ABSTRACT

An abstract of the thesis of Rashawn Lee Knapp for the Master of Science in Computer Science presented June 15, 2006.

Title: Towards Environment-Aware Performance Analysis: Improving Parallel Performance Diagnosis by Including Knowledge of the Runtime Environment

Tools used for conducting parallel performance analysis do not adequately convey an understanding of an application’s performance in relation to the status and behavior of the runtime environment in which an application executes. This limits the quality of diagnosis that tools are able to offer regarding the causes of performance problems.

This work presents Environment-Aware Performance Analysis, a new approach that targets improving the quality of diagnosis offered by performance analysis tools. This thesis focuses on Environment-Aware Performance Analysis for parallel applications.

Environment-Aware Performance Analysis includes analysis of an application’s execution behavior and analysis of the environment in which an application executes.

The performance diagnosis incorporates both sets of analyses, and it seeks to identify root causes of performance behavior. Environment-Aware Performance Analysis extends traditional methods of application performance analysis, and, by doing so, has potential to improve the quality of performance diagnosis. In this work, we investigate including analysis about the status and behavior of the runtime environment.

A primary goal is to demonstrate a strong need for performance analysis tools to utilize information about the runtime environment. Through simulated experiments and by citing published and anecdotal scenarios, we describe situations in which traditional performance analysis tools either provide misleading diagnostic information or are unable to provide adequate guidance to analysts in determining the causes of performance problems. With proof of concept demonstrations, we show the potential for performance analysis tools to improve the quality of diagnosis by incorporating analysis of the runtime environment with analysis of application behavior. We present a layered view of the runtime environment and describe how this view can be used to frame performance problems in terms of root causes and potential optimizations. We discuss considerations relate development and design of environment-aware tools. In our presentation of related work, we provide a historical perspective and describe the state of the art for parallel performance analysis tools.

TOWARDS ENVIRONMENT-AWARE PERFORMANCE ANALYSIS: IMPROVING PARALLEL PERFORMANCE DIAGNOSIS BY INCLUDING KNOWLEDGE OF THE RUNTIME ENVIRONMENT

by

RASHAWN LEE KNAPP

A thesis submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE
in
COMPUTER SCIENCE

Portland State University
2006

ACKNOWLEDGMENTS

I wish to thank my advisor Dr. Karen Karavanic who offered encouragement and support through every step of this process; Dr. Douglas Pase who reviewed and advised on the research from the beginning; Dr. Bryant York and Dr. Robert Bertini who offered valuable advice as committee members; and Kathryn Mohror who shared her performance analysis experiences with me.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1 INTRODUCTION

This work presents Environment-Aware Performance Analysis, a new approach that targets improving the quality of diagnosis offered by application performance analysis tools. The research presented in this thesis focuses on Environment-Aware Performance Analysis for parallel applications.

1.1 Overview

Environment-Aware Performance Analysis includes both analysis of an application's execution behavior and analysis of the environment in which an application executes. The performance diagnosis incorporates both sets of analyses, and it seeks to identify root causes of performance behavior. Environment-Aware Performance Analysis extends traditional methods of application performance analysis, and, by doing so, has potential to improve the quality of performance diagnosis. A broad definition of Environment-Aware Performance Analysis interprets environment analysis to include the runtime environment and aspects about the build and compile time environments. In this work, we investigate the runtime environment portion of the environment analysis component.

A primary goal of this thesis is to demonstrate a strong need for performance analysis tools to utilize information about the runtime environment in the phases of performances analysis. Through simulated experiments and by citing published and anecdotal scenarios, we describe situations in which traditional performance analysis tools either provide misleading diagnostic information or are unable to provide adequate guidance to analysts in determining the causes of performance problems. With proof of concept demonstrations, we show the potential for performance analysis tools to improve the quality of diagnosis by incorporating analysis of the runtime environment with analysis of application behavior. We present a layered view of the runtime environment and describe how this view can be used to frame performance problems in terms of root causes and potential optimizations. We discuss considerations related to design and development of environment-aware tools. In our presentation of related work, we describe other parallel performance analysis tools.

The contributions of this thesis are:

- A test suite for evaluating the ability of tools to accurately diagnose performance problems;
- Experimental results showing the limitations of current tools;
- A set of case studies illustrating situations that might have benefited from Environment-Aware Performance Analysis;
- A discussion of the challenges and concerns for development of environment-aware tools; and
- A survey of the state of the art in parallel performance analysis tools.

1.2 Background

In this section we introduce the key concepts of parallel computing. We describe some of the motivating factors driving the implementation of large-scale parallel computing systems. We introduce performance analysis concepts, and we discuss challenges related to conducting performance analysis in large-scale environments. We present key features in the cycle of performance tuning, and we describe how Environment-Aware Performance Analysis extends the scope of traditional performance analysis.

In parallel computing, multiple compute resources act simultaneously to solve computational problems. Compared to serial or sequential processing, in which one central processing unit (CPU) is employed, the advantage offered by parallel computing is the ability to solve problems in less time. There are many computational problems which cannot be solved on sequential machines simply because the time required is too great. However, by combining computational resources, parallel computing creates an environment in which it is possible to solve large and complex problems in reasonable amounts of time. Parallel computing is often referred to as *high end* or *high performance* computing.

High performance systems combine parallelism with advances in hardware design to achieve faster computation. Parallelism has several basic forms, each with many variations. One common form is shared memory parallelism, in which processors can access memory as a global address space and processes view the shared memory as local memory. Alternatively, in distributed memory parallel computing, memory is not shared and communication networks are required for connecting cooperating processes. Many of today’s high performance systems employ a combination of these two basic forms, in which a single node comprises several shared memory compute resources and nodes are interconnected via a communication network. This is a common configuration for cluster computing. The term *massively parallel* is used to describe high end systems containing many processors. This notion of “many’ is subjective and changes over time, as advances in the design of processors, memory, and interconnect technology allow new systems to contain more processors than the previous generation of high performance systems. In general, there is a positive correlation between the number of processors and the computational power of a system. The TOP500 project ranks the computational capacity of high end systems by ordering the top 500 systems according to performance on a benchmark measuring the number of floating point operations per second (Flop/s)[87]. With each successive publication of the list, the current generation of massively parallel systems rises to the upper ranks. In June of 2002, Earth-Simulator, a 5,120 processor high performance system installed at the Earth Simulator Center in Japan, placed first in the TOP500 ranking [86]. With sustained performance at 35.86 Tera Flop/s (TFlop/s), Earth-Simulator ranked first on the list for five consecutive publications and held the seventh position in the most recent release, in November of 2005 [87]. BlueGene/L, with 131,072 processors, earned first place. Installed at Lawrence Livermore National Laboratory, in the United States, this machine’s sustained performance is rated at 280.6 TFlop/s [87]. Size and intricacy in hardware design lend to the overall complexity of these systems. The welcome benefit of such complexity is that as more combinations and variations of achieving computational power and parallelism are developed, computationally harder problems have the potential to be solved within reasonable time frames.

A class of problems, referred to as “Grand Challenge Problems,” is a driving motivation in the implementation and design of today’s high performance systems. These problems are characterized as computational problems of significant social and economic impact which cannot be solved in reasonable amounts of time using current state of the art resources, but whose solutions would be advanced by the application of high performance computing techniques and resources [83]. To promote active research in the pursuit of solutions to Grand Challenges, the United States enacted the High Performance Computing Act of 1991[24]. This legislation launched the federally supported High Performance Computing and Communications (HPCC) programs, providing significant support and attention to research and development in many areas of high performance computing. One of the identified goals of the act was to enable the research and development of software and hardware necessary to address Grand Challenges. Many of the Grand Challenges originating from the legislation required simulation, and as a result of the successes of these initiatives, simulations of important physical phenomena are now possible. In many cases where true experimentation is severely limited or impossible, simulation has provided the first real understanding of certain phenomena. The successes of these research endeavors are outlined in a 1996 HPCC report [36], and include aeropropulsion system simulation, combustion modeling, atmospheric modeling, ocean modeling, modeling of air pollution dynamics, global climate modeling, earthquake ground motion modeling, analysis of protein and nucleic acid sequences, and protein folding prediction. An update to the 1991 Grand Challenges was recently published by the National Coordination Office for Networking and Information Technology Research and Development (NITRD) [20].

The critical nature of these Grand Challenge problems requires that the applications and systems involved in these computations produce results that are both timely and accurate. Time constraints are imposed upon applications in several ways. Often, the data produced by a simulation is needed within some specific time frame. For example, in predicting the location along a coast line and with how much force a hurricane will hit, the prediction, in order to be useful, must be made well before the actual event occurs. Other time constraints include job scheduling parameters and system mean time to failure. There is a general desire to improve the performance of these large-scale scientific applications by decreasing execution time. Additionally, if the execution time, also known as wall-clock time, of a critical high performance application exceeds an imposed time constraint or unexpectedly takes longer, a need to gain understanding about the application’s behavior arises.

Performance analysis techniques are used to gain understanding about an application’s execution behavior. Performance analysis involves measurement, interpretation, and communication [43]. *Measurement* is both deciding what to measure and how to obtain the measurement. *Interpretation* refers to understanding the measured results and discerning their significance and validity. *Communication* is the process of presenting the results. Even for a simple application, it would be extremely difficult to conduct a performance analysis study using only manual methods. At most, one might be able to gather wall-clock timing information; and this would be at the granularity of the stop watch used, a granularity much courser than that of the computing system. Understanding application behavior, even for the simplest of applications, requires a degree of automated performance analysis support.

Performance analysis support can be provided by hardware, by the operating system, by library facilities, and by high level tools. Each mechanism offers varying levels of assistance in conducting the three phases of performance analysis. The measurement phase includes instrumentation and data collection. Instrumentation can be accomplished by compiling explicit instrumentation points into the executable code, by inserting the instrumentation into an already compiled application, known as binary rewriting, and by inserting the instrumentation dynamically, as the application executes.

The four basic approaches to data collection are timing, counting, profiling, and tracing [74, 76]. *Timing* techniques measure execution time. A common timing measurement is total execution time, also called wall-clock time. It is also common to measure the time spent in certain activities: loops, function calls, time spent waiting, etc. With a clock of adequate resolution, timing techniques provide sufficient information regarding where time is spent. Timing, however, does not encapsulate notions of when events occur, with what frequency events occur, or why events occur. *Counting* captures the number of times an event occurs. Used by itself, counting does not provide insight into the dynamic behavior of an application or a system. *Profiling,* also called sampling, is a way to approximate values that would be captured by strict counting. Profiling measurements are obtained by periodically sampling some aspect of the system state and incrementing a counter associated with the observed state. At the completion of a profiling interval, the value of a counter is proportional to the total time spent executing in the associated state. Profiling provides approximated information about the number of times events occur. The primary limitation of profiling is the potential for errors, which is inherent in sampling. For example, profiled statistics would be skewed if the number of samples obtained is quite low and the profiled application is a short-lived application. *Tracing* records the occurrences of events and saves a descriptive record about each event. Trace records typically contain an event identifier, indicating what event occurred; a timestamp, indicating when the event occurred; and data regarding the node, process, and source code line associated with the event. Tracing mechanisms provide detail about what events occur, when events occur, and where events occur. Additionally, counting and timing measurements can be computed from the trace data. Tracing has potential to be highly invasive, depending on the number of events that are traced and the frequency of event occurrences. This could lead to unacceptable perturbation in the application. Tracing mechanisms also have potential to produce large volumes of data, leading to challenges in data storage and data analysis.

Two existing approaches for facilitating data interpretation are post-mortem analysis and runtime analysis. Two techniques for carrying out the communication phase are visualization of results and automated diagnoses.

Many facilities are available for conducting performance analysis in high end computing. These include high level tools which assist with all three phases of performance analysis; library interfaces which provide access to lower level facilities like timers and hardware counters; and low-level methods for timing and counting. We describe some of the performance tools used in high performance computing in Chapter Two.

Even with performance analysis support, there are notable difficulties for conducting performance analysis in high end computing. There are challenges presented due to factors of scale, including system size, application size, and the potential to produce large volumes of analysis data. There are also difficulties related to the steps taken and tools used in the performance analysis process. Performance analysis often involves the use of several tools and many manual steps, both of which add delay to the performance analysis process. Examples include: associating an application execution with the data produced by all of the tools used for analysis; keeping track of analysis information for separate runs of an application; knowing what varies from one run to the next; and deciding, based on the tool's presentation of results, what to target for optimization. The reason that applications often need to be instrumented with more than one tool is that tools tend to be dependent upon the underlying system configuration, the types of applications that will be analyzed, and the types of parallelism involved.

The processes involved in performance analysis form a cycle of events that compose a performance tuning cycle. The cycle displayed in Figure 1 shows a simplified view of the typical steps involved; in practice it is common to have additional internal cycles. For example, given that several tools may be required to collect the appropriate measurement data, a cycle may occur between the phases of determining what the application does and determining why a performance behavior is exhibited. In tuning an application, analysts first want to obtain a description about the execution behavior of the application. This step asks “What does the application do?” To create this description, an analyst can use the performance analysis techniques described earlier to gather data which might be used to construct a timeline of application events, report the percentage of time spent in functions, or report function call counts. The information from this step is pulled forward to the diagnosis step and is used to answer the question “Why is some behavior exhibited?” Once an answer to this question is obtained, analysts can target something for optimization. The optimization could involve an application level change, a change to a library, an operating system change, or could involve changing some aspect of the hardware. After implementing the optimization, the application’s execution is re-assessed. In this step, analysts determine if the optimization improved performance.

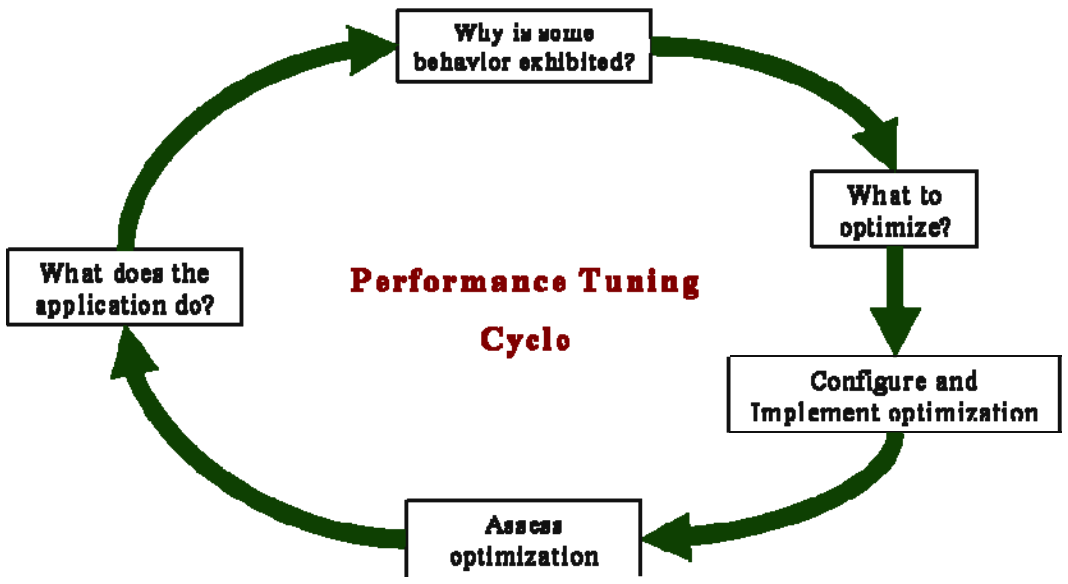

Why is some behavior exhibited?
What to optimize?
Configure and Implement optimization
Assess optimization
What does the application do?
Performance Tuning Cycle

Figure 1: Steps Involved in a Performance Tuning Cycle

As shown in Figure 2, today's performance tools cover part of the scope of performance tuning. Today's tools are capable of describing the execution behavior of an application, but because they do not include analysis about the status and behavior of the runtime environment, they are only able to report the causes of application performance behavior in terms of the application.

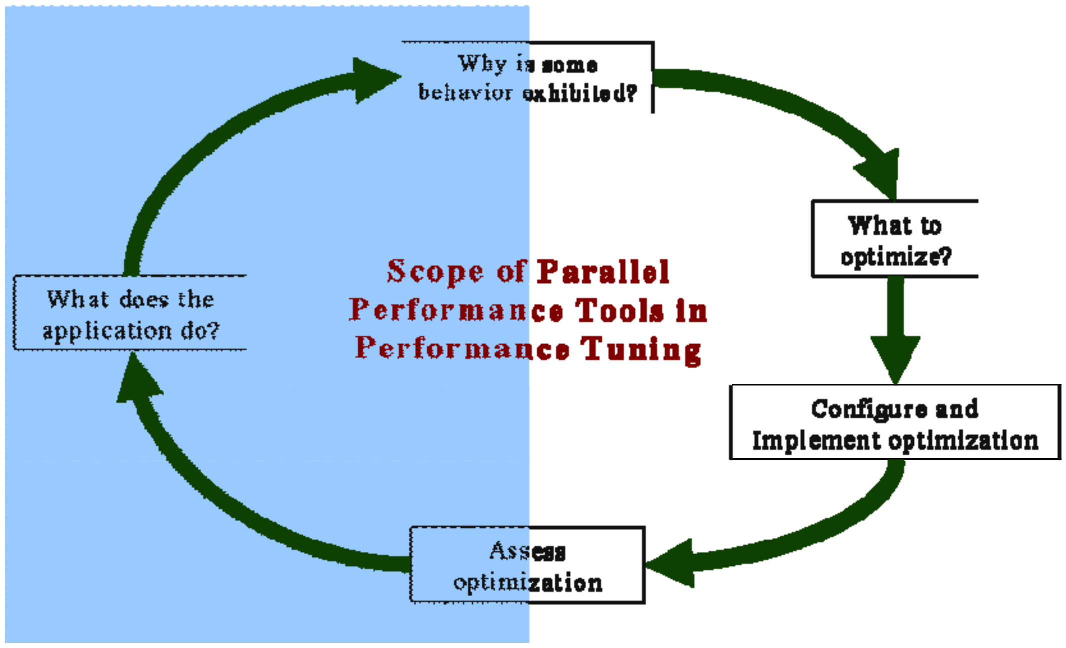

Why is some behavior exhibited?
What to optimize?
Configure and Implement optimization
Assess optimization
What does the application do?
Scope of Parallel Performance Tools in Performance Tuning

Figure 2: The Scope of Today's Parallel Performance Tools

Environment aware performance analysis improves the quality of diagnosing why a performance behavior is exhibited, and as such, extends the scope of traditional performance analysis methods. This is done by collecting information about the status and behavior of the runtime environment, and in particular asks an additional question in the first phase of data collection: "What is the status and behavior of the runtime environment?" The scope of environment-aware performance analysis tools is shown in Figure 3.

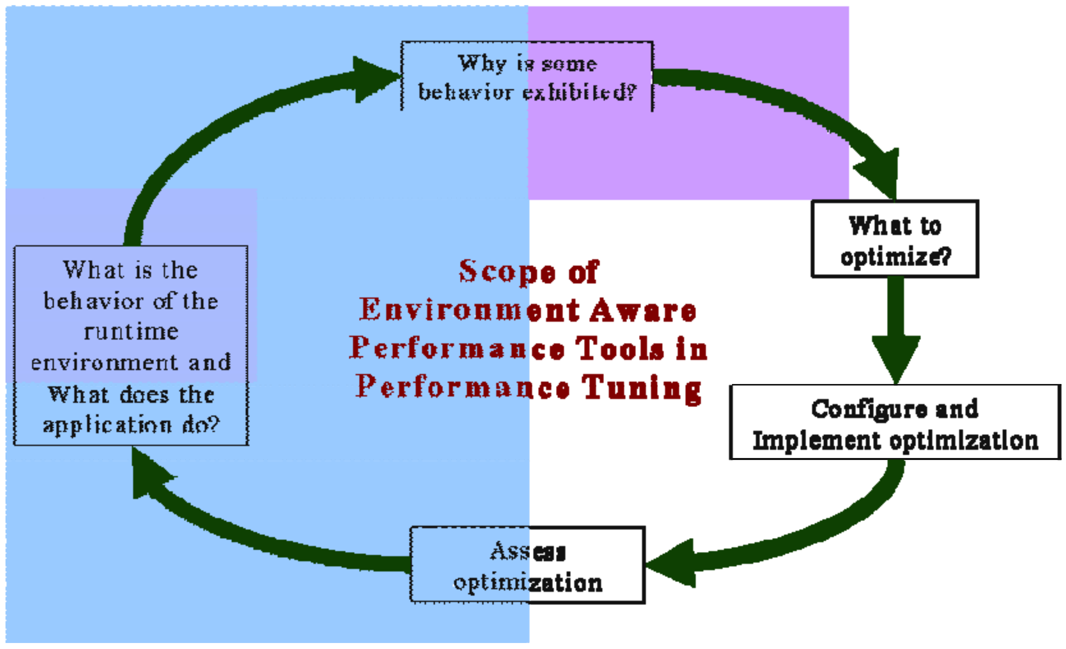

Why is some behavior exhibited?
What to optimize?
Configure and Implement optimization
Assess optimization
What is the behavior of the runtime environment and What does the application do?
Scope of Environment Aware Performance Tools in Performance Tuning

Figure 3: The Scope of Environment-Aware Parallel Performance Tools

There are many open challenges for high end performance analysis; this thesis focuses on the difficulty in accurately diagnosing the underlying cause(s) of poor performance. We propose a new approach that incorporates analysis of the runtime system into traditional application performance analysis. Including both analysis of an application's execution behavior and analysis of the runtime environment in which an application executes, allows for understanding of an application's behavior in context of its runtime environment. Furthermore, the use of both analysis sets in the construction of a performance diagnosis has potential to identify root causes of performance behavior; and by doing this, the quality of diagnosis offered by environment-aware techniques is improved. The available set of performance analysis tools used in today's high end computing environments do not convey adequate understanding of an application's actual performance in relation to the runtime environment in which an application executes.

In this chapter, we presented an introduction to Environment-Aware Performance Analysis, a new technique for enhancing the quality of diagnosis offered by application performance analysis tools. We defined this term and stated the scope of the research in this area for this thesis. We presented the goals of this research and enumerated the contributions of this thesis. Additionally, we introduced the key concepts of high performance computing and performance analysis within this environment. We described motivation for the implementation of large-scale parallel systems, and we discussed some of the performance analysis challenges that exist in these large-scale environments.

In the next chapter we present related work. In Chapter 3, we present a view of the runtime environment and discuss issues related to development of environment-aware performance analysis tools. Case studies which illustrate candidate situations for environment-aware performance analysis are presented in Chapter 4. This is followed, in Chapter 5, by a description of a proposed test suite for evaluating the ability of performance analysis tools to accurately diagnose performance problems. Our experiments and results are presented in Chapter 6. We present our conclusions and identify areas of future work in Chapter 7.

CHAPTER 2 RELATED WORK

This chapter presents research related to our goals and contributions. An environment-aware tool conducts both application level analysis and analysis of the environment in which the application executes and can correlate the analysis of the runtime environment to the analysis of the application. Related work falls into three categories: tools that measure the environment, tools that measure applications, and tools that include some combination of both. We also identify tools which conduct automated diagnosis techniques and tools which provide facilities for experiment management. This chapter not only presents related work, but also depicts the current state of the art for parallel performance tools.

2.1 Environment and System Analysis

Tools for environment and system analysis fall into three categories: system administration utilities, system monitoring packages, and hardware counters.

2.1.1 System Administration Utilities

It is common for operating systems to provide a wide range of system administration utilities [27]. These types of utilities report on the active environment, including process, memory, I/O, and network activity, as well as on machine, device, and operating system configuration details. Although generally categorized as administrative tools, application performance analysts use many of these facilities in the process of gaining understanding about the environments in which their applications execute. Examples of utilities available on Unix and Linux operating systems that measure operating system performance include *proc*, *sar*, *top*, *ps*, *vmstat*, and *iostat*. The *proc* file system is a highly informative tool. This virtual file system resides in the kernel memory and provides comprehensive reports on a full range of configuration details and process activities. The *sar* utility samples selected operating system counters at regular intervals. By default, *sar* reports CPU utilization as percentages of time for specific intervals for user level utilization, kernel level utilization, idle time during which the system had an outstanding disk I/O request, and idle time in which the system did not have an outstanding disk I/O request. The *top* utility is another tool used by performance analysts. This utility, which updates values while it is running, provides a view of the process table which can be ordered by CPU usage, memory usage, or by user. The continually refreshed header is useful for capturing a summary of system load, user and system utilization of the CPUs, and memory activity. The *ps* utility, available on Unix and Linux systems, presents a snapshot of current processes. Other Linux and Unix system utilities commonly used by performance analysts and application developers include *vmstat* to report virtual memory statistics and *iostat* to report on CPU and device utilization.

Most platforms provide tracing utilities, allowing users and administrators to gain insight about the behavior of certain events. For example, Linux offers *strace* to trace system calls made by a process and *ltrace* to trace the dynamic library calls made by a process. AIX has *trace* which traces user-specified events like function entry and exit, kernel routines, and interrupt handlers. The Solaris operating system provides *truss* for tracing system calls invoked by a process and *sotruss* for tracing library calls made by a process. Newer versions of Solaris also incorporate a utility called *DTrace* for dynamic tracing of user level, operating system, and kernel events.

It is typical for operating systems to provide administration utilities for monitoring and measuring network activity. Utilities commonly used for these tasks include *tcpdump*, *netstat*, *ping*, and *traceroute*. The *tcpdump* utility prints the header information of packet traffic on a network interface. The utility allows network administrators to inspect the frequency and kind of traffic occurring on a network. To obtain status information about an entire network, the *netstat* utility can be used. Information can be obtained regarding the status of active sockets, network interfaces, internal data structures, routing tables, and multicast memberships. Another useful utility is *ping*, which can be used to determine if a host is in operation and what the round trip time is for sending to and receiving packets from the host. The network administration utility *traceroute* is used to determine a possible path a packet may take to reach a destination.

In general, system administration utilities, most of which operate with a text interface, are very good at reporting raw data and leave the task of interpretation to higher level functions. They report some of the kinds of information about the runtime environment that environment-aware performance analysis tools need in order to correlate the runtime environment to application performance. However, these utilities do not provide mechanisms for coordinated analysis across runtime layers, nor are there provisions for relating the measurements to the performance of an application.

2.1.2 System Monitoring

System monitoring packages generally facilitate gathering several system level measurements and reporting statistics about where time is spent.

OProfile, bundled with the Linux kernel, is a system-wide profiler, capable of profiling all running programs and kernel activity [38, 60]. OProfile produces post-mortem reports of profiled activities, and it provides an option to output annotated source, showing profile statistics next to lines of source code. OProfile does not attempt to determine the causes of what it terms “hot spots.” OProfile supports single and multi processor systems, but does not support distributed memory or message passing systems. OProfile contains several elements which would be desirable in a high performance environment-aware tool: It captures measurement information across several layers for an entire system; it does not have to be invoked for or by a specific application; and it has the ability to collect statistics continually. However, it does not include the ability to coordinate measurements across processes in a distributed memory environment.

The Network Weather Service (NWS) is a research project focused on producing accurate short-term performance forecasts for distributed systems [95, 94]. The goal is to characterize and forecast the performance that an application can expect to achieve from a set of distributed computational and network resources. This requires a perspective of the entire runtime environment in which the application executes; in particular knowledge of the capacity and availability of resources and how these restrictions change over time. The NWS research is applicable to Grid computing, and the design of the NWS architecture is composed of services and sensors. NWS does not provide application level instrumentation or analysis, but rather predicts the availability of resources that an application can expect to leverage in the short term.

CODE is a software framework for Control and Observation in Distributed Environments, supporting Globus-based grids [79]. This framework is a toolkit, enabling performance monitoring and management of resources. The design targets system administrators, providing an interactive GUI that allows for resources to be clicked on and current status reports to be viewed. Through the GUI, the current status of the grid, as perceived by the set of monitors deployed, can be viewed; notification messages about problems are logged; and historical grid performance can be viewed. CODE targets resource utilization, and does not relate this analysis to applications executing in the grid environment.

NWPerf, developed at Pacific Northwest National Laboratory, is a system wide performance monitoring tool for Linux clusters [53]. NWPerf seeks to address the problem of gathering fine-granularity system performance metrics while sustaining low and acceptable levels of interference to user applications. Similar to OProfile, NWPerf is not invoked for a specific application, but rather monitors all applications running on a Linux cluster, providing course-grained profiling information about individual application performance. One of the research goals is to be able to determine how well a system is utilized by the jobs that run on it. The prototype implementation collected two kinds of metrics: CPU performance counter information, like percent of peak flops and memory bytes per cycle; and information analogous to that reported by *vmstat*. Similar to the other monitoring facilities discussed, NWPerf does not relate the behavior of the monitored system to events and behaviors within an application's execution.

2.1.3 Hardware Counters

Hardware counters provide another means by which to obtain data about the runtime environment. Many of the high performance tools discussed in this chapter provide support for hardware counter data collection. Low level measurements can be obtained by the hardware counters provided by individual system architectures. On any given platform, hardware counters exist as a limited number of registers which count events affecting processor functionality. Examples of events that hardware counters can capture include cache hits and cache misses for each cache level; counts for completed integer, floating point, load, and store instructions; and instruction stalls. The number of counters available and the types of events that can be counted are architecture dependent. Since there is great variability among platforms, interfaces for accessing hardware counters have been developed. This enables application level code to access a single interface to obtain hardware counter data, and, as a result increases portability across architectures.

The Portable Interface to Hardware Performance Counters (PAPI) [61, 4] and the Performance Counter Library (PCL) [65, 3, 2] are hardware counter interfaces specifically targeted to performance analysis in high end computing. They share the same goal of easing the task of incorporating hardware counter information into high end performance analysis tools. Both PCL and PAPI provide access to raw hardware counter data. Additionally, they also provide a number of derived metrics, rates, and ratios which are compositions of the raw metrics. Although providing hardware counter access for a wide range of platforms, PCL does not easily support newer architectures, as it has not been upgraded for several years. PCL is supported in the Tuning and Analysis Utilities (TAU) suite of parallel performance tools [78]. PAPI support is enabled in a handful of parallel computing performance analysis tools, including KOJAK and TAU [93, 78, 61].

2.1.4 Discussion

In this section, we have discussed three typical mechanisms by which information about the runtime system can be obtained. With operating system provided utilities, a wide variety of data can be captured regarding processor, memory, I/O, and network usage. There are also a number of tracing facilities for tracing system calls and library calls. System utilities generally report raw data and provide very little data analysis. System monitoring packages facilitate simultaneous collection of much of the same information that can be obtained using system utilities. Monitoring packages usually include analysis facilities for providing summary information. Application developers can use hardware counters and higher level interfaces to access the counters to obtain system event count data. With these mechanisms it is possible to understand the behavior of computing systems. However, these mechanisms do not include facilities for correlating system events and application events.

2.2 Application Focused Performance Tools

Many of the performance analysis tools developed for high end computing focus primarily on application level performance. In this section we present several tools which represented early advances in parallel performance research; we provide an overview of the MPI profiling interface and present several of the commonly used MPI profiling libraries; and we present some of the application focused parallel performance tools used in today's high performance environments.

2.2.1 gprof

Although not a parallel performance tool, gprof [18, 19] deserves mention for several reasons. It is widely used in performance tuning of sequential codes; it is supported on Unix and Linux platforms; it has been a viable tool for over twenty years; and it is used by parallel performance analysts, even though it does not support parallel codes. gprof is a simple to use tool which conducts profiling, by sampling the program counter at regular intervals. It records the measured data in text files. gprof extends the Unix *prof* utility. This utility is capable of providing a listing of each function, the number of times each function is called, the time spent in each call, and the average time per function call. However, it is not possible, with *prof* data, to determine the callers or callees of functions. This knowledge can be somewhat ascertained by knowing the call structure of the source code. However, this view does not predict the exact nature of the runtime behavior of function caller and callee relationships. To provide this knowledge, gprof maintains caller information and reports call graph detail, showing caller and callee relationships. Providing call graph information and the tool's ease of use enabled gprof to become a widely used tool.

2.2.2 AIMS

One of the early tools developed for distributed memory parallel performance analysis was AIMS, an acronym for Automated Instrumentation and Monitoring System [96]. AIMS was developed by NASA Ames Research Center and supported parallel programs written in C and FORTRAN which used message passing implementations that pre-dated the MPI standard. AIMS used trace based instrumentation so that events of an application's execution could be reconstructed during post-mortem visualization. The instrumentation points were automatically inserted into source code before compilation. After applying the instrumentation, applications were compiled and linked with the AIMS run-time performance monitoring library. This library facilitated data collection across the participating nodes of an application's execution, and managed the process of writing collected events to a single trace file. Events that could be traced included entry and exit points for subroutines, loops, and user-defined code segments; time spent in communication routines; time spent in file read and write operations; and time spent in barrier operations. Post-mortem visualization of traced events was enabled by the built-in visualization tool kit. The *vk* feature of the tool kit provided several views for understanding an application's behavior over time. These included an animation facility, in which the events could be viewed at several speeds; and a source code mapping facility, in which trace events were associated with exact points in application source code. The visualization tool kit also provided a mechanism to generate lists of resource utilization statistics for nodes and routines, with the intention that the data provided in these lists could be used as input to statistical graphing packages.

For the most part, AIMS focused entirely on application behavior. It did not collect any information about the runtime environment in which an application executed. However, the final version of AIMS [97] included a utility called *sysconfig* which enabled a static view of the likely routes for packets to take for communications between nodes. *sysconfig* examined either a host file or an AIMS trace file to extract host machine names and then used *traceroute* to determine packet routes for every machine pair. The data were saved in a file and could be viewed in conjunction with execution trace records.

2.2.3 MPP Apprentice

The original MPP Apprentice [91] was designed for application tuning on the Cray T3D. Data collection was accomplished via a “stop-watch” technique, which is the combination of timing and counting instrumentation methods. As such, performance information was presented from the perspective of the source code, rather than from the perspective of time. The MPP Apprentice collected stop-watch data for code sections, including the whole program, sub routines, and code blocks. The instrumentation methods were built into the compiler, and instrumentation was turned on by a compile time flag. During execution of an instrumented application, collected data were aggregated within each processor and maintained locally in each processor’s memory. Either by user request or at the end of execution, the per-processor data were aggregated across all processors and stored in a file, which was used as input to the post-mortem visualization component. The visualization component provided a summary report and graphical display of count and timing statistics for the entire execution and all subroutines, ordering “critical” subroutines at the top of the list. Long running subroutines were categorized as “critical,” and users could tune this value to suite an expectation about application runtime performance. Additionally, for the instrumented subroutines, summary information for time spent in overhead, parallel work, I/O, and called routines was presented. Instrumented subroutines could be expanded to view details about nested code sections, like loops and “if” clauses. Upon user request, the MPP Apprentice displayed the source code associated with any viewable construct. Another possible display with the MPP Apprentice was the call sites view which showed all the call sites for a selected subroutine and all the calls made by the selected subroutine. The MPP Apprentice included a knowledge base which served two primary purposes. First, derived metrics from the base statistics collected could be calculated and reported in the visualization session. For example Flop/s and cache usage metrics could be computed and displayed. Second, the MPP Apprentice knowledge base offered tips and suggestions to users. The tool could inform the user about properties of the architecture design that could affect performance. For certain observed performance problems, the MPP Apprentice provided suggestions on how to pursue performance problem resolutions.

Cray has continued to provide performance tools as part of a Cray system. In newer architectures, the instrumentation, data collection, and file generation components are done by the Cray PAT tools, while the Apprentice^2 acts as a visualization GUI [12].

The MPP Apprentice provided summary information about event counts and execution times, but did not have the ability to reconstruct the events of an application execution according to time. However, its visual displays, acute knowledge of the system architecture, and ability to provide tips and suggestions regarding critical performance areas assisted users in performance problem detection and resolution.

2.2.4 The MPI Profiling Interface and MPI Profiling Tools

The MPI standard [81, 21] specifies a profiling interface for MPI implementations where all MPI functions have a second name identical to the normal name, but preceded with a ”P”. Within an MPI implementation these “P” functions are wrapper functions which simply call the normal named MPI function. MPI profiling libraries contain functions that are named the same as the normal named MPI functions. The functions are implemented so that they call the “P” named functions that exist in the MPI implementation. Besides being simple wrappers to the “P” functions, typical implementations conduct profiling activities, maintaining event counts and execution times. The library functions call the MPI implementation’s “P” named functions. This makes it fairly easy to develop profiling libraries separate from an MPI implementation, and it enables MPI applications to be developed without needing to know specific details about the kinds or types of instrumentation that might be conducted via the profiling interface.

The MPI Parallel Environment (MPE) package is a tool kit for MPI applications [54]. The profiling component of MPE is an example of a library that takes advantage of the profiling interface provided by MPI implementations. MPE collects and logs statistics related to message passing activities during a program’s execution. MPE includes the Upshot, Nupshot, and Jumpshot [67, 23, 35, 98, 99] viewers for post-mortem visualization of the profiled data. MPE is part of the MPICH distributions, but can be downloaded separately and works with most MPI implementations, including Lam/MPI [41] and OpenMPI [59].

Another MPI profiling library is mpiP [55]. Developed at Lawrence Livermore National Laboratory (LLNL), this profiler collects statistical information about MPI function calls and presents the summary data to the user in tabular format after execution of a profiled application. mpiP can relate function call statistics to source code line numbers. Tool Gear, also developed at LLNL, is a graphical viewer for mpiP output [85]. Tool Gear uses the mpiP data regarding source code line numbers to display the specific source code involved with a call site statistic reported by mpiP.

2.2.5 Upshot, Nupshot, Jumpshot

Argonne National Laboratory (ANL), promotes a research program called Performance Visualization for Parallel Programs [66]. This research focuses on post-mortem visualization of parallel application performance. Over the course of several years, Upshot [23], Nupshot [35], and Jumpshot [98, 99] have been developed and implemented for this purpose. Chronologically, Upshot was developed first and is written in Tcl/TK. Nupshot was developed as an improved version of Upshot, in which part of the Tcl code was re-written in C to improve graphic performance. Jumpshot, a Java-based tool, is the most recent viewer in the series and expands the capabilities of both its predecessors. Currently, all three tools are part of the MPE suite of performance tools [54]. These tools focus entirely on post-mortem visualization of trace records. They do not provide facilities for instrumentation, data collection, or trace generation. The tools support several trace file formats and present visual representations of the events contained in a trace file. The primary visualization feature is the zoomable timeline which shows the interaction of events among participating processes and nodes.

2.2.6 Discussion

In this section we presented application focused parallel performance tools. The tools in this section only collect application related measurement data. Since these tools do not have knowledge about the runtime environment in which applications execute, they are only able to present performance behavior in terms of the application. In the following section, we will identify parallel performance tools which collect varying degrees of information about the runtime environment.

2.3 Integrated System and Application Level Performance Tools

In this section we present systems and tools which support collecting measurement data related to application performance and system related metrics. Most tools in this section collect runtime metrics via hardware counter support. Even though these tools integrate system level metrics into data collection techniques, they do not, for the most part, provide performance diagnoses which consider both the application and the runtime environment. It is most often the case that the analysis results are displayed and diagnosis of application performance behavior is conducted by the user of the tool.

2.3.1 IPS-2

IPS-2 was an early research tool developed at the University of Wisconsin for parallel application performance analysis [47]. IPS-2 used tracing as the mechanism for collecting performance data. Events, such as synchronization operations and procedure entries and exits, were logged to trace files. Instead of modifying applications to insert instrumentation manually, users only had to specify a compiler option and IPS-2 applied the instrumentation automatically. The interactive user interface allowed the user to specify what aspects of the program should be measured and displayed performance analysis results after the program's execution. Results could be viewed in tabular form or as a time histogram.

Critical path analysis and phase behavior analysis were two automated techniques, offered by the interactive interface, which guided users toward deciding how best to improve an application's performance. In an execution, the critical path is the execution path that consumes the most time. From an execution's trace history, IPS-2 identified this path and identified the execution times for components along the path. Since decreasing the execution time of the critical path causes a decrease in an application's total execution time, users could use the critical path analysis to decide which component along the critical path should be targeted for optimization. Phase behavior analysis, a second automated guidance technique introduced in IPS-2, identified the phases of a program’s execution. With this technique, each phase could be evaluated as a smaller sub-problem of the entire execution. For example, critical path analysis could be applied to a phase.

IPS-2 was later extended to support external metrics, a feature which allowed users to define metrics not provided in the default set of metrics [26]. In the original implementation, only events at the application level were measured. With this added support, it was possible to define metrics at other levels of the runtime environment, including the operating system, the hardware, and the network level. One such external metric, described in [26] used filtered data from the `vmstat` system call. The user interface allowed for transparent viewing of the performance results of both the application level and the external metrics.

IPS-2 provided graphical displays for all metric values, which was useful for viewing aspects of program behavior along side aspects of system behavior. This feature combined with the critical path and phase behavior analysis techniques improved the degree of assistance offered to the user in deciding how an application could be modified.

2.3.2 Paradyn

The Paradyn parallel performance measurement tool [46], developed at the University of Wisconsin, finds it roots in IPS-2, but is quite different in the way it provides instrumentation and analysis. Paradyn uses an instrumentation technique called dynamic instrumentation, in which instrumentation is inserted into a running executable. Additionally, Paradyn presents the results of performance analysis in an online fashion, while the application executes. User specified metrics can be viewed, online, via bar charts, tables, and histograms. Additionally, the Paradyn Performance Consultant conducts automated bottleneck search and detection; and presents the results online. We discuss the Performance Consultant in greater detail in section 2.4.2. Paradyn can be used to analyze MPI and sequential applications; and it supports a range of Solaris, Linux, Windows, and AIX platforms [62].

By default, Paradyn does not provide metrics for measurements external to the application. However, Paradyn allows users to create new metrics using the Metric Definition Language (MDL). With this mechanism, it is possible to collect measurements about aspects of the runtime environment. All metrics can be viewed with the Paradyn visualizations.

Although system level metrics can be collected and visualized, two important features of environment-aware performance analysis are missing: Paradyn does not provide system level integration by default, and Paradyn does not utilize the data associated with external metrics in its automated diagnosis techniques.

2.3.3 DPCL

The Dynamic Probe Class Library (DPCL) is a dynamic instrumentation infrastructure for performance tools [11]. At the time of DPCL's development, the Dyninst Application Program Interface (API) [6] could be used to provide dynamic instrumentation. Dyninst facilitates dynamic instrumentation of code that executes on a single machine. However it does not provide a means for managing dynamic instrumentation of executing code that spans multiple nodes. Prior to DPCL, parallel tools which conducted dynamic instrumentation had to provide the necessary support for managing dynamic instrumentation across multiple nodes. DPCL utilizes Dyninst, but eases the task of supporting dynamic instrumentation in parallel tools by providing the necessary infrastructure to manage dynamic instrumentation across multiple nodes of an executing application. Originally implemented by IBM and distributed as an IBM licensed product, DPCL was later made available under an open source license.

2.3.4 DCPI

The Digital Continuous Profiling Infrastructure (DCPI) was designed to run continuously on multi-processor systems, conducting sample-based profiling of entire system activity, including user programs, shared libraries, and operating system kernels [1]. DCPI was designed for Digital Alpha processors and was supported on the Digital Unix platform. The data collection component sampled program counters with high frequency and stored the profiled data to disk. DCPI also supported hardware counter sampling. DCPI provided several utilities for post-mortem analysis of the stored data. These tools displayed the number of samples per procedure; calculated cycles per instruction; identified instruction stalls and provided possible causes for stalls; and analyzed profile data across multiple sessions.

Although DCPI supported a narrow range of processors and operating systems, DCPI showed that it is possible to conduct fine-grained sampling for long periods of time with minimal overhead. This is an important consideration for environment-aware performance tool development.

2.3.5 VAMPIR

VAMPIR (Visualization and Analysis of MPI Resources) is a trace-based tool for post-mortem visualization of MPI programs [57]. Originally developed and maintained as a commercial product by Pallas GmbH, VAMPIR is now an Intel product and is distributed as part of the commercial Intel Trace Analyzer and Collector [28]. In this form, only Intel based architectures are supported. VAMPIR supports a variety of platforms and implementations of MPI, and tracing is not limited to just MPI routines. Trace data can be viewed in several ways, offering generalized summaries of events on each node and time-line displays which show activities for all nodes over the execution interval. The time-line displays incorporate a zoom feature, allowing users to zoom in and out on program details. VAMPIR can collect and display counter data, and it supports the PAPI hardware counter interface. In the time-line display, counter data is displayed in the same view as application events. Since its initial development, a next generation version of the tool, VAMPIR NG (VNG), has been released [5]. Accomplishing much of what VAMPIR did, VNG operates as a distributed and parallel application. With VAMPIR and its successors, visualization of program activity is the primary goal. The post-mortem visualization sessions allow users to analyze the behavior of application executions. Although it has the ability to present system level information, through its display of counter data, and application event data, VAMPIR does not interpret the data or detect performance problems. The user is responsible for this type of analysis.

2.3.6 SvPablo

Source View Pablo (SvPablo), developed at the University of Illinois, is a graphical environment for instrumenting application source code and viewing performance data for sequential and parallel applications[10]. SvPablo incorporates components of the Pablo Performance Analysis Environment, also developed at the University of Illinois [75].

Data collection in SvPablo is similar to profiling, capturing only statistical information rather than detailed event traces. Additionally, SvPablo supports hardware counter data collection. Instrumentation is inserted into source code, prior to compilation, and can be done either automatically or interactively. After execution, the application level data and hardware counter data collected from each processor are merged and additional metrics are computed. These results are correlated to application source code and are viewable via the tool's browser facility.

The visualization interface displays application summary statistics and hardware performance data. The interface also supports multi-execution displays, so that users can compare analysis data for multiple executions. To display source code associated with a function, users click on the function name.

2.3.7 SCALEA and SCALEA-G

ASKALON is a research project aimed at developing a programming environment and tool set for cluster and grid computing [16]. The vision for the framework contains five components: AKSUM, an automatic performance bottleneck analysis tool; Performance PROPHET, a performance modeling and prediction system; SCALEA, a performance instrumentation, measurement, and analysis tool; ZENTURIO, an automatic performance experiment management system; and an integrated graphical user interface for the tool set.

The performance analysis tool SCALEA [88] supports OpenMP, MPI, HPF, and mixed parallel/distributed programs. Performance analysis is done in a post-mortem fashion, and visualizations are generated from trace files. Instrumentation can be inserted automatically for pre-defined code regions and it can be inserted manually, using SCALEA directives. SCALEA also supports hardware counter profiling. Post-mortem analysis relates results to source code lines. This appears seamless to the user as all collected profiling and trace data are correlated with the corresponding code regions.

SCALEA-G is a system monitoring and performance analysis package designed to encompass Grid computing [89, 90]. In Grid computing, system monitoring and application performance analysis techniques have developed separately, although both are viewed as crucial to Grid performance. SCALEA-G is a first attempt at integrating both system monitoring and performance analysis into a unified tool for Grid computing. System monitoring is accomplished through a network of services and sensors; and instrumentation can be inserted into original source code, requiring recompilation, or it can occur dynamically, while an application executes. The interactive GUI allows users to create and add monitoring sensors to running systems, and it provides online application analysis and system monitoring. Application performance and the Grid system as a whole can be monitored via the same interactive session.

Both SCALEA and SCALEA-G include aspects of environment-aware performance analysis. SCALEA incorporates hardware counter profiling and SCALEA-G packages system monitoring and application performance analysis for grids into one tool. Both tools provide integrated visualizations of application and system level analysis results.

2.3.8 PerfSuite

PerfSuite is tool kit for Linux desktop and cluster applications [40, 39]. It is in active development at the National Center for Supercomputing Applications (NCSA). PerfSuite conducts hardware performance event counting and profiling of applications. It supports MPI, single-threaded, and POSIX multi-threaded applications. Raw performance data are written to XML files, and the analysis of the data is done by a post-mortem process which calculates summary and derived results; maps event data to source code lines; and combines multi-file collected data that may span processes and machines to calculate summary data for the whole set. PerfSuite provides summary results of counts and profile statistics in text format, but supports the export of data to other formats. In 2005, NCSA moved PerfSuite into full scale production, automatically measuring all applications that use the NCSA SGI Altix system, including parallel shared and distributed memory applications [68].

PerfSuite provides hardware counter support and application profiling; and it presents summary results of hardware counter performance and application profiled statistics.

2.3.9 Active Harmony

Active Harmony is a research project that focuses on online and automatic adaptation of applications and work loads to changing conditions in parallel and distributed computing environments [25]. We discuss the techniques for conducting automated adaptation in Section 2.4.6. Active Harmony considers the application and the resources utilized by the application as tunable parameters of the execution environment.

In order to facilitate online adaptation of applications and resources, Active Harmony requires a perspective of the interactions and relations of the application and the runtime environment. This includes ideas of environment-aware performance analysis. The goals of Active Harmony, however, differ from the goals presented in this thesis. Active Harmony focuses on efficient resource utilization for a whole system, while our goals target parallel performance improvement.

2.3.10 TAU

Tuning and Analysis Utilities (TAU) is a research tool for parallel and distributed performance analysis [78]. TAU can be configured to run on many platforms, and it supports MPI and shared memory parallel applications.

TAU supports profiling and trace-based data collection. It offers many methods for applying instrumentation, including source-based, allowing users to manually annotate source code following a TAU specification; preprocessor-based, where a user selects the kinds of instrumentation to apply, but the source code is instrumented automatically before compilation; compiler-based, where the compiler inserts instrumentation into the object code that it generates; wrapper library-based, where standard library routines are substituted with wrapper routines that include instrumentation and call the original routine; and dynamic instrumentation. TAU includes hardware counter support and provides interfaces to PAPI and PCL hardware counter libraries.

Performance analysis and visualization is done post-mortem. TAU includes profiling data support with ParaProf and PerfDMF. ParaProf supports single experiment visualization, and multi-experiment visualization can be conducted with PerfDMF. TAU does not implement its own support for trace analysis and visualization, but supports export of trace collected data to other analysis and visualization tools, including Vampir, Jumpshot, Paraver, and EPILOG.

In TAU's profiling analysis and visualization tools, TAU reports and shows summary information about function calls and hardware events. Similar to other visualization based analysis tools, TAU does not identify causes of performance problems, and interpretation of analysis is largely the responsibility of the user.

2.3.11 PEM and CPO

A collaborative research project between IBM and the University of Colorado has resulted in several key ideas related to the integration of environment and application performance analysis, with the goal being automated and online optimization of a whole system [92, 8]. The paradigm presented by this work is referred to as Continuous Program Optimization (CPO), which involves cyclic phases of monitoring and optimization. In the monitoring phase, information across system layers is obtained and analyzed; and in the optimization phase, this information is used in a continual process of feed-back directed adaptation where applications adapt to the execution environment and the execution environment can change to improve application performance. The monitoring facility of the architecture is termed PEM, an acronym for Performance and Environment Monitoring. This component is responsible for capturing the behavior of the runtime system. PEM is presented as a vertical set of layers in which the interactions among hardware and software are understood and optimizations are relevant based on the analysis of the interactions. The presentation of the runtime environment as a set of layers is similar to the notion presented in this thesis work for environment-aware diagnosis. The framework presented in the CPO and PEM work does not include explicit instrumentation techniques for capturing application level performance data. The experimental work presented to date does not include deployment on cluster or distributed systems, but only single machines. Nor have thoughts been presented about how this framework will scale to large cluster and distributed parallel systems.

2.3.12 Discussion

In this section we presented tools, infrastructure systems, and research ideas that support data collection of system level metrics and application level metrics. Most of these systems present the analysis data for both sets of metrics in single visualization sessions, but largely rely on users to infer diagnoses.

2.4 Automated Performance Diagnosis

In this section we present tools which conduct automated performance diagnosis. This means that the tool uses some form of automated analysis to determine where performance problems exist, when they occur, and why they occur. This strategy is called the W^3 approach [46]. It has generally been implemented with respect to the application that a performance tool analyzes. Most of the tools presented in this section automatically determine where, in the application source code, performance problems exist; when, during an execution, performance problems occur; and why is generally answered by determining the state of the executing application at the point of the performance problem. We believe that this approach is a necessary component of environment-aware diagnosis, but that the strategy must be expanded along each dimension to incorporate the status of the runtime environment and its impact on application level performance.

2.4.1 ATExpert

Designed for Cray platforms, the goal of the ATExpert was to assist users in application tuning so that the best possible speedup could be achieved [37]. The tool worked in conjunction with the Autotasking compiler which automatically instrumented applications if a specific compiler option was used. ATExpert post-processed Autotasking's runtime trace files and graphically displayed the performance of parallel programs and predictions for speedup. The key features of ATExpert were its ability to relate performance characteristics back to source code, its ability to simulate what Autotasking's measured results would have been if all possible numbers of processors were available for the application execution, its graphical displays of performance and predicted speedup, and its ability to associate suggestions for improving performance with specific performance observations. For the most part, ATExpert focused on application performance and did not consider how and to what extent the runtime environment affected application performance. However, the rules-based mechanisms employed for relating specific actions to take for improving application performance to specific performance behaviors demonstrated by the application is an idea that can be carried forward into the design of environment-aware performance analysis tools.

2.4.2 Paradyn

Key to Paradyn, is the Performance Consultant (PC) [46, 62], a module which automates the process of discovering performance problems. Paradyn introduced the W^3 method for bottleneck search and detection. This method searches for bottlenecks along three dimensions: the "why" axis, the "where" axis, and the "when" axis. The results of the PC search are graphically displayed to the user, as the application executes, by the Search History Graph.

The PC operates on a set of hypotheses which are predefined performance problems and are evaluated for values of true or false. The hypothesis set composes the “why” axis and refers to the types of performance problems that Paradyn can identify. The current hypotheses include evaluations for comparing CPU time, total synchronization waiting time, total I/O waiting time, and average number of bytes per I/O operation to threshold values. The “where” axis contains the program resources which can be considered for evaluation. Paradyn’s default setting is to start automated searches by considering the whole program. The “when” contains the phases of a program. Detected performance problems are associated with specific phases of an application’s total execution. Each phase can be inspected by a different PC. The phase associated with the entire execution is called the global phase.

The PC begins its search along the “why” and “where” axes, by evaluating the top level hypotheses for the whole program. If any one of the hypotheses evaluates to true, further refinement along that hypothesis and associated resource hierarchy is conducted. It is possible to have more than one true hypothesis. If a performance bottleneck is detected, the Search History Graph displays the component of the resource hierarchy where the bottleneck occurs, identifies the hypothesis that evaluates to true for the bottleneck, and indicates the phase in which the problem occurs.

In Paradyn, the “why” axis contains the types of performance problems included in the PC’s search, and the “where” axis includes the components of the application which are instrumented and subsequently evaluated for performance problems. In its current form, the Performance Consultant does not include performance problem hypotheses which test aspects of the runtime environment. This limits its ability to provide quality diagnosis about the true causes of performance problems; especially if the causes are rooted in layers of the runtime environment external to the application.

2.4.3 KOJAK

KOJAK (Kit for Objective Judgment and Automatic Knowledge-based detection of bottlenecks) is a generic environment for automatic performance analysis of parallel programs [17, 51, 93]. The goal of KOJAK is to automatically identify bottlenecks which fall into the dual categories of well-defined and typical. This goal is motivated by the fact that most parallel performance tools at the time of KOJAK's initial development did not apply automated techniques to identifying common case bottlenecks, such as those due to load balancing problems or message passing overhead.

2.4.4 ASL

The working group on Automatic Performance Analysis: Resources and Tools (APART), introduced a specification in 1999 for automatic performance analysis. The specification is called the APART Specification Language (ASL) [15]. The specification is a formalization of performance bottlenecks and the data required to detect them. Although the specification acknowledges that performance data encompasses metrics about applications and the runtime environment, it does not address the issue of diagnosing the cause of bottlenecks. The goal is to formalize what a bottleneck is so that automated bottleneck detection is easier to build into a tool.

2.4.5 AKSUM

AKSUM [14], a component of the ASKALON project [16] , is a performance analysis tool for parallel applications which conducts automatic searches for bottlenecks over multiple experiments. In its current implementation, automatic instrumentation is facilitated by ASKALON’s SCALEA [88], a single experiment performance analysis tool. AKSUM assumes the existence of an experiment management system and works with ZENTURIO [73], the experiment management component of ASKALON.

To guide the automated search for performance bottlenecks, AKSUM uses performance property specifications. A performance property is a predefined description of a specific negative performance behavior that could occur in an application execution. In an execution a performance property is “true” if the behavior specified occurs. Additionally with each performance property instance, AKSUM assigns severity and confidence values, indicating how severe the problem is and the degree of confidence in the indication that the problem exists. AKSUM defines several default properties and allows users to custom define new properties. Users are also able to specify what performance properties should be searched for in an analysis session. Once instrumentation is complete, the application is compiled and executed by the experiment engine, which interfaces with the experiment management system. For every normally terminating execution, the performance data is stored in the experiment management system. This multi-experiment data is available to ASKUM's performance property search mechanism.

2.4.6 Active Harmony

From an application perspective Active Harmony [25] provides automated diagnosis through an exported metric interface which allows applications to access metrics about processors, network activity, and operating system activities. Applications are able to export tuning options to the execution environment, specifying expected consumption of system resources. Then the system adapts applications to changes in the environment. Making the decision about how to adapt an application requires a prediction of the effectss of a proposed adaptation. Active Harmony provides metrics to facilitate this decision making. In the prototype implementation Active Harmony had developed a Load Balancing Factor (LBF) metric. One of the variations of the LBF computes the impact of reconfiguring the assignment of processes to processors. This impact is presented as the potential improvement in execution time of each proposed change in configuration.

2.4.7 Self-propelled Instrumentation

The goal of self-propelled instrumentation [49] is to automate anomaly detection and root cause identification of anomalies. This is an aspect of automated diagnosis which is being researched at the University of Wisconsin. Self propelled instrumentation is a form of dynamic instrumentation, in which an autonomous agent follows the program's flow of control instrumenting to a fine level of granularity. Data collection is similar to tracing, where each instrumented event is recorded. Trace data are analyzed to look for differences among processes, for example one process stops earlier or later than others. Two techniques have been developed for automating the process of determining the root cause of a failure, and both show promise in identifying root causes in the application level.

2.4.8 Discussion

In this section we presented tools which conduct automated performance diagnosis. The parallel performance tools discussed in this section automatically determine where, in the application source code, performance problems exist; when, during an execution, performance problems occur; and provide diagnoses as to why performance problems exist. The diagnoses, however, lack specificity if the root cause of a performance problem is external to the application. Environment aware performance analysis seeks to identify root causes regardless of where they exist in the runtime environment.

2.5 Tools for Experiment Management

An important aspect of performance analysis is the ability to compare different executions. In high performance computing, multi-execution comparison is cumbersome and error prone in the absence of automated experiment management. In this section, we discuss how experiment management applies to parallel performance analysis and we present two experiment management systems.

2.5.1 Early Work

Early work in experiment management involved laying out a comprehensive framework for comparing executions of an application and determining differences in performance between executions [32]. The initial work in this area proposed the idea that every run, simulated result, and model are experiments and can be organized and compared via an experiment management system. Three key ideas were proposed and demonstrated by Karavanic and Miller [32, 33]. First, an experiment management system must be able to map the full space of an application, spanning all of its experiments which differ by both execution environment and code structure. Second, the experiment management system should automate the task of comparing two or more experiments. Third, the experiment management system should provide a mechanism for making its stored data available to performance analysis tools to use in enhancing performance diagnosis.

As high performance systems continue to get larger and more complex, automated experiment management becomes more and more necessary in large-scale performance analysis endeavors. Experiment management, however, is still largely an area of research. Recent work in the capabilities and extensions of automated experiment management include ZENTURIO [73] and PerfTrack [34].

2.5.2 ZENTURIO

ZENTURIO is an automatic experiment management system designed with the goals of cluster and grid computing in mind [73]. It is one of the components of the ASKALON infrastructure [16]. ZENTURIO is designed as three grid services: a registry service, an experiment generator, and an experiment executer. The client entry point into the system is through a graphical interface that facilitates experiment preparation, monitoring, and visualization. Using a directive language, users specify what aspects of an experiment event should be analyzed and maintained in the experiment management system. These aspects include things like execution parameters and performance metrics. Once this specification is defined, the generator and executer services carry out automatically building and launching all executions pertaining to the application. After execution, the collected performance data are analyzed, further computed, and stored using a component of SCALEA [88]. With post-mortem visualization, the performance results and output data of the most recent experiment and a subset of the full experiment set can be viewed. In the graphical session, the user can visualize a single experiment or compare multiple experiments by specifying the metrics of interest for the comparison.

2.5.3 PerfTrack

PerfTrack is a data store and interface for managing performance data and associated data related to the environment for large-scale parallel applications [34]. As an experiment management system, PerfTrack enables comparative analysis of performance results of multiple experiments. The PerfTrack model for cataloging resources and associating resources to performance results is generic and easily extendable. PerfTrack is not a performance analysis tool in the traditional sense, as it does not provide facilities for application instrumentation or measurement. PerfTrack is a means for storing data and maintaining relationships that exist between collected data and associated environment information, so that performance results can be considered in the context of the environment in which they occurred, and multiple experiments can be compared simultaneously. Using the data collection interface, PerfTrack builds a specified application, collecting and storing information about the build environment, and PerfTrack launches an application, collecting and storing any performance data that results from however the application is instrumented. PerfTrack is flexible and designed to support data collection from any tool.

2.6 Conclusion

In this chapter we identified tools and research initiatives containing components related to environment-aware performance analysis. We presented system administration utilities, system monitoring packages, and accessing hardware counters as methods available for collecting data about an application's runtime environment. These facilities, however, do not incorporate application performance analysis. We then discussed parallel performance tools which focus entirely on application performance and do not incorporate knowledge about an application's runtime environment. There are parallel performance tools which support the collection of metrics related to the runtime environment, and we presented these in Section 2.3. The majority of these tools integrate system level data via hardware counter performance monitoring. These tools support analysis of application and runtime environment data via visualizations, but performance problem diagnosis is largely the responsibility of the user. There are a few systems which incorporate automated diagnosis, but no parallel performance tool identifies root causes that exist in runtime layers external to the application. We closed this chapter with a discussion about experiment management.

CHAPTER 3 A FRAMEWORK FOR ENVIRONMENT-AWARE PERFORMANCE ANALYSIS

In this chapter, we describe environment-aware performance analysis and discuss issues related to environment-aware tool development. In the first section we present a layered view for describing the structure of a runtime environment. We define each layer of the runtime environment, identify the main components of each layer, and describe how this view can be used to frame performance problems in terms of root causes and potential optimizations. Although the idea of representing a system with layers in not new, our approach of using a layered model for performance analysis is. In the second section we discuss development considerations for the design of environment-aware tools.

3.1 The Layers of an Application and its Environment

An environment-aware approach to application performance analysis includes analysis of an application's execution behavior and analysis of the environment in which an application executes. Data collection involves application performance data capture and the collection of data about the status and behavior of the application's runtime environment. In diagnosing application performance behavior, an environment-aware approach utilizes both sets of data to determine potential root causes for performance behavior. Following the premise that the causes of poor application performance can be application rooted or can exist in areas of the runtime environment that are not part of the application itself, environment-aware performance analysis requires an understanding of the structure of the runtime environment and an approach for identifying the area of the runtime environment in which root causes exist.

We consider a runtime environment to be composed of four basic layers: an application layer, a library layer, an operating system layer, and a physical layer. Figure 4 shows this basic structure. In the following sections we describe each layer and identify the common components of each layer. The components of each layer are shown in Figure 5. The structure presented here is flexible as far as the components that can be associated with each layer. We express a general environment. For specific environments, the components could be different.

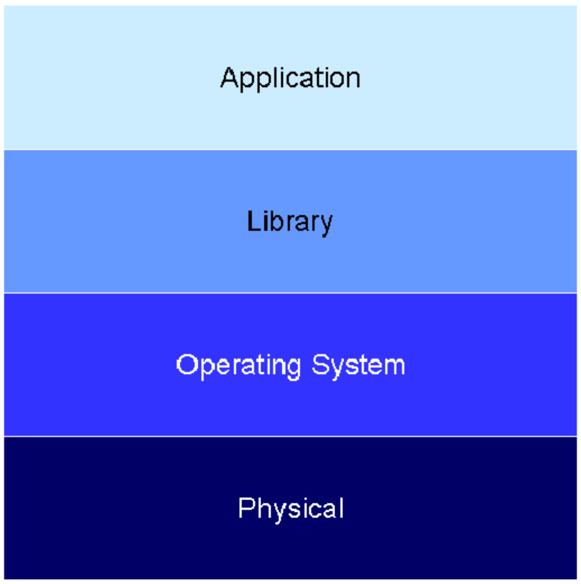


Figure 4: The Four layers of the Runtime Environment

3.1.1 Application

The application layer represents applications which are available to users for execution. This includes user developed programs, other applications that users can launch, and system utilities. An MPI program executed by a user is in the application layer. System utilities like *vmstat*, *top*, and *proc* are also in the application layer.

3.1.2 Library

The library layer is composed of modules that are required by the application layer. These include libraries users develop and build into their codes and other libraries that user programs call upon. Examples of libraries include MPI implementation libraries and the system library.

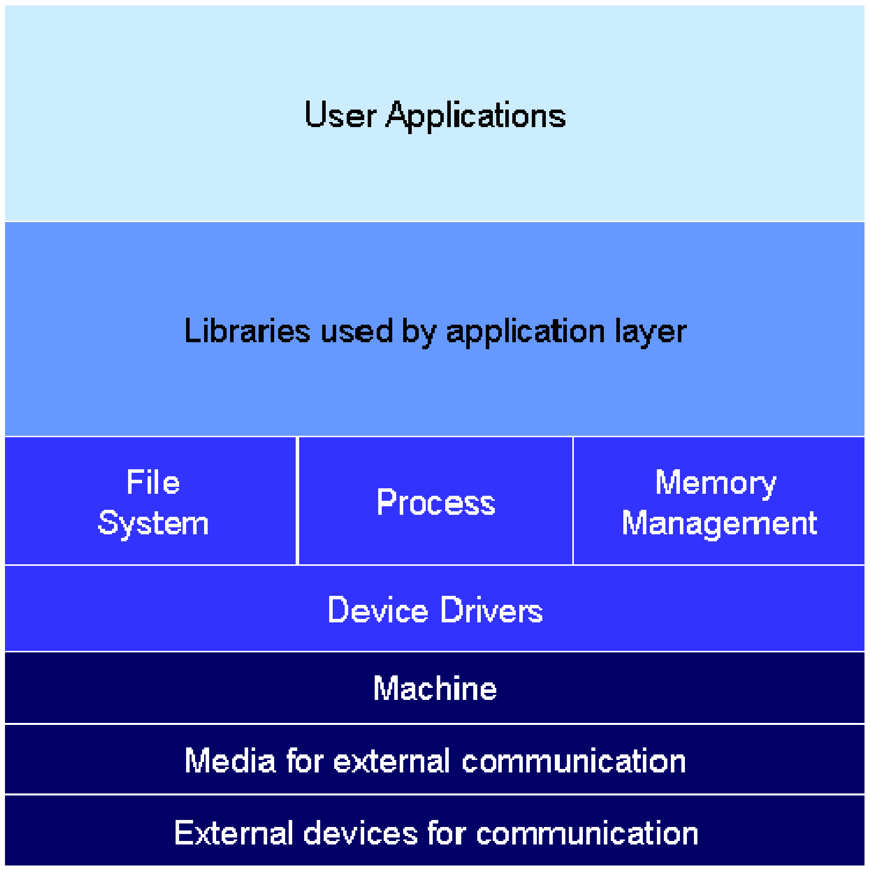

User Applications
Libraries used by application layer
File System
Process
Memory Management
Device Drivers
Machine
Media for external communication
External devices for communication

Figure 5: Main Components of the Layers of the Runtime Environment

3.1.3 Operating System

The operating system layer is composed of distinct components reflecting the properties and functions of an operating system. The lowest level incorporates the system's device drivers. In modern operating systems, all accesses to physical devices pass through this layer. Abstractions for file systems, memory management, processes, and other operating system functionality are located in the level above the device drivers. As an example of a device driver in the operating system layer, consider a system configured for Ethernet communication between nodes. In this scenario, the operating system has a device driver for the associated Ethernet hardware. In the layered structure, a representation for the Ethernet device driver exists within the device driver level of the operating system layer. Configuration data can be associated with the abstractions for file system, memory management, and process components of the operating system layer. For example, within the file system component, this data could include the name, version, and configuration settings for a particular file system. Additionally, data related to operating system state can be associated with specific components of the operating system layer. For example, the process component is an ideal categorization for the number of running processes over a time interval.

3.1.4 Physical

The physical layer contains the tangible components of the system. The model for the physical layer contains three tiers. The upper level is the machine. This level contains CPUs, memory, disk storage and I/O devices, network interface cards, and any other physical properties of a machine. The second level contains media for external communication with other devices. This level is intended to represent the physical means by which machines are connected to external machines. The wires and cables are in this level. The lowest level represents the machines and devices that a machine connects to. Routers, switches, and other machines are included in this level.

3.1.5 Specifying Root Causes and Potential Optimizations by Layers

We consider the runtime environment as a set of layers for the purpose of using a simple representation for showing where root causes of performance problems can exist and where optimizations can occur. Today's parallel performance tools often identify where, in an execution, an application spends most of its time. However, the place at which a performance problem is detected is not necessarily the place from which the performance problem emanates. Nor is it necessarily the place where an optimization should or can be applied.

The layered view of the runtime environment can be used to categorize root causes of performance problems by layer. We believe that knowing the root cause of a performance problem is fundamental in deciding what to target for optimization. Furthermore, there is often more than one way to improve application performance. Of all possible optimizations that would result in improved application performance, only a subset may be practical or available to the analyst. For example, if a performance problem is caused by an inefficient MPI implementation, an application developer without root privileges would be unable to modify the generally available MPI library. However, with the knowledge that the MPI library is at the root of the performance problem, the developer can decide upon several reasonable optimization approaches. These might include deciding to use a different MPI implementation; deciding to optimize a copy of the MPI library source code and install the modified library in a space controlled by the user; or deciding to change the application so it does not utilize the inefficient MPI components. In this example, the performance problem was caused by something in the library layer, but optimizations were possible at several layers. The layered view can be used to categorize potential optimizations by layer.

3.1.6 Application Execution and the Runtime Environment

Intuitively we understand that an application execution involves multiple passes through each of the layers of the runtime environment. From an application analysis perspective, it may not be possible to instrument and collect performance data along the entire execution path. For example, most dynamic instrumentation tools do not support dynamic instrumentation of kernel calls. However, it can be useful to know what components of the runtime environment are touched by an application's execution path. To illustrate this point, we present a simple `MPI_Send` example, in which process A on node 1 calls `MPI_Send` to send a message to process B on node 2. The components of the runtime environment involved in the `MPI_Send` on process A's machine include the following: the MPI application in the application layer; the MPI library in the library layer; memory management in the operating system layer and memory in the physical layer for accessing the contents of the send buffer; the Sockets library in the library layer and subsequent low level `write` from the System library for writing the data; the TCP and IP protocols in the operating system layer for handling the message contents; the Ethernet driver in the operating system layer; the Ethernet adapter in the physical layer representing node 1; CAT 5 cable in the external media level of the physical layer; the local switch in the external device level of the physical layer; and node 2 in the external device level of the physical layer. Figure 6 shows a view of the runtime layers with the components involved in an MPI_Send added to the view. Knowing the runtime components involved in an application's execution is useful because these are aspects of the runtime environment that can directly affect application performance. During the MPI_Send operation, there are a variety of components spread across all the layers which can affect performance. Additionally, application performance can be indirectly affected by other components of the runtime environment. It is important to make this distinction, as it is not possible to determine which components are indirect contributors to negative performance only by following the execution flow of control. For example, processes not associated with a running MPI application can negatively affect the application's performance if the processes share the same CPUs, filesystem, memory, or interconnect as the MPI processes.

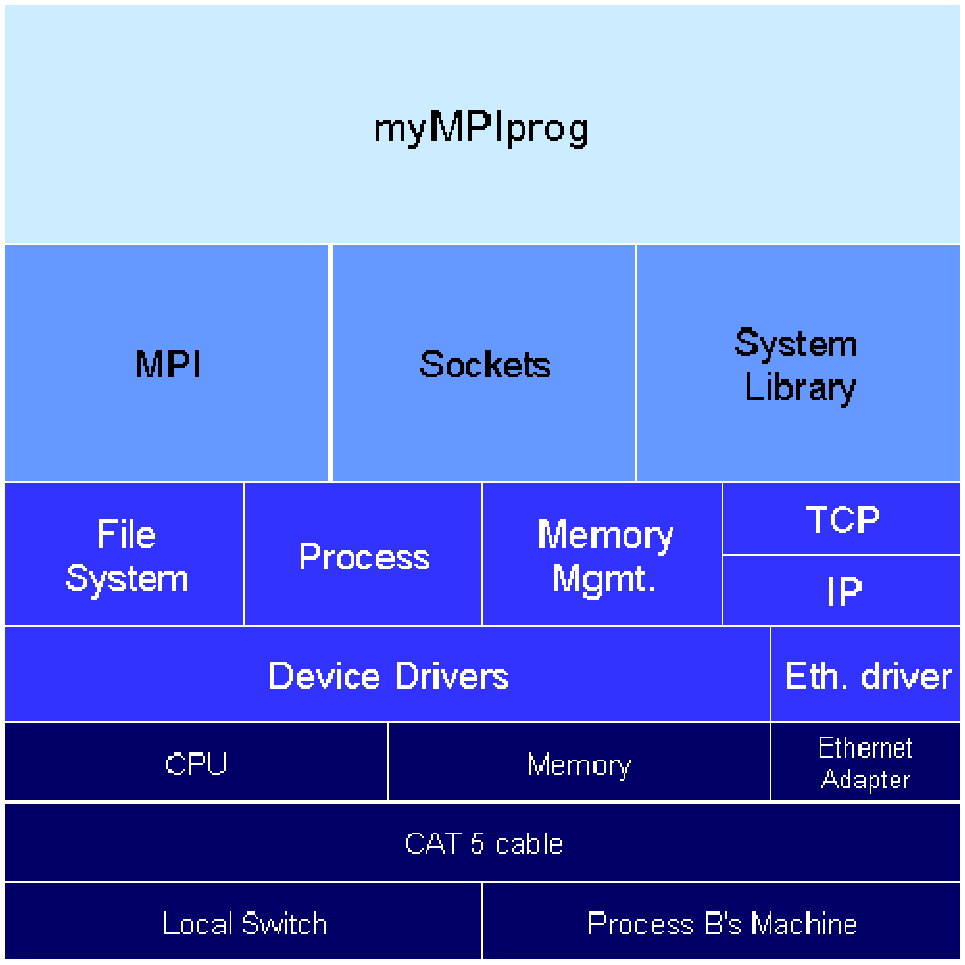


Figure 6: Example of Runtime Environment Components Involved in MPI_Send on an Ethernet based Cluster

3.1.7 Discussion

In this section we presented a layered view of the runtime environment. We described the four basic layers of the runtime environment: application, library, operating system, and physical. We discussed this view in terms of a way to categorize the root causes of performance problems and potential optimizations by layer. We also considered how this view can be used to identify those components which directly affect application performance and those components which indirectly affect application performance.

3.2 Important Issues for Environment-Aware Performance Tuning Approaches

Approaches for design and development of environment-aware analysis tools involve important considerations regarding time, the application, the runtime environment, measurement, analysis, and diagnosis.

3.2.1 Time

An important concern in all of distributed computing is determining the order of distributed events. Clocks among distributed systems are not always synchronized, so it may be impossible to order events by clock timestamps. However, it is possible to obtain partial orderings of events; and it is possible to obtain total orderings in some cases [42]. This same concern is present in environment-aware analysis. In particular, data collected with different tools must be mapped onto a common time scale to correlate activities at different layers of the system.

3.2.2 Application

Knowing what the application does during execution is an important issue for environment-aware approaches. Application analysis tools currently construct a good picture of what an application does during execution. This information is necessary for environment-aware approaches to use in determining why behavior happens, as it is difficult to ask why if there is no context for the question. In order to understand why a behavior occurs, more than just a description of an application's execution is required. Environment aware analysis enhances this description knowledge about what is happening in the rest of the runtime environment while an application executes.

3.2.3 Runtime Environment

Capturing a description of the status and behavior of the runtime environment and associating it to an application execution is an important concern of environment-aware analysis. To capture this description it will be important to consider what aspects of the runtime environment can be measured and what metrics will be most appropriate for enhancing understanding of an application's performance. Another consideration is deciding how to collect measurements. A final, but very important consideration is how to relate the description of the runtime environment to the description of an application execution.

3.2.4 Measurement

With regard to measurement, the measurement strategy is an important issue. It will be important to consider how data should be collected. It is important to choose data collection methods which will support creating a mapping between the description of the environment and the application based on the two sets of measurements.

3.2.5 Analysis

In the analysis phase, an area of consideration is how raw data are combined and calculated upon to compose derived measurements and determining if there are valid derived metrics composed of data from both application level measurement and runtime environment measurement. The analysis component is also concerned with determining what values, in both the application and the runtime environment data set, fall outside the bounds of an acceptable level of performance.

3.2.6 Diagnosis

The important issues involved in diagnosis involve determining if out of bounds application metrics correspond to out of bounds runtime environment metrics for a time slice of interest, and determining how likely it is that an application experienced performance problem is caused by conditions in the runtime environment, and not by conditions with in the application.

3.3 Discussion

We described environment-aware performance analysis and discussed issues related to environment-aware tool development. In the first part of the chapter we presented a layered view for describing the structure of a runtime environment. This view contains the application layer, the library layer, the operating system layer, and the physical layer. We defined each layer and described how this view can be used to frame performance problems in terms of root causes and potential optimizations. In the second section we discussed development considerations for the design of environment-aware tools. These include considerations regarding time, the application and the runtime environment, measurement, analysis, and diagnosis.

CHAPTER 4 PERFORMANCE ANALYSIS CASE STUDIES

This chapter presents case studies in which application performance was unexpectedly poor and diagnosing the causes of performance problems was difficult. In each of these situations, the analysts applied several iterations of performance testing before gaining a first key insight into what was causing the unexpected performance. We believe that each of these cases is an ideal candidate for the application of environment-aware diagnosis. Table 1 lists, for each case, the type of performance problem noticed and what the root cause of the problem was.

	Exhibited Performance Behavior	Root Cause
1.	Writing large buffers to disk incurred greater execution time than writing many small buffers to disk	Version of parallel file system
2-4.	Utilizing n of n processors per node achieved worse performance than using n-1 processors per node	System interference
5.	New machine reached only 50% of expected performance on a memory performance benchmark	Faulty hardware on system board
6.	Latencies for database transactions were inconsistent	Shared physical disk by two servers. Database users on one server experienced longer latencies when the other server was involved in I/O with the shared disk.

Table 1: Case Studies Organized by Exhibited Performance Behavior and Root Cause of Performance Problem

The cases presented here are examples of difficult diagnosis situations. This set of cases is not an exhaustive set, but it is sufficient for demonstrating the need for performance analysis methods to incorporate knowledge about the runtime environment.

4.1 OS and File System Upgrade

In this situation, a performance analysis researcher had constructed a series of experiments designed to study the overheads associated with tracing applications [52]. The study investigated trace buffer sizes and the time required to write trace buffers to disk.

The analyst selected the SMG2000 ASC Purple benchmark because it generates many MPI function calls. TAU 2.15.1 was used to trace function entries and exits. The experiments were conducted on the batch partition of MCR, an 1152 node Linux cluster at LLNL, running the CHAOS operating system, and configured to use Luster, a parallel file system. The application was configured with 4 MPI processes for all of the experiments and associated runs. For each buffer size of interest, three types of experiments were conducted: one where the application was not traced, one where the application was traced, but the buffers were not written to disk, and one where the application was traced and the trace buffers were written to disk.

In the first few experiment sets, it was discovered that the wall-clock time for the large buffer with write experiments was worse than wall-clock time for the small buffers with write experiments. This did not make sense. In general, fewer larger sized writes are more cost effective than many small sized writes. The experiments were re-run several times, but similar results were obtained each time.

After several rounds of investigation, over the course of several weeks, the analyst obtained unexpected results that showed the large buffer with write experiments performing better than the small buffer with write experiments. These results were perplexing because the analyst knew that the application code had not changed. The same executable and tracing configurations had been used in all of the experiments. Fortunately, the analyst had used PerfTrack, an experiment manage system, to automatically collect characteristic information about the execution environment each time the application was executed. In reviewing the attribute information associated with each execution, the analyst discovered that the previous experiments had run on CHAOS 3.0 and the new results were obtained from CHAOS 3.1. Further investigation into the details of the CHAOS upgrade revealed that the Lustre file system had also been upgraded. The analyst contacted the system administrators and was able to obtain operating system benchmark data for the CHAOS versions installed over the course of the experiments. With this data, the analyst ruled out the operating system as the cause of the unexpected performance and concluded that the file system was the most likely cause.

4.2. Inconsistent I/O Rate Service Time

This case study is an example of a problem that evidenced itself in a Unix server environment, but could also certainly happen in a high performance cluster environment [72]. The situation is summarized, in [72], as “a generic problem set faced by many Unix engineers.” The observed performance problem is that, at times, database transaction latencies on server A are longer than expected. In this scenario the file systems of server A and server B share some of the same physical disk resources. In the case presented, five physical disks had each been partitioned into two slices. Seven of the slices were allocated to server A and three slices were allocated to the file system of server B. However, one of the physical disks was shared by both server A and server B. A commercial tool, RICHPse SymbEL, was used to analyze “service time” measurements of I/O requests in context of the physical disks on which the requests occurred. Several additional scripts were written that enabled service time measurements to be recorded for the disk slices. This then lead to the discovery that Server B’s sporadic use of the disk slice on the shared disk was the cause for the inconsistent database transaction latencies noticed on Server A.

This problem could not have been thoroughly investigated using traditional application analysis tools, as the scope of the problem involved a machine external to the application’s execution.

4.3 System Interference Related to Variability in Application Performance

The three accounts presented in this section are very interesting because they each involve nearly identical problems, and the lengths to which the analysts go to track down the causes of the performance problems is extensive. All three cases deal with scaling fine-grained applications to run on thousands of nodes, while utilizing the full set of processors per node. The applications involved experienced poor performance when configured to use *n* processors per node, the full set of processors per node. In each case performance was actually better when *n-1* processors per node were configured, and with this configuration, the applications scaled to high numbers of processors.

4.3.1 ASCI Q

An interesting story of how operating system interference can affect application performance is presented in the published account of a performance study on ASCI Q, a 2048 node high performance computer at Los Alamos National Laboratory (LANL) [69]. At the time of ASCI Q's initial deployment, a performance study was designed to accurately determine the performance that should be expected when running a hydrodynamics code on the full ASCI Q system. This code had been run on other platforms, so there was a notion of expected performance based on previous experience with the code. Additionally, the ASCI Q hardware had undergone performance testing prior to the production phase. Using the expectations from previous experiences and the hardware measurement data, a prediction for the application's performance was made using a model that had been developed for the application. The model had been used successfully on other large-scale systems, so there was confidence in the model.

The first experiments of running the application on the full system performed much worse than the prediction. In these experiments, four MPI processes were assigned to each node, thus involving the full amount of processors per node. As nodes were added to the problem set, the performance continued to deteriorate. At 4096 processors, the time to completion of one application cycle was twice the predicted value. At first, the analysts considered that the model might be inaccurate. A series of tests were conducted to determine the accuracy of the model. The first important finding was that the model was accurate if one, two, or three processors per node were utilized. Each cycle in the application was configured to perform a consistent amount of work, and so the expectation was that each executed cycle would complete in the same amount of time. In analyzing the variability of cycle completions, the analysts discovered that there was great variability. Further inspection, breaking up the time spent in each cycle by function, revealed that as processors increased, the time spent in MPI_Allreduce and MPI_Reduce increased. This lead the team to consider the collective operations of the MPI implementation as factors of the poor performance. This resulted in several changes made to the MPI_Allreduce function, changing the way synchronization was handled. With this change, the application's performance was expected to improve by 78%. However, the difference in performance was negligible. Although the problem was not yet solved, the analysts were able to eliminate the MPI implementation as the problem source. The next suspicion was that system interference was causing the problem.

This seemed reasonable, as the noise would go largely undetected as long a free processor was available for system activities.

In several refinements of tests, using synthetic benchmarks, to isolate system noise, it was discovered that certain nodes in 32-node clusters experienced similar noise patterns. In every 32-node cluster, comprising ASCI Q, nodes 0, 1, and 31 showed consistently longer execution times than the other nodes. In a series of "techniques" that are not described in the accounting, the analysts were able to identify most of the system activities that were causing application processes to be switched out, and they were able to identify the delay imposed on the application due to the context switches. The system activities included two events generated by the Quadrics resource management system; a cluster management event; and parallel file system events.

Once they understood the problem, they were able to construct reasonable solutions. Through a combination of removing some of the daemons, co-scheduling some of the system processes to occur simultaneously, and distributing the processes differently, they were able to get reasonable performance from the application running on the full ASCI Q system.

4.3.2 IBM SP Machines and AIX Trace

A story very similar to the ASCI Q experience played out at Lawrence Livermore National Laboratory (LLNL) on the IBM SP machines [30]. The machines analyzed in this study were Blue, a 1120 processor machine, Frost, a 1088 processor machine, and White, a 8192 processor machine. Frost and White are each configured with 16 processors per node, and Blue is configured with four processors per node. This performance study was designed to investigate performance variability encountered in applications for which very little variability was expected. They expected little variability because the codes were deterministic in the work they did, the work was balanced among the available processors, and the jobs were scheduled to run on dedicated nodes.

The investigation into variability lead to the discovery that variability was minimized and within the expectation if not all of the processors per node were utilized by the application. At first, the analysts suspected the communications operations of the IBM MPI implementation. They constructed an experiment to test this idea. The experiment focused on the collective operation, MPI_Allreduce. The AIX *trace* facility was used to examine the behavior of the MPI_Allreduce test. As a way to improve repeatability of the test, the entire machine was run in dedicated mode; all *cron* jobs and monitoring activities that were not part of the experiment were turned off; and the traffic on the machine interconnect was mostly minimized to just the test traffic. Similar to the ASCI Q study, it was found that the MPI implementation was not a factor in the variability problems.

The test data revealed that MPI_Allreduce was susceptible to some variability, namely there were a few reduction operations in each test that were much longer than the others. These "long-lived" reductions were analyzed with the AIX trace facility.

This showed that one or more of the participating processes in each of the long reduction operations was delayed by operating system interference. They were then able to identify two kinds of interference. The first was interference caused by the AIX timer decrement interrupts. These happened at 100 Hz and were staggered across every CPU in a node. The second type of interference identified was system daemon scheduling. These interference activities were much longer in duration than the timer interrupts.

With further testing, they were able to assess the impact of both types of interference. These results showed that the daemon scheduling interference was the larger contributor to the variability experienced by the application.

4.3.3 NAMD

As a final example of the difficulties in detecting operating system interference as a factor in poor application performance we present the experience of scaling NAMD, a molecular dynamics code, on Lemieux, a 3000 processor cluster at the Pittsburgh Supercomputing Center (PSC) [70]. In this work, the goal was to achieve optimal performance of NAMD on Lemieux. The scientists describe several of the optimizations employed at the application and communication layers to achieve scaling performance similar to results achieved on ASCI Red. As in the previous two examples, the application scaled well, utilizing three of the four processors per node, but when the full set was attempted performance deteriorated. The analysts identified that the inability to utilize all of the processors was due to some operations taking much longer to complete than expected. They were able to establish that the majority of these “stretches” took place in send and receive communications operations. Since they achieved reasonable scaling at three processors per node, they did not further investigate the cause of the “stretches”. It is likely that father investigation would have yielded results showing what kinds of system interference were causing the problems.

However, in a later study, the scientists decided to pursue optimizations at the application and communications library layers [31]. By making several changes at these layers, the scientists were able to get the NAMD molecular dynamics code to scale to 3000 processors, utilizing the full set of processors per node. The application level optimization involved changing non-blocking receives to blocking receives. The library level optimizations involved changes to the Quadrics elan communications library.

4.4 Faulty Hardware

This situation was described by Dr. Douglas Pase, with IBM’s server performance team [64]. In this case study, actual performance was far below expected performance in benchmarking memory performance on a new server system. The cause of the poor performance was difficult to identify using a routine set of performance analysis methods.

The goal of the performance study was to obtain memory performance measurements for a new machine, the IBM eServer 325. The analysis team used STREAM, a well known memory performance benchmark [45]. The STREAM benchmark measures sustainable memory bandwidth (in MB/s). For the eServer 325 tests, the OpenMP threaded version of STREAM was used.

The eServer 325 contained two 2.0 GHz single-core AMD Opteron processors and six GB of DDR333 memory. In the BIOS, SRAT, DRAM Interleave, ECC and ChipKill were enabled, while Node Interleave was disabled. The server was not connected to a network and it was tested as a stand-alone system. This was a new machine, and the STREAM benchmark had not yet been tested on it.

The actual performances for the STREAM tests were less than 50% of the calculated theoretical performance. The team had calculated peak memory bandwidth to be 10.6 GB/s. The team had prior experience using this benchmark, and they knew that typically, a system achieves 75-85% of the theoretical performance.

The analysts conducted a series of tests to investigate the unexpected behavior. One of the things they discovered was that single threaded configurations performed as expected. However, any configuration with more than one thread obtained poor memory performance. The analysts checked the BIOS settings looking for configuration settings that didn’t make sense; they inspected the BIOS logs; and they swapped out memory, thinking bad memory could be the cause. None of these checks proved to be causing the poor performance.

Eventually, the team realized that the system was experiencing a very high number of interrupts. These were on the order of 10,000 to 100,000 interrupts per second, and the rate was high regardless of the benchmark executing or not. Once the high number of interrupts was discovered, the team was able to more precisely direct their investigation. They quickly discovered that the majority of the interrupts were related to a USB device. In discussion with the developers of the system, it was discovered that the mother board had a minor design flaw. The developers were able to easily resolve the problem.

4.5 Discussion

In this chapter we presented several case studies which illustrate situations which would have benefited from environment-aware performance analysis. Each of these cases is an example of difficult diagnosis. The cases demonstrate the need for additional information about the runtime environment to be included in performance analysis. We also see common themes shared among these situations. It was often the case that the analysts conducted several iterations of performance tuning before reaching a key insight point. These cases also showed that finding good solutions required learning what the root causes of the problems were. In the cases where tools were used, the diagnoses offered by the tools were not sufficient for determining what to target for optimization. These studies support our claim that the quality of diagnosis in parallel performance tools can be improved by incorporating knowledge about the runtime environment.

CHAPTER 5 A TEST SUITE FOR PERFORMANCE DIAGNOSIS

In this chapter we present a test suite that can be use to determine how well application performance analysis tools determine causes of poor application performance. Ideally, a tool should direct the user to the layer of the runtime environment in which the root cause of a performance problem exists, and identify the specific conditions within the layer that are most likely causing the performance problem. Each test simulates or constructs a condition existing in a particular layer that will negatively affect application performance. The results are interpreted by comparing the diagnosis of the evaluated tool to the real diagnosis. To fully pass a test, a tool must correctly identify the root cause layer and correctly identify at least one condition within the layer responsible for the performance problem. We envision the test suite to be of value to developers of environment-aware performance tools and we anticipate that this type of test suite will be most useful in the first prototype implementations of environment-aware parallel performance tools.

For each test, we present a brief description of what the test targets; describe how to construct the experiment; explain how to interpret the results; and state what application performance analysis tools, without knowledge of the entire runtime environment, would generally report about the performance behavior.

5.1 Application Layer: Naive Broadcast

Test Overview. The Naive Broadcast test is designed to verify that performance problems originating in the application layer can be correctly diagnosed by commonly used tools. For these kinds of problems it is possible to optimize the application performance by directly modifying the application source code, but we want a tool to inform us that this is the appropriate layer to investigate.

Experimental Setup. The application focus of this experiment is an MPI application with this communication pattern: the one sending process uses a point to point communications operation to send the same message content as separate messages to every other participating MPI process, rather than using the collective operation MPI_Bcast.

Interpretation of Results. To fully pass this test, the tool must identify the application layer as the layer containing the root cause and identify that the MPI_Send could potentially be replaced by MPI_Bcast.

Diagnosis without Environment-Aware Analysis. Application level tools indicate that one process does a majority of the work in many MPI send calls, and other processes are spending time waiting to receive messages. Although the detection of where the application is spending most of its time is correct, tools do not point to the important aspect of the MPI application: that the MPI broadcast function should be used instead of the individual MPI_Send calls.

5.2 Library Layer: MPI Library Bottleneck

Test Overview. The MPI Library Bottleneck test is designed to verify that performance problems originating in the Library Layer can be correctly diagnosed by commonly used tools.

Experimental Setup. In this test, we take advantage of the MPI profiling interface. In this test we consider the case of poorly implemented MPI collective operations. We use an open source MPI profiling library and modify the MPI_Allreduce function so that it wastes some time before calling the real collective operation in the MPI library. We link the modified profiling library to an MPI application that uses the MPI_Allreduce operation.

Interpretation of Results. To fully pass this test, the tool must identify the library layer as the layer containing the root cause and identify that the MPI_Allreduce function is a costly function.

Diagnosis without Environment-Aware Analysis. Application level tools identify the user level call made to MPI_Allreduce as a bottleneck. In instances where the library has been compiled with debugging options, some application level tools are able to traverse to this level and identify the collective operation in the library layer as a bottleneck.

5.3 Operating System Layer: NFS Server Experiment

Test Overview. The NFS Server Test is designed to verify that performance problems in the Operating Systems Layer can be correctly diagnosed by commonly used application level tools. This experiment creates a scenario where an NFS server receives a high number of requests for the same file from various clients. One of the clients happens to be an MPI application that needs to read the file. Other clients, not associated with the MPI application, are also trying to access the file for writing. The experiment investigates the affects of this situation on the MPI application's performance and looks into how well application analysis tools guide an analyst in understanding that application performance is negatively affected by the high demand on the file by unrelated processes.

Experimental Setup. In this test one machine that is separate from the application's cluster is designated as the NFS server. The MPI application is configured to runs on one node in the cluster, and it does a series of file reads on a single file. Two other nodes, on which the MPI application is not running, have processes writing to the same file that the MPI application is trying to read.

Interpretation of Results. To fully pass this test, the tool must identify the operating system layer as the layer containing the root cause and identify that the file system has several processes trying to access the same file.

Diagnosis without Environment-Aware Analysis. Application performance analysis tools show that the MPI application performs poorly in the application level calls to read the file. Without additional information from the runtime environment, these tools are unable to indicate that the likely cause of the poor performance is contention for the file resource by processes running on other machines and not part of the MPI application.

5.4 Physical Layer: Memory Difference

Test Overview. The Memory Difference Test is designed to verify that performance problems in the Physical Layer can be correctly diagnosed by commonly used application analysis tools. In this example, an MPI application's performance is negatively affected by a node that has less memory than the other nodes on which the application's MPI processes execute.

Experimental Setup. This experiment constructs a physical environment where one node in a cluster has less memory than the other nodes. Other than the memory difference, the nodes are homogeneous. The MPI application is configured to run on a set of nodes that includes the node with less memory, and one process is assigned to each node. The application is constructed so that every MPI process does the same set of memory intensive tasks. The process assigned to the node with less memory takes longer to complete its work than the other processes.

Interpretation of Results. To fully pass this test, the tool must identify the physical layer as the layer containing the root cause and identify performance is constrained by the size of memory available on one of the participating nodes.

Diagnosis without Environment-Aware Analysis. Application level tools identify the process that takes more time than the other participants to complete its work. The tools also identify where in the application this slowness is attributed. The tools do not indicate that a machine configuration difference is what causes one process to work longer than the others.

5.5 Physical Layer: Interconnect Test

Test Overview. The Interconnect Test is designed to verify that performance problems in the Physical Layer can be correctly diagnosed by commonly used tools. This experiment was designed to consider the interconnect component of a cluster from the perspective of available network bandwidth.

Experimental Setup. The experiment is constructed to examine the performance of a communication intensive MPI application when available bandwidth is far below the theoretical capacity. In this test a very large message is passed back and forth between two processes using the blocking MPI_Send and MPI_Recv communication methods. This experiment is configured to run on two nodes with one process per node. Available network bandwidth for the participating processes is reduced by generating extra traffic on the links between the two nodes.

Interpretation of Results. To fully pass this test, the tool must identify the physical layer as the layer containing the root cause and identify that the media for communication is constrained.

Diagnosis without Environment-Aware Analysis. Application performance analysis tools identify the MPI send and receive routines in the application as the locations of slower performance in this reduced bandwidth setting. Without additional information about the availability of network bandwidth, tools cannot correctly identify the cause of this problem.

5.6 Discussion

This test suite that can be used to evaluate how well application performance analysis tools identify the causes of poor application performance. These tests construct conditions at different layers of the runtime environment that will negatively impact the performance of an application. We described the goals of each test; explained how to construct the experiment; gave an overview of what constitutes passing a test; and stated what would generally be reported by application performance analysis tools that do not have access to additional knowledge about the runtime environment.

CHAPTER 6 EXPERIMENTAL RESULTS

In this section, we describe our experimental methods and results. The general design of each experiment is similar. With these experiments we want to show that application performance analysis tools do not adequately incorporate information about the runtime environment and we want to demonstrate that when status information about the runtime environment is used in performance analysis, there is great potential to accurately diagnose the causes of application performance problems. To do this we construct example situations, in which we have two environments: a normal environment that can be used as a baseline for application performance; and an environment where some aspect of the normal environment has been changed in order to dramatically affect application performance. We call this second environment the sub-optimal environment. The general approach taken in all of the experiments was to evaluate an application's performance in both environments using one or more application performance analysis tools, compare the results given by the tools between the two environments, and then supplement the tools' results with additional information collected about an execution's runtime environment to demonstrate the value added to the overall problem diagnosis.

We chose to construct the sub-optimal environment for several reasons. First, since our goal is to demonstrate our hypothesis through well-designed experiments, we need our experiments to be repeatable and we need to obtain results that are comparable. It is much easier to achieve repeatability and comparability if we have tight control over the experiment environment. Secondly, the construction of the sub-optimal environment suits the needs of the study, in which we want to show specific examples of performance analysis tools unable to capture how the runtime environment contributes to an application's performance behavior.

The goals of these experiments are two-fold. We first want to show that difficult diagnosis scenarios likely to go undetected by application performance analysis tools. Secondly, by building up this experiment set with several concrete examples where we have injected additional information about the runtime environment, we are able to support the claim that environment-aware diagnosis is a necessary component for automated performance analysis

6.1 Available network bandwidth for Inter-node communication

This experiment was designed to consider the interconnect component of a cluster from the perspective of available network bandwidth. In this experiment we show that overall performance of an MPI application is affected by actual capacity of the inter-node communication links, but that it is difficult to assess the contribution of available bandwidth on application performance using only application performance analysis tools. In the normal environment for this experiment available bandwidth was consistently high, near the theoretical peak, and in the sub-optimal environment, the available bandwidth for the communicating processes was severely reduced. The application performance analysis tools identified the MPI send and receive communication routines in the application as the locations of slower performance in the reduced bandwidth setting. However, without additional information it was impossible to determine why these application level routines were so much slower in one setting than in the other. When we supplemented the results provided by the application performance tools with data collected about available bandwidth, it became clear, from a user perspective, that the poor performance observed in the sub-optimal environment was related to availability of bandwidth during the application's execution.

The experiment was constructed to examine the performance of big-message, a communication intensive MPI micro-benchmark from the PPerfMark 2.0 benchmark suite [71], when available bandwidth was close to the theoretical peak and when available bandwidth was much less than the theoretical peak. In this test, a bottleneck in the system is created by passing very large messages back and forth between two processes. The actual size of the message is 100,000 bytes. The message is sent and received between two processes for a specified number of iterations, using MPI_Send and MPI_Recv, which are blocking communication routines. In this experiment set, we used 1,000 iterations so that the wall-clock time in the normal environment was at least 1 minute. All of the executions in this experiment were configured to run on two nodes with one process per node. A description of the static structure of the main loop in big-message is shown in Figure 7. The Gsend_message and Grecv_message calls are wrapper functions to MPI_Send and MPI_Recv.

```
CALL MPI_Wtime()
IF rank == 0
   LOOP until message[] is filled
      Assign ‘1’ to message[i]
   END LOOP
   LOOP ITERATIONS times
      CALL Gsend_message
      CALL Grecv_message
   END LOOP
ELSE
   LOOP ITERATIONS times
      CALL Grecv_message
      CALL Gsend_message
   END LOOP
END IF
   CALL MPI_Wtime()
```

Figure 7: Static structure of main loop in big-message

We analyzed application performance with mpiP 2.8.2 [56], a MPI profiling tool, and with Paradyn 4.2.1 [63, 62], a dynamic instrumentation and online performance analysis tool. We viewed the mpiP results with mpipview, a graphical viewer provided by Tool Gear 1.30 [84]. Additionally, the micro benchmark included several source code level measurement components. Elapsed wall-clock time for the entire program for each process is measured using MPI_Wtime which has microsecond resolution on the systems used in this experiment. The micro benchmark also records user process time and system process time for the entire program for each process using `getrusage()`.

The bandwidth measurements were collected using NetPIPE 3.6.2 [80,58]. NetPIPE is a suite of applications aimed at gathering throughput and latency measurements of the underlying network. NetPIPE's TCP and MPI modules were used to collect throughput and latency measurements between the two nodes on which big-message was executed. Each module runs a series of ping pong tests between two nodes, in which the number of bytes sent is increased in each successive test.

This experiment utilized the LAM/MPI 7.0.6 [41] implementation of MPI. The experiments were conducted in the computer science Linux Laboratory at Portland State University on homogeneous single processor Linux nodes, running kernel version 2.6.9-22.0.2.EL. Each node was configured with 786 MB of memory, one network interface card capable of 100 Mbps transmission, and a 1400MHz Intel Pentium 4 processor with 256 KB cache. The laboratory network environment is 100 megabit Ethernet, and every node is directly connected to the switch serving the laboratory.

To construct the sub-optimal environment we reduced the bandwidth available to big-message's communicating processes by generating traffic on the subnet using Iperf 1.7.0 [29]. Iperf is a research tool designed to measure TCP bandwidth. In our use, we did not analyze the measurements collected by the tool, but rather used the tool as a means to create network traffic between clients and a respective server. We configured four of the laboratory's nodes to be servers and then used all of the remaining available nodes as clients, including the two nodes that were involved in the big-message executions.

To understand what the baseline environment for bandwidth capacity was for the laboratory configuration we measured TCP and LAM/MPI throughput and latency using NetPIPE. Since, these experiment were conducted in a general purpose instructional laboratory, we found that the most consistent normal environment existed at non-peak hours, generally between 2:00am and 8:00am. Therefore, all of the tests for the bandwidth experiments were conducted during these hours. Three representative trials from the TCP measurements for the normal environment are presented in Figure 8. These graphs show that as message sizes increased, the throughput measurements approached the theoretical peak of 100 Mbps (Mega bits per second). Latency is less than 0.1 seconds for message sizes 1 MB and smaller. The largest message tested was 8 MB and the incurred latency was 0.714 seconds. Figure 9 shows TCP measurements, using the same representative runs as earlier, and three representative runs from the LAM/MPI measurements. In these measurements, LAM does not add significant overhead to TCP. After gathering the baseline bandwidth measurements for the laboratory, we ran ten trials of the big-message application on the machines paulus and jess. Each trial was configured to utilize two nodes with one process per node and to execute the MPI message loop for 1,000 iterations. In each trial the master process executed on paulus. The summary wall-clock results for these trials are presented in Table 2. We then analyzed the performance of the big-message application in the normal environment with mpiP and Paradyn. With Tool Gear we viewed the results of the profiled big-message benchmark. Figure 10 is a screen shot of the Tool Gear display. Here, we see that 58.89 % of the time spent in all MPI profiled functions was assigned to the MPI_Recv function and 41.11% of MPI time was assigned to the MPI_Send function. The screen shot shows the details mpiP calculated for MPI_Recv in big-message. We see that the function was called 2,000 times, which is because each process called it 1,000 times. The counts presented for each process make sense. The maximum, mean, and minimum timing values for MPI_Recv are presented for each process and for both processes combined. Tool Gear also shows the location in the source code of the MPI_Recv call.

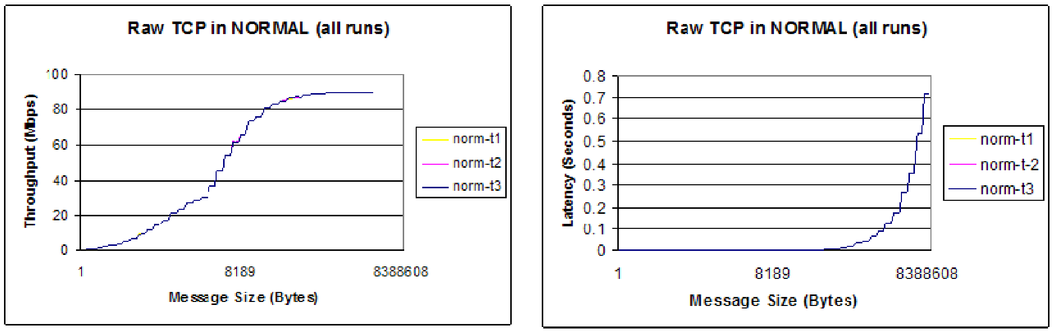


Figure 8: TCP Throughput and Latency in the Normal Environment

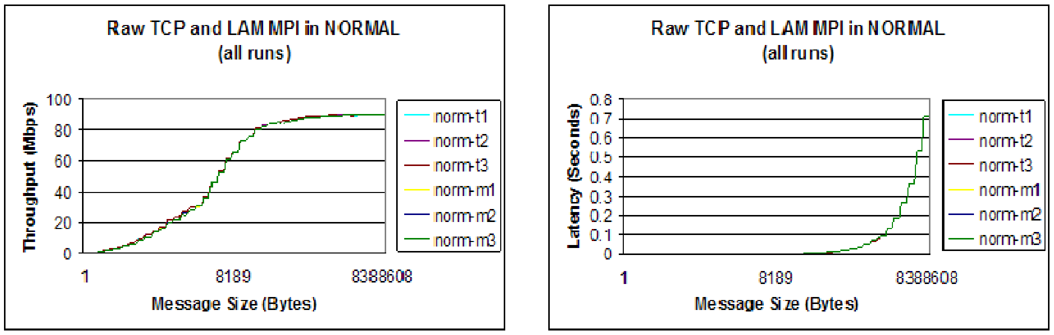
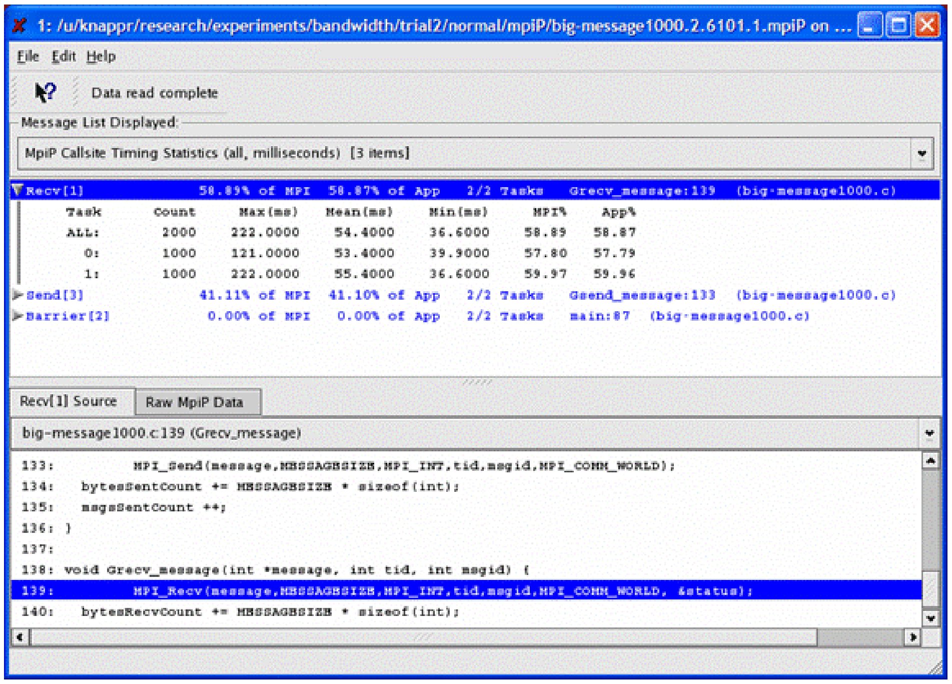


Figure 9: TCP and MPI Throughput and Latency in the Normal Environment

	paulus	jess
Min	68.912230	68.904881
Max	69.623708	69.616357
Med	69.233780	69.260735
Avg	69.254693	69.273031

Table 2: Summary Statistics. Summary wall-clock statistics, reported in seconds, for 10 trials of big-message in the normal environment


1: /u/knappr/research/experiments/bandwidth/trial2/normal/mpiP/big-message1000.2.6101.1.mpiP on ...
File Edit Help
Data read complete
Message List Displayed:
Recv[1] Source
Raw MpiP Data

MpiP Callsite Timing Statistics (all, milliseconds) [3 items]

```
Recv[1]             58.89% of MPI   58.87% of App   2/2 Tasks   Grecv_message:139   (big-message1000.c)
       Task      Count      Max(ms)     Mean(ms)     Min(ms)      MPI%     App%
       ALL:       2000     222.0000     54.4000      36.6000      58.89    58.87
         0:       1000     121.0000     53.4000      39.9000      57.80    57.79
         1:       1000     222.0000     55.4000      36.6000      59.97    59.96
Send[3]             41.11% of MPI   41.10% of App   2/2 Tasks   Gsend_message:133   (big-message1000.c)
Barrier[2]           0.00% of MPI    0.00% of App   2/2 Tasks   main:87   (big-message1000.c)
```

big-message1000.c:139 (Grecv_message)

```
133:         MPI_Send(message,MESSAGESIZE,MPI_INT,tid,msgid,MPI_COMM_WORLD);
134:   bytesSentCount += MESSAGESIZE * sizeof(int);
135:   msgsSentCount ++;
136: }
137:
138: void Grecv_message(int *message, int tid, int msgid) {
139:         MPI_Recv(message,MESSAGESIZE,MPI_INT,tid,msgid,MPI_COMM_WORLD, &status);
140:   bytesRecvCount += MESSAGESIZE * sizeof(int);
```

Figure 10: Tool Gear 1.30 screen shot of big-message in the normal environment

We also used Paradyn 4.2.1 to dynamically instrument big-message, and to view, as the program executed, the results of the automated performance bottleneck search and detection. To use Paradyn effectively, we needed big-message to run for a longer period of time. The benchmark source was modified slightly, to enable the iteration count to be infinite. The automated bottleneck search and detection component of Paradyn is presented to the user as the Search History Graph. The Search History Graph results for the modified big-message in the normal environment are presented in Figure 11. This shows that ExcessiveSyncWaitingTime is the problem hypothesis which tested *true* for being a bottleneck. Paradyn uses a color coding scheme to denote this. In this scheme a blue background indicates that the hypothesis has evaluated to true. A pink background indicates that the hypothesis evaluated to false. The bottlenecks are big-message's wrapper functions around MPI_Send and MPI_Recv. We also configured a table visualization to present the results of a few metrics of interest, as shown in Figure 12. This table shows the number of bytes sent and received; the total CPU time, including the time of all called procedures; and the total time, including the time of called procedures, spent waiting on synchronization operations.

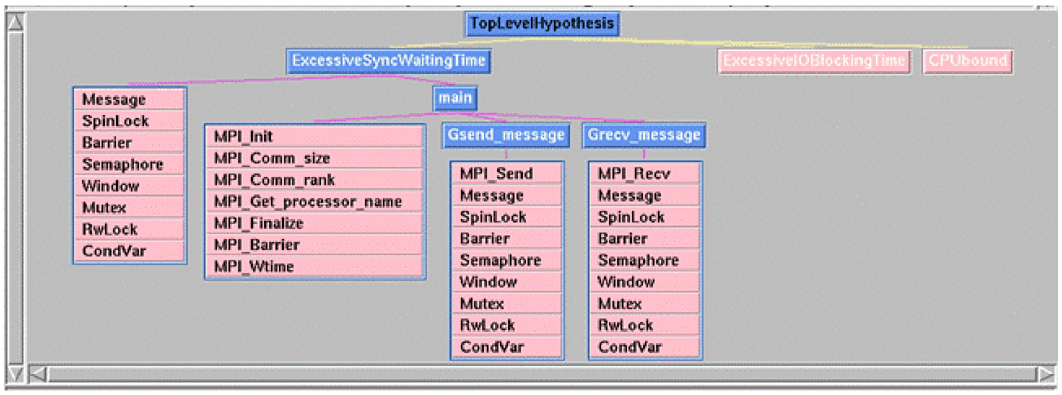
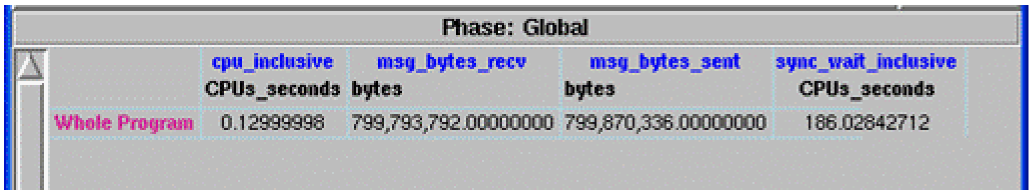


Figure 11: Paradyn 4.2.2 screen shot of Search History Graph of big-message in the normal environment



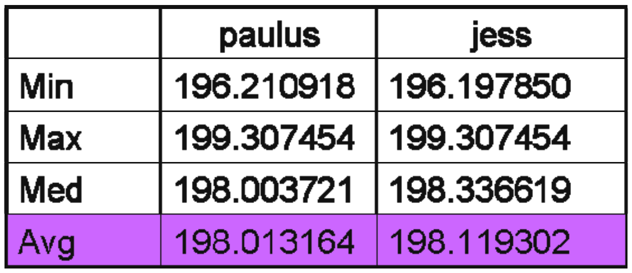
Phase: Global

	cpu_inclusive	msg_bytes_recv	msg_bytes_sent	sync_wait_inclusive
	CPUs_seconds	bytes	bytes	CPUs_seconds
Whole Program	0.12999998	799,793,792.00000000	799,870,336.00000000	186.02842712

Figure 12: Paradyn 4.2.2 screen shot Table Visualization of big-message in the normal environment

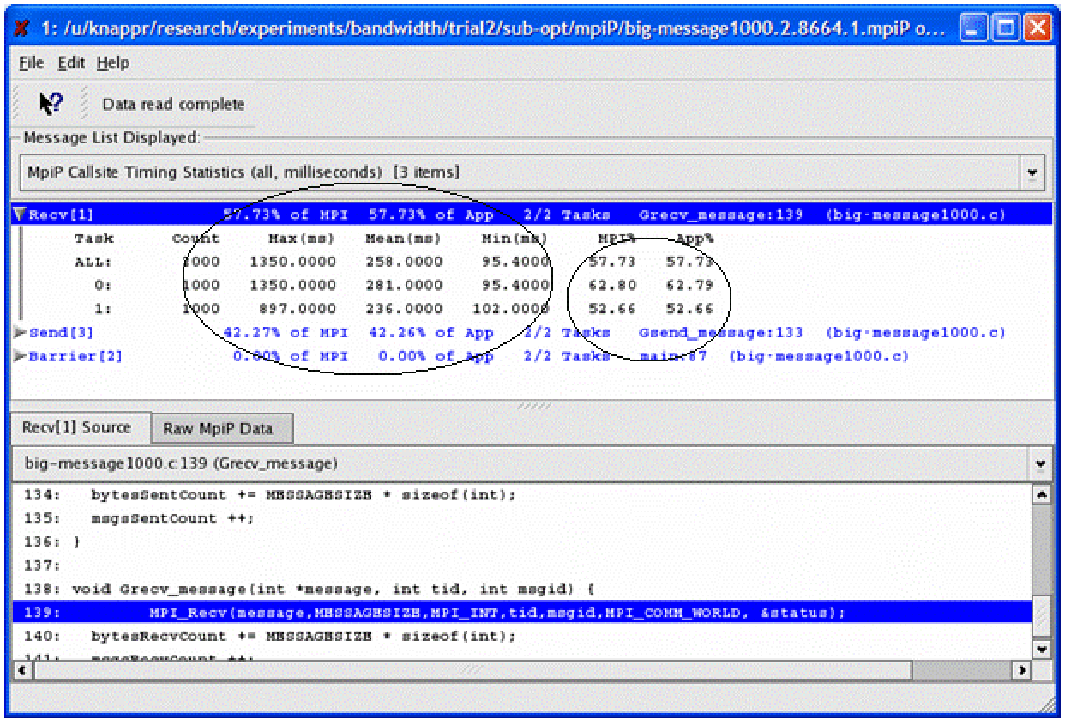
The previous results are all reported about the normal environment. Now we present the results of running big-message in the sub-optimal environment. We present the same set of data: the summary statistics from big-message's self instrumented wall-clock timing in Table 3; the profile from mpiP and Tool Gear in Figure 13; and the Search History Graph and table visualization from Paradyn in Figure 14 and Figure 15. We see readily that it took much longer to execute 1,000 iterations. This is evident from the summary statistics shown in Table 3.

From the Tool Gear screen shot, we see that the maximum, minimum, and mean times for the MPI_Recv function are much larger than before. The percentage breakdown of MPI work assigned to functions has not changed too much. We do not expect this to change since the program repeats the same steps in each execution. The Paradyn collected data in the sub-optimal environment also has changed. We can see that MPI_Send and MPI_Recv evaluated to true for being potential bottlenecks. Whereas, in the normal environment results, the wrapper functions tested true, but MPI_Send and MPI_Recv tested false. Since MPI_Send and MPI_Recv tested true in the changed environment, their associated resource trees are expanded out. The green background for the sub trees of MPI_Send and MPI_Recv indicates that the truth value of the hypothesis is unknown for theses resources..



	paulus	jess
Min	196.210918	196.197850
Max	199.307454	199.307454
Med	198.003721	198.336619
Avg	198.013164	198.119302

Table 3: Summary Statistics. Summary wall-clock statistics, reported in seconds, for 6 trials of big-message in the sub-optimal environment


1: /u/knappr/research/experiments/bandwidth/trial2/sub-opt/mpiP/big-message1000.2.8664.1.mpiP o...
File Edit Help
Data read complete
Message List Displayed:
MpiP Callsite Timing Statistics (all, milliseconds) [3 items]
Recv[1] Source
Raw MpiP Data
big-message1000.c:139 (Grecv_message)

```
Recv[1]          57.73% of MPI   57.73% of App   2/2 Tasks   Grecv_message:139   (big-message1000.c)
      Task      Count     Max(ms)     Mean(ms)     Min(ms)      MPI%     App%
      ALL:       2000   1350.0000     258.0000     95.4000     57.73    57.73
        0:       1000   1350.0000     281.0000     95.4000     62.80    62.79
        1:       1000    897.0000     236.0000    102.0000     52.66    52.66
Send[3]          42.27% of MPI   42.26% of App   2/2 Tasks   Gsend_message:133   (big-message1000.c)
Barrier[2]        0.00% of MPI    0.00% of App   2/2 Tasks   main:87   (big-message1000.c)
```

```
134:    bytesSentCount += MESSAGESIZE * sizeof(int);
135:    msgsSentCount ++;
136: }
137:
138: void Grecv_message(int *message, int tid, int msgid) {
139:          MPI_Recv(message,MESSAGESIZE,MPI_INT,tid,msgid,MPI_COMM_WORLD, &status);
140:    bytesRecvCount += MESSAGESIZE * sizeof(int);
```

Figure 13: Tool Gear 1.30 screen shot of big-message in the sub-optimal environment

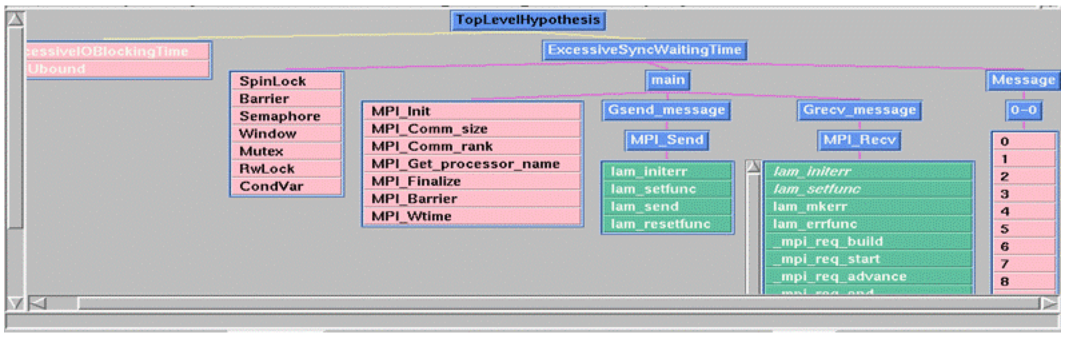
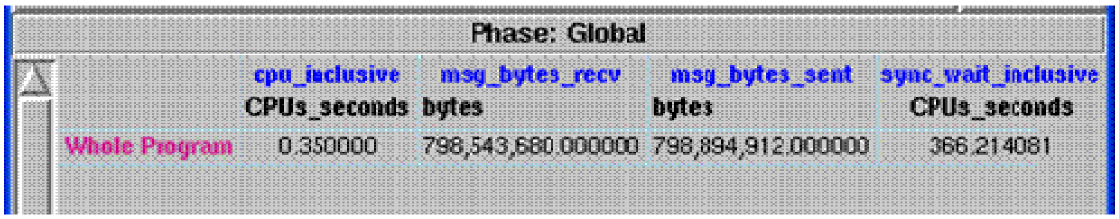


Figure 14: Screen shot of Search History Graph of big-message in the sub-optimal environment

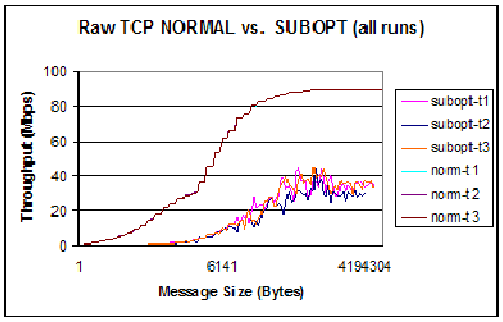


Phase: Global

	cpu_inclusive	msg_bytes_recv	msg_bytes_sent	sync_wait_inclusive
	CPUs_seconds	bytes	bytes	CPUs_seconds
Whole Program	0.350000	798,543,680.000000	798,894,912.000000	366.214081

Figure 15: Paradyn 4.2.2 screen shot Table Visualization of big-message in the sub-optimal environment

In this experiment set, we examined the performance of an MPI benchmark application in two environments. In the analysis from the runs in the normal environment, we are able to identify that the MPI_Recv and MPI_Send functions are where performance bottlenecks could potentially exist. When the application was executed in the sub-optimal environment, it was obvious from wall clock time that something was different between the two execution environments. The analysis tools were helpful in showing us where the performance problem existed in relation to the source code. We were able to ascertain from both mpiP and Paradyn that MPI_Recv and MPI_Send were the functions executing the largest percentage of time. Paradyn provided more detailed information than mpiP about what the performance problem really was. In both the normal and suboptimal environments the hypothesis test for ExcessiveSyncWaitingTime tests to true, indicating that the default threshold for synchronization waiting time had been exceeded by the total synchronization waiting time of the executing application. We are not able to determine from the analysis results presented what would be the best course of action to pursue to get the application’s wall clock time back to what had been experienced in the first set of runs. An analyst is likely to suspect the MPI library implementation or possibly the application itself as potential targets for optimization. To be able to make a decision about what course of action to take, the analyst really needs some other information brought into the description of the application's performance. To illustrate this point, we have supplemented the analysis information provided by the tools with information collected about the available network bandwidth between the communicating processes of the big-message application. These results are shown in Figure 16. The graphs present TCP throughput and latency measurements that occurred in the normal environment and in the suboptimal environment. It is clear that bandwidth throughput became constrained by more than 50% and that latency was severely impacted. With access to this piece of supplemental information, an analyst would know that the application's available bandwidth had been reduced. Provided that the analyst knows that nothing had changed in the MPI implementations and the application executable between the executions in the normal environment and the executions in the suboptimal environment, the analyst can safely eliminate both the application and the MPI library as possible sources of the problem.



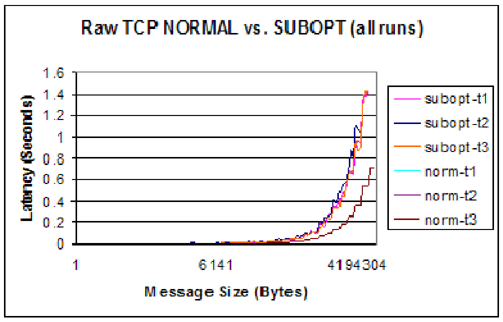


Figure 16: TCP throughput and latency in the normal and the sub-optimal environments

6.2 Gathering system info with MPI processes

6.2.1 Overview

In this experiment we explored the idea of using MPI processes to track system information. The idea is that an MPI application could have one process per node reserved just for system monitoring activities. In this scheme, the monitor processes only do monitoring activities, while the worker processes do the work of the application. In this experiment, all MPI processes are specified as arguments to program which launches MPI executions. It is possible to create the monitor processes dynamically, but this is not a fully supported feature of all MPI implementations.

There are several advantages gained by allowing the application to perform the monitoring tasks. One of the advantages is that the gathered information correlates with the application's execution. The application starts and stops the monitoring activity. Another advantage is that the application controls what is monitored and with what frequency data is collected.

There are several drawbacks to this method. In systems composed of single CPU nodes, the cost of running two MPI processes on a node could negatively impact application performance.

6.2.2 Purpose

The purpose of this experiment is to determine if a mapping between application performance and system activity can be done using generally available methods. In the results presented we execute an MPI monitor program. The program is constructed such that the worker tasks do the targeted work of the application and the monitor process collects some system level information and writes this data to a log file. In our test program, the monitor processes execute the *vmstat* system level utility.

6.2.3 Methods

In the *vmstat* monitor application, the monitor runs the *vmstat* utility at one second intervals for a specified duration. The worker processes do a simple loop computation and do not perform any MPI communication operations. By collecting *vmstat* data, we want to observe the application's performance in view of virtual memory activities. The *vmstat* utility reports information on processes, memory, paging, block IO, traps, and cpu activity. In this experiment we configured vmstat to update at one second intervals, and we configured it to produce the basic vmstat report. The kinds of things reported in each update, in the basic report, include the number of processes waiting, the number of processes in uninterruptible sleep, the amount of virtual memory used, the amount of idle memory, amount of memory swapped in from disk and out to disk, blocks received and sent to block I/O devices, the number of interrupts and context switches, and total CPU time percentages.

We executed the experiment in two environments. The first environment was considered the normal environment, and the second environment was the sub-optimal environment. For executions in the normal environment, the wall-clock times for the worker processes in each execution were nearly identical. In the sub-optimal environment, the wall-clock times for the worker tasks on one of the nodes was consistently longer over several executions than the corresponding value for the worker process on the other node. This experiment utilized the LAM/MPI 7.0.6 [41] implementation of MPI. The tests were conducted in the computer science Linux Laboratory at Portland State University on homogeneous single processor Linux nodes, running kernel version 2.6.9-34.0.1.ELsmp. Each node was configured with 1 GB of memory, one network interface card capable of 100 Mbps transmission, and a 3.20GHz Intel Pentium 4 processor with 1 MB size cache. The laboratory network environment is 100 megabit Ethernet, and every node is directly connected to the switch serving the laboratory.

6.2.4 Results

From the application timing results, we could see that the worker process on one of the nodes was taking longer to complete its simple loop computation. This is unexpected behavior for the application for three reasons. First, the application is deterministic. Second, the nodes selected for execution were homogeneous. Third, from previous experiences of running the application on the two nodes, we knew that the worker processes normally completed their tasks at about the same time.

Table 4 presents the wall-clock times, in seconds, for the worker process on each node for a representative execution from the normal environment. Table 5 presents similar metrics for the sub-optimal environment. We see that the worker process on andrea takes about 11 seconds longer to complete than it did in the normal environment, resulting in a 10% slowdown for the simple loop computation of the application.

	paulus	andrea
Average	113.516906	113.578591

Table 4: Wall Clock Times. Wall-Clock times, in seconds, for worker process on each node for two different executions, in the normal environment.

	paulus	andrea
Average	113.527086	124.952760

Table 5: Wall Clock Times. Wall-Clock times, in seconds, for worker process on each node for two different executions, in the normal environment

To investigate the problem further, we inspected the monitoring information that was gathered during these representative executions. Table 6 and Table 7 present the average values from the vmstat reports for the normal and sub-optimal environments respectively. Table 8 provides an explanation of the *vmstat* measurements and units.

	r	free	buf	cache	bo	in	cs	us	sy	id
paulus normal	1	542506	72080	213400	5	1015	37	49	0	51
andrea normal	1	528829	67081	238159	4	1024	42	48	0	52

Table 6: Vmstat reported values. Average values, by node, from vmstat for two different executions in normal environment

	r	free	buf	cache	bo	in	cs	us	sy	id
paulus sub-opt	1	543683	72410	208130	5	1012	40	49	0	51
andrea sub-opt	2	522449	68171	239669	4	9181	16346	47	6	47

Table 7: Vmstat reported values. Average values, by node, from vmstat for two different executions in sub-optimal environment

r	Number of processes waiting to run
free	Amount of idle memory (*KB*)
buf	Amount of memory used as buffers (*KB*)
cache	Amount of memory used as cache (*KB*)
bo	Blocks sent to a block device (*blocks/s*)
in	Number of interrupts, including the clock (*count/s*)
cs	Number of context switches (*count/s*)
us	Time spent running non-kernel code (*percent of total cpu time*)
sy	Time spent running kernel code (*percent of total cpu time*)
id	Time spent idle (*percent of total cpu time*)

Table 8: Explanation of Vmstat measurements and units

By inspection of Table 6 and Table 7, we noted some values that looked different enough to warrant further inspection. We can see that in the sub-optimal environment andrea, on average, has two process waiting to run. Normally, only one process is waiting to run. We also notice an increase in blocks sent to I/O devices for andrea, in the sub-optimal environment. Two dramatic changes are noticed in the number of interrupts and context switches logged on andrea during the application's execution in the sub-optimal environment. It is also evident that the break down of CPU time percentages for andrea in the sub-optimal environment changed: the percentage of time spent executing user code decreased, the percentage of time spent executing kernel code increased, and the percentage of time spent idle decreased. Just from the information presented in the averages table, we can theorize with a high degree of confidence that the addition of a second process on andrea is the cause of the performance problem. However, we can view these metrics of interest with a simple time-line representation. These views are displayed in Figure 17.

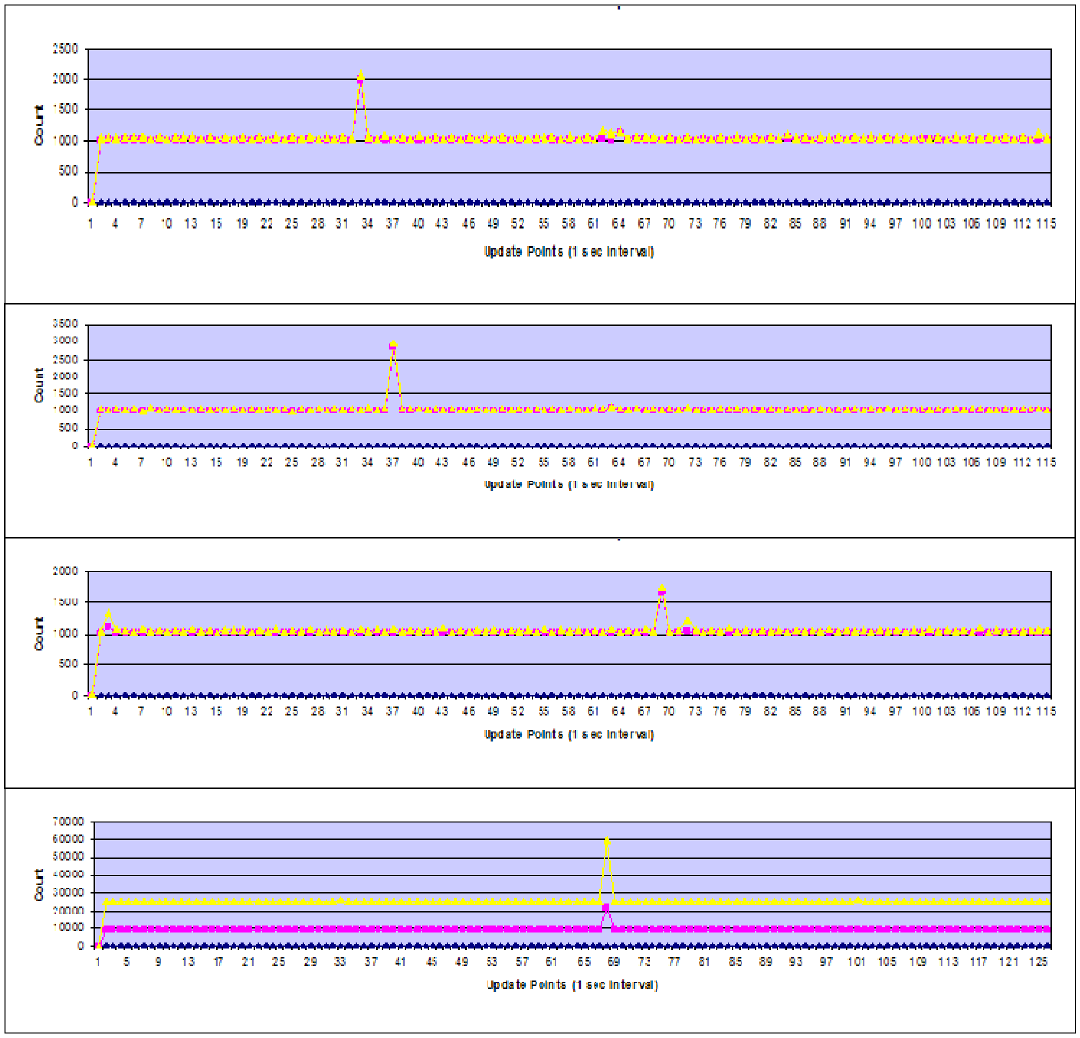
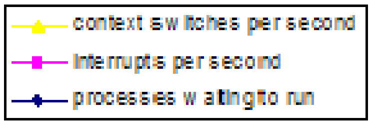


Figure 17: Processes, Interrupts and Context Switches for paulus and andrea in the Normal and Sub-optimal Environments. The first graph shows data for paulis in the normal environment. The second graph shows data for andrea in the normal environment. The third graph shows data for paulus in the sub-optimal Environment. The fourth graph shows data for andrea in the sub-optimal environment.



6.2.5 Discussion

We conclude that it is possible to use generally available system utilities to gain insight into the performance behavior of applications, however, integrating this information with application level data requires new tool support.

CHAPTER 7 CONCLUSIONS AND FUTURE WORK

In this thesis we proposed a new approach for performance analysis called Environment-Aware Performance Analysis. This approach focuses on improving the quality of diagnosis offered by performance analysis tools. We limited the focus of this thesis to the investigation of environment-aware performance analysis for parallel applications.

Environment-Aware Performance Analysis includes analysis of an application's execution behavior and analysis of the environment in which an application executes. The performance diagnosis incorporates both sets of analyses, and it seeks to identify root causes of performance behavior. Environment-Aware Performance Analysis extends traditional methods of application performance analysis, and, by doing so, has potential to improve the quality of performance diagnosis.

This thesis demonstrates a need for performance analysis tools to utilize information about the runtime environment in the analyzing performance data and formulating performance diagnoses. To support this claim we presented real situations in which traditional performance analysis tools either provided misleading diagnostic information or were unable to provide adequate guidance to analysts in determining the causes of performance problems. We also provided proof of concept examples to show the potential for performance analysis tools to improve the quality of diagnosis by incorporating analysis of the runtime environment with analysis of application behavior.

We presented a layered view of the runtime environment which can be used to frame performance problems in terms of root causes and potential optimizations. We also discussed development considerations for the design of environment-aware tools.

The contributions of this thesis include a test suite for evaluating the ability of tools to accurately diagnose performance problems; experimental results showing the limitations of current tools; a set of case studies illustrating situations that might have benefited from environment-aware performance analysis; a discussion of the challenges and concerns for development of environment-aware tools; and a survey of the state of the art in parallel performance analysis tools.

As we continue our research in this area we have identified several areas for future work. We would like to conduct a tool evaluation study using the test suite, and we want to complete other experiments that are currently in progress. We plan to develop a prototype tool that encompasses the key features of environment-aware performance analysis. This effort will entail preliminary work in the development of techniques for integration of data collected from different layers and by different tools. An example of this is integrating application performance trace data, MPI library profiling data, vmstat-like data collected from a system monitoring facility, and hardware counter performance data. These data sets span the layers of the runtime environment and could potentially be collected by different tools. We want to develop methods for automated analysis of integrated data which detect performance problems and identify the runtime layers in which likely causes of performance problems exist.

This work focused on performance of high end parallel applications. Environment aware analysis is also applicable to other computing domains, including databases, server applications, operating system benchmarking, and Grid computing. Environment aware performance analysis can support user perspectives other than the application developer/user. For example, a system engineer, designing new systems or benchmarking systems, is concerned with the performance of system hardware components, and is less concerned with improving the performance of individual applications.

REFERENCES

1. Anderson, J. M., Berc, L. M., Dean, J., Ghemawat, S., Henzinger, M. R., Leung, S. A., Sites, R. L., Vandevoorde, M. T., Waldspurger, C. A., and Weihl, W. E. 1997. Continuous profiling: where have all the cycles gone?. *ACM Trans. Comput. Syst.* 15, 4 (Nov. 1997), 357-390. DOI= http://doi.acm.org/10.1145/265924.265925.

2. Berrendorf, R. and Mohr, B. *PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors (version 2.2)*. 2003. Zentralinstitut für Angewandte Mathematik (ZAM), . Retrieved 6-20-06 from ZAM: http://www.fz-juelich.de/zam/PCL/doc/pcl/pcl.pdf.

3. Berrendorf, R. and Ziegler, H. *PCL - The Performance Counter Library: A Common Interface to Access Hardware Performance Counters on Microprocessors*. Technical Report FZJ-ZAM-IB-9816, Zentralinstitute Fur Angewandte Mathematik, October, 1998.

4. Browne, S., Dongarra, J., Garner, N., Ho, G., and Mucci, P. A portable programming interface for performance evaluation on modern processors. *The International Journal of High Performance Computing Applications*. 14, 3 (Fall 2000), 189-204.

5. Brunst, H., Kranzlmüller, D., Nagel, W. E. 2004. Tools for Scalable Parallel Program Analysis - Vampir VNG and DeWiz. In Distributed and Parallel Systems: Cluster and Grid Computing, Z. Juhasz, P. Kacsuk and D. Kranzlmüller, Eds. Kluwer International Series in Engineering and Computer Science, Springer, New York, NY, 777, 93-102.

6. Buck, B. R. and Hollingsworth, J. K. An api for runtime code patching. *Journal of High Performance Computing Applications*, 14 (4). 317-329.

7. Cascaval, C., Duesterwald, E., Sweeney, P.F, and Wisniewski, R. *Vertical Performance Monitoring*. Presentation. WAPA, 2005, IBM T. J. Watson Research Center, December 16, 2005.

8. Cascaval, C., Dusterwald, E., Sweeney, P. F., and Wisniewski, R. W. Performance and environment monitoring for continuous program optimization. *IBM Journal of Research and Development.* 50, 2-3 (March/May 2006), 239-247.

9. Cheng, D. Y. and Pase, D. M. 1991. An evaluation of automatic and interactive parallel programming tools. In *Proceedings of the 1991 ACM/IEEE Conference*

on Supercomputing (Albuquerque, New Mexico, United States, November 18 - 22, 1991). Supercomputing '91. ACM Press, New York, NY, 412-423. DOI= http://doi.acm.org/10.1145/125826.126052.

10. DeRose, L. and Reed, D. A. 1999. SvPablo: A multi-language architecture-independent performance analysis system. In *Proceedings of the International Conference on Parallel Processing (ICPP'99),* (Fukushima, Japan, September 21-24, 1999). IEEE Computer Society Press, 311-318.

11. DeRose, L. Hoover, Jr., T., and Hollingsworth, J. K. 2001. The Dynamic Probe Class Library – An Infrastructure for Developing Instrumentation for Performance Tools. In *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01)* (San Francisco, CA, April 23-27, 2001). IEEE Computer Society, 10066b. DOI=http://doi.ieeecomputersociety.org/10.1109/IPDPS.2001.925012

12. DeRose, L. http://www.ccs.ornl.gov/workshops/FallCreek05/presentations/l_derose.pdf, retrieved 5-31-06.

13. DeRose, L., Hoover Jr., Hollingsworth, J. K. 2001. The dynamic probe class library – an infrastructure for developing instrumentation for performance tools. *Proceedings of the 15th International Parallel and Distributed Processing Symposium (IPDPS'01).* 1:10066.2, 2001

14. Fahringer, T. and Seragiotto, C. 2004. AKSUM: A performance analysis tool for parallel and distributed applications. In *Performance Analysis and Grid Computing*, V. Getov, M. Gerndt, A. Hoisie, A. Malony, and B. Miller, Eds. Kluwer Academic Publishers, Norwell, MA, 189-208.

15. Fahringer, T., Gerndt, M., Riley, G. and Traff, J. *Knowledge Specification for Automatic Performance Analysis*. Technical Report FZJ-ZAM-IB-9918, APART ESPRIT IV Working Group on Automatic Performance Analysis: Resources and Tools, November, 1999

16. Fahringer, T., Jugravu, A., Pllana, S., Prodan, R., Seragiotto Jr., C. and Truong, H-L. ASKALON: a tool set for cluster and grid computing. *Concurrency and Computation: Practice and Experience*, 17, 2-4 (2005), 143-169.

17. Gerndt, M., Mohr, B., Pantano, M. and Wolf, F. Automatic performance analysis for CRAY T3E. In *Proceedings of the Seventh Workshop on Compilers for Parallel Computers (CPC'98)*, University of Linköping, Sweden, June 29 - July 1 1998, pp. 69-78

18. Graham, S. L., Kessler, P. B., and Mckusick, M. K. 1982. gprof: a call graph execution profiler. In *Proceedings of the 1982 SIGPLAN Symposium on Compiler Construction (SIGPLAN '82)* (Boston, Massachusetts, United States, June 23 - 25, 1982). ACM Press, New York, NY, 120-126.

19. Graham, S. L., Kessler, P. B., and McKusick, M. K. 2004. gprof: a call graph execution profiler. *SIGPLAN Not*. 39, 4 (Apr. 2004), 49-57. DOI= http://doi.acm.org/10.1145/989393.989401.

20. *Grand Challenges: Science, Engineering, and Societal Advances Requiring Networking and Information Technology Research and Development*, 2004. Howe, S. E. and King, F. D. Eds. National Coordination Office for Information Technology Research and Development (NITRD), Arlington, VA. Retrieved 5-10-06 from NITRD: http://www.nitrd.gov/pubs/200311_grand_challenges.pdf

21. Gropp, W., Huss-Lederman, S., Lumsdaine, A., Lusk, E., Nitzberg, B., Saphir, W. and Snir, M. *MPI: The Complete Reference (Vol. 2), Volume 2 - The MPI-2 Extensions.* The MIT Press,Cambridge, MA, 1998.

22. Hauswirth, M., Diwan, A., Sweeney, P. F., and Mozer, M. C. 2005. Automating vertical profiling. In *Proceedings of the 20th Annual ACM SIGPLAN Conference on Object Oriented Programming, Systems, Languages, and Applications* (San Diego, CA, USA, October 16 - 20, 2005). OOPSLA '05. ACM Press, New York, NY, 281-296. DOI= http://doi.acm.org/10.1145/1094811.1094834

23. Herrarte, V. and Lusk, E. *Studying Parallel Program Behavior with* `Upshot`. Technical Report ANL-91/15, Argonne National Laboratory, Argonne, IL, 1991.

24. High Performance Computing Act of 1991. National Coordination Office for Networking and Information Technology Research and Development (NITRD), Arlington, VA. Retrieved 5-10-06 from NITRD: http://www.nitrd.gov/congressional/laws/pl_102-194.html.

25. Hollingsworth, J. K. and Keleher, P. J. 1998. Prediction and Adaptation in Active Harmony. In *Proceedings of the Seventh IEEE international Symposium on High Performance Distributed Computing* (July 28 - 31, 1998). HPDC. IEEE Computer Society, Washington, DC, 180.

26. Hollingsworth, J. K., Irvin, R. B. and Miller, B. P. *The Integration of Application and System Based Metrics in a Parallel Program Performance Tool.* Technical Report TR996, University of Wisconsin-Madison, Computer Sciences Department, Madison, WI, 1991.

27. http://bhami.com/rosetta.html, retrieved 6-1-06.

28. Intel Trace Analyzer and Collector. Retrieved 5-24-06 from Intel: http://www.intel.com/cd/software/products/asmo-na/eng/cluster/tanalyzer/index.htm, retrieved 5-24-06.

29. Iperf. http://dast.nlanr.net/projects/Iperf/, retrieved 3-6-06.

30. Jones, T. R., Brenner, L. B. and Fier, J. M. Impacts of Operating Systems on the Scalability of Parallel Applications, Technical Report, Lawrence Livermore National Laboratory, 2003.

31. Kale, L. V., Kumar, S., Zheng, G. and Lee, C. W. 2003. Scaling molecular dynamics to 3000 processors with projections: a performance analysis case study. In *Proceedings of the Terascale Performance Analysis Workshop, International Conference on Computational Science* (*ICCS 2003*) (Melbourne, Australia, June 2-4, 2003). Lecture Notes in Computer Science, Springer, Berlin/Heidelberg, 2660. 25-32.

32. Karavanic, K. L. and Miller, B. P. 1997. Experiment management support for performance tuning. In *Proceedings of the 1997 ACM/IEEE Conference on Supercomputing* (Cdrom) (San Jose, CA, November 15 - 21, 1997). Supercomputing '97. ACM Press, New York, NY, 1-10. DOI= http://doi.acm.org/10.1145/509593.509601.

33. Karavanic, K. L. and Miller, B. P. 1999. Improving online performance diagnosis by the use of historical performance data. In *Proceedings of the 1999 ACM/IEEE Conference on Supercomputing* (Cdrom) (Portland, Oregon, United States, November 14 - 19, 1999). Supercomputing '99. ACM Press, New York, NY, 42. DOI= http://doi.acm.org/10.1145/331532.331574.

34. Karavanic, K. L., May, J., Mohror, K., Miller, B., Huck, K., Knapp, R., and Pugh, B. 2005. Integrating Database Technology with Comparison-based Parallel Performance Diagnosis: The PerfTrack Performance Experiment Management Tool. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing* (November 12 - 18, 2005). Conference on High Performance Networking and Computing. IEEE Computer Society, Washington, DC, 39. DOI= http://dx.doi.org/10.1109/SC.2005.36.

35. Karrels, E. and Lusk, E. Performance Analysis of MPI Programs. 1994. In *Proceedings of the Workshop on Environments and Tools for Parallel Scientific Computing* (Townsend, TN, May 1994). J. Dongarra and B. Tourancheau, Eds. SIAM Publications, 195-200.

36. Kitchens, T. A., Austin D. M., Fitzmaurice, J. M., Glick, N. S., Johnson, F. C., Kelley, R. L., Martino, R. L., Milstead, P. L., Novak, J. H., Poliakoff, A. T., Turnbull, W. T., Ricart, G. and Voigt, R. G. 1996. *High Performance Computing and Communications: Foundation for America's Information Future.* S. Howe, Ed. The National Coordination Office for High Performance Computing and Communications (NCO), Bethseda, MD. Retrieved 5-10-06 from NITRD: http://www.nitrd.gov/pubs/bluebooks/1996/.

37. Kohn, J. and Williams, W. ATExpert. Journal of Parallel and Distributed Computing, 18, 2 (June 1993), 205-222.

38. Kola, G., Kosar, T., and Livny, M. Profiling grid data transfer protocols and servers. Europar 2004.

39. Kufrin, R. 2005. Measuring and improving application performance with PerfSuite. Linux J. 2005, 135 (Jul. 2005), 4

40. Kufrin, R., PerfSuite: An Accessible, Open Source Performance Analysis Environment for Linux, 6th International Conference on Linux Clusters: The HPC Revolution 2005. Chapel Hill, NC, April 2005

41. LAM/MPI. http://www.lam-mpi.org, retrieved 3-17-06.

42. Lamport, L. 1978. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM* 21, 7 (Jul. 1978), 558-565. DOI= http://doi.acm.org/10.1145/359545.359563.

43. Lilja, D. J. *Measuring Computer Performance. A Practitioner's Guide.* Cambridge University Press, Cambridge, UK, 2000.

44. M. Hauswirth, P. F. Sweeney, A. Diwan, M. Hind. "Vertical Profiling: Understanding the Behavior of Object-Oriented Applications". *OOPSLA '04: Proceedings of the 19th annual ACM SIGPLAN Conference on Object-oriented programming, systems, languages, and applications*. 251-269, Vancouver, BC, Canada, October 2004.

45. McCalpin, J. D. *STREAM: Sustainable Memory Bandwidth in High Performance Computers*. Retrieved 6-01-06 from http://www.cs.virginia.edu/stream/.

46. Miller, B. P., Callaghan, M. D., Cargille, J. M., Hollingsworth, J. K., Irvin, R. B., Karavanic, K. L., Kunchithapadam, K. and Newhall, T. The Paradyn parallel performance measurement tool, *IEEE Computer* 28, 11 (November 1995), 37-46. Special issue on performance evaluation tools for parallel and distributed computer systems.

47. Miller, B. P., Clark, M., Hollingsworth, J., Kierstead, S., Lim, S.-S. and Torzewski, T. IPS-2: the second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, 1, 2 (April 1990), 206 - 217.

48. Miller, B., Cortes, A., Senar, M. A., and Livny, M. 2003. The Tool Dæmon Protocol (TDP). In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing* (November 15 - 21, 2003). Conference on High Performance Networking and Computing. IEEE Computer Society, Washington, DC, 19.

49. Mirgorodskiy, A. V. and Miller, B. P. Autonomous analysis of interactive systems with self-propelled instrumentation. *Multimedia Computing and Networking Conference (MMCN '05),* (San Jose, California, 2005).

50. Mirgorodskiy, A. V., Maruyama, N. and Miller, B. P. Miller, "Problem Diagnosis in Large-Scale Computing Environments", Submitted for publication.

51. Mohr, B., Wolf, F., 2003. KOJAK – A Tool Set for Automatic Performance Analysis of Parallel Programs. In Harald Kosch, László Böszörményi, Hermann

Hellwagner (Eds.): Euro-Par 2003. Parallel Processing, 9th International Euro-Par Conference, Klagenfurt, Austria, August 26-29, 2003. Proceedings. Springer. Lecture Notes in Computer Science, Vol. 2790, 1301 - 1304, 2003.

52. Mohror, K., Personal Communication, June 2, 2006.

53. Mooney, R., Studham, R. S., Schmidt, K. P., Nieplocha, J. 2004. NWPerf: A System Wide Performance Monitoring Tool for Large Linux Clusters. *Cluster Computing*, 2004 IEEE International Conference (San Diego, California, September 20-23, 2004). 379-389.

54. MPE. http://www-unix.mcs.anl.gov/perfvis/download/index.htm#MPE, retrieved 5-25-06.

55. mpiP current version. http://www.llnl.gov/CASC/mpip/, retrieved 5-25-06.

56. mpiP earlier verison. http://www.llnl.gov/CASC/mpip/, retrieved 3-02-06.

57. Nagel, W. E. Nagel, Arnold, A., Weber, M., Hoppe, H.-C. and Solchenbach, K. VAMPIR: Visualization and analysis of MPI resources. *Supercomputer*, 12(1):69--80, January 1996.

58. NetPIPE. http://www.scl.ameslab.gov/netpipe/, retrieved 1-3-2006.

59. Open MPI: Open Source High Performance Computing. Retrieved 6-20-06 from Open MPI: http://www.open-mpi.org/.

60. OProfile. http://oprofile.sourceforge.net , retrieved 5-24-06.

61. PAPI. http://icl.cs.utk.edu/papi/, retrieved 5-25-06.

62. *Paradyn Parallel Performance Tools User's Guide*, Release 4.2, Paradyn Project, Computer Sciences Department, University of Wisconsin, Madison, WI 53706-1685. March 2005. Retrieved 8-5-05 from University of Wisconsin: http://www.paradyn.org/html/manuals.html.

63. Paradyn Parallel Performance Tools. Paradyn Project, Computer Sciences Department, University of Wisconsin, Madison, WI. Retrieved 3-20-06 from University of Wisconsin: http://www.paradyn.org.

64. Pase, D., Personal Communication, 2006.

65. PCL - The Performance Counter Library Version 2.2, January 2003. Zentralinstitut für Angewandte Mathematik (ZAM), . Retrieved 5-25-06 from ZAM: http://www.fz-juelich.de/zam/PCL/.

66. Performance Visualization for Parallel Programs. Argonne National Laboratory, Argonne, IL. Retrieved 6-18-06 from ANL: http://www-unix.mcs.anl.gov/perfvis/

67. Performance Visualization for Parallel Programs: Viewers. Argonne National Laboratory, Argonne, IL. Retrieved 6-18-06 from ANL: http://www-unix.mcs.anl.gov/perfvis/software/viewers/index.htm

68. PerfSuite. http://perfsuite.ncsa.uiuc.edu/, retrieved 6-1-06.

69. Petrini, F., Kerbyson, D. J., and Pakin, S. 2003. The case of the missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q. In Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (November 15 - 21, 2003). Conference on High Performance Networking and Computing. IEEE Computer Society, Washington, DC, 55.

70. Phillips, J. C., Zheng, G., Kumar, S., and Kalé, L. V. 2002. NAMD: biomolecular simulation on thousands of processors. In Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (Baltimore, Maryland). Conference on High Performance Networking and Computing. IEEE Computer Society Press, Los Alamitos, CA, 1-18.

71. PPerfMark. http://www.cs.pdx.edu/~karavan/pperfdb/PPerfMark20.tar.gz, retrieved 2-21-06.

72. Purdy, M. 2006. Modern Performance Monitoring. Queue 4, 1 (Feb. 2006), 48-57. DOI= http://doi.acm.org/10.1145/1117389.1117404.

73. Radu Prodan and Thomas Fahringer. 2002. ZENTURIO: An experiment management system for cluster and grid computing. In *Proceedings of the 4th International Conference on Cluster Computing (CLUSTER 2002)* (Chicago, USA, September 2002). IEEE Computer Society Press.

74. Reed, D. A. 1993. Performance instrumentation techniques for parallel systems. In *Performance Evaluation of Computer and Communication Systems*, Joint Tutorial Papers of Performance '93 and Sigmetrics '93 (May 10 - 14, 1993). L. Donatiello and R. D. Nelson, Eds. Springer-Verlag, London, 729 (1993), 463-490.

75. Reed, D. A., Aydt, R. A., Noe, R. J., Roth, P. C., Keith A. Shields, Bradley Schwartz, and Luis F. Tavera, "Scalable Performance Analysis: The Pablo Performance Analysis Environment," Anthony Skjellum(ed), Proceedings of the Scalable Parallel Libraries Conference, IEEE Computer Society, October 1993, pp. 104-113.

76. Roth, P. C. *Etrusca: event trace reduction using statistical data clustering analysis*. 1996. Thesis.

77. Roth, P. C., Arnold, D. C., and Miller, B. P. 2003. MRNet: A Software-Based Multicast/Reduction Network for Scalable Tools. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing* (November 15 - 21, 2003). Conference on High Performance Networking and Computing. IEEE Computer Society, Washington, DC, 21.

78. Shende, S. and Malony, A. D. 2005. The TAU parallel performance system. (submitted to) *International Journal of High Performance Computing Applications, ACTS Collection Special Issue*, 2005.

79. Smith, W. A system for monitoring and management of computational grids. *International Conference on Parallel Processing (ICPP'02)*. 55, 2002.

80. Snell, Q., Mikler, A., and Gustafson, J. Netpipe: a network protocol independent performance evaluator. In *IASTED International Conference on Intelligent Information Management and Systems*. June 1996.

81. Snir, M., Otto, S., Huss-Lederman, S., Walker, D., and Dongarra, J. *MPI: The Complete Reference (Vol. 1) - 2nd Edition, Volume 1 - The MPI Core*. The MIT Press,Cambridge, MA, 1998.

82. Srivastava, A. and Eustace, A. 2004. ATOM: a system for building customized program analysis tools. *SIGPLAN Not*. 39, 4 (Apr. 2004), 528-539. DOI= http://doi.acm.org/10.1145/989393.989446

83. Tapia, R., Lanius, C., McZeal, M. and Parks, T. A. *Computational Science: Tools for a Changing World. A High School Curriculum*. Rice University, 2001. Retrieved 5-2-06, from Rice University: http://ceee.rice.edu/Books/CS/index.html.

84. Tool Gear earlier version. http://www.llnl.gov/CASC/tool_gear/, retrieved 3-02-06.

85. Tool Gear later version. http://www.llnl.gov/CASC/tool_gear/, retrieved 6-1-06.

86. TOP 500, 2002. http://www.top500.org/list/2002/06/, retrieved 5-15-06.

87. TOP 500. http://www.top500.org/, retrieved 5-15-06.

88. Truong, H-L, Fahringer, T, Madsen, G., Malony, A. D., Moritsch H. and Shende, S. "On using SCALEA for Performance Analysis of Distributed and Parallel Programs", Supercomputing 2001 Conference (SC2001), Denver, Colorado, USA. November 10-16,2001

89. Truong, H-L. and Fahringer T. SCALEA-G: a unified monitoring and performance analysis system for the grid. *Scientific Programming*, 12, 4 (2004), 225-237.

90. Truong, H-L., Fahringer, T. and Dustdar, S. Dynamic instrumentation, performance monitoring and analysis of grid scientific workflows. *Journal of Grid Computing*, 3, 1-2 (June 2005), 1-18. DOI= 10.1007/s10723-005-5299-6

91. Williams, W., Hoel, T., Pase, D. 1994. The MPP apprentice performance tool: delivering the performance of the Cray T3D. Programming Environments for Massively Parallel Distributed Systems, K. Decker and R. M. Rehman, Eds. Birkhauser Verlag, Basel, 333-346.

92. Wisniewski, R.W., Sweeney, P. F., Sudeep, K., Hauswirth, M., Duesterwald, E., Cascaval, C., and Azimi, R. "Performance and Environment Monitoring for Whole-System Characterization and Optimization", *PAC2 Conference on Power/Performance interaction with Architecture, Circuits, and Compilers*. IBM T.J. Watson Research Center, Yorktown Heights, New York, September, 2004.

93. Wolf, F. and Mohr, B. 2003. Hardware-counter based automatic performance analysis of parallel programs. In *Proceedings of the Minisymposium 'Performance Analysis' Conference on Parallel Computing (PARCO '03),* (Dresden, Germany, September 3, 2003), Elsevier.

94. Wolski, R. 2003. Experiences with predicting resource performance on-line in computational grid settings. *SIGMETRICS Perform. Eval. Rev.* 30, 4 (Mar. 2003), 41-49. DOI= http://doi.acm.org/10.1145/773056.773064

95. Wolski, R., Spring, N and Hayes, J. "The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing", *Journal of Future Generation Computing Systems*, 15(5-6):757-768, October, 1999.

96. Yan, J. C. 1994. Performance tuning with AIMS -- an automated instrumentation and monitoring system for multicomputers. In *Proceedings of the 27th Hawaii International Conference on System Sciences* (Wailea, Hawaii, January, 4 - 7, 1994). 2. 625-633.

97. Yan, J. C., etal. The automated instrumentation and monitoring system version 3.7 users' guide. 1997. Retrieved 6-18-06 from NAS Systems Division, NASA Ames Research Center: http://www.nas.nasa.gov/Groups/Tools/AIMS/manual/TableTest.html.

98. Zaki, O., Lusk, E., Gropp, W., and Swider, D. Toward Scalable Performance Visualization with Jumpshot. Mathematics and Computer Science Division, Argonne National Laboratory, 1999. Retrieved 6-18-06 from ANL: http://www-unix.mcs.anl.gov/perfvis/software/viewers/jumpshot-2/paper.html.

99. Zaki, O., Lusk, E., Gropp, W., and Swider, D. Toward scalable performance visualization with jumpshot. *High-Performance Computing Applications*, 13(2). 277-288.