EXPERIMENT MANAGEMENT SUPPORT FOR PARALLEL PERFORMANCE TUNING

BY

KAREN L. KARAVANIC

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

(Computer Sciences)

at the University of Wisconsin—Madison

1999

EXPERIMENT MANAGEMENT SUPPORT FOR PARALLEL PERFORMANCE TUNING

Karen L. Karavanic

Under the supervision of Professor Barton P. Miller

at the University of Wisconsin—Madison

The development of a high-performance parallel system or application is an evolutionary process. It may begin with models or simulations, followed by an initial implementation of the program. The code is then incrementally modified, and continues to evolve throughout the applications's lifespan. At each step, a key question for developers is: *how and how much did the performance change?* This question arises while comparing an implementation to models or simulations; considering versions of an implementation that use a different algorithm, communication or numeric library, or language; studying code behavior by varying number or type of processors, type of network, type of processes, input data set or work load, or scheduling algorithm; and benchmarking or regression testing. We present a design and prototype implementation of an experiment management environment designed to answer performance questions that span multiple program executions from all stages of the lifespan of an application.

We have developed a concise representation for the set of executions collected over the life of an application. In our model, information from all experiments for one application, including the components of the code executed, execution environment, and performance data collected, is gathered in the Program Space.

We developed techniques for automating comparison between measured executions. The structural difference operator determines differences in the source code and the runtime environment; the performance difference operator compares performance results and reports results that differ by more than a specified amount. We present several case studies exploring the use of these operators with large-scale parallel applications.

We also developed a novel approach to automated performance diagnosis that uses application data gathered in previous executions to guide the search for performance bottlenecks. Adding historical knowledge about an application provides a means for the tool to perform more effective diagnosis. We evaluated our technique using different versions of an MPI application on an IBM SP/2, and found reductions of 31% to 98% in the time needed to locate performance bottlenecks.

Acknowledgments

I thank my advisor, Bart Miller, for these past years of encouragement, guidance, and support. By guiding me through many periods of intense, deadline-driven work, he has given me the gift of knowing my own strength and wisdom.

I thank David Wood, Marvin Solomon, and Miron Livny for their time and effort spent reading my dissertation, and for providing valuable criticisms and suggestions for improving my work. I thank my entire committee -- Marvin Solomon, Miron Livny, David Wood, and Gregory Moses -- for a lively and thorough discussion of my dissertation during my defense.

I gratefully acknowledge financial support provided by the NASA GSRP Fellowship Program, the United States Department of Energy, and the National Science Foundation.

The members of the Paradyn Performance Tools research group, past and present, have provided me with technical discussions, ideas, advice, and encouragement, and have attended more practice talks than anyone might reasonably expect. Many other people have shared with me technical discussions, guidance, and interesting ideas, including Douglas Pase, the members of the IBM Parallel Tools Group in Poughkeepsie, Joann Ordille, John May, Mary Zosel, Mary Vernon, Janet Wiener, Jerry Yan, Bob Hood, Steve Huss-Lederman, and Doug Kimmelman.

I am grateful to many special people who have provided me with meals, rides, beverages, laughter, the occasional roof over my head, and lots of positive energy through the various stages of my dissertation work: Elizabeth Webster, Richard Russell, Phil Kaveny, Mary LoSardo, Mom, Beth Martinson, Terry Mika, Steve Swartz, Michael Rawdon, Doug Pase, Alain Kagi, Doug Burger, Anne at Portabella, Rich Maclin, Beth Cole, Tia Newhall, all of my Systers, and my fellow Starfish. The folks in Nancy Pullen's dissertators group shared their lives with me and taught me to eat live toads.

I thank Sam Pottle for the abundance of loving encouragement, chocolate, and champagne that saw me through.

Contents

Chapter 1

Introduction

1.1 Motivation

The development of a high-performance parallel system or application is an evolutionary process. It may begin with models or simulations, followed by an initial implementation of the program. The code is then incrementally modified to tune its performance, and continues to evolve throughout the applications's lifespan. At each step, the key question for developers is: *how and how much did the performance change?* This question arises while comparing an implementation to models or simulations; considering versions of an implementation that use a different algorithm, communication or numeric library, or language; studying code behavior by varying number or type of processors, type of network, type of processes, input data set or work load, or scheduling algorithm; and benchmarking or regression testing. Despite the broad utility of this type of comparison, the current generation of parallel performance tools focuses on measuring the performance of a single program execution. This dissertation describes a design and prototype implementation of a performance tool designed to answer performance questions that span multiple program executions from all stages of the lifespan of an application.

Recent work in the more general area of scientific experiment management focuses on providing the means to record data used in or generated by experiments[38,43,56]. This includes a potentially distributed, large, and varying store of experimental data; descriptions of the experimental methodology; and indexing information to link the data to its source. We view performance tuning as a specialized instance of scientific experimentation, with each complete or partial application execution viewed as an experiment. This changes our definition of a performance tool from the traditional, single program execution approach to a new broader, inherently multi-execution approach. The engineers or scientists engaged in studying and improving the behavior of medium to large scale parallel scientific codes, are analogous to chemists or biologists carrying out experiments at a lab bench. Each complete or partial program run, simulation result, or program model is an experiment, analogous to the biologist's gene-expression experiment or the chemist's chromatography study. We have used this scientific experimentation archetype as a basis for designing an *Experiment Management* environment for Parallel Performance. Such an environment would be directly useful to both application and system developers, as well as creators of models and simulation environments, and represents a qualitative change from the state of the art.

In our model, information from all experiments for one application, including the components of the code executed, the execution environment, and the performance data collected, is gathered in a *Program Space*. Performance data of many forms may be stored in the Program Space, including scalars, tables, traces, and graphs. The Program Space also includes descriptive data, or metadata, which characterize the execution. Examples of possible types of metadata are environment variable settings, compiler options used to build the executable, or a description of the machine used for the run. The possible combinations of code and execution environment form the multi-dimensional Program Space, with one dimension for each axis of variation and one point for each individual experiment. The Program Space represents instances of a program selected from different points in its lifetime, during which it may be modified, ported to different architectures, tuned, or compiled with different options or using a different compiler. Program Space views serve as a user interface to the system.

An Experiment Management tool enables exploration of this space with a simple naming mechanism, a selection and query facility, and a set of visualizations. Examining the behavior of more than one version of a program, or the behavior of one version of a program in different environments, is a common task; for example, developers of libraries that will be used across a variety of platforms require a complete performance picture in which performance on a single platform is less important than the performance across all platforms.

Performance tuning across multiple executions must answer the deceptively simple question: what changed in this run of the program? A key component of this work is the ability to automatically describe the differences between two runs of a program, both the structural differences (differences in program source code and the resources used at runtime), and the performance variation (how were the resources used and how did this change from one run to the next). The difference information is not necessarily a simple measure such as total execution time, but may be a more complex measure derived from details of the program structure. The items being compared may include an analytical performance prediction, a previous execution of the code, a set of performance thresholds that the application is required to meet or exceed, or an incomplete set of data from selected intervals of an execution.

These automated comparison techniques have several potential applications: they can compare an actual execution with a predicted or desired performance measure for the application, or compare dictinct time intervals of a single program execution. An example of a more complex environment in which these comparison techniques might be useful is resource allocation for metacomputing [16], the use of distributed heterogeneous computers and high-speed networks to perform scientific computation previously possible only on dedicated supercomputers. Effective resource allocation in such an environment requires knowledge of past performance on a variety of architectures and machine configurations.

1.2 Contributions

In this dissertation, we describe our solutions to several key problems that arise in constructing a complete experiment management based performance tool: First, we developed a representation for the space of executions; second, we developed techniques for quantitatively and automatically comparing two or more executions, and tested them out in a variety of performance related tasks. Finally, as further evidence of the utility of a well-organized and accessible store of application performance data, we developed and studied a new approach to automated performance diagnosis using historic performance data and dynamic instrumentation.

The first research contribution of this thesis is a concise representation for the set of executions collected over the life of an application. In our model, information from all experiments for one application, including the components of the code executed, the execution environment, and the performance data collected, is gathered in a Program Space. We refer to each experiment as a *Program Event*. A Program Space comprises a SpaceMap, one or more Event-

Maps, and a collection of Performance Results. The *SpaceMap* contains descriptive data that characterizes Program Events, distinguishing the various Program Events described in the same Program Space. Examples of descriptive data stored in the SpaceMap are: input data set characteristics, environment variable settings, compiler options used to build the executable, a description of the machine used for the run, code version number, or laboratory name. Each *EventMap* contains a detailed list of one Program Event's resources, such as individual function names and machine components. It serves as an interface to the performance results, and provides a naming scheme for the potentially large collection of performance data. *Performance Results* are measured values for an execution's behavior, for example, total CPU time for a specified function. The Program Space is designed to accommodate stored performance data of many forms, including scalars, tables, traces, and graphs.

The second research contribution is a set of techniques for automating comparison between measured executions. We developed techniques for determining the difference between two or more program runs. Difference is computed by two operators: the structural difference operator compares two EventMaps and reports resources that differ; and the performance difference operator compares two collections of performance results and reports results that differ by more than a specified amount. We implemented a prototype that computes the difference operators, and performed several case studies in which we explore their use with large-scale parallel applications.

The third contribution of this research is an investigation of the use of the historical data contained in the Program Space for improving performance diagnosis. Harvesting useful historical knowledge requires an available store of performance data gathered from one or more previous program runs. Our research explores novel opportunities for exploiting this collection of data to focus data gathering and analysis efforts to the critical sections of a large application. This approach allows a complex performance evaluation to be specified and conducted with a minimum of user intervention. We present a novel approach to automated diagnosis that uses application data gathered in previous executions to guide the search for performance bottlenecks. This method leverages off of the repetitive nature of the performance tuning process — it is rare for a parallel application to be examined with a performance tool only once. Adding historical knowledge about an application provides a means for the tool to become more effective that does not rely on assumptions about all possible applications with which it might be used.

Our starting point was an existing diagnostic research tool, the Paradyn Parallel Performance Tool [54]. Paradyn's Performance Consultant performs online, automated bottleneck detection in a single execution of a parallel or serial program. We modified the Performance Consultant, incorporating several different types of historical knowledge about an application's performance into the tool's search for performance problems: *pruning directives* that tell the tool to ignore some resources entirely; *priorities* that tell the tool which aspects of the application and run-time environment to look at first; and *thresholds* that tell the tool specific values against which to measure the application's actual performance. We use the directives to guide online performance diagnosis with an enhanced version of Paradyn. We evaluated our technique by testing different versions of an MPI application on the IBM SP/2, and found reductions of 31% to 98% in the time needed to locate performance bottlenecks.

1.3 Roadmap

We discuss related work in the following chapter. Chapter 3 presents the Program Space, a flexible and uniform mechanism for describing and naming selections from the space of all executions throughout the lifetime of an application. It includes a flexible canonical representation, called Resource Normal Form, for storing the variety of program performance data for which our automatic analysis techniques have been developed. We also describe our techniques for describing, representing, and comparing multiple versions and runs of a program. In Chapter 4 we present results of three case studies using an initial prototype implementation of an experiment management system. In Chapter 5 we describe the use of historical performance data in improving the diagnosis abilities of Paradyn’s Performance Consultant. We summarize our results and discuss future research directions in Chapter 6.

Chapter 2

Related Work

In this chapter we discuss the array of research efforts most closely related to this dissertation. In Section 2.1 we examine parallel performance tools. In Section 2.2 we survey approaches for automating parallel performance diagnosis. In Section 2.3 we discuss scientific experiment management. Section 2.4 surveys work that involves comparing different program versions. We summarize in Section 2.5.

2.1 Parallel Performance Tuning

Performance tuning is a cyclic process that involves repeating a series of steps until acceptable application performance is achieved: (1) formulate one or more hypotheses about application behavior; (2) gather data to support or dismiss each hypothesis; (3) analyze that data (by direct inspection, analytical techniques, or some combination); and (4) change the code or the environment experimentally based on these hypotheses. Parallel applications may be tuned many times, for example when ported to a new platform or after code revision. We can categorize performance tool research efforts by identifying the particular steps (1)-(4) the tool is designed to accomplish or guide. For the most part, performance tools are rather narrowly defined as a mechanism for gathering data (step 2) or as an aide to a human formulating hypotheses about an application’s behavior (step 1) during a single observed program run [27,54,55,61,69,77,80]. In this common approach, much of the work is done manually by a knowledgable expert conducting the tuning study, frequently with the use of visualization tools [27,41,51,73]. Recently, research has focused on automating the diagnostic process (steps 1, 2, and 3) [20,28,45,50,70]. Several projects [30,25,44,49] are attempting to redefine performance tuning as an interactive run-time activity that may perform steering adjustments or debugging fixes as the application runs (step 4). Completing the spectrum, there is current research focused on complete automation of all four steps, by a tool that includes performance measurement, analysis, and code or environment adjustments to improve performance[66]. There is also research into self-tuning applications that include code to trigger changes based on measured performance values, either as a prelude to a full execution [75] or online as the application is running [31].

The tools detailed above address single iterations of one or more steps in the tuning process. Our approach is to provide a framework that can incorporate existing techniques for each step of the performance tuning cycle, and extend the scope to include repeated iterations. Hondroudakis and Procter [33,34] have proposed the term “tuning in the large” to refer to this complete process of tuning, as opposed to a single instance of running a performance tool with one iteration of a program. Their recent survey of high performance computing professionals cited the need for maintaining performance related information from tuning sessions. They proposed storing tuning information in a generally accessible database so that techniques used in tuning similar applications might be tried by others.

2.2 Automating Parallel Performance Diagnosis

A variety of research projects address the issues in automating the diagnosis of parallel programs. Some of this work is limited to proposed designs and architectures for tools that might automate the process; however there have been several tools implemented that at least partially automate the diagnostic portion of the tuning task. In this section, we survey these tools.

Cray developed two tools that provide automated diagnosis of performance bottlenecks. ATExpert [45] uses an expert system to analyze performance data gathered from their Autotasking Fortran compiling environment. Performance characteristics were matched to the source and to a set of performance rules, and the results were used to generate specific advice for the programmer tuning the code. Cray's MPP Apprentice [77] also generates diagnostic feedback and advice for the programmer, necessarily more general (and therefore providing less direct guidance to the programmer) than that of ATExpert because Apprentice is not closely tied to a parallelizing compiler. Both of these are post-mortem tools, that is, they generate some performance measurements during the application's execution and perform an analysis after execution has completed.

S-Check [50,71], a tool developed at NIST, uses a partially automated approach to diagnose performance-critical parts of large-scale applications. The tool uses artificially introduced delays together with the statistical technique known as Design of Experiments or DEX [4] to focus attention on synchronization points that are the cause or potential cause of performance bottlenecks. In their approach, called Synthetic-Perturbation Screening, parts of the experimental setup, such as narrowing down potentially fruitful locations to test, are done manually through a GUI. The tool provides a single metric, sensitivity to artificially introduced delays.

The relationship to our experiment management system is that our system might implement S-Check functionality as one of the types of experimentation.

Chitra [2,52] provides descriptions of the performance of multiple executions in the form of parameterized models using semi-Markov chains and CHAID-based models. The calculation uses data from summarized trace files. Aggregation, filtering, and reduction of model parameters provide visual feedback to the tool user that focuses attention to potential problem areas. This is one of the few multiple-execution performance tools.

Paradyn's Performance Consultant [54] performs an automated search through an application, inserting and removing instrumentation as it tests hypotheses to focus the programmers attention on a small set of performance bottlenecks. It is a completely automated, single button system that requires no user interaction to complete a diagnosis. We chose the Performance Consultant as the testbed for our technique of incorporating historical knowledge in diagnosis, described in Chapter 5.

The Projections:Expert project [70] developed partially automated performance diagnosis capabilities, via an automated postmortem analysis. The tool was developed for parallel programs written in the Charm object-oriented programming language. It narrows down the list of performance problems found by estimating the reduction in execution time that would result from removing each bottleneck, and outputs a phase-specific list of bottlenecks including location and cause.

Several architectures have been proposed for a tool to perform automated diagnosis. The Poirot project [28] proposed an architecture for a tool to automatically diagnose parallel applications across a range of platforms. The project ended before a prototype was developed; however the work contributed an analysis of existing diagnostic approaches and an approach to constructing a more general tool to include the methods found in several different tools. Their design included a store of application version data. The approach combines a tool interface and problem-solving environment within a knowledge-based system. Our approach differs in choosing as its foundation a scientific experiment management environment, and in addressing the inherent multi-execution nature of performance diagnosis.

KOJAK [20] (Kit for Objective Judgement and Automatic Knowledge-based detection of bottlenecks) is a proposed generic automatic performance analysis environment currently under development. This work builds on Gerndt's earlier work [19] that also described a design for an automated analysis tool, but was never implemented. The work is in early stages; a first component, a programmable tool for event trace analysis of message passing programs called EARL [78], has been implemented. Each specific type of analysis is implemented as a TCL script. They demonstrated scripts that calculate the execution time for each program region, locate out-of-order message-passing, and determine synchronization delays incurred by issuing an MPI_recv before issuing an MPI_send. New scripts may be added to target the tool to specific needs.

During the course of our work on using historical data in automated performance diagnosis, a multi-organization effort has started in Europe, the Esprit IV Working Group on Automatic Performance Analysis: Resources and Tools (APART). This project is focusing increased attention to the potential benefits of automating part or all of the currently time-consuming and difficult diagnostic process. The project is currently focused on defining the goals of performance analysis [67], and the types of bottlenecks typical to parallel codes written with OpenMP, MPI, or HPF [13]. The overall objective is a tool that will be used for completely automated performance diagnosis on a variety of common platforms. Such a tool will require some of the same infrastructure that we have been developing for our Experiment Management system, such as a representation for program resources. As members of the group, we are actively engaged in sharing our results and participating in their design. In contrast to the stated goals of the working group, we have not focused on developing an inclusive set of bottlenecks *per se*, rather we have focused on the effects of adding historical knowledge given some existing mechanism for locating bottlenecks.

2.3 Scientific Experiment Management

A variety of ongoing research efforts are attempting to provide flexible computer support for scientific experimentation. This work can be categorized by the part of the scientific experimentation process addressed: Some work focuses on support for the individual scientist in the lab or office, while other work focuses on techniques for the organization, storage and access for a large central data store to be accessed by a large number of distributed scientists.

Work on support for large-scale scientific data stores includes the DataFoundry project at LLNL[15]. Researchers are developing solutions for storing large collections of scientific data that includes the use of metadata, summary information that characterizes the much larger full set of stored data. The Extensible Computational Chemistry Environment (ECCE) project at Pacific Northwest National Laboratory [43] includes design and implementation of an Experiment Management system for computational chemists. The project includes a focus on scientific metadata, and attempts to provide a central store in which scientists record experimental methods as well as the (raw) input and output data. They use an object-oriented database system as the underlying storage engine. Their Basis Set Advisor stores information on different algorithms that have been used to calculate chemical results, and performs an automated “best fit” selection for chemists trying to solve related problems.

Ioannidis and Livny describe scientific experimentation and identify unmet technological opportunities [37]. Their desktop experiment management system, called ZOO[39,38], is a proposed solution for the needs of an individual scientist’s desktop. The researchers define a life cycle of local experimental studies, which iterates through experimental design, data collection, intialization request, data analysis, and follow-up. They point to the significant and expanded role that conceptual schemas must play in providing the rich set of data services needed for scientific experiment support. They elevate the role of schemas, so that they can be manipulated by the user for forming queries or used to display query answers. They chose an object-oriented data model for their database, which sets them apart from other major experiment management efforts[76]. While there is some overlap in functionality between this project and ECCE, ECCE focuses on issues related to serving as a national repository whereas ZOO focuses more on the process happening locally as a single scientist works. We discuss the potential for using the ZOO system in our work in more detail in Section 3.5.

The other main branch of experiment-related research falls under the heading of electronic notebooks. An Electronic Notebook System is “a system to create, store, retrieve and share fully electronic records in ways that meet all legal, regulatory, technical and scientific requirements” [7]. The goal of completely replacing scientists notebooks entails development of signature, verification, and security features to allow use of such data as evidence, as is currently possible with paper notebooks. The DOE 2000 Electronic Notebook Project [18] has developed electronic notebooks currently in use in a variety of sites, that enable sharing of heterogeneous data among geographically dispersed collaborators. They have focused on a simple notebook interface through web browsers, and handling non-text data such as sketches. They are developing a general-purpose information sharing utility, as opposed to our work which is targeted specifically to parallel performance tuning. However, a notebook-like approach is one possibility for an eventual GUI to our system.

2.4 Comparing Program Versions and Runs

Several research efforts have developed approaches for comparing different programs and program executions. In this section, we briefly survey approaches that might be incorporated into our structural and behavioral comparison operators.

The most closely related research effort to our own takes a multi-execution approach to correctness debugging, referred to as “relative debugging.” GUARD [72] compares the program currently being debugged with a reference version of the same program, to detect differences in variable values at user-defined points during the execution. Comparisons are made of simple or complex data types; the programs compared may be written in different languages or run across heterogeneous environments. Program comparison is limited to user-defined <variable name, location> pairs. They have defined difference operators for simple and complex data types that can compare complex structures and factor out machine dependent data representations. The difference operation determines only “different” or “equal”; there is no notion of quantifying the difference. This approach, while useful in a correctness debugging environment, is not sufficient for performance debugging.

Comparison of programs is inherent to the performance prediction problem. We seek to include performance predictions and models in the scope of our tuning approach by broadening our definition of Program Execution to include models and predictions. Some examples of predictive tools are the MK Toolkit [3] and MTOOL [21]. The MK Toolkit developed by Block and Sarukkai automates the predictive analysis task. They compare actual versus predicted total execution time. Their work does not include any effort to compare results across platforms, environments, or code changes. MTOOL includes a metric that compares observed and predicted execution times at the granularity of individual basic blocks. Their approach, useful only for memory tuning, defines the difference between predicted and observed execution time as a memory bottleneck.

A different approach to comparing program executions is found in the trace transformation approach to performance prediction of Mendes [53]. They represent a parallel program by a directed graph in which each node represents a trace event and each edge defines a happens-before relation as defined by Lamport[47]. A stable program is defined as a program for which the resulting execution graphs will always be identical or nearly identical. Similarity s between two graphs is defined as the degree of the largest isomorphic graphs that are induced subgraphs of each of the two original graphs. The distance d between the graphs is defined as $d = n - s$, where n is the number of vertices of each of the original graphs (so if the two original graphs are isomorphic, $d = 0$). This is a different approach from our method of comparing first structural, then quantitative differences.

Two program similarity metrics are described in work by Saavedra and Smith [68]; they developed a machine-independent model of sequential program execution to characterize both machine performance and program execution. They developed a benchmark that consists of abstract operations, and model the code on each platform based on the number of each type of abstract operator included in the code. They use two metrics for program similarity: program characteristic similarity, which is a normalized squared Euclidean distance; and execution time similarity, which uses a coefficient of variation of each variable, which represents time for a particular abstract operation. They defined thirteen parameters to be used in calculating program similarity, which are types of operations such as memory bandwidth and single precision addition and division. Their approach might be extended to develop a program-level performance distance metric for use in our experiment management system.

The Wisconsin Program Slicing Project at UW-Madison [35] has investigated methods for determining syntactic and semantic differences between two versions of a sequential source code. Foundational work [57,74] defined "the slice of a program with respect to a given component c" as the set of program components that might affect the values of the variables used at component c. Horwitz [36] described an alternative partitioning algorithm with the goal of identifying changed parts of a sequential pascal program from an old version to a new version, and classifying the changes as either semantic or textual. The difference between two code versions is defined as the number of semantically or textually changed components of the new version plus the number of new flow or control dependence edges in the graph representation of the new version. Wang [81] developed a comparison algorithm that detects syntactic differences between two sequential Pascal program versions by constructing and comparing a variant of their parse trees. The results are visualized using a simple textual side-by-side display of the two versions with differences color coded. Reps [64] investigated program integration, or merging different versions of a pascal program in a semantically acceptable way. The Slicing Project's work focuses on precise characterization and identification of the differences between two program versions. An interesting possibility for future research is to incorporate program slicing techniques into our tool as a means of performing a more precise structural difference of the code hierarchies.

Several research efforts have examined methods for using visualization of performance data to compare different versions of programs. Ribler *et al* at Virginia [65] have developed alternative methods for visualizing categorical trace data, including recent work on simultaneously visualizing multiple traces. IPS-2 [40] allowed visualization of data from different runs, matching them by normalizing to total execution time. Our previous work combined Paradyn and Devise [42] to enable side-by-side, linked visualization of data from multiple executions of a parallel application. Open problems remain, for example in matching phases of the program across two runs of different execution time.

The problem of mapping resources between different program versions is similar to the problem of mapping resource requirements to resources available. This problem has received recent attention because of the advent of metacomputing environments [10, 16, 23]. In such environments, mapping is used to fit a specified job request onto the available resources. Work by Brune *et al* at Paderborn Center for Parallel Computing [5,6,17] describes a language and API for Resource and Service Description (RSD). Their goal is to provide a cross-platform mechanism for specifying a resource request and for returning information about a machine platform, appropriate for use in a metacomputing environment. Gehring presents an algorithm for generating formal descriptions of the dynamic execution behavior of distributed programs, with the goal of using these in schedules, load balancers, or routing systems. The Resource and Service Description project has the goal of allowing resources and services to be specified for complex heterogeneous computing systems and metacomputing environments. This description language is designed to specify resources for both provider and requestor.

The classad matchmaking framework used for Condor [60] uses a semistructured data model to represent both potentially available resources and resource requests. (The term semistructured data refers to data that has some structure, but for which the structure is not as explicitly defined as in relational databases; one example is the data contained in BibTex files[1].) The classad task is complex because, in addition to resource attributes, there are constraints that must be met for a match to be made. These constraints represent a variety of conditions such as “only people in my research group can use my machine,” and they are independently specified by different parties with different goals.

2.5 Summary

In this chapter we have reviewed the state of the art in research areas related to our work: parallel performance tools, experiment management systems, and techniques for comparing program versions or executions. Viewing Parallel Performance Tuning as a specialized instance of scientific experiment management is a novel approach that has not previously been investigated. Similarly, creating an adaptive parallel performance tool that changes its approach based on historical application performance data has not been previously suggested or implemented.

Chapter 3

The Program Space

The *Program Space* is a representation for a collection of performance results and the various program versions, runs, and runtime environments to which they refer. The Program Space provides the functionality needed to name, store, and retrieve a large collection of data pertaining to an application's structure and behavior. It is directly useful to a developer studying an application's behavior, as a means of organizing and navigating a potentially large amount of data. Also, it can be incorporated into performance tools to allow past performance to be included in conclusions about current performance. At the topmost level, the Program Space provides a way to describe program executions that allows us to distinguish between different runs and versions of an application. At a fine-grained level, it provides a detailed description of each individual program run that allows us to refer to specific sub parts for which we have data. Finally, it provides a representation for the collected performance data.

In creating the Program Space we were motivated by the following design goals:

- to provide an intuitive user interface that highlights differences in application behavior between program runs, and helps relate them to differences in code and environment;

- to provide a uniform naming mechanism and interface for performance data that can accommodate a large quantity of heterogeneous, incomplete, and distributed data, and that can be used by a variety of visualization and data exploration tools;
- to address the dynamic nature of the data (new program runs and new types of performance results will be added over time);
- to allow an efficient and practical implementation to be built on top of commonly available database management systems;
- to provide answers to common performance related questions that cannot be addressed by tools that incorporate only a single program run. Such questions include: What is the scaling behavior of my code? How do the results I just gathered compare to other platforms? Did the performance of this function improve when we compiled with optimization, and if so, by how much? Are the performance requirements being met? Whats the highest amount of I/O waiting time seen in a run on more than 16 nodes? How accurate was my predictive model?

To address these goals we developed the Program Space. The Program Space contains data describing a collection of one or more Program Events. A *Program Event* is an individual complete or partial run of a program, simulation, or predictive model. Although there may be any number of Program Events represented in the space, all Program Events are from a related group of code versions (as defined by the user). We do not combine unrelated programs in one Program Space. The Program Events represented in one Program Space may be different repeated runs with identical parameters, or they may be runs performed using modified source code, different input data sets, or different platforms.

A Program Space comprises a SpaceMap, one or more EventMaps, and a collection of Performance Results. The *SpaceMap* contains descriptive data that characterizes Program Events, distinguishing the various Program Events described in the same Program Space. Examples of descriptive data stored in the SpaceMap are: input data set characteristics, platform description, code version number, or laboratory name. Each *EventMap* contains a detailed list of one Program Event's resources, such as individual function names and machine components. It serves as an interface to the performance results, and provides a naming scheme for the potentially large collection of performance data. *Performance Results* are measured values for an execution's behavior, for example, total CPU time for a specified function.

From one point of view, the SpaceMap and EventMap each contain different classes of information collected from different sources. From another view, they represent a unified body of attributes over which we can query (see Section 3.4.6).

We examine each of the three components of the Program Space — the SpaceMap, the EventMap, and the Performance Results — in more detail in the next three sections. In Section 3.4 we describe retrieving information from the Program Space, in Section 3.5 we discuss implementation considerations, and in Section 3.6 we present a chapter summary.

3.1 The SpaceMap

The SpaceMap is an organized collection of *metadata*: descriptive data that characterizes a collection of one or more Program Events. Examples are input data set characteristics, platform description, code version number, algorithm name, compiler options used, or laboratory name. Information useful in distinguishing a particular Program Event within the entire collection is considered metadata in this context. The list of attributes that serve as semantically meaningful metadata varies for different applications. For this reason, we do not require any particular attributes, or limit the total number of attributes. The only restriction on the SpaceMap is that each execution must be uniquely specifiable by selecting some combination of attribute values. This property is not restrictive in practice, since it may be enforced by adding the attribute “EID” and assigning a unique identifier as the value for each Program Event.

Each attribute stored in the SpaceMap labels one axis in the Program Space. The possible attribute values translate into coordinates in the resulting multi-dimensional space defined by these axes. Each execution is therefore one point in the space, with coordinates determined by its specific attribute values.

We illustrate this concept of a multi-dimensional space with a simple example. The Program Space pictured in Figure 1 contains data from four Program Events with the following attributes and values:

EID	PlatformSize	InputData	Code Version
1	8	A	original
2	16	A	original
3	8	A	newSolver
4	8	B	original

The first column shown contains the EID. Conceptually, if we zoom in on one particular point of the Program Space, we find an EventMap and Performance Results pertaining to a single Program Event.

Our characterization of the complete set of application data as a multi-dimensional space allows us to apply multi-dimensional database techniques such as the data cube operator to efficiently perform queries over the Program Space. The functionality of the data cube operator [22] allows efficient aggregation operators over an N-dimensional space.

In this simplified example, we have used simple types for attribute values: integer, character, and character string. However, in practice, values can be more complex user-defined or aggregate types. For example, rather than the simple identifiers A and B used in this example for inputData, we might describe the input data set used with a string name plus a description of the input consisting of the size of a grid being represented, the name of the data source, and the start time in seconds, plus the name of the file in which the actual input data set is located. In a heterogeneous environment, attribute “platformSize,” shown here as an integer, might instead be represented with a list of processor types and counts.

The SpaceMap serves as an interface to the data contained in the Program Space. Navigating the SpaceMap allows the user to visualize and query the Program Space, both its structure and its contents. Selecting a valid combination of values from the SpaceMap corresponds to selecting one or more Program Events with data stored in the Program Space. The result of such a selection is an EventMap display showing details of either a single program execution or a collection of executions, depending upon how many of the stored executions match the specification selected. We discuss extracting information from the Program Space in more detail in Section 3.4 and underlying implementation issues in Section 3.5.

3.2 The EventMap

The *EventMap* contains structural information about a Program Event gathered from compile- and execution-time information. As described in the previous section, the SpaceMap represents the experiment metadata. The EventMap, on the other hand, represents the

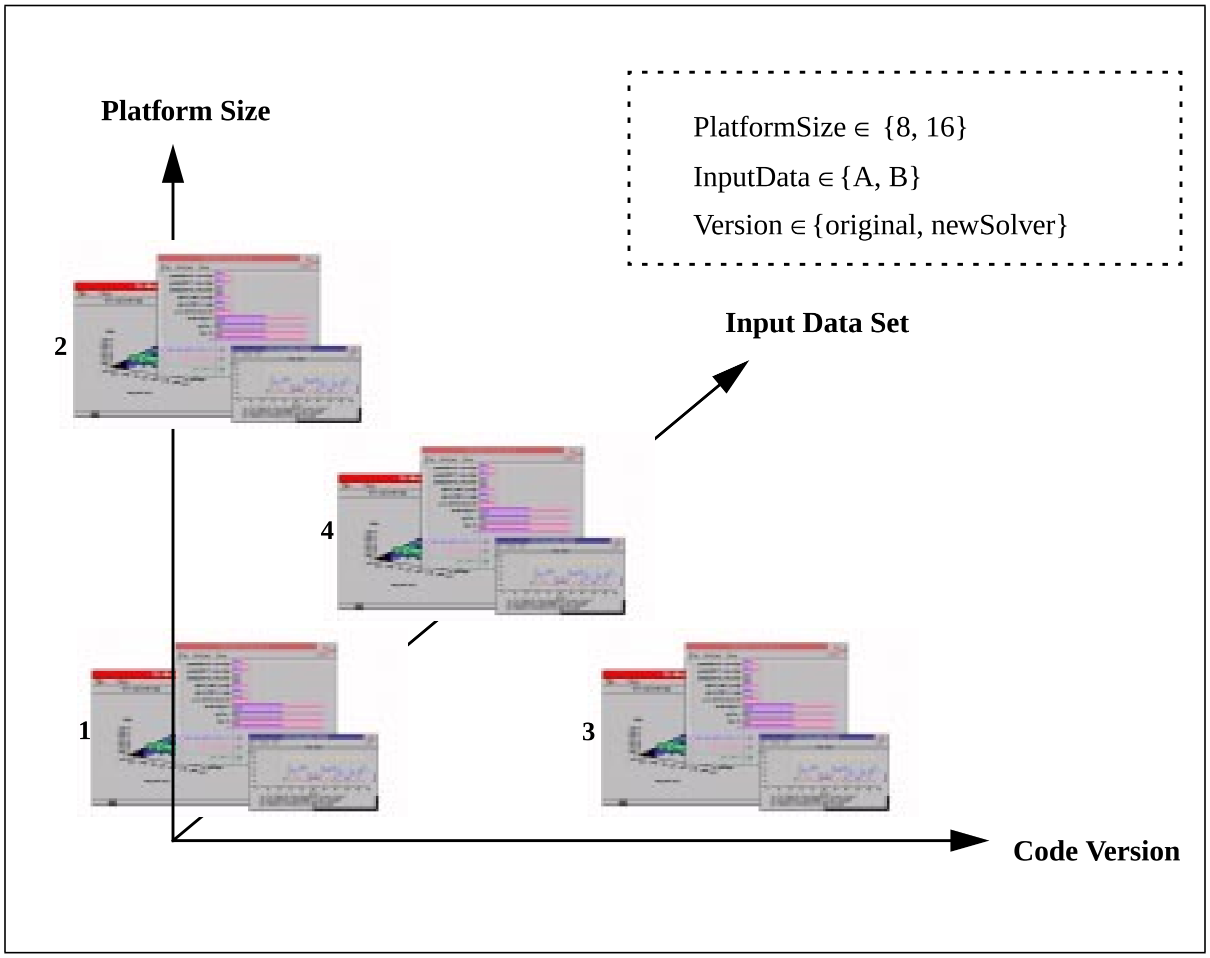


Figure 1: The Complete Collection of Data forms the Program Space. In this illustration each cluster of data visualizations represents all of the data, both EventMap and Performance Results, for an application. The numbers refer to the "EID" column of the table. The inset shows the contents of the SpaceMap for this Program Space.

components of a program execution: the program's structure and execution environment. A Program Space contains one EventMap for each Program Event it describes. It represents the program structure and serves as an index to the Performance results. As new Program Events are added to the Program Space, new EventMaps are created, adding data for one Program Event at each step. Definitions related to the EventMap are listed in Figure 2.

We organize the program resources into classes according to the aspect of the application they represent, and structure each class as a tree, called a *resource hierarchy (*similar to the representation used in Paradyn [54]). Possible program resources include the program code, application processes, machine nodes, synchronization points, data structures, and files. Each class of resources provides a unique view of the application. For example, a Code hierarchy provides a source code based view of the application, while a Machine hierarchy provides a view of the application runtime environment.

Definition 1. An *EventMap, E,* is a forest composed of zero or more unique resource hierarchies:

EventMap E = $\{R_0, R_1, \ldots R_n\}$. ❑

Definition 2. A *resource hierarchy* R is a tree of the form

$R = (r, T)$

where r is a resource and T is the set of all children of r in the resource hierarchy:

$T = \{(r_0,T_0), (r_1,T_1), \ldots (r_m,T_m)\}$. ❑

Definition 3. A *resource r* is a pair

$r = (l, X)$

where l is the resource label and $X \in \{e_0, e_1 \ldots e_i\}$, a set of unique event identifiers. ❑

Figure 2: Definitions related to the EventMap.

A resource hierarchy is a collection of related program resources. The root node of each resource hierarchy represents the complete program execution and therefore we label it with the name of the entire resource hierarchy. Each descendant of the root node represents a particular program resource within that view. As we move down from the root node, each level of the hierarchy represents a finer-grained description of the program. For example, a code hierarchy might have one level for nodes that represent modules, below that a level with function nodes, and below that a level for loops or basic block nodes.

In the resource hierarchy of Figure 3, the root level (level 0) is the program-level view of an application, which represents the behavior of the whole program. Level one is the module-level view of the source code, and level two is its function-level view. Each level of a

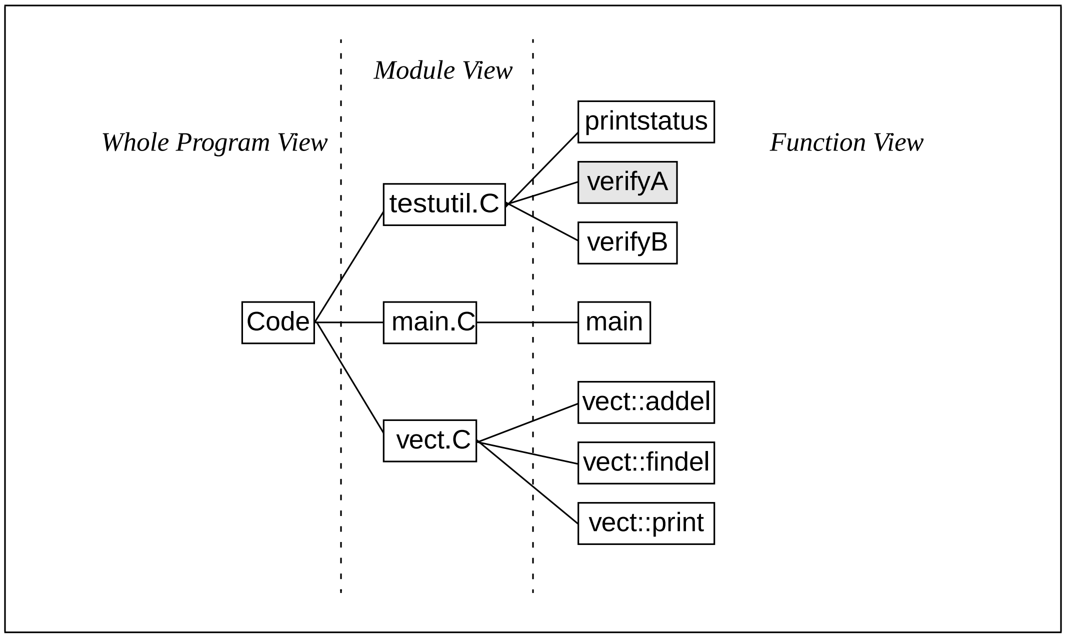


Figure 3: Each level of a Resource Hierarchy represents a different view of the application.

resource hierarchy is a set of resources, and each level above the leaf level is a partition of the set of nodes in the next lower level. For example, the module level of Figure 3, which contains testutil.C, main.C, and vect.C, is a partition of the set of all leaf nodes. We specify a particular level of a hierarchy using a superscript notation: R_0^1 refers to level one (the children of the root node) of resource hierarchy zero. The Code hierarchy in Figure 3 shows the set of functions partitioned into modules:

R_0^0 = {Code}

R_0^1 = {testutil.C, main.C, vect.C}

R_0^2 = {printstatus, verifyA, verifyB, main, vect::addel, vect::findel, vect::print}.

Figure 4 shows a sample EventMap for a parallel application called *Tester*. The EventMap is the set {Code, Machine, Process}. The Code hierarchy contains nodes that represent the program's modules and functions; the Machine hierarchy contains one node for each CPU on which Tester executed; and the Process hierarchy contains one node for each process. The resources and hierarchies shown here follow the pattern of Paradyn data, though we can easily incorporate data from other environments. For example, it may be natural to have process resources as children of machine resources.

Each resource is a representation of a logical or physical component of a program execution. A single resource might be used to represent a particular aspect of the program or the environment in which it executes: a process, a function, a CPU, or a variable. In Figure 4, the leaf node labeled "verifyA" represents the resource "function verifyA in module testutil.C." The semantic meaning attached to a particular resource is relevant only to the tool user and does not affect the model functionality; however, a program execution component must be uniquely represented by a single resource. Each internal node of a resource hierarchy tree represents a set of one or more resources; for example, testutil.C is a single resource that represents the aggregation of the set {printstatus, verifyA, verifyB}, and we define a measurement of CPU time for testutil.C as the sum of CPU time for printstatus, verifyA, and verifyB.

A *resource name* is formed by concatenating the labels along the unique path within the resource hierarchy from the root to the node representing the resource. For example, the resource name that represents function verifyA (shaded) in Figure 4 is </Code/testutil.C/verifyA>.

3.3 Performance Results

Each performance datum collected during a program execution is stored in the Program Space as a tuple in the form [*e, m, f, t, r*], where:

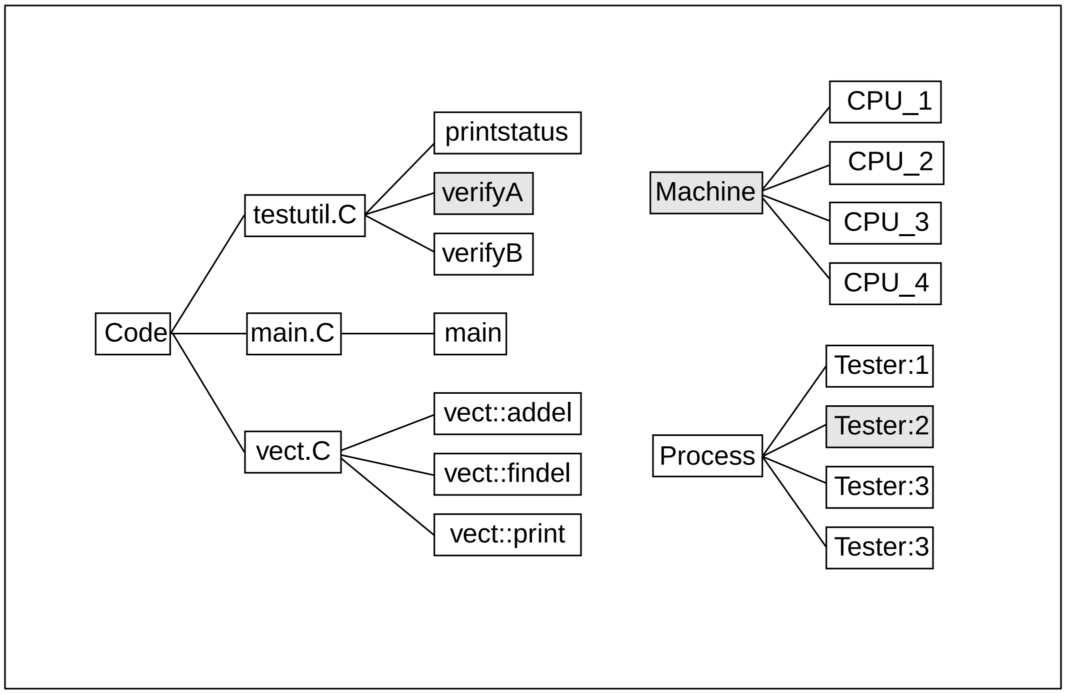


Figure 4: EventMap for Program *Tester*

- *e* is a unique Program Event identifier or *EID*;
- *m* is the name of the *metric*, which is a measurable execution characteristic, such as CPU time;
- *f* is the *focus*, a constraint specification that narrows the scope of the data to a particular part of the application, such as one function;
- *t* is the *time interval*, which specifies the time during an execution the data was collected; and
- *r* is the *performance result*, which may be a scalar or more complex object. Each performance result is uniquely identified by the EID, metric, focus, and time interval, and is represented PR(*e, m, f, t*). The collection of performance results may be heterogeneous, and results for even a single program execution may include different types of data such

as time series and traces. In practice there is rarely a complete set of Performance Results, so PR may be ∅ (null) for some valid combinations of e, m, f, and t.

We want our system to be extensible in the types of performance results that it may include. For this reason, we allow new types of performance results to be added, with access methods serving as a uniform interface.

3.3.1 The Focus

For a particular performance measurement, we need the ability to specify the particular parts of a program to which it applies. For example, we may be interested in measuring CPU time as the total for an entire program run, or as the total for a single function. We constrain our view of the program to a selected part with a *focus* (see Figure 5). A focus is a selection of resources from an EventMap that follows certain restrictive rules. Selecting the root node of a resource hierarchy represents the unconstrained view, the whole program. Selecting any other node narrows the view to include only those leaf nodes that are descendents of the selected node. For example, the shaded nodes in Figure 4 represent the constraint: functions verifyA and verifyB of process Tester:2 running on any CPU. Between hierarchies, a focus defines the state where the condition selected in each hierarchy is true at the same time. For example, selecting machine_1 and function_foo results in narrowing the scope to data pertaining to function_foo only as it ran on machine_1.

Because our naming of performance results is based on the EventMap's resource hierarchies, we call the naming scheme *resource normal form*. We convert a selected set of resource nodes to resource normal form by concatenating the selections from each resource hierarchy. For example, the shaded selection of Figure 4 is represented as:

Definition 4. A *focus F* is formed by selecting one resource node from each of the resource hierarchies in an EventMap:

$\{r1 \in R_1^{j1}, r2 \in R_2^{j2}, r3 \in R_3^{j3}, \ldots rn \in R_n^{jn}\}$

where *jx* is a level of resource hierarchy R_x and *n* is the total number of resource hierarchies. The nodes selected may be in different levels of the different hierarchies.❑

Figure 5: Definition of a Focus.

< /Code/testutil.C/verifyA, /Machine, /Process >.

We call the focus that results from selecting the root node of each resource hierarchy the *root focus*.

3.4 Retrieving Information from the Program Space

Once we have populated a Program Space with data from a collection of Program Events, we can view and extract that data in a variety of ways. Here we discuss some basic queries and a few specialized operators: structural difference, discrete distance, and performance difference. Querying and viewing data from more than one execution can take the form of asking the same question of more than one execution and comparing the answers. One of the features of the Program Space structure is that it also allows us to ask questions that are inherently multi-execution. We defer a discussion of implementation of the underlying storage and query facility to Section 3.5.

3.4.1 Choosing Program Events Using the SpaceMap

First we consider questions asked by making selections over the collection of attributes and values in the SpaceMap. Queries over this metadata are made by selecting one or more particular attribute values from the SpaceMap. The result is in the form of a filter over the Event-

Map that screens out all but the Program Events with the selected values. In our example of Figure 1, selecting “Platform=8, InputData=A, CodeVersion=newSolver” yields an EventMap representing a single Program Event, Program Event 1, whereas selecting “Platform=8” yields an EventMap representing Program Events 1, 3, and 4. Note that there may be more than one selection for a given result, for example, Program Event 1 is also the result of selecting “CodeVersion=newSolver.”

Questions of the form “How do these results compare to other platforms?” are asked by specifying the different platforms, including the most recent, from the SpaceMap, then specifying the particular metrics you want to compare.

3.4.2 Combining EventMaps with the Structural Merge Operator

If a selection made from the SpaceMap refers to more than one Program Event, the answer is computed by merging two or more EventMaps to form a single representation for the multiple Program Events. We combine two or more EventMaps with the *Structural Merge Operator (+)*. The result is a single, merged EventMap that represents two or more distinct program executions.

To calculate the structural merge of two EventMaps, $E_1 + E_2$, we compare the two sets of resource hierarchies in a top-down manner. First we isolate those resource hierarchies that are members of both program executions by matching up common pairs of root nodes. For each such pair, we do a level by level comparison of the two resource hierarchies, continuing in a top down manner following the rule: if two resource hierarchy nodes match, we merge them into one node in the result, then check the children of the two nodes, until either the leaf nodes have been examined or we fail to find a match for a node. When a node without a match is identified at any level of the hierarchy, the entire subtree rooted at the node is added to the result and labeled with the execution's label. Applying the structural merge operator to two EventMaps yields a single EventMap. This result contains all resources from the original two EventMaps, so the structural merge operation works as a hierarchical set union operation. Some or all of the resources in the result EventMap are merged resources and represent resources found in multiple executions.

```
+ (EventMap $E_1$, $E_2$ ) returns EventMap
[1]  $E \leftarrow \{ \}$
[2]  $\forall (r_i, T_i) \in E_1$
[3]        **if** $\exists (r_j, T_j) \in E_2$, s.t. match $(r_i, r_j)$ **then**
[4]              $E \leftarrow E \cup \{ r_i + r_j, T_i + T_j \}$
[5]              $E_2 \leftarrow E_2 - (r_j, T_j)$
[6]        **else** $E \leftarrow E \cup \{(r_i, T_i)\}$
[7]  $E \leftarrow E \cup E_2$
[8]  **return** E
```

Figure 6: Algorithm to find the Structural Merge of two EventMaps, $E_1 + E_2$

The *Structural Merge Operator,* +, takes two EventMaps as operands and yields an EventMap. Figure 6 shows the algorithm for the structural merge operator, which forms $E = E_1 + E_2$ given two EventMaps $E_1 = \{(r_0, T_0), (r_1, T_1), \ldots (r_n, T_n)\}$ and $E_2 = \{(r_0, T_0), (r_1, T_1), \ldots (r_m, T_m)\}$. Note that it is not required that $n = m$. We define a merge operator + for individual resources in Figure 7. The comparison function used to determine if two resources match may be defined for each resource hierarchy, and therefore may use attributes that are unique to a particular hierarchy, as well as common attributes. As an example, we show a simple match function in Figure 8. This match function uses resource string labels as the basis for determining resource difference. We discuss resource matching further in Section 3.4.5.

```
+ (resource $r_{0,}$ $r_1$ ) returns resource
[1] given $r_0 = (l_0, X_0)$, $r_1 = (l_1, X_1)$:
[2]  $r \leftarrow (l_0, X_0 \cup X_1)$
[3]  return r
```

Figure 7: Merging Two Resources, $r_1 + r_{2.}$ Only matching pairs of resources are merged.

The structural merge operation has the following properties:

- commutativity: $\mathbf{E_1 + E_2 = E_2 + E_1}$
- associativity: $(\mathbf{E_1 + E_2}) + \mathbf{E_3} = \mathbf{E_1} + (\mathbf{E_2 + E_3})$
- idempotency: $\mathbf{E_1 + E_1 = E_1}$

It can be applied iteratively to build up a single EventMap that contains all of the resources for any number of distinct Program Events.

```
match(resource $r_{0,}$ $r_1$ ) returns boolean
[1] if $l_i = l_j$ then
[2]          return true
[3]  else return false
```

Figure 8: Algorithm *match* (($r_1 = (l_i,X_i)$, $r_2 = (l_j, X_j)$))

Figure 9 shows an application of the structural merge operator to two EventMaps E_1 and E_2. The top set of resource hierarchies describes execution E_1, and the middle set describes E_2. The bottom of the figure shows the result. Resources common to both executions are lightly shaded; <Code> and <Code/Module2> are examples of such resources. Resources unique to E_1 or E_2 are drawn darkly shaded or clear respectively; for example, < Code/Module2/Bar > is unique to E_1 and < Code/Module2/Car > is unique to E_2. Note the results for the Semaphores and Messages hierarchies: although both have children labeled “one” and “two”, we do not want to merge these nodes; because the root nodes are different the Semaphores and Messages hierarchies are not considered equivalent and non-equivalent hierarchies are never merged.

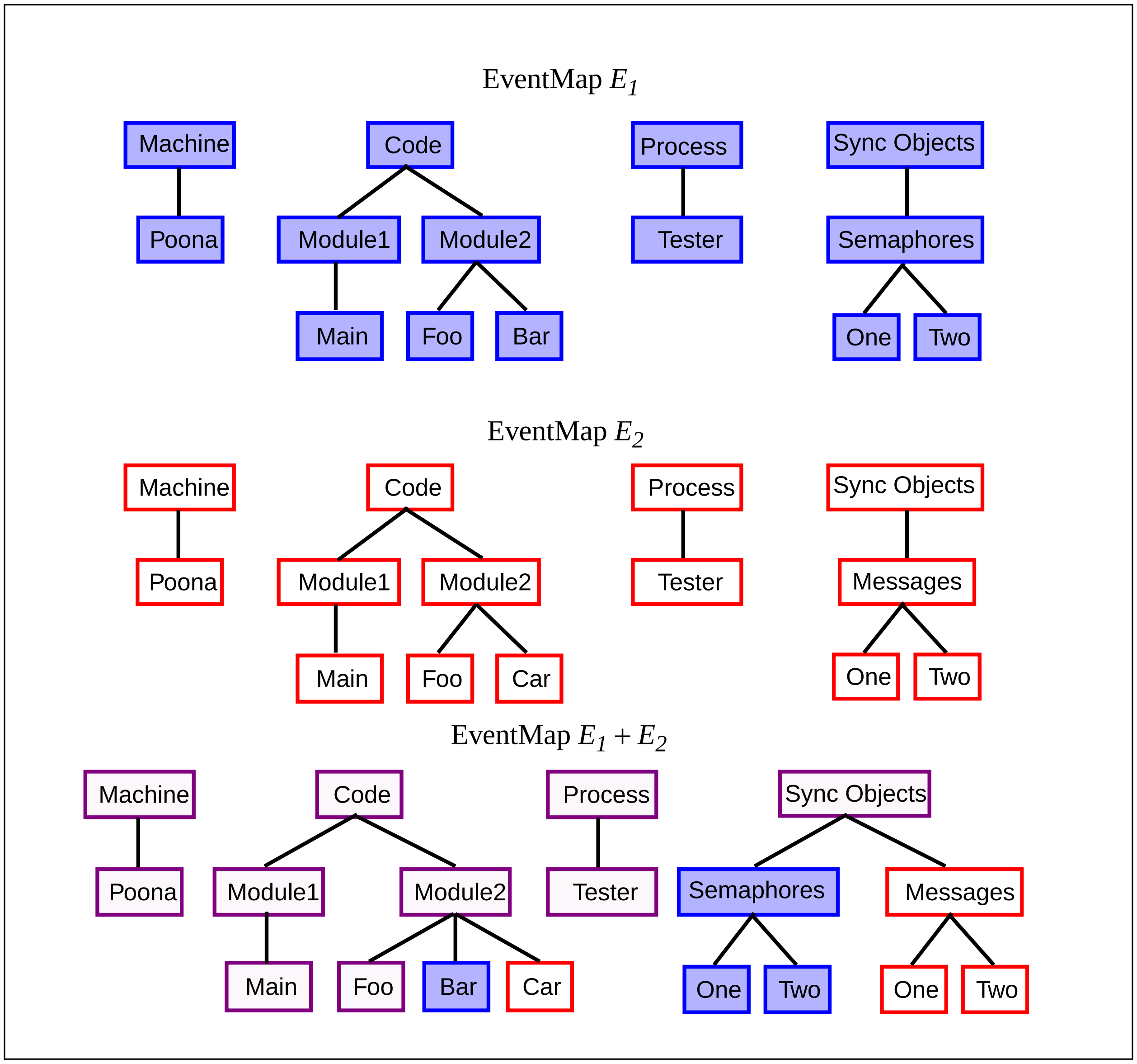


Figure 9: An Example of the Structural Merge Operator

3.4.3 The Structural Difference Operator

The Structural Difference Operator compares two or more EventMaps and returns a list of resources that do not occur in all of the EventMaps. Given our overall goal of comparing the performance of two or more program executions, a natural first question is, how different were the code and environments used in the two tests, and where did they differ? If we perform two test runs using identical code running on identical, dedicated platforms, every resource hierarchy of the first execution has an identical counterpart in the second execution; performance data may be meaningfully compared for every focus that is valid for one of the individual executions. The comparison becomes more complex if we consider cases in which either the code or the run time environment (or both) differ between our two test runs. We need to determine the common set of valid resources. This is accomplished with the Structural Difference Operator. From the SpaceMap, the user can invoke the Structural Difference Operator. This operator takes as input two or more EventMaps and returns an EventMap that contains only nodes that are not valid for at least one of the input EventMaps. The Structural Difference Operator is implemented using the Structural Merge Operator (see Section 3.4.2). The result of the merge is filtered so that only the differing resource nodes are displayed. We present examples of the structural difference operator in Chapter 4.

3.4.4 The Performance Difference

The *performance difference* operator answers the question, how did performance vary between these two different Program Events? It automates the otherwise overwhelming task of detecting all performance changes between two potentially large datasets. It takes as inputs a merged EventMap, filters out resources that are not valid for all of the included Program Events, then returns all foci for which the discrete distance metric (see Figure 10) yields an answer of true. The list is computed hierarchically as shown in Figure 11. We iterate through the resource nodes of the EventMap, starting with the focus that represents the entire program execution, the root focus. If there is a performance difference noted, we then check more specific foci for the same metric. This metric is useful to draw attention immediately to performance changes from one version to the next.

$$dd(x,y) = \begin{cases} true & if \quad |x-y| \geq \delta \\ false & if \quad |x-y| < \delta \\ undefined & if \quad (x = \emptyset) \vee (y = \emptyset) \end{cases}$$

Figure 10: The Discrete Distance Operator is a binary function that indicates whether or not two specified performance results differ by more than a specified interval. Here x and y are two different performance results. The Discrete Distance is undefined if either performance result is null. We use the discrete distance metric as a building block for clustering and differencing of performance results.

```
perfdiff(EventMap , metric ) returns set of focus
[1] answer ← {}
[2] enqueue (pendingQueue, EventMap->getRootFocus()
[3] while ! (isEmpty (pendingQueue))
[4]         currentFocus ← dequeue (pendingQueue)
[5]         pr1 ← PR($E_1$, m, currentFocus, t = all)
[6]         pr2 ← PR($E_2$, m, currentFocus, t = all)
[7]         If dd(pr1, pr2) = true
[8]                 answer ← answer ∪ {currentFocus)
[9]                 for each f in magnify(currentFocus)
[10]                        enqueue (pendingQueue, f)
[11] return answer
```

Figure 11: Algorithm for the Performance Difference Operator perfDiff (($E_1 + E_2$,) – ($E_1 \oplus E2$), m). The performance difference operator searches through the performance results for the specified metric *m* and the requested executions *E1* and *E2*, returning a list of all foci for which the results are different. The input is an EventMap and a collection of performance results, and the output is a set of foci.

With this approach, it is possible for detailed performance differences to go undetected. This situation could occur if two or more performance differences cancel each other out, so that the higher-level focus tests false for performance change. For example, CPU time might be higher for one function and lower in another, with a net change of zero for the entire module. In this case, comparison of the two modules will show no performance difference, and we will not test the individual functions at all. The trade-off here is between the performance gain from stopping the testing earlier, and a possibility for false negative results. We will re-examine this issue after we have more experience with the performance difference operator applied to large scale applications.

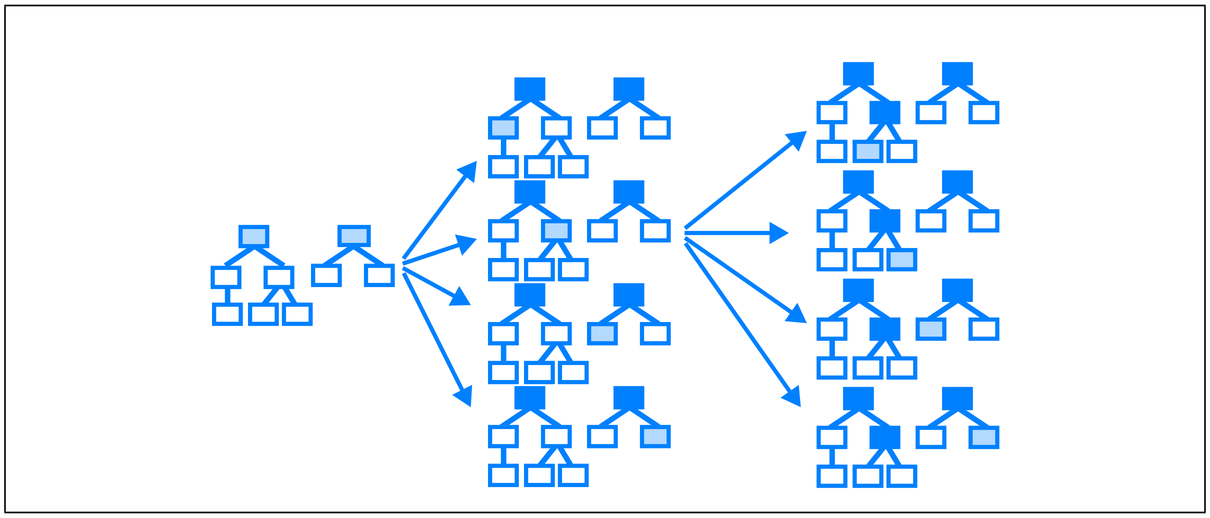


Figure 12: The focus provides a partial ordering of the data. This diagram demonstrates ordering based on the focus. The original nodes are darkly shaded and the newly added nodes are lightly shaded. The focus on the left, the root focus, represents the whole program. The middle group contains foci formed by taking a single step from the root focus, along each of the possible paths. The group on the right shows the foci formed by taking a single step from one of the four foci in the middle group. Stepping through the set of all possible foci in this way is called magnification.

We construct more specific foci using an operation we call magnify (see Figure 13). As specified in Definition 4, a focus contains one node from each resource hierarchy. For each resource hierarchy in the original focus, we form a set of new foci by replacing the given resource with the each of its children in the next lower level of the hierarchy. The result of the magnify operation applied to a focus is a set of foci.

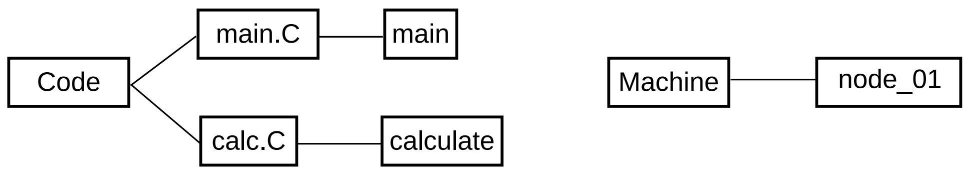
```
    given focus $f = \{r1 \in R_1^{j1}, r2 \in R_2^{j2}, r3 \in R_3^{j3}, \ldots rn \in R_n^{jn}\}$:
[1] $answer \leftarrow \{\}$
[2] for $i = 1$ to $n$
[3]     $f \leftarrow f - n$
[4]     for each $j$ in children($ri$)
[5]         $answer \leftarrow answer \cup (f + j)$
[6] return $answer$
```

Figure 13: Algorithm for magnify(*f*). Magnify returns the set of foci constructed by making all possible one step descents down the resource hierarchy from the given starting focus.

To ask questions such as “Are the performance requirements being met?” and “How accurate was my predictive model?” you select the Program Event you are testing plus the Program Event that represents a simulation or previous run, then apply the performance difference operator. To ask the question “What is the scaling behavior of my code?” you might select the different platform sizes from the SpaceMap, and apply the performance difference operator for the relevant metric (probably Execution Time, although you might also select something more specific).

We designed the Performance Difference Display to focus a tool user’s attention to the particular parts of the application for which behavior changed from one run to the next. Each node represents a metric-focus pair with differing performance. Metric-focus pairs with equal performance results are omitted. In this way, a potentially large amount of information about application performance is distilled so the user is not overwhelmed with the full set of data. To illustrate the performance difference operator and display, we present a simple example. We

CONTENTS OF EVENTMAP



APPLICATION DATA

STEP	MAGNIFICATION AXIS	NEW FOCUS
1	Machine	/Code,/Machine/node_01
2	Code	/Code/calc.C,/Machine/node_01
3	Code	/Code/calc.C/calculate,/Machine/node_01

FOCUS	CPUTIME	IOTIME
oldAlgorithm: /Code, /Machine	5 sec	.2 sec
newAlgorithm: /Code, /Machine	4 sec	.2 sec
oldAlgorithm: /Code/main.C, /Machine/NeverDown	1 sec	.2 sec
newAlgorithm: /Code/main.C, /Machine/NeverDown	1 sec	.2 sec
oldAlgorithm: /Code/calc.C, /Machine/NeverDown	4 sec	0 sec
newAlgorithm: /Code/calc.C, /Machine/NeverDown	3 sec	0 sec
oldAlgorithm: /Code/main.C/main, /Machine/NeverDown	1 sec	..2 sec
newAlgorithm: /Code/main.C/main, /Machine/NeverDown	1 sec	.2 sec
oldAlgorithm: /Code/calc.C/calculate, /Machine/NeverDown	4 sec	0 sec
newAlgorithm: /Code/calc.C/calculate, /Machine/NeverDown	3 sec	0 sec

Figure 14: Explanation of the Performance Difference Display. We show a complete set of information for our example application. The top box shows the resources of the merged EventMap (in this example the two EventMaps are identical). The middle box details the particular refinement steps. The bottom box shows the performance data.

show all of the structural and performance data for two Program Events in Figure 14. In the top box, we show the EventMap. The code is contained in two modules, main.C and calc.C, and two functions, main and calculate. The application ran on a single machine node. The

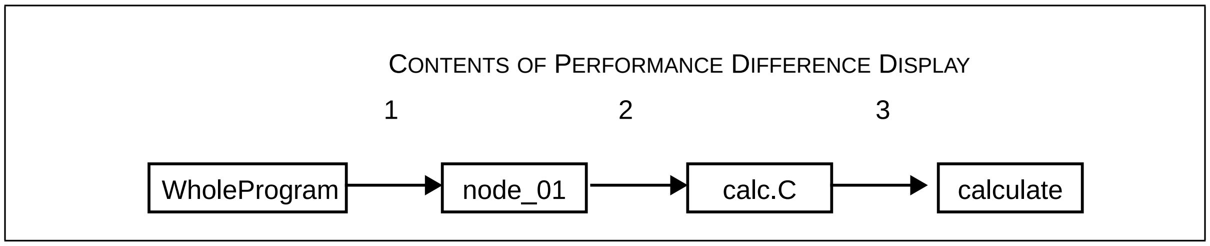


Figure 15: Sketch of performance difference display contents for example of Figure 14. We have numbered each step to match the textual explanation.

middle of the figure shows the focus magnification that resulted in each pair of Performance Results being checked for a difference. The Program Space contains performance data for metrics CPUtime and IOtime, for two program versions. The second version is the result of attempting to optimize the function calculate; the code for function main was not changed at all between versions. performance data for this example is shown in the bottom of the figure. In Figure 15, we sketch the contents of the Performance Difference Display that results from applying the performance difference operator for the metric CPUtime to the two Program Events shown. We have annotated the display by numbering the steps. The starting point is the root focus, represented here by the left most node labeled “WholeProgram.” Step 1 in Figure 15 represents a magnification along the Machine hierarchy, to node_01; step 2 represents a magnification along the Code hierarchy to calc.C; and step 3 represents further magnification along the Code hierarchy to function calculate. We label each node with the name of the resource that results from the magnification step taken to reach it. The Program Space contains performance data for metrics CPUtime and IOtime, for two program versions. We show a snapshot of a Performance Difference Display with our prototype in Chapter 4.

3.4.5 Matching and Mapping Resources

In the previous section, we described the structural merge operator, and its use in general queries on two or more EventMaps. The matching technique we described is appropriate for simpler cases, however, it is not sufficient for all cases. In practice, the particular resources used often change from one run of a program to the next. For example, an 8-node application might run on nodes 0-7 of a machine during one run and on nodes 123-130 of the same machine on the next run. Similarly, process ID’s are likely to be different for each run. If we are to relate performance results from a previous run to the current run, for example by using the performance difference operator, we must be able to establish an equivalency between (*map*) the differently named resources. Mapping allows us to link resources from different executions with different names, and treat them as equivalent. We have experimented with preliminary forms of mapping, and believe that this is an area that warrants further study. Here we discuss our current approach to mapping.

There are different approaches to resource mapping. One approach is to map conservatively. One-to-one mappings between resource nodes that match exactly fit into this category. Another approach is to map more aggressively, allowing mapping between nodes that are slightly different, including one-to-many or many-to-many mappings. Which approach to use is determined by the performance-related task being performed. For example, if we are interested in learning what has changed structurally between two runs of a program, conservative mapping will yield the most inclusive list of changes. We might determine that performance is differing for all program runs that include a particular machine node, pointing to a localized failure or congestion. In constrast, when comparing the performance of two program runs, using an aggressive mapping strategy will result in performance comparisons for a greater number of pairs of resources. Mapping different machine nodes with an inexact, categorical definition allows performance per node to be compared; mapping different versions of a function will allow the performance change that resulted from the code change to be determined.

When we apply the structural merge operator to two resource hierarchies, we invoke a *match* function on each pair of resources to find the nodes that differ. Resources that remain unmatched after this step can be manually mapped. All mappings are from one EventMap to a single virtual EventMap. In this way, we avoid the exponential explosion of maintaining a list of mappings for each pair of EventMaps in the Program Space. Mapping of individual resources is specified as a set of directives of the form “map *resourceName virtualResourceName”* specified by the user in an input file or through a GUI by selecting pairs of resources to map. The virtual resource name refers to one name that will be used to refer to all resource nodes with a mapping to the same node, regardless of the original resource name. Applying a complete set of mappings to an EventMap yields a virtual EventMap, each node of which may map to several differently named nodes from distinct Event Maps. This approach allows us to use only one list of mappings for each Program Space.

We allow manual mapping of resource nodes, however, we also want the ability to map certain kinds of resource nodes automatically. Consider the Machine hierarchy. We might define a match function (=) based on attributes, say machineNode1 = machineNode2 if machineNode1.cpu = machineNode2.cpu and machineNode1.os = machineNode2.os. We can map the two Machine hierarchies as follows: (1) independently divide the child nodes of each hierarchy into equivalence classes. This can be performed by adding an intermediate level to the hierarchy, one node for each resulting equivalence class. (2) Test selected pairs from each hierarchy against each other for equality, matching up the equivalence classes into pairs, one from each hierarchy. (3) If all of the classes pair up (that is, there is an equal number of equivalence classes for each of the two original resource hierarchies), we check the cardinality of each equivalence class. If the cardinality is equal for each pair, we consider that pair of classes *mappable*. (4) For each mappable pair of equivalence classes, generate a list of pair wise mappings.

3.4.6 Making Selections from the EventMap

Performance Results are retrieved by selecting a focus from the EventMap and a metric from the list of available metrics for that focus. Selecting a valid focus from the EventMap acts as a filter over the list of available metrics. The answer will contain one or more performance results. The user performs queries over the performance results in this uniform manner, regardless of the particular types of performance results stored for the requested metric/focus; the answer to a query may mix results from different types of performance results. The particular EventMap used is retrieved by first selecting one or more particular Program Event(s) via the SpaceMap.

To ask questions of the form "Did performance of function foo improve when we compiled with optimization?" you would select the runs with optimization levels of interest from the SpaceMap, then select function foo from the resulting EventMap, and the relevant metric (perhaps CPU time) from the list of available metrics. The answer will include performance results for function foo, for all of the Program Events with the selected optimization levels. To ask "What's the highest I/O waiting time we've seen when we run on more than 16 nodes," select all runs with values of platform size greater than 16 from the SpaceMap, select the I/O waiting time metric, then aggregate over the resulting list of I/O waiting time values. To ask the question “What is the scaling behavior of my code?” you might select the runs with different platform sizes from the SpaceMap, the root focus from the EventMap, and the relevant performance metric, for example, Execution Time.

3.5 Implementation Considerations

As with most real world problems, there is more than one approach to designing underlying database support for the Program Space. Here we discuss the relevant issues. First we briefly survey the features of two Experiment Management Systems, and compare them to the needs for our experimental performance environment. Next we present a design for implementing the Program Space. In Section 3.5.3 we briefly discuss other possible approaches.

3.5.1 Existing Experiment Management Systems

We looked at two existing approaches to providing database support for scientific experimentation: the ZOO project at UW-Madison and the OPM project at Lawrence Berkeley Laboratory. Each of the projects researched the particular database needs of scientists running experiments[8, 38], and there is much similarity in the reported results. Not all of the features mentioned by the researchers are implemented in their released tools, however, all were mentioned as particularly important to a full implementation of an experiment management system. Here we list the features related to our performance environment and comment on their applicability.

Schemas as commodity objects. In both ZOO and OPM, schemas can be manipulated, shared, and reused by the scientist. They are stored as objects in ZOO’s meta database, and can be accessed through a web browser in OPM. This is a radical departure from the traditional database approach, in which the user is not given direct access to the schema. Our performance tuning environment shares the need for available schemas, to enable the schemas to evolve over the course of experimentation.

Database sharing. Both projects mention the need for databases to be shared by different scientists and projects. Also a single experiment may involve data from multiple sources. This sharing directly applies to our performance environment.

Queries can span multiple databases. Allowing queries to span databases would enable different application users and developers to each have their own experimental results locally, yet still have the ability to use results from other laboratories. (See Section 3.5.3.)

Mechanism for launching new experiments. This feature could usefully be added to our performance environment, although it is not part of the current design. Experiments in our environment are program or simulation runs with monitoring tools in place. A simple example of the practicality of automating their launch is that a scalability study could be done with the push of a single button.

Mechanism for converting data from ASCII files into database objects. In our environment this means the ability to add new Program Events to the database. Resource hierarchies and performance results may all be input from ASCII files.

Levels of integrity enforcement to allow incomplete information or human error. This is a must in our performance environment, which should allow for metric/focus pairs without any data at all, as well as gaps in data for different time intervals in a single Program Event. For example, the Performance Difference Operator must allow for the condition that a focus is valid for the two Program Events being compared yet no performance results exist for that focus.

Versioning. Part of the potential for our extensible resource definition approach is that resources representing source code can be enhanced to include the actual filenames or versions of the files. Integration with functionality of a revision control system could be incorporated into the match function for such resources, and would be a useful extension.

Object oriented features such as ZOO's data model (MOOSE) and query language (FOX). We also found a need for object oriented features, as described in more detail in the following section.

In summary, we found much overlap between the needs of a performance study and the needs determined for scientific experimentation. In principal, the storage engine for our system might be implemented using either ZOO or OPM. We determined this to be impractical, however, because neither project has produced a complete version with all of the features needed for us to fully implement our tool. To illustrate the relationship of the Program Space and ZOO, we present a design for our experiment management system implemented over ZOO.

A schema for the Program Space is shown in Figure 16. We follow the representation used in the ZOO documentation [76], with the addition of ovals to represent abstract data types and user defined methods. The three main classes are EventMap, SpaceMap, and Performance Result. Each new type of performance result stored in the database requires a new ADT be defined for it, as a child of type PerformanceResult, with method functions for common lookup features such as getValue (startTime, endTime) and getMaxValue(startTime, endTime). This enables queries to be performed over a heterogeneous collection of data.

Here are examples of queries we have been discussing, written in ZOO's Fox query language.

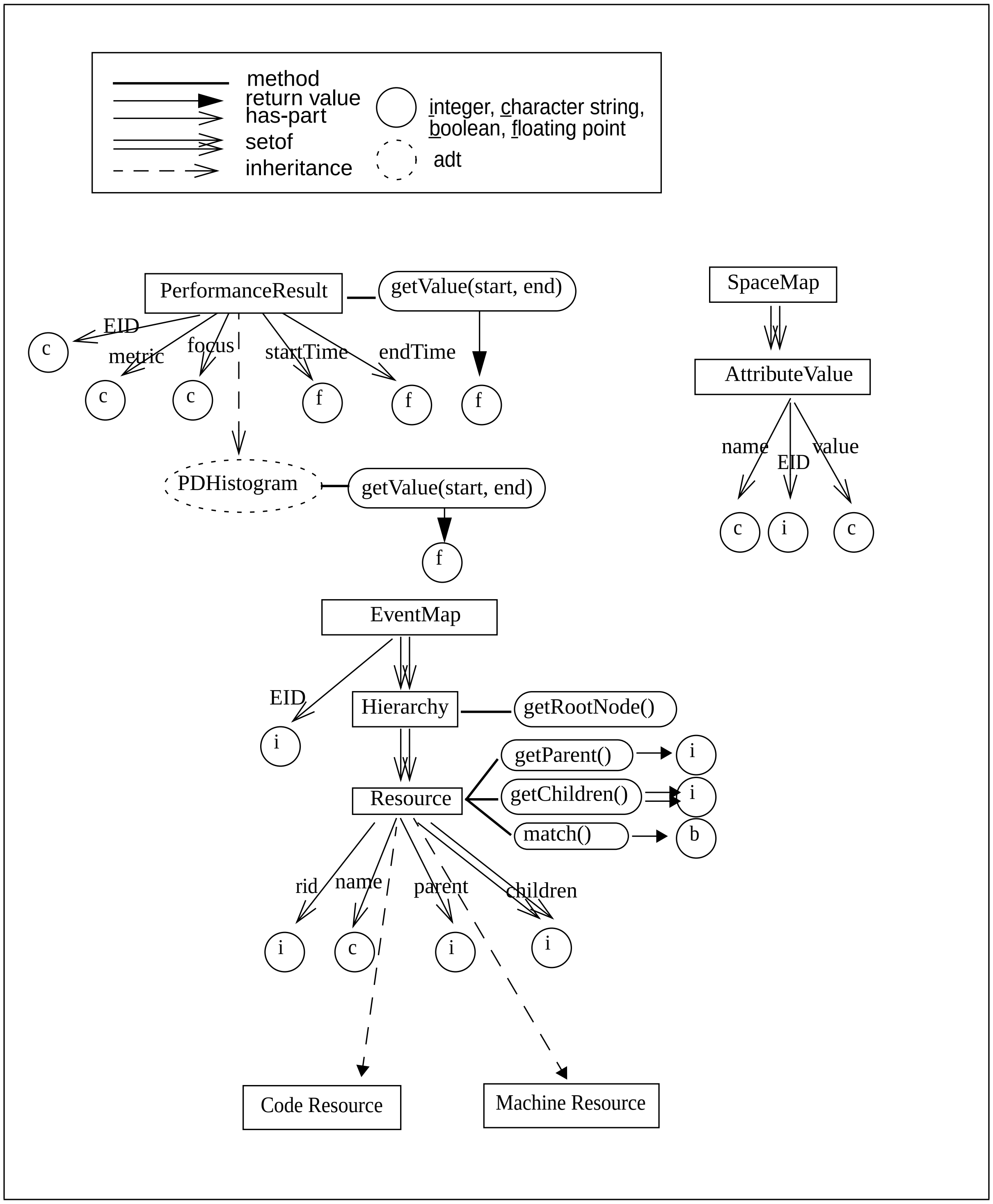


Figure 16: Schema for a Program Space containing Paradyn data.

1. Select zero or more EventMaps by choosing attributes from the SpaceMap. Here we show an attribute and value selected as *selectedAttribute* and *selectedValue* respectively.

```
for e in EventMap, s in SpaceMap, av in s.members
select e.hierarchy
where av.name = selectedAttribute and av.value = selectedValue and e.eid = s.eid
```

2. Select zero or more Performance Results from an EventMap.

```
for e in EventMap, p in PerformanceResult
select p.GetValue(startTime, endTime)
where selectedFocus = p.focus and selectedMetric = p.metric and e.eid = p.eid
```

3. Select the greatest CPUtime for the Whole Program recorded for any Program Event

```
select max
      (for p in PerformanceResult
      select p.GetValue(start, end)
      where p.focus = rootFocus and p.metric = “CPUtime”)
```

3.5.2 Implementing the Program Space with an Object Relational DBMS

An object-relational database management system (ORDBMS) is a combination of a traditional relational database management system with newer object-related functionality. This additional functionality may include user-defined abstract data types (ADTs), user-defined methods; constructed types such as sets, tuples, arrays, and sequences; and inheritance [59].

All of the major database vendors (IBM, Oracle, Informix) now provide some amount of object support, in at least partial compliance with the current SQL3 draft standard[63]. Although the particulars of the implementation vary, as long as we have the necessary object features discussed here, we can implement the program space on top of that database product.

At present, however, not every vendor supplies the full range of object functionality, for example, Oracle does not provide inheritance[63].

3.5.3 Other Implementation Strategies

It is possible to use a relational database management system (RDBMS) as the storage engine underlying an object oriented model. The ZOO project took this approach for their experiment management software, for example. The difficulties are the amount of work required to implement the necessary transformations from object-oriented queries to the underlying database, and the poor performance. There are several examples of a poor fit for an RDBMS. First is the dynamic nature of the schemas. The difficulty is that the schema is changing over time; that is, attributes may be added after the initial database design is complete. As long as the total database size is not large, this can be practically achieved by creating a new schema and copying all data as needed. However, in a traditional RDBMS the schema is not generally manipulated by users, who typically are not familiar with the Data Definition Language used. We want users of the system to be able to add performance result types and resource types while using the system.

The EventMap structure and the resources it contains introduce additional difficulties for an RDBMS. We need to define the match function for each type of resource node, or at the least for each hierarchy. This requires an ADT for clean implementation. The Performance Results require user-defined methods to be extensible, so that a new type of performance result may be added to the system. This can be done with inheritance.

We would like to leave open the possibility, as a future enhancement, for a distributed version of the Program Space that would enable scientists in different laboratories working on the same application to share their performance data. There has been much recent research into XML[26], an interface specification for describing data over the web. Because it is designed for data interchange on the web, it includes features for handling incomplete data and varying data schemas. An XML document is analogous to a relation in a database and a DTD is the equivalent to a database schema. As in the scientific experiment database projects discussed in Section 3.5.1, this approach provides direct access to the schema. Research currently underway will add the necessary features to make sharing data through the web a realistic alternative. For example, XML-QL [9] is a proposed query language for XML with many of the features needed for a shared, distributed data store. In the field of chemistry, for example, researchers have constructed the Chemical Markup Language as a set of XML definitions to be commonly used within their research community.

3.6 Summary

In this chapter, we presented a representation for a set of related program executions called the Program Space. The complete Program Space contains an EventMap, a collection of Performance Results, plus a SpaceMap describing execution-level characteristics that allow the different program runs and versions to be distinguished. We described the structural merge operator, its ordered comparison of two or more program executions, and its use in building up an EventMap that represents multiple program runs. Next we described our method for representing individual Performance Results, and operators for retrieving information from the Program Space: simple queries, discrete distance, and performance difference. We concluded with a discussion of implementation issues. In the next chapter we present examples of a prototype implementation of the Program Space performing a variety of tasks common in performance studies.

Chapter 4

Case Studies: Applying the Experiment Management Approach to Common Performance Activities

Demonstrating the utility of our experiment management approach for performance studies necessitates testing it in practical situations with existing parallel codes. To this end, we implemented a prototype experiment management system and used it with structural and performance data from parallel applications. We designed each small study as representative of common and essential tasks carried out in laboratories involved in writing and using parallel applications. For example, the project from which we gathered data for our first study has the following characteristics common to many environments in which performance related studies are carried out: The overall project is solving problems in physics, not computer science, but to do so high end computing technology must be used; many of the scientists involved are not primarily computer scientists; the computation is large enough that performance and scalability are key concerns — the more efficiently the simulation can be computed, the higher the quality of the result that can be obtained, and therefore the more useful the simulation will be;

and the specific algorithms underlying the code will evolve over time concurrently with the various porting and tuning efforts.

In the first study, we used our techniques for structural and performance data to examine results from several runs of the same program on two different platforms. The data is from a research project involving engineers and computer scientists from the University of Wisconsin and Syracuse University, that is developing an application to perform radiation hydrodynamics simulations. We examined data from a sequential version of the application on two different platforms. Our second study evaluated performance changes as a shared memory program evolved through versions during a performance tuning study. Our third study used the structural difference operator to compare program versions constructed using different communication libraries.

Our prototype provides core functionality of the Program Space. It reads in resource hierarchies and test data from Paradyn sessions, and provides a SpaceMap, one or more EventMap displays, a facility for retrieving and viewing performance data, and a performance difference operator. We wrote the code using Tcl/Tk [19] extended with a tree widget [20] and the BLT library [12].

4.1 Draco

In this study, we examined two versions of a scientific code called Draco, currently under development by a team of researchers at the University of Rochester and the University of Wisconsin. Developed at the Laboratory for Laser Energy, Draco is an adaptive Lagrangian-Eulerian code to perform radiation hydrodynamics simulations, especially related to direct laser driven inertial confinement fusion. A variety of physics packages are employed in each time step to simulate the development of the system. The various computational methods implemented in the different physics packages result in some packages being more computationaly intensive than others. The simulation code is being performance tuned and parallelized. The focus of our Draco study is a sequential code version currently being improved to enable finer-grained solutions to the physics problem being simulated to be computed.

Here we describe using our prototype tool with a sequential version of the code, written in Fortran 90. The application was ported from SGI Irix to SPARC Solaris by Paradyn researchers to begin examining application performance and to allow further study of Paradyn's dynamic instrumentation with such a large and complex application. There are approximately 50,000 lines of source code. The 289 functions (not counting libraries) are divided into four modules: 1d, 2d, 3d, and main. The resource hierarchies for the SPARC platform contained 10,152 nodes in total, including resource nodes representing libraries and library functions.

In the remainder of this section, we describe using the prototype to examine the data from three runs of the Draco application: (1) an execution of the SGI version, (2) an execution of the SPARC version, and (3) a repeated execution of (2). These different runs were assigned EID's, 1, 2, and 4 respectively.

We used the Paradyn Export feature to save performance data and resource information from several runs of the code. Export writes each internal Paradyn histogram into a file with header information (metric, focus, time interval size, start time, number of values) followed by a list of time/data value pairs. It also writes an index file, that provides a list of the individual data files, and a resource definition file, that lists the names of all of the resources for the Paradyn session. Our first step was to load the application data into our prototype. We specified an application name, DRACO, that refers to the entire Program Space we are building.

Next we initialized each of three Program Events with a string name, the name of the directory that contains its data files, and a list of application attributes for the SpaceMap.

Figure 17 shows two views of the resulting SpaceMap. The view on the left lists the

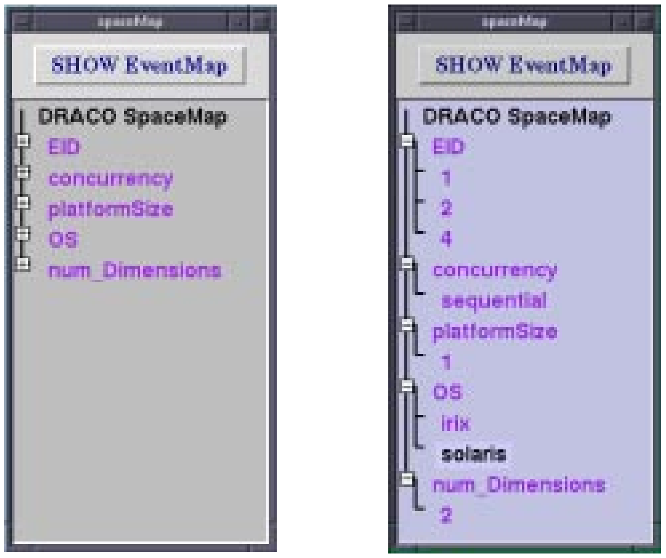


Figure 17: SpaceMap for the DRACO application.

attributes used to describe individual Program Events. The view on the right shows the same SpaceMap with the attribute headings expanded to show a list of all of the currently valid values for these attributes. We recorded four attributes: concurrency (sequential or parallel), platformSize (the number of machine nodes used in each particular program run), OS (operating system used) and num_Dimensions (referring to the underlying physics simulation algorithm). EID is a unique identifier assigned by the system to each Program Event. The first step in navigating the experimental data is selecting attributes from the SpaceMap, then viewing the resulting EventMap.

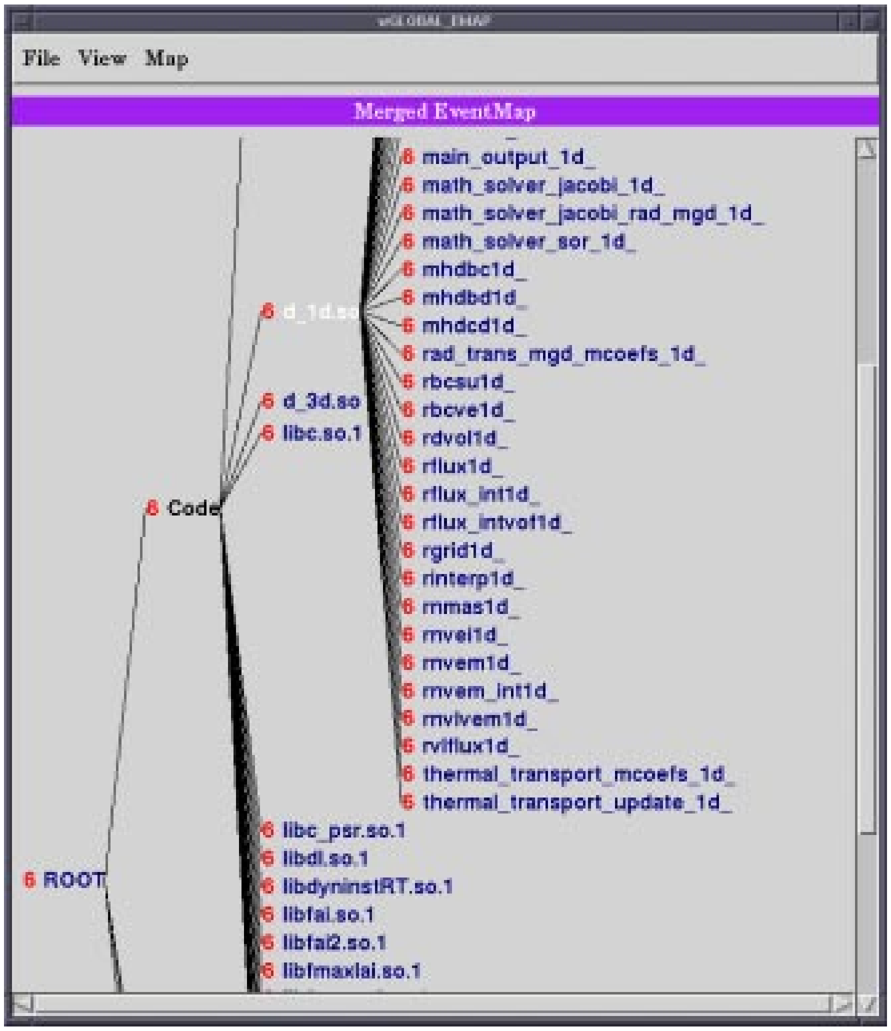


Figure 18: A Merged EventMap for three different Draco runs.

We chose the two Program Events from the SPARC platform by selecting “solaris” under the OS attribute and pressing the “SHOW EventMap” button. The result is shown in Figure 18. In this display we show the result of applying the Structural Merge Operator to the two Program Events. Each Program Event is assigned a unique identifier 2^i ($i \geq 0$). We tag each resource in the structural display with a numeric label showing the sum of the EIDs of the Program Events that contain that resource. In this case, we have merged Program Events with EIDs 2 and 4, so a numeric label 2 or 4 indicate a resource unique to one Program Event

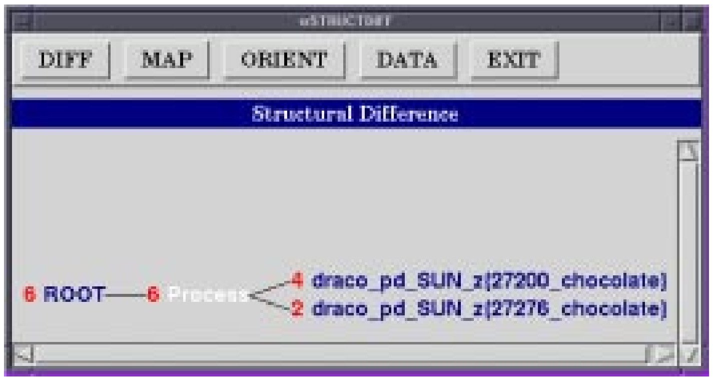

DIFF
MAP
ORIENT
DATA
EXIT
Structural Difference
6 ROOT
6 Process
4 draco_pd_SUN_z[27200_chocolate]
2 draco_pd_SUN_z[27276_chocolate]

Figure 19: Result of the Structural Difference Operator applied to two Draco Program Events.

and a numeric label 6 indicates a resource common to both. Children of resource nodes can be shown or hidden by clicking on the parent node. The snapshot we display here was taken after expanding the Code module node “d_1d.so.”

Our expection is that nothing has changed between these two versions, since no source code revisions occurred, and both times the program was run on the same machine. However, we want a quick way to verify this. Using the merged EventMap is impractical, because it would require us to visually check the numeric labels for all ten thousand nodes of the hierarchies. Instead we display the Structural Difference (Figure 19). This shows us only what has changed between the two Program Events; in this case, only the process identifiers are different. Note that the nodes “ROOT” and “Process” appear here only as placeholders. They are displayed to show the location of the process-related resources within the overall EventMap. We can distinguish such placeholders by their numeric labels: a label of 6 indicates that the resource is common to Program Events 2 and 4.

Figure 20 shows the EventMap for all three of the DRACO Program Events, with the Code hierarchy expanded. The different program runs are distinguished by their unique identifiers:

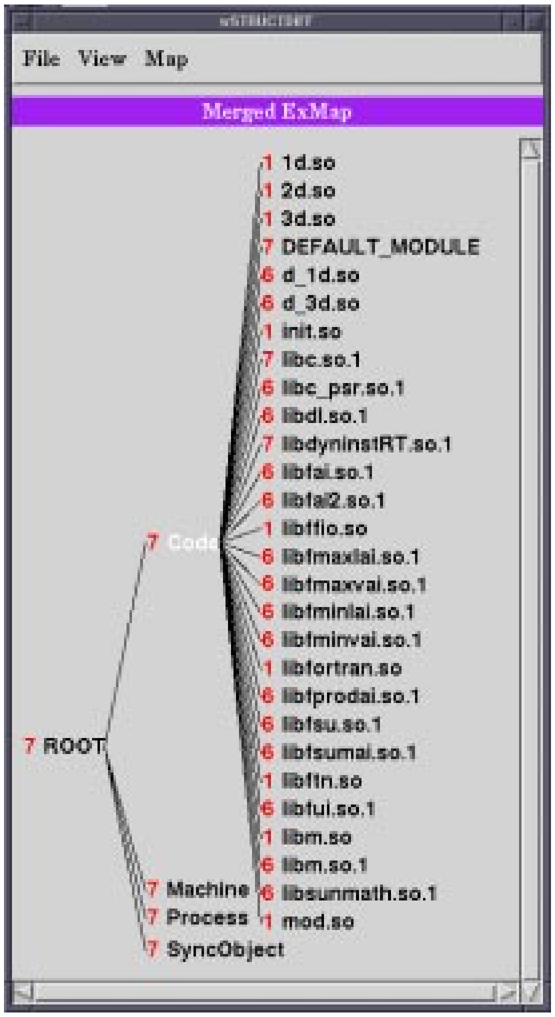


Figure 20: Merged EventMap for the three Draco Program Events (before mapping).

1 for the program run on the SGI platform, and 2 and 4 for the two program runs on the SPARC platform. Note that the first three modules from the top of the list, 1d.so, 2d.so, and 3d.so, only occurred in the IRIX-based runs, and the modules d_1d.so and d_3d.so only occurred in the SPARC-based runs. This is an example of common software engineering practices affecting our ability to match equivalent resources. The module d_1d.so is equivalent to the module 1d.so from a semantic perspective, however, on one platform a slightly different

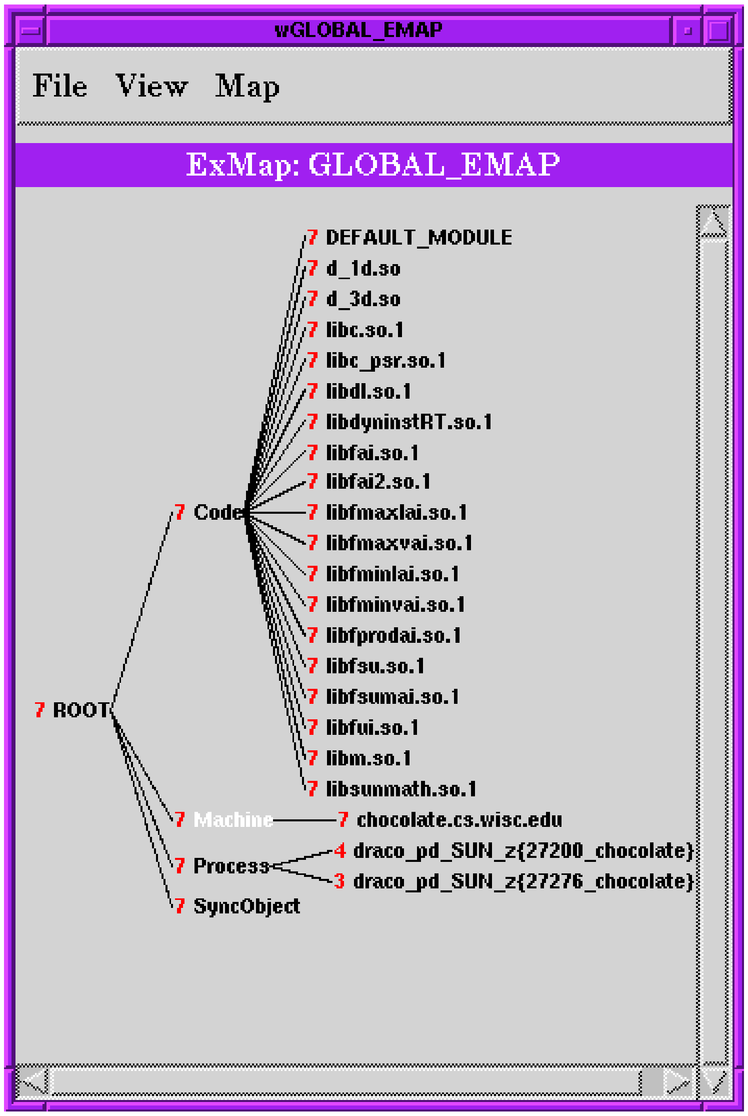


Figure 21: EventMap for the three DRACO Program Events (after mapping).

naming convention was used. Also, the functions listed in 2d.so in the IRIX version are included in DEFAULT_MODULE in the SPARC version. We used mapping to transform the resources so that the semantically similar modules would be considered equivalent. We needed only three simple mapping directives to accomplish our goal:

- map /Code/d_1d.so /Code/1d.so

- map /Code/d_3d.so /Code/3d.so
- map /Code/2d.so /Code/DEFAULT_MODULE.

The result of this mapping is shown in Figure 21.

4.2 Performance Tuning a Shared Memory Application

The goal of this study was to test out our performance difference operator, to examine its potential utility in a performance study. We used data from a previously completed performance tuning study of a protein-folding application, called Fold4, developed in our Chemical Engineering Department. The eventual target platform for the application was the SGI PowerChallenge. The first, sequential, version of the application was written in Fortran 77. A performance tuning study, reported in detail elsewhere [79], was conducted. The engineers ported the application from the SGI PowerChallenge to the Wisconsin Cluster of Workstations (COW), a cluster of Sun SPARCstation 20s, each with two 66-MHz Hypersparc processors and a Myricom Myrinet interface, running Solaris release 5.4. On the COW, the application was run on the Blizzard distributed shared memory system, and Paradyn was used to gather performance data using Blizzard. Once tuned, the application was then ported back to the SGI PowerChallenge. We analyzed the program versions and performance data from this study after it was completed.

We ran three versions of Fold4, taken from different steps in the performance tuning study. Version 1, the starting point of the tuning study, resulted from porting to the COW. A problem was identified with a serial portion of this code version that consumed 40% of the execution time on 8 nodes. The source code was modified to change to data partitioning to try to relieve the bottleneck, resulting in Version 2. Version 2 exhibited a problem with false sharing of data blocks. Data was padded and aligned to improve the cache behavior, resulting in Version 3. We present selected results here to demonstrate the benefit of the experiment management approach in navigating the large space of resources and data involved in a complete performance tuning study.

To consider the changes from Version 1 to Version 2, we merged these two EventMaps and applied the Structural Difference Operator (Figure 22). The resulting EventMap display distinguishes foci valid for both EventMaps. The Memory hierarchy shows the data partitioning change that occurred, in the form of eighteen new data structures storing particle data (GM->part_x). This restructuring was implemented to alleviate a performance bottleneck caused by frequent data movement between nodes.

Next we applied the performance difference operator with metric memoryBlockingTime. The Performance Difference Operator compares individual pairs of performance results and reports any that differ by more than a specified threshold (see Section 3.4.4). It progresses through the available performance data according to the partial order defined by resource normal form. Each magnification path is followed until no further magnification is possible or until the measured difference in the data is within a specified delta.

In Figure 23, we present a Performance Difference Display for the protein folding application study. This display shows that memory blocking behavior differed overall in the two runs; further, it differed in each process, and those differences were localized to five data structures (GM, GM->part0, GM->part1, GM->part2, GM->part3). (Note that the symbol “->” is part of the data structure name itself, and does not imply that GM is a parent node of part0, for example, in our resource hierarchy.) The ability to verify the performance changes at this level of granularity automates what was a common and time consuming task in the actual tuning project.

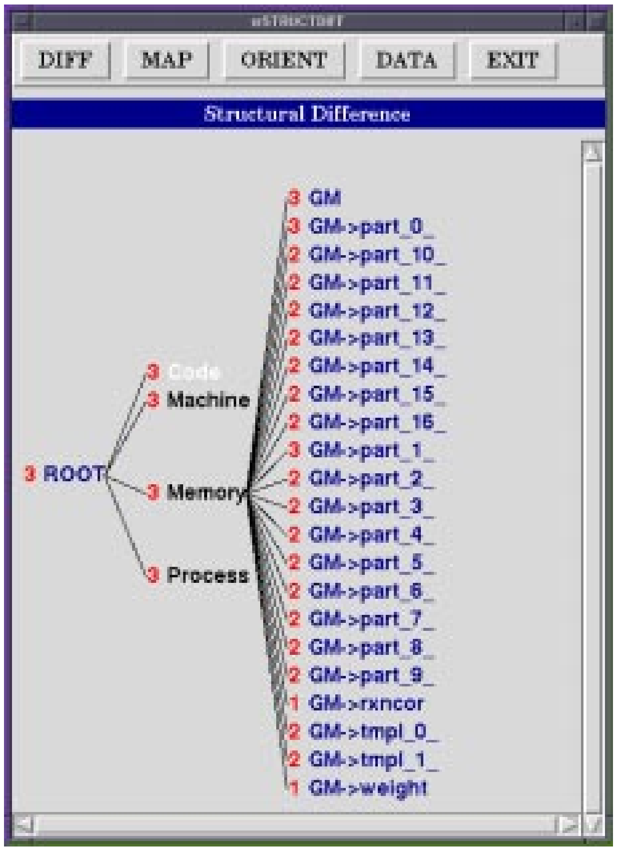


Figure 22: Result of Applying Structural Difference Operator to Fold4 Versions 1 and 2.

In the actual performance study, zeroing in on the performance changes between the versions was a time-consuming task: the actual study took approximately three to four person-weeks. Here we have demonstrated that our technique is useful for actual, not just toy, problems, and that it can be used to shorten the time required to draw meaningful conclusions about an application's evolving behavior. This validates the use of our technique for similar porting/tuning efforts, which are a common facet of high performance software development.

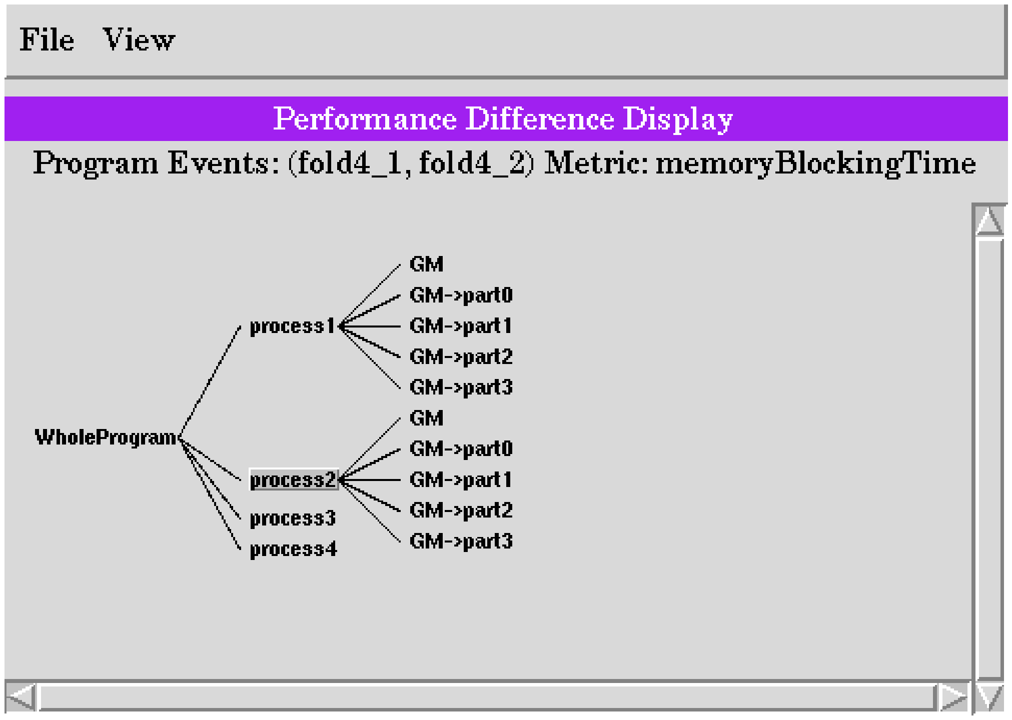


Figure 23: Results of the Performance Difference Operator for metric MemoryBlockingTime. The nodes shown represent resource combinations for which there is a performance difference detected between the Program Events. Starting at the left, the node WholeProgram means that there is some performance change between the two runs. The process nodes indicate that performance changed in some way for each of the four processes. The next level details individual shared data structures: GM is a shared index structure, and GM->part0, GM->part1, GM->part2, and GM->part3 are the shared data structures listed in the index. The data structures listed are those common to both runs for which memoryBlockingTime changed. This snapshot was taken after selecting two nodes, process1 and process2, to see a detailed display of their children. A visualization of the performance data for any node, showing the plots of the values for each run, can be launched by selecting the node on this display.

4.3 Comparing Alternate Implementations: Porting a PVM Application to MPI

In this study we examine two versions of a parallel application, developed with two different communication libraries, PVM and MPI. Our goal was to provide feedback that would be of use to a tool user, directing their attention to the structural changes in the application. That is, what has been changed in the application in porting it to a new communication library? The application, ns, is a parallel message-passing FFT code that solves the Navier-Stokes equation in three dimensions. We obtained this code from scientists at IBM Research at Haifa. Its history and use illustrate a typical candidate for an experiment-management based approach to performance study — the first version of the application was written in sequential Fortran, then it was transformed into C, parallelized using MPI, and finally rewritten using PVM. We used two versions of the application, the MPI and PVM versions, and examined what we could learn of the differences between the versions using our newly developed methods.

We used our tool to compare the structure of two versions of the application, before and after a port from the PVM to the MPI message passing libraries. A scientist porting an application wants directed feedback about the resulting changes in performance to the application, and hopefully some idea of the cause of any performance degradation. They may or may not be the original authors of the code, and therefore they may have only minimal knowledge of the application design and code structure.

In Figure 24, we show the EventMap that resulted from applying the Structural Difference Operator to EventMaps representing a 4-node run of nspvm (PVM version) and a 4-node run of nsmpif (MPI version), both on the IBM SP-2. This display provides a quick way to see what differed in both the code and the environment between the two runs. Figure 24 shows two different snapshots of the resulting EventMap. In this example, the PVM version has been assigned EID “1” and the MPI version has been assigned EID “2,” so resources common to both are labeled with a “3.” The display on the left shows a portion of the Code hierarchy expanded. In the Code hierarchy, three modules (dfft.c, ns3d.c, and p3d.c) appear in both runs. At the leaf level, we can see four procedures (strip, and strip1, strip2, strip3) appeared only in execution 1. The Message hierarchy on the right shows the changes in message tags: tags 0_1, 0_3, 0_5, and 0_2 represent MPI message tags, and the rest represent the message tags for the PVM version. By selecting a focus from the EventMap, we can display the performance data.

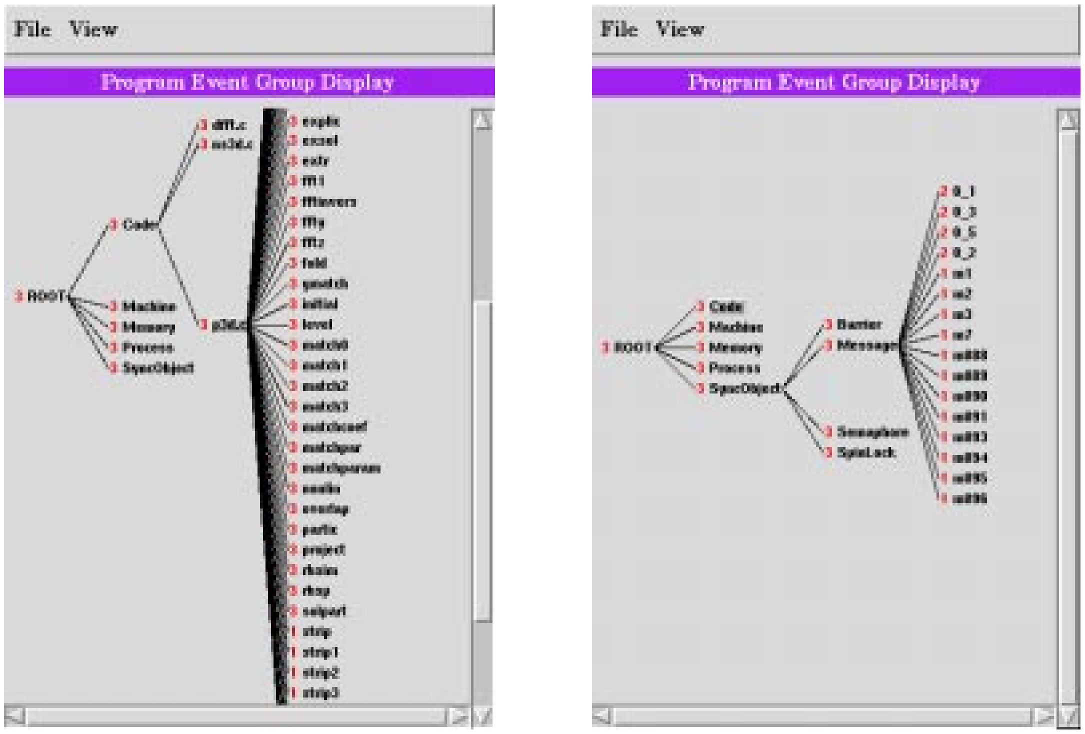


Figure 24: Resource Hierarchies for EventMap: nspvm + nsmpif. The EventMap Display allows the developer to navigate resource hierarchies and quickly see what differed structurally between two (or more) program runs. The display is organized like the resource hierarchies from Figure 4 in Chapter 3, with the addition of an integer Event Identifier (EID). Each run is labeled with a value 1, 2, 4, 8, etc. Resources that appeared in only one run, are labeled with the EID of that run (1 or 2 in this example). Resources that appear in more than one run are labeled with the sum of the EIDs; the resources labeled with 3 appeared both runs 1 and 2. *[Note: in this early prototype version, the term "Program Event Group" was used to refer to the EventMap.]*

In this section we have shown the utility of the EventMap interface for highlighting differences in code versions necessitated by porting to a new communication library.

4.4 Summary

In this chapter we have presented examples of using our prototype to (1) examine performance data gathered before and after a scientific application was ported to a new platform, (2) compare implementations based on alternate communication libraries, and (3) evaluate performance as a program evolves through versions. In each case, the use of an experiment management system allowed tasks typically involved in performance tuning and developing parallel applications to be completed simply and quickly.

Chapter 5

Using Historical Data in Performance Diagnosis

5.1 Introduction

This chapter describes how historical performance data, i.e., data gathered in one or more previous executions of an application, can be used to increase the effectiveness of automated performance diagnosis. Accurate performance diagnosis of parallel and distributed programs is a difficult and time-consuming task. In a recent survey of scientists actively engaged in parallel performance tuning[33], 50% reported an average time per tuning task of several weeks or longer. Recent research [28, 32, 70, 77] has examined possible approaches for automating, and thereby simplifying, the process of diagnosing a single program run. We present a novel approach to automated diagnosis that uses application data gathered in previous executions to guide the search for performance bottlenecks. This method leverages off of the repetitive nature of the performance tuning process — it is rare for a parallel application to be examined with a performance tool only once. Adding historical knowledge about an application provides a means for the tool to perform a more effective diagnosis.

Our starting point was an existing diagnostic research tool, the Paradyn Parallel Performance Tool [54]. Paradyn's Performance Consultant performs online, automated bottleneck detection in a single execution of a parallel or serial program. We modified the Performance Consultant, incorporating several different types of historical knowledge about an application's performance into the tool's search for performance problems.

We chose the Performance Consultant as our foundation for several reasons. First, using an online tool presents all of the same challenges as other types of tools, such as trace-based post-morem analysis and application steering, so our approach might be generalized to a wide array of uses. Second, there are some challenges unique to an online approach, so by using one we assure that our work might be applied to other online tools. The development of an API for dynamic instrumentation [29] and IBM's Dynamic Probe Class Library (DPCL) [58], currently in beta release, that provides a set of library functions for tool developers to use to incorporate dynamic instrumentation into their tools, suggest that more tools using an online approach will be available in the near future.

The general search strategy used in the Performance Consultant works well for studying new and unfamiliar applications. It provides systematic investigation of an application that does not depend on any assumptions about the application or the runtime environment, so it yields useful information for a wide range of programs. In practice, we noticed that the second time we sat down with the same application, it would miss data for interesting events and possibly stop before completion due to inherent instrumentation cost limits. There is a natural tension between a generally useful, single button approach to performance diagnosis and a more application-specific, knowledge-dependent approach. Our goal is not to replace the Performance Consultant's single button model, rather, to augment the search strategy in cases where prior knowledge of the program being studied is available. We use this knowledge to incorporate several different types of historical performance data into the tool’s search for performance problems.

The goals for this set of studies were:

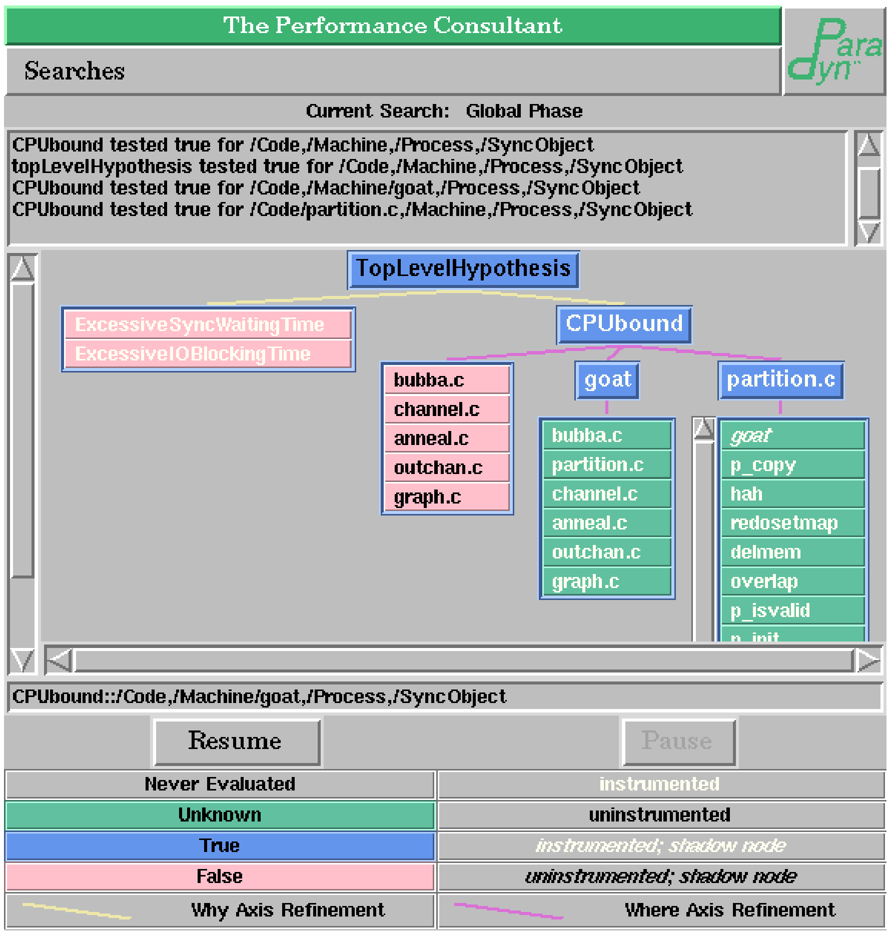
- Shorten the time required to identify important bottlenecks. We evaluate this strategy by measuring and comparing the total time to find bottlenecks with and without historical information.
- Decrease the amount of unhelpful instrumentation. There is a practical limit to the total amount of instrumentation in place at one time, to minimize inaccuracy of results due to perturbation. Decreasing unhelpful instrumentation in some cases will allow the search to continue where it might otherwise reach a limit and halt. We evaluate this strategy by measuring the total amount of instrumentation and the time to find bottlenecks.
- Determine the precise location of all significant bottlenecks. Results most useful for performance tuning are obtained when testing identifies a reasonably small number of well-defined potential problem areas. Practical limits on the total amount of instrumentation can result in important bottlenecks not being fully explored because the limited resources are being used to test less useful bottlenecks. We measure this by identifying a set of important bottlenecks for a particular execution, then evaluating the effect of historical information on finding the bottlenecks in that set.

We save performance and structural data from successive executions of an application, then extract knowledge useful for diagnosis from this collection of data, in the form of search directives. There are three types of directives: *pruning directives* that tell the tool to ignore some resources entirely; *priorities* that tell the tool which aspects of the application and runtime environment to look at first; and *thresholds* that tell the tool specific values against which to measure the application's actual performance. We use the directives to guide online performance diagnosis with an enhanced version of Paradyn. We evaluated our technique by testing an MPI application on the IBM SP/2, with reductions of 31% to 98% in the time needed to locate performance bottlenecks.

Next, we provide a brief description of the Performance Consultant and how we changed it. We describe the three mechanisms for including historical data in a diagnostic tool in Section 5.3. Then we present our experiments and results in Section 5.4. We finish with discussions and conclusions in Section 5.5.

5.2 Paradyn's Performance Consultant

Paradyn is an application profiler that uses *dynamic instrumentation* to insert and delete measurement instrumentation as a program runs. This approach results in a relatively small amount of data, in contrast to most tracing methods that may result in (possibly unusably) large data files. In Paradyn, a program is represented by *resource hierarchies*, and specific parts of a program are identified using a *focus*. This is a simpler, single execution version of our representational scheme as described in Chapter 3; in fact, it was the starting point we used in developing our multi-execution model. Paradyn's Performance Consultant (PC) [32] capitalizes on dynamic instrumentation to automate bottleneck detection during a program execution. The PC starts searching for bottlenecks by issuing instrumentation requests to collect data for a set of pre-defined performance hypotheses for the root focus or whole program. Each hypothesis is based on a continuously measured value computed by one or more Paradyn metrics, and a fixed threshold. The full collection of hypotheses is organized as a tree, where



```
CPUbound tested true for /Code,/Machine,/Process,/SyncObject
topLevelHypothesis tested true for /Code,/Machine,/Process,/SyncObject
CPUbound tested true for /Code,/Machine/goat,/Process,/SyncObject
CPUbound tested true for /Code/partition.c,/Machine,/Process,/SyncObject
```

```
CPUbound::/Code,/Machine/goat,/Process,/SyncObject
```

Figure 25: A Performance Consultant search in progress. The three items below TopLevelHypothesis have been added as a result of refining the hypothesis. Nodes ExcessiveSyncWaitingTime and ExcessiveIOBlockingTime have tested false, as indicated by node color (pink), and node CPUbound (blue) has tested true and has been expanded by refinement. The nodes bubba.c, channel.c, anneal.c, outchan.c, and graph.c all tested false, whereas the nodes goat and partition.c tested true and were refined.

hypotheses lower in the tree identify more specific problems than those higher up.

For example, the PC starts its search by measuring total time spent doing computation, synchronization, and I/O waiting, and compares these values to predefined (user set) thresholds.

Instances where the measured value for the hypothesis exceeds the threshold are defined as *bottlenecks*.

Each node in a PC search represents instrumentation and data collection for a (hypothesis:focus) pair. If a node tests true, meaning a bottleneck is found, the Performance Consultant tries to determine more specific information about the bottleneck. It considers two types of expansion: a more specific hypothesis, and a more specific focus. A child focus is defined as any focus obtained by moving down along a single edge in one of the resource hierarchies. Determining the children of a focus by this method is referred to as *refinement*. If a pair (h : f) tests false, testing stops and the node is not refined. The PC refines all true nodes to as specific a focus as possible.

Each (hypothesis : focus) pair is represented as a node of a directed acyclic graph called the Search History Graph (SHG). The root node of the SHG represents the pair (TopLevelHypothesis : WholeProgram), and its child nodes represent the refinements chosen as described above. Paradyn displays the SHG in list box form; we show an example in Figure 25.

Depending on the number of resources needed to represent an application, the number of hypothesis/focus pairs to be explored might be quite large. To prevent the PC data requests from overwhelming the system capacity or perturbing the application to a point where reliable results cannot be determined, the cost of instrumentation enabled by the PC is continually monitored. Search expansion, which generates new instrumentation requests, is halted when the cost reaches a critical threshold, and restarted once instrumentation deletion (initiated when nodes test false) causes the cost to return to an acceptable level.

5.3 Types of Search Directives

We have developed three mechanisms for including historical data in a diagnostic tool: *pruning directives* that tell the tool to ignore some resources entirely; *priorities* that tell the tool which aspects of the application and runtime environment to look at first; and *thresholds* that tell the tool specific values against which to measure the applications's actual performance.

Pruning directives instruct the diagnostic tool to ignore a subtree of a resource hierarchy in its evaluation of a specific hypothesis. They are a mechanism for conveying information about insignificant parts of an application. The total number of hypothesis/focus pairs tested by the Performance Consultant may become large if the total number of resources is large. In practice, this is frequently true. The top-down approach taken by the PC has the effect of excluding part of the potentially huge search space, since false nodes are never refined. Prunes further shrink the size of the search space. For example, we can avoid the overhead of instrumenting small, infrequently executed functions by pruning them from the search. Pruning directives can also be used to customize the search strategy for a particular environment. For example, the static process model of MPI version 1 [24] leads to a one-to-one correspondence between process and machine node for *n*-process programs run on *n* machine nodes. It is not necessary to investigate relative performance by both process and machine, so we can prune out the machine hierarchy. Pruning does not dictate the overall search strategy employed – what to examine first or next – rather it reduces the size of the total search space. One possible side effect of pruning is incorrectly eliminating something important. For this reason we also investigated other methods with better robustness. We investigated pruning based on historical data, such as functions with short execution time and redundant hierarchies (e.g. machine hierarchy if processes and machines map one-to-one) or sections of hierarchies. We also investigated pruning based on general rules, such as pruning the /SyncObject hierarchy from all but synchronization-related hypotheses.

Priorities assign a relative level of importance to specified focus-hypothesis pairs. This allows resources more likely to be responsible for behaviors of interest to be studied first, allowing data to be collected for a longer time interval. Unlike prunes, priorities do not exclude any foci from consideration; they instruct the diagnostic tool to consider certain hypothesis-focus pairs first. Each hypothesis-focus pair is given priority: High if it tested true in at least one previous execution; Low if it tested false in all previous executions; otherwise, Medium. High priority pairs are instrumented at search start and are persistent (i.e., testing continues throughout the entire program run, regardless of whether a true or false conclusion is reached). Starting up high priority pairs immediately, rather than waiting for the default top down search order to refine down to them, results in more control over the overall search order. By comparison, setting priority to medium or low only ensures an ordering between the node and its siblings.

Thresholds are the values used to determine if a hypothesis is true or false for a given focus. In the standard version of Paradyn, there is a threshold value that can be set by the user for each hypothesis. The goal is to keep the number of bottlenecks reported in a practically useful range. Reporting a large number of different bottlenecks yields inadequate guidance to the tuning effort, i.e., what to look at first, and also drives up the cost of instrumentation. Reporting only one or two bottlenecks, or failing to refine the bottlenecks to a detailed level, provides less information than might reasonably be obtained through simple visualization. We investigated automatically setting the thresholds based on historical data.

We added new functionality to the Performance Consultant that allows hypothesis definitions to be custom defined, instead of built-in as they are in the standard Performance Consultant. We also added infrastructure to input search directives, and to write Paradyn session data to files. Since Paradyn's front end is implemented using Tcl/Tk commands, we implemented these new features as Tcl commands, as detailed below.

- `shg setPriority` *`hypothesisName focus`* `[high|medium|low]`

  Specifies a priority level (see Section 5.3) for the specified hypothesis-focus pair.

- `whyAxis addHypo` *`hypothesisName parent metric1 metric2 threshold comparisonOperator expandPolicy`*

  Adds a new hypothesis with the specified name to Paradyn's why axis. Parent is the name of the parent node on the whyAxis. The two metrics are used to calculate the value of the hypothesis using performance data for a specific focus. The resulting value is compared to a threshold via the comparison operator (>, < , or =). If specified, the expand policy dictates whether the search will continue by testing the same focus with the child of the current hypothesis, or the same hypothesis with the set of refinements to the current focus.

- `whyAxis addPrune` *`resourceName hypothesisName`*

  Specifies a part of the search space to ignore (see Section 5.3).

- `save data [global|phase|all]` *`directoryName`*

  Writes out the contents of all Paradyn time histograms to files created in the specified directory. The options shown dictate whether global data, phase-specific data, or both will be written.

- `save resources all` *`dirName`*

  Writes the where axis contents, a list of resources, to a file in the specified directory.

- `save shg [global|phase]` *`dirName`*

  Writes the data contained in the Performance Consultant's search history graph to a file in the specified directory.

5.4 Experiments and Results

We performed a set of experiments to evaluate our introduction of prior knowledge into the Performance Consultant's search. We investigated the effectiveness of adding pruning and priority directives to the Performance Consultant, by measuring and comparing the time to locate the application's performance bottlenecks with the different methods. We also studied the advantages of using application-specific thresholds formulated using historical data, by measuring and comparing the number of bottlenecks successfully diagnosed and the number of locations instrumented to gather the needed data. Finally, we studied the use of pruning, prioritization, and generated thresholds in diagnosing different versions of the same application, to simulate the common practice of performance tuning successive versions of an implementation. We describe these experiments in more detail in the remainder of this chapter.

5.4.1 Using Pruning and Priority Directives

We ran our enhanced version of the Performance Consultant on an MPI application that solves the 2-D Poisson problem[24], running on four nodes of an IBM SP/2. First we ran the PC on the application with no modifications, and saved the resource hierarchies, search history graph, and performance results. This run forms our base case and was allowed to run to completion to identify the complete (100%) set of possible bottlenecks. Then we tested three variations of directed searching: first we generated only pruning directives, second only priorities, and third a combined version with both prunes and priorities. Identical search thresholds were used in all runs. In each experiment, we recorded the time each bottleneck was reported by the tool. The times we recorded are the timestamps assigned by Paradyn to the data, and reflect execution (elapsed) time. Since Paradyn performs dynamic instrumentation, the starting timestamp is determined by the instant of the instrumentation request, plus the time required to actually insert the instrumentation into the application code. Each conclusion about a performance hypothesis is reached only after data has been collected from the running application for a specified minimum time interval.

% B'necks Found	No Directives	Prunes Only			Priorities Only	Priorities & All Prunes
		All	General	Historic		
25%	115.2	80.0 (-30.6%)	102.4	108.8	80.0 (-30.6%)	51.2 (-55.6%)
50%	182.4	83.2 (-54.4%)	121.6	204.8	124.8 (-31.6%)	57.6 (-68.4%)
75%	1011.2	140.8 (-86.1%)	169.6	281.6	211.2 (-79.1%)	86.4 (-91.4%)
100%	2611.2	169.4 (-93.5%)	236.8	470.4	560.0 (-78.6%)	147.2 (-94.4%)

Table 1: Time (in seconds) to Find all True Bottlenecks with Search Directives

The results are reported in Table 1 and Figure 26. In Table 1, each row presents the time in seconds for the tool to locate the percentage of total bottlenecks specified in column 1, from 25% to 100% of the total. We were most interested in measuring the time to discover all the bottlenecks, since the more detailed level bottlenecks are particularly important and tend to be found later in the search. Because Paradyn reports results to the tool user as the application is running, we were also interested in seeing how quickly the tool would discover some or most of the bottlenecks. Column 2 of the table reports the time needed to find bottlenecks without any prior knowledge, which serves as the base case against which we are measuring our new techniques. The remaining columns list results for experiments of adding different types of search directives, as we describe in more detail below. In Figure 26 we graph percentage of true bottlenecks found versus time in seconds, using the full set of data points for each of four cases: no directives, prunes only, priorities only, and prunes and priorities combined.

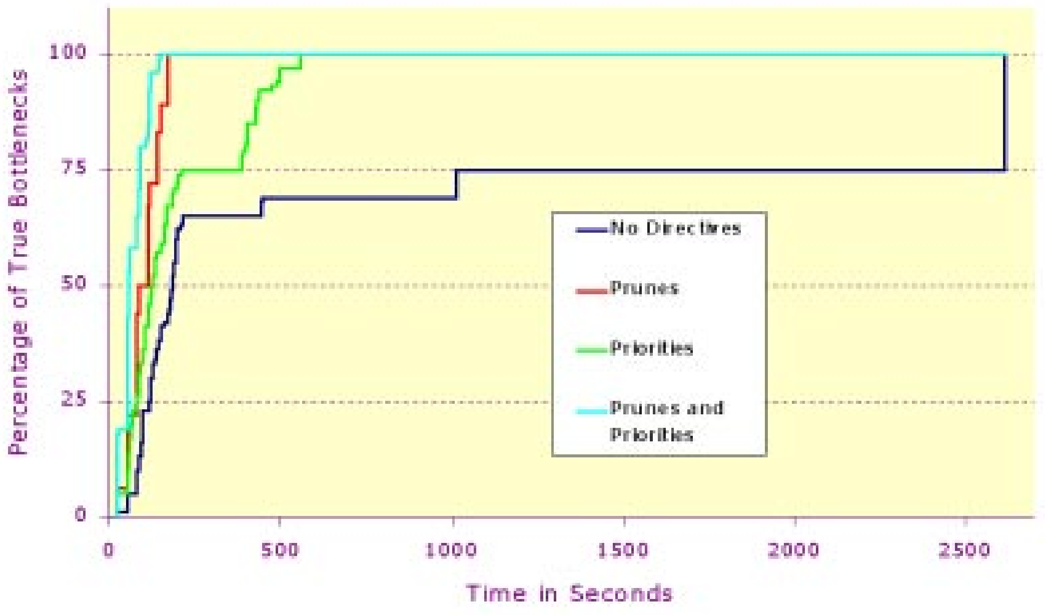


Figure 26: Percentage of True Bottlenecks found over time using different types of search directives. Each line shows the time to find some percentage of the complete set of true bottlenecks, from 0 to 100. Prunes included both general and historic prunes. Data from this study is also summarized in Table 1.

The first experiment investigated the performance advantages obtained using pruning directives. We used data from previous runs to generate a list of pruning directives. Then we ran Paradyn, providing the list of pruning directives as input to the modified Performance Consultant. General prunes, such as pruning the /SyncObject hierarchy from all but synchronization-related hypotheses, are not specific to a particular application or environment. Historic prunes, such as pruning a specific function with low execution time, are formulated based on data gathered in one or more previous executions of the same application. We evaluated the effects of each of the two types of pruning; the results are listed in the “Prunes Only” column of Table 1. The use of general and historic pruning directives resulted in improvements in time to locate 100% of the bottlenecks of 91% and 82% respectively. We see a substantial improvement with either type of pruning, and note that in this case general prunes alone performed better than historic prunes alone. The ability to specify general prunes varies with the overall type of application. For example, since the application in this study was implemented using MPI version 1, which has a static process model, we could add a general prune to remove the process hierarchy, and that yielded good results. However, we would not be able to specify this general prune for other types of applications. The combination of general and historic yields the best results: comparing the subcolumns of “Prunes Only,” we see that it took 236.8 seconds to find 100% of the bottlenecks with general prunes, compared to 169.4 seconds with both type of prunes, an improvement of 28%. Combining the two types of pruning directives resulted in a reduction of 93.5% in time to locate all true bottlenecks compared to the “No Directives” case.

In the second experiment, we studied the effects of ordering the search for bottlenecks using priorities. We used historical data to generate priorities for each hypothesis/focus pair as outlined in Section 5.3. We expected that, compared to using the PC with no historical data, we would reduce the time required to find the major (true) bottlenecks. Priorities do not reduce the number of potential bottlenecks tested, they change the order in which the candidates are tested. As shown in column 4 of Table 1, we obtained a reduction of 79% in time to locate all true bottlenecks. The improvement is more modest than the reduction of 93.5% we obtained using pruning directives. However, reordering the search does not introduce the possibility of missing bottlenecks, which is an important advantage to the method over using pruning directives.

In the final experiment, we tested a combination of prunes and priorities. Our goal was to improve on the time reduction obtained using only priorities, yet avoid the possibility of pruning important tests from the search. We included general pruning of redundant and irrelevant hierarchies, but did not include historic prunes for previously false hypothesis/focus pairs. This combined approach may result in some retesting of false nodes, however, it will never miss new behaviors due to pruning. We obtained a reduction of 94.4% from the base case for finding 100% of the true bottlenecks, which is a reduction of 22 seconds from the 169 seconds it took using pruning without priorities.

5.4.2 Using Thresholds Determined from Historical Data

We studied the behavior of the Performance Consultant while varying threshold values to determine the potential benefit of automatically setting thresholds based on historic data. Our application was the 2-D decomposition code from the previous section run on four nodes of an IBM SP/2. This sample application is strongly dominated by synchronization waiting time, which accounts for approximately 75% of the total execution time. 45% of the total execution time for all four processors is spent waiting in function exchng2, and 20% in function main. This waiting time is split between three message tags, 3/0, 3/1, and 3/-1 (27%, 19%, and 20% respectively). Individual processes 3 and 4 are dominated by wait time (81% and 86%) and significant waiting also occurred in processes 1 and 2 (46% and 47%).

We investigated the quality of the PC's diagnosis by checking for the number of these areas reported as bottlenecks, either individually (e.g., function main) or in combination (e.g., message tag 3/0 for function main). Full results are shown in Table 2. When a threshold setting greater than 10% was used, bottlenecks we previously determined to be significant were not reported by the PC. When the threshold was set to 12% the tool reported close to the full set of bottlenecks; the default Paradyn setting of 20%, in contrast, resulted in 7 of the 26 bottlenecks being missed. The third column shows how much instrumentation was used to diagnose the program run. Setting the threshold to 12% (shaded) yields good results and also uses noticeably less instrumentation than a setting of 10% or 5%. The final column shows an efficiency metric determined by dividing the number of bottlenecks found by the number of hypothesis/ pairs tested. Efficiency decreases with thresholds below 12%, an indication that lowering the threshold below 12% increases the amount of instrumentation but does not improve the result.

In earlier studies, we found similar results for an ocean circulation modeling code using PVM, running on SUN SPARCstations. We found an optimal synchronization threshold at 20%, from a starting point of 30% (which yielded an incomplete diagnosis). Efficiency decreased below 20%, for example the number of metric-focus pairs instrumented was 326 for 20% and jumped to 373 for 10%. The useful threshold in this case differs from that found for the MPI application, showing the advantage of application-specific historical performance data.

Synchronization Bottleneck Threshold Setting (% of total execution time)	Number of Bottlenecks Reported by the Performance Consultant	Total Number of Hypothesis/Focus Pairs Tested	Efficiency (Bottlenecks Found Per Pair Tested)
30%	9	30	0.3
20%	19	66	0.29
14%	22	76	0.29
12%	25	85	0.29
10%	26	107	0.24
5%	26	105	0.25

Table 2: Bottlenecks Found with Varying Threshold Values. Number of bottlenecks reported are rounded, averaged values calculated from three repeated tests. Row 4 (shaded) is a crossover point for efficiency: increasing the number of pairs tested beyond 85 does not yield any significant gain in results.

5.4.3 Using Historical Data with Different Code Versions

We studied the use of historical performance data in the situation where the application has been revised over time. While tuning an application, a developer repeats through a cycle of profile-analyze-change. We performed a series of performance diagnoses using different versions of an MPI application on the IBM SP/2. The application implements an iterative Poisson function decomposition. We used several of the different versions of the implementation presented by Gropp *et al*[6]. In each step of the study, we used results from previous runs of the Performance Consultant to direct subsequent PC runs. There were four versions of the application: Version A is a 1-dimensional version that uses blocking send and receive operators; Version B is a non-blocking 1-dimensional version; Version C performs a 2-dimensional decomposition; and Version D runs the same code as Version C across 8 nodes (all others run on 4 nodes). We changed all versions to compute a fixed number of iterations, rather than stopping as soon as a solution is reached. The complete set of results for the study is contained in Table 3, and the results are presented graphically in Figure 27.

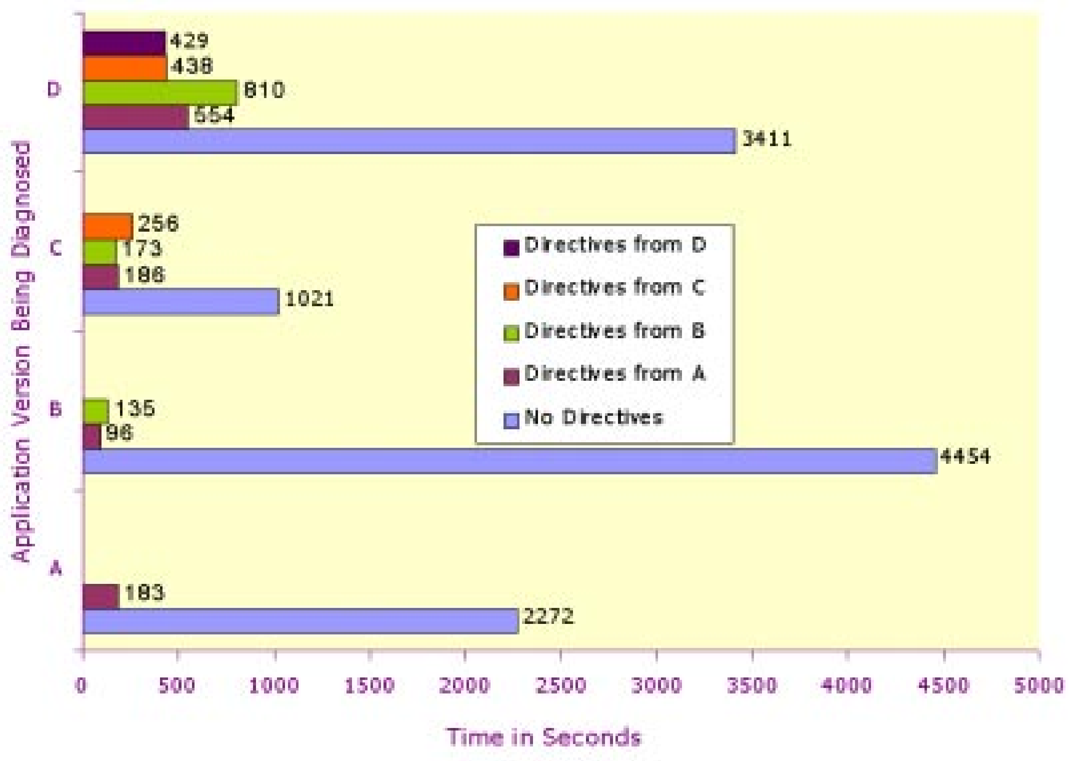


Figure 27: Time to complete diagnosis using search directives from different application versions. Each group of bars represents the program version being diagnosed: A, B, C, or D. The colors of the bars indicate the source of the search directives used. See also Table 3.

We started by running the Performance Consultant on Version A without search directives, which we denote A_{none}, resulting in a time to locate true bottlenecks of 2272 seconds. Next, we repeated the diagnosis of version A, this time including general prune and priority directives generated from the previous execution. This run, A_A, showed a decrease in the diagnosis time by approximately 92%.

Next we examined B_A, Version B using search directives extracted from runs of Version A, and found a 98% improvement in diagnosis time over the original base case that used no directives. We continued for Versions C and D, each time running the Performance Consultant with search directives extracted from each individual prior run.

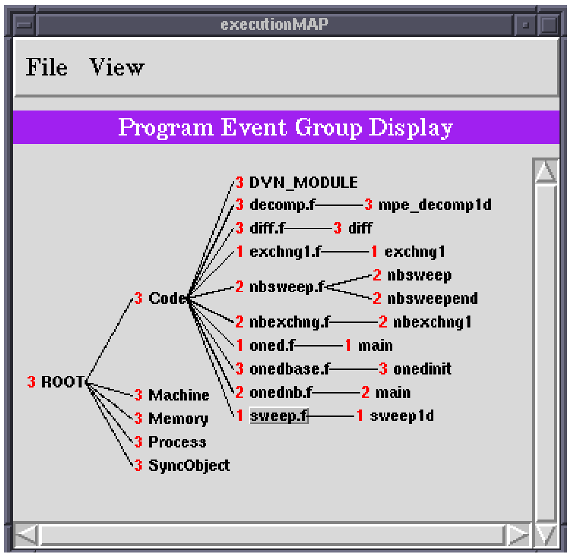
		Source of Search Directives				
		None	A	B	C	D
Application Version	A	2272	183 (-92%)			
	B	4454	96 (-98%)	135 (-97%)		
	C	1021	186 (-82%)	173 (-83%)	256 (-75%)	
	D	3411	554 (-84%)	810 (-76%)	438 (-87%)	429 (-87%)

Table 3: Time (in seconds) to complete diagnosis with search directives from different application versions. Some times reported are median values for several runs. Standard Deviations ranged from 3 to 17 seconds. Each row contains the data for a particular application version, A through D. Each column contains the data for a particular source of the search directives used with the Performance Consultant. For example, the cell found at row "C" and column "B" contains the time to diagnose C using directives from a previous run of B. Time relative to the base version (column "None") is shown in parentheses.

After each run of the Performance Consultant, we use the resulting search history graph and the program's resource hierarchies to generate search directives to be used in subsequent runs. We added new functionality to the Performance Consultant to map focus names found in these directives onto names valid in the current environment (see Section for a more detailed discussion of this mapping). For these tests mapping was implemented as a set of directives of the form "map *resourceName1 resourceName2*" specified by the user in an input file. After starting Paradyn, we applied the specified mappings to the list of extracted search directives, then read the directives into the Performance Consultant. For increased efficiency, we applied specified pruning directives, if any, to the resulting list of search directives before we read it into the Performance Consultant.

For the tests described in this section, we mapped each pair of machine resources so that search directives generated in one run could be meaningfully used to refer to machine resources discovered in a subsequent run. We also mapped functions and modules between the different code versions, since the names had been changed for the different code versions.

Figure 28 shows resource hierarchies combined using the Structural Difference Operator for two versions of the application, Versions A and B. Each resource is tagged with execution identifier 1, 2, or 3 if the resource is found in Version A, Version B, or both, respectively.



Mappings Used

```
map /Code/exchng1.f /Code/nbexchng.f
map /Code/exchng1.f/exchng1 /Code/nbexchng.f/nbexchng1
map /Code/oned.f /Code/onednb.f
map /Code/sweep.f /Code/nbsweep.f
map /Code/sweep.f/sweep1d /Code/nbsweep.f/nbsweep
```

Figure 28: Mappings for Versions A and B. On the left we show the execution map for Versions A and B of the Poisson decomposition application, with the Code hierarchy expanded. Each resource is tagged with an execution identifier: resources unique to version A are labeled with “1,” those unique to version B are labeled with “2,” and those common to both are labeled with “3.” We map unique nodes which refer to code that was modified between versions, including a change of name. The mapping directives we used are shown on the right.

Resources unique to one execution are candidates for mapping. For example, the module containing function main is named “oned.f” in Version A, and “onednb.f” in Version B. We mapped these two resources, /Code/oned.f and Code/onednb.f, so that search directives extracted from runs of A could be used in diagnosing runs of B. The full set of mappings we used in this example is shown to the right of the resource hierarchies. Our mapping functionality is currently restricted to one-to-one mappings, so for example when we mapped machine names between the 4 and 8 node runs, the result was a set of search directives including 4 of the total of 8 machine nodes. This is seen in the bottom row of Table 3 as longer times required to reach a diagnosis due to a less complete set of search directives for the 8 node machine.

In every case we tested, adding historical knowledge to the Performance Consultant greatly improved its ability to quickly diagnose performance bottlenecks: diagnosis time was reduced a minimum of 75% in all executions using historical knowledge.

In Table 3, each row represents the version of the application currently being diagnosed. Each column represents the source from which we extracted the search directives used. The first column contains the time to reach a diagnosis using no search directives, and subsequent columns contain the time to reach a diagnosis using search directives from different sources. We used dedicated machine time and therefore saw relatively low variability in run time for repeated executions of the same version.

After completing the test runs, we analyzed the Performance Consultant behavior to determine how it was affected by the search directives we added. First we examined the effects of using search directives from the base run of A, A_{none}, to diagnose a second run of A, A_A. 81

Priority Setting	A only	B only	C only	A, B only	A, C only	B, C only	A, B, C	TOTAL
High	16	13	3	10	10	9	46	107
Low	32	72	24	28	20	13	92	281
Total	48	85	27	38	30	22	138	388

Table 4: Similarity of Extracted Priorities Across Code Versions. We show here the number of priority directives extracted from the base runs A_{none}, B_{none}, and C_{none}. The first row includes only directives that assign a high priority, the second includes only directives that assign a low priority, and the bottom column shows the combined total. Each column represents the source(s) of the priority directives: A_{none}, B_{none}, $C_{none,}$ or some combination of these. For example, of the total 107 different high priority directives, 16 were unique to version A and 46 were common to versions A, B, and C.

hypothesis/focus pairs tested true in A_A, resulting in 81 search directives that set priority to high. In A_B, a total of 103 hypothesis/focus pairs tested true. 78 were pairs that tested true in A_A (and were included in the 81 search directives); of the remaining 25, 3 had been set to low priority, 6 were intermediate level nodes not tested in A_A, and the remaining 16 were more detailed/refined answers not tested in A_A due to cost limits. In this case, using search directives resulted in a more detailed diagnosis than could be performed without the directives.

Although we had anticipated search directives from different versions would not be as effective as search directives from the same version, we saw only small differences in most cases. In fact, we were surprised to see that run C_C performed more poorly than either C_A or C_B. We analyzed some of the data more closely to account for this. We examined the sets of search directives extracted from runs A_{none}, B_{none}, and C_{none} to see how they differed. As shown in Table 4, of the 388 total priority directives, 138 (36%) were common across all three sets of directives, 160 (41%) were unique to a single set, and the remaining 90 (23%) occurred in two out of three sets. Of the 107 total high priority directives, 46 (43%) were common to all three, 32 (30%) were unique to one, and the remaining 29 (27%) were common to two. We noticed that most of the differences involved more detailed focuses, for example, in run C_{none} the Performance Consultant found the bottleneck for function exchng across all nodes, however it failed to refine this to a particular message tag, as it did in run A_{none}, and found only half as many particular node/message tag combinations as it did in run A_{none}. The tool also found fewer true bottlenecks in run C_{none}: 67 as compared to 81 found in run A_{none} and 68 found in run B_{none}.

We also examined the diagnoses obtained in runs C_A, C_B, and C_C. The bottlenecks found did not vary significantly between these runs. Of 115 total bottlenecks diagnosed as true by the Performance Consultant in any of these runs, 113 were common across all three, and the remaining 3 were common to two of the three. So despite the differences in the time it took to reach the diagnosis, the Performance Consultant yielded similar answers in all three cases.

For this example, despite modifications to the communications primitives (blocking or non-blocking), and modifications to the algorithm (1-dimensional or 2-dimensional decomposition), the bottleneck locations remained the same. Although total synchronization time and total execution time varied between versions, the set of resources responsible for the time was similar.

We also investigated using results from multiple previous runs to guide the current run. We looked at two different approaches to combining search directives from different versions: $A \cap B$ sets to a high/low priority only those hypothesis/focus pairs that tested true/false in *both* Versions A and B. $A \cup B$ sets to a high priority those hypothesis/focus pairs that tested true in either A or B, and sets to low priority those hypothesis/focus pairs which tested false in either version and did not test true in A or B. We used the resulting set of directives to diagnose Version C. In this particular example, the lists of priorities that result from the two methods of combination have 59 common directives, with 38 additional directives unique to $A \cup B$. The resulting diagnosis times were 176 seconds for $A \cup B$ and 179 seconds for $A \cap B$. This difference is too small for us to conclude the superiority of one combination method over the other. Which performs better is related to the similarity of the sets of directives generated using data from runs A and B, not the similarity in code or platform of the versions.

5.5 Discussion and Conclusions

We have described a new approach to automated performance diagnosis that incorporates knowledge from previous runs of the same application. The result is a performance tool that learns from each diagnostic program run, adapting its search strategy to obtain more useful diagnoses more quickly. We show performance gains of up to 98% obtained by incorporating historical knowledge into the Performance Consultant’s search strategy. The results presented demonstrate the utility of our approach for repeated performance diagnosis of similar program runs, a common scenario when tuning parallel applications. The improvements achieved show that our new approach to gathering and storing historical application data can be successfully applied to the problem of automating performance diagnosis.

Chapter 6

Summary and Directions for Future Research

6.1 Dissertation Summary

Developing efficient parallel programs is a challenging and time consuming task. Despite advances over the past decade in compiler and tool technology, tuning a parallel code for a particular platform generally takes weeks or months of the time of specialists. Furthermore, most major laboratories aggressively replace their machines with newer models to gain the advantage of processor speed increases. This situation creates an opportunity for a tool that would simplify the process of parallel performance tuning large-scale scientific applications. Improvements in the ability of a tool to provide meaningful, focused feedback to a programmer tuning a code may result in a better implementation, and that leads to savings in machine time, more detailed simulation results, or both. In this dissertation, we have presented a new approach for parallel performance tools that directly addresses the key observations that the tuning process itself is inherently repetitive, and involves more than one program execution. Our thesis is that the scientific experiment management paradigm is a useful approach for parallel performance tools. Incorporation of data from multiple program executions together with the ability to meaningfully and practically navigate the data is a useful approach in common tasks related to parallel performance: porting applications, tuning for a particular platform, comparing versions of a code under development, and investigating scalability behavior.

The major contributions of this thesis are:

- the Program Space, a flexible and extensible representation for a complete multi-dimensional space of program runs;
- a set of mechanisms for quantitatively and automatically comparing two or more program runs, in terms of both structure and performance;
- a demonstration of the use of the Program Space and our program comparison methods with large scale parallel applications under development and in use; and
- results of a study of the incorporation of historical data contained in the Program Space into automated performance diagnosis, in which we demonstrated performance improvements of up to 98% in the time needed for the tool to completely diagnose a parallel application.

6.2 Directions for Future Research

Many interesting research ideas emerged during the course of the work described in this dissertation. Here we discuss interesting potential avenues of exploration and describe some of our ongoing work.

Work outside of the scope of this dissertation and already in progress is extending this research in several directions. A prototype currently under development incorporates the DEVise visualization tool [48] to allow a rich variety of visualizations to be used for navigating performance data, and includes the results of user interface research into techniques for displaying the contents of the Program Space.

We are continuing to research additional approaches to resource mapping. The goal is to automate the mapping to the furthest extent possible, while continuing to allow user-specified mappings. We have demonstrated resource mapping performed at the start of each new execution, and further study is warranted to extend this approach to cover cases in which new resources are discovered later in an application run. Future work might explore using richer resource descriptions to enable more semantically meaningful mappings to be performed automatically.

The prototype implementation can be usefully extended to include performance data gathered with a variety of monitoring and predictive tools. The techniques we have developed can be used to compare an actual execution with a predicted or desired performance measure for the application. Incorporation of data from performance predictions or performance models into the Program Space would add significantly to the utility of our tool. Uses for this include performance tuning efforts, automated scalability studies, and performance model validation studies.

In addition to its application to cross-execution studies, comparison-based performance analysis can be used to compare distinct time intervals of a single program execution. The ability to name, lookup, examine, and compare performance results from different time intervals within one program execution has many direct applications. Both the environment and the code may vary during the course of a single program run. In the range of computational models now available, especially the more dynamic environments of the near future, processes may be created, destroyed, migrated [11]; communication patterns and use of distributed shared memory may be optimized [62]; data values or code may be changed by a steering adjustment [25,46]; or loop behavior may change as matrix distribution changes over the course of the computation [14].

We plan to investigate the incorporation of an experiment launching mechanism into a prototype implementation, to allow completely automated performance diagnosis that may involve more than one program execution. Experiment launching is a feature found in the scientific experiment management systems that we examined; we believe that this technique could be usefully incorporated into the parallel performance tuning environment, allowing us to incorporate approaches such as design of experiments into our tool.

One of the goals of our approach is comparison, and a metric that describes *how much* performance has changed would be useful. We briefly examined approaches to a *performance distance* metric: a Euclidean distance, and a weighted average of performance result values. The goal is a quantitative measure of how much performance differs between two or more executions as a whole, that can be used to weigh or rank performance bottlenecks. More detailed study with actual examples of different types of parallel applications must be carried out to determine which approaches will be useful here.

References

[1] S. Abiteboul. Querying semi-structured data. In F. Afrati and P. Kolaitis, editors, *Proceedings of The International Conference on Database Theory,* Delphi, Greece, 1997.

[2] M. Abrams, A. Batongbacal, R. Ribler, and D. Vazirani. Chitra94: A tool to dynamically characterize ensembles of traces for input data modeling and output analysis. Technical Report Computer Science Department TR 94-21, Virginia Polytechnic Institute and State University, June 1994.

[3] R. J. Block, S. Sarukkai, and P. Mehra. Automated performance prediction of message-passing programs. *Proceedings of Supercomputing '95*. IEEE Computer Society Press, 1995.

[4] G.E.P. Box, W.G. Hunter, and J.S. Hunter. **Statistics for Experimenters**. John Wiley and Sons, New York, 1978.

[5] M. Brune, J. Gehring, A. Keller, B. Monien, F. Ramme, and A. Reinefeld. Specifying resources and services in metacomputing environments. *Parallel Computing*, 24:1751–1776, 1998.

[6] M. Brune, J. Gehring, A. Keller, and A. Reinefeld. RSD – Resource and Service Description. J. Schaeffer, editor, *High Performance Computing Systems and Applications, HPCS '98: The 12th Annual International Symposium on High Performance Computing Systems and Applications*, pages 193–206. Paderborn Center for Parallel Computing, Kluwer Academic Publishers, 1998.

[7] CENSA. The definition of an electronic notebook system. http://www.censa.org, 1999.

[8] I.A. Chen and V.M. Markowitz. An overview of the Object Protocol Model (OPM) and the OPM Data Management Tools. *Information Systems*, 20(5):393–418, 1995.

[9] A. Deutsch, M. Fernandez, D. Florescu, A. Levy, and D. Suciu. A query language for XML. Technical report, AT&T Labs, 1998.

[10] H.G. Dietz, W.E. Cohen, and B.K. Grant. Would you run it here... or there? (AHS: Automatic heterogeneous supercomputing). *Proceedings of the International Conference on Parallel Processing*, pages 217–221, 1993.

[11] J. Dongarra, G.A. Geist, R. Manchek, and V.S. Sunderam. Integrated PVM framework supports heterogeneous network computing. *Computers in Physics*, 7:166–74, 1993.

[12] G. Howlett, The BLT Toolkit. In M. Harrison et al., **Tcl/Tk Tools**. O'Reilly, 1997.

[13] T. Fahringer, M. Gerndt, G. Riley, and J.L. Traff. Knowledge specification for automatic performance analysis. Technical report, ESPRIT IV Working Group on Automatic Performance Analysis: Resources and Tools (APART), http://www.fz-juelich.de/apart, November 1999.

[14] S.J. Fink, S.R. Kohn, and S.B. Baden. Flexible communication mechanisms for dynamic structured applications. *Proceedings of IRREGULAR '96*, Santa Barbara, CA, August 1996.

[15] Lawrence Livermore National Laboratory Center for Applied Scientific Computing. Datafoundry: Data warehousing and integration for scientific data management. http://www.llnl.gov/CASC/datafoundry/, 1999.

[16] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 11(2):115-128, 1997.

[17] J. Gehring. Dynamic program description as a basis for runtime optimization. Technical Report TR-002-97, Paderborn Center for Parallel Computing, Paderborn, Germany, May 1995.

[18] A. Geist, E. Mendoza, J. Myers, N. Nachtigal, and S. Sachs. DOE 2000 Electronic Notebook Project March 1999 Status Report. http://www.epm.ornl.gov/enote/status.html, August 1999.

[19] M. Gerndt and A. Krumme. A rule-based approach for automatic bottleneck detection in programs on shared virtual memory systems. *Proceedings of the International Workshop on High-Level Programming Models and Supportive Environments (HIPS '96) in conjunction with IPPS '96*, pages 10–16, Hawaii, 1996. IEEE Computer Society Press.

[20] M. Gerndt, B. Mohr, F. Wolf, and M. Pantano. Performance analysis on Cray T3E. *Proceedings of the Seventh Euromicro Workshop on Parallel and Distributed Processing*. IEEE, 1988.

[21] A.J. Goldberg and J.L. Hennessy. Performance debugging shared memory multiprocessor programs with MTOOL. *Proceedings of Supercomputing '91*, pages 481–490, Albuquerque, NM, November 1991.

[22] J. Gray, A. Bosworth, A. Layman, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. Technical Report MSR-TR-95-22, Microsoft Research, November 1995.

[23] A. S. Grimshaw, W.A. Wulf, and the Legion Team. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1):39–45, January 1997.

[24] W. Gropp, E. Lusk, and A. Skjellum. **Using MPI: Portable Parallel Programming with the Message-Passing Interface**, chapter 4. The MIT Press, 1994.

[25] W. Gu, G. Eisenhauer, E. Kraemer, J. Stasko, J. Vetter, and K. Schwan. Falcon: On-line monitoring and steering of large-scale parallel programs. *Proceedings of the Symposium on the Frontiers of Massively Parallel Computation*, McLean, Virginia, February 1995.

[26] D. Hay. XML: What is it, anyway? *Intelligent Enterprise*, 2(11), August 1999. http://www.iemagazine.com/990308/online1.shtml.

[27] M.T. Heath and J.A. Etheridge. Visualizing the performance of parallel programs. *IEEE Software*, 8(5):29–39, September 1991.

[28] B. R. Helm, A.D. Malony, and S.F. Fickas. Capturing and automating performance diagnosis: the Poirot approach. *Proceedings of the 1995 International Parallel Processing Symposium*, pages 606–613, April 1995.

[29] J.K. Hollingsworth. An application program interface for runtime code patching. Unpublished technical report, 1998.

[30] J.K. Hollingsworth and B. Buck. *DyninstAPI Programmer's Guide*. Computer Science Department, University of Maryland, College Park, MD, release 1.0 edition, July 1997.

[31] J.K. Hollingsworth and P.J. Keleher. Prediction and adaptation in Active Harmony. *Proceedings of the 7th International Symposium on High Performance Distributed Computing*, pages 180–188, Chicago, IL, 1998.

[32] J.K. Hollingsworth and B.P. Miller. Dynamic control of performance monitoring on large scale parallel systems. *Proceedings of the International Conference on Supercomputing*, Tokyo, July 1993.

[33] A. Hondroudakis and R. Procter. The tuner's workbench: A tool to support tuning in the large. P. Fritzson, editor, *Proceedings of the ZEUS-95 Workshop on Parallel Programming and Computation*, pages 212–221, Linkoping, May 1995. IOS Press.

[34] A. Hondroudakis and R. Procter. An empirically derived framework for classifying parallel program performance tuning problems. *Proceedings of the SIGMETRICS Symposium on Parallel and Distributed Tools*, pages 112–121. ACM Press, August 1998.

[35] S. Horwitz and T. Reps. Efficient comparison of program slices. *Acta Informatica*, 28:713–732, 1991.

[36] S. Horwitz. Identifying the semantic and textual differences between two versions of a program. *ACM SIGPLAN Notices volume 25: Proceedings of the ACM SIGPLAN '90 Conference on Programming Language Design and Implementation*, pages 234–245, White Plains, New York, 1990.

[37] Y. Ioannidis and M. Livny. Conceptual schemas: Multi-faceted tools for desktop scientific experiment management. *International Journal of Intelligent and Cooperative Information Systems*, 1(3):451–474, December 1992.

[38] Y. Ioannidis, M. Livny, S. Gupta, and N. Ponnekanti. ZOO: A desktop experiment management environment. *Proceedings of the 22nd International VLDB Conference*, pages 274–285, Bombay, India, September 1996.

[39] Y. Ioannidis, M. Livny, E. Haber, R. Miller, O. Tsatalos, and J. Wiener. Desktop experiment management. *IEEE Data Engineering Bulletin*, 16(1):19–23, March 1993.

[40] R. Bruce Irvin and Barton P. Miller. Multi-application support in a parallel program performance tool. Technical Report CS-TR-93-1135, Computer Sciences Department, University of Wisconsin - Madison, February 1993.

[41] M. Itzkowitz, J. Yu, A. McNaughton, P. Orelup, and C. Hanna. Visualizing performance on parallel supercomputers. In M.L. Simmons, A.H. Hayes, J.S. Brown, and D.A. Reed, editors, **Debugging and Performance Tuning for Parallel Computing Systems**, pages 181–197. IEEE Computer Society Press, 1996.

[42] K. Karavanic, J. Myllymaki, M. Livny, and B. Miller. Integrated visualization of parallel program performance data. *Parallel Computing*, 23:181–198, 1997.

[43] T. Keller and D. Jones. Metadata: The foundation of effective experiment management. Technical report, Environmental Molecular Sciences Laboratory, Pacific Northwest National Laboratory, 1996.

[44] D.J. Kerbyson, E. Papaefstathiou, and G.R. Nudd. Application execution steering using on-the-fly performance prediction. In P. Sloot, M. Bubak, and B. Hertzberger, editors, **Proceedings of the High Performance Computing and Networking International Conference and Exhibition**, Lecture Notes in Computer Science #1401, pages 718–727. Springer-Verlag, 1998.

[45] J. Kohn and W. Williams. ATExpert. *Journal of Parallel and Distributed Computing*, 18:205–222, 1993.

[46] K. Kunchithapadam and B.P. Miller. Integrating a Debugger and a Performance Tool for Steering. In M.L Simmons, A.H. Hayes, J.S. Brown, and D.A. Reed, editors, **Debugging and Performance Tools for Parallel Computing Systems**, pages 53–64. IEEE Computer Society Press, 1996.

[47] L. Lamport. Time, clocks and the ordering of events in a distributed system. *Communications of the ACM*, 21(7), July 1978.

[48] M. Livny, R. Ramakrishnan, K. Beyer, G. Chen, D. Donjerkovic, S. Lawande, J. Myllymaki, and K. Wenger. Devise: Integrated querying and visual exploration of large datasets. *Proceedings of ACM SIGMOD*, May 1997.

[49] T. Ludwig, R. Wismuller, V. Sunderam, and A. Bode. OMIS – on-line monitoring interface specification (version 1.0). Technical report, Institute Fur Informatik der Technische Universitat Munchen, February 1996.

[50] G. Lyon, R. Snelick, and R. Kacker. Synthetic-perturbation tuning of MIMD programs. *The Journal of Supercomputing*, 8:5–28, 1994.

[51] A.D. Malony, D.H. Hammerslag, and D.J. Jablonowski. Traceview: A trace visualization tool. *IEEE Software*, 8(5):19–28, September 1991.

[52] A. Mathur and M. Abrams. Toward a machine assisted software performance diagnosis methodology. Technical Report TR 93-12, Virginia Polytechnic Institute and State University Department of Computer Science, 1993.

[53] C. L. Mendes. Performance prediction by trace transformation. *Proceedings of the Fifth Brazilian Symposium on Computer Architecture*, Florianopolis, September 1993.

[54] B. Miller, M. Callaghan, J. Cargille, J. Hollingsworth, R. Irvin, K. Karavanic, K.Kunchithapadam, and T. Newhall. The Paradyn parallel performance measurement tool. *IEEE Computer*, 28(11):37–46, November 1995.

[55] W. Nagel, A. Arnold, M. Weber, H. Hoppe, and K. Solchenbach. Vampir: Visualization and analysis of MPI resources. *Supercomputer 63*, 12(1):69–80, 1996.

[56] D. Nelson. The laboratory notebook technical manual. Technical Report LA-UR 88-1256, Los Alamos National Laboratory, Los Alamos, NM, 1990.

[57] K.J. Ottenstein and L.M. Ottenstein. The program dependence graph in a software development environment. *Proceedings of the ACM SIGSOFT/SIGPLAN Software Engineering Symposium on Practical Software Development Environments*, volume 19 of *ACM SIGPLAN Notices*, pages 177–184, Pittsburgh, PA, May 1984.

[58] D.M. Pase. *Dynamic Probe Class Library (DPCL): Tutorial and Reference Guide*. IBM Corporation, RS/6000 Development, Poughkeepsie, New York, version 0.1 edition, June 1998.

[59] R.Ramakrishnan. **Database Management Systems**. WCB/McGraw-Hill, 1998.

[60] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing (HPDC)*, pages 140–146, Chicago, IL, July 1998.

[61] D.A. Reed, R.A. Aydt, R.J. Noe, P.C. Roth, K.A. Shields, B.W. Schwartz, and L.F. Tavera. Scalable performance analysis: The Pablo performance analysis environment. IEEE CS Press, editor, **Proceedings of the Scalable Parallel Libraries Conference**, pages 135–142, Los Alamitos CA, 1993.

[62] S.K. Reinhardt, J.R. Larus, and D.A. Wood. Tempest and Typhoon: User-level shared memory. In *Proceedings of the 21st Annual International Symposium on Computer Architecture*, April 1994.

[63] M. Rennhackkamp. Extending relational DBMSs. DBMS Magazine, http://www.dbmsmag.com, October 1997.

[64] T. Reps. Algebraic properties of program integration. *Science of Computer Programming*, 17:139–215, 1991.

[65] R. Ribler, A. Mathur, and M. Abrams. Visualizing and modeling categorical time series data. Technical report, Department of Computer Science, Virginia Polytechnic Institute and State University, August 1995.

[66] R.L. Ribler, J.S. Vetter, H. Simitci, and D.A. Reed. Autopilot: Adaptive control of distributed applications. *Proceedings of the High-Performance Distributed Computing Conference*, July 1998.

[67] G.D. Riley and J.R. Gurd. Requirements for automatic perfomrance analysis. Technical report, ESPRIT IV Working Group on Automatic Performance Analysis: Resources and Tools (APART), http://www.fz-juelich.de/apart, November 1999.

[68] R. H. Saavedra and A.J. Smith. Analysis of benchmark characteristics and benchmark performance prediction. Technical Report Computer Science Technical Report USC-CS-92-524, University of Southern California, 1992.

[69] S. Shende, A.D. Malony, J. Cuny, and K. Lindlan. Portable profiling and tracing for parallel, scientific applications using C++. *Proceedings of SPDT98*, pages 134–145, Welches, OR, 1998.

[70] A.B. Sinha and L.V. Kale. Towards automatic performance analysis. *Proceedings of the 1996 International Conference on Parallel Processing*, pages III–53–60, 1996.

[71] R. Snelick, J. Jaja, R. Kacker, and G. Lyon. Synthetic-perturbation techniques for screening shared memory programs. *Software - Practice and Experience*, 24(8):679–701, August 1994.

[72] R. Sosic and D. Abramson. Guard: A relative debugger. *Software Practice and Experience*, 27(2):185-206, February 1997.

[73] A. Waheed and D.T. Rover. Performance visualization of parallel programs. *Visualization '93*, San Jose, CA, October 1993.

[74] M. Weiser. Program slicing. *IEEE Transactions on Software Engineering*, SE-10(4):352–357, July 1984.

[75] R.C. Whaley and J. Dongarra. Automatically tuned linear algebra software. *Proceedings of SC98*, Orlando FL, Nov. 1998. ACM/IEEE.

[76] J. Wiener and Y. Ioannidis. A moose and a fox can aid scientists with data management problems. *Proceedings of the 4th International Workshop on Database Programming Languages*, pages 376–398, New York, NY, August 1993.

[77] W. Williams, T. Hoel, and D. Pase. The MPP Apprentice performance tool: Delivering the performance of the Cray T3D. In K.M. Decker and R.M. Rehmann, editors, **Programming Environments for Massively Parallel Distributed Systems**. Birkhauser, 1994.

[78] F. Wolf and B. Mohr. Earl: A programmable and extensible toolkit for analyzing event traces of message passing programs. Technical Report FZJ-ZAM-IB-9803, Forschungszentrum Juelich GmbH, April 1998.

[79] Z. Xu, J. Larus, and B. Miller. Shared-memory performance profiling. *Proceedings of the 6th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, Las Vegas, Nevada, June 1997.

[80] J. Yan, S. Sarukhai, and P. Mehra. Performance measurement, visualization and modeling of parallel and distributed programs using the AIMS toolkit. *Software – Practice and Experience*, 25(4):429–461, April 1995.

[81] W. Yang. Identifying syntactic differences between two programs. *Software – Practice and Experience*, 21(7):739–755, July 1991.