instrumentation system JEWEL is fed to a dynamic scheduler. The scheduler uses this information to adaptively control the real-time system to be responsive to the variation of important system variables. **Reference**: http://borneo.gmd.de:80/RS/Papers/direct/direct.html and [7,15]
Dynamic Resource Scheduling	RMON	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
Visualizing Corporate Data	AT&T visualization systems	Visualization tools use monitored data to locate long-distance calling frauds through unusual calling patterns, to find communities of interest in local calling, to retain customers, and to compare databases consisting of textual information. **Reference**: http://www.att.com/att-tj/ and [6]

overheads of a functionally similar tool, *gprof* [19]. Gu et al. use synthetic workloads to exercise specific features of the *IS* of the Falcon steering tool and measure the *IS* performance [8].

This study of Paradyn's *IS* follows previous work by Waheed and Rover to view the *IS* as enabling technology, or *middleware* [2], and to establish an approach for characterizing, evaluating, and understanding *IS* operation, including its overheads [28]. This approach emphasizes a separation of the high-level tool requirement and usability issues from the low-level design and test issues. We applied this two-level approach for modeling and evaluating the Paradyn *IS.*

Acknowledgments

We would like to acknowledge the contribution of Bart Miller of the University of Wisconsin, who helped initiate this collaborative work on Paradyn *IS* modeling, evaluation, and testing. We also thank Tia Newhall for implementing the *batch-and-forward* policy in Paradyn.

References

[1] Belanger, David G., 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.

[2] Bernstein, Philip A. "Middleware: A Model for Distributed System Services," *Communications of the ACM*, 39(2), Feb. 1996.

[3] Bhatt, Devesh, 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.

[4] Crovella, Mark E. and Thomas J. LeBlanc, "Parallel Performance Prediction Using Lost Cycles Analysis," *Proceedings of Supercomputing '94*, Washington, DC, Nov. 14–18, 1994.

[5] Dimpsey, Robert T. 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.

[6] Eick, Stephen G. and Daniel E. Fyock, "Visualizing Corporate Data," *AT&T Technical Journal*, January/February 1996, pp. 74–85.

[7] Gergeleit, Martin, 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.

[8] Gu, Weiming, Greg Eisenhauer, Eileen Kramer, 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.

[9] Hao, Ming C., 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.

[10] Harrison, R., 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.

[11] Hollingsworth, J. K., B. P. Miller, and Jon Cargille, "Dynamic Program Instrumentation for Scalable Performance Tools," *Proc. of Scalable High-Performance Computing Conference*, Knoxville, Tenn., 1994.

[12] Hollingsworth, J. K. and B. P. Miller, "An Adaptive Cost Model for Parallel Program Instrumentation," *Proceedings of EuroPar '96*, Lyon, France, August 1996.

[13] Hughes, Herman D., "Generating a Drive Workload from Clustered Data," *Computer Performance*, 5(1), March 1984.

[14] Jain, Raj, *The Art of Computer Systems Performance Analysis—Techniques for Experimental Design, Measurement, Simulation, and Modeling*, John Wiley & Sons, Inc., 1991.

[15] Lange, F., 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.

[16] Law, Averill M. and W. D. Kelton, *Simulation Modeling and Analysis*, McGraw-Hill, Inc., 1991.

[17] Malony, A. D., 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.

[18] Mercer, Clifford W. 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.

[19] Miller, B. P. et al., "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.

[20] Miller, Barton P., 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.

[21] *OMIS—On-Line Monitoring Interface Specifications*. Accessible from http://www-bode.informatik.tu-muenchen.de/~omis.

[22] Pancake, Cherri M. "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.

[23] Reed, Daniel A., 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.

[24] Reed, Daniel A., "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.

[25] Reed, Daniel A., Keith A. Shields, Will H. Scullin, Luis F. Tavera, and Christopher L. Elford, “Virtual Reality and Parallel Systems Performance Analysis,” *IEEE Computer*, 28(11), November 1995.

[26] Ries, Bernhard, 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.

[27] Saini, Subhash 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.

[28] Waheed, A. and Diane T. Rover, “A Structured Approach to Instrumentation System Development and Evaluation,” *Proceedings of Supercomputing ‘95*, San Diego, California, Dec. 3–8, 1995.

[29] Waheed, A., 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.

[30] Waheed, Abdul, Diane T. Rover, and Jeff Hollingsworth, “Modeling and Evaluation of Paradyn Instrumentation System,” Technical Report, April 1996. Available on-line from http://web.egr.msu.edu/VISTA/Paradyn/paradyn.html.

[31] *Workshop on Debugging and Performance Tuning of Parallel Computing Systems*, Chatham, Mass., Oct. 3-5, 1994.

[32] Yan, Jerry C. 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.

[33] Yan, Jerry C., 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.