VII. Tools for Parallel Computing: A Performance Evaluation Perspective

Allen D. Malony

Department of Computer and Information Science
University of Oregon

Summary. To make effective use of parallel computing environments, users have come to expect a broad set of tools that augment parallel programming and execution infrastructure with capabilities such as performance evaluation, debugging, runtime program control, and program interaction. The rich history of parallel tool research and development reflects both fundamental issues in concurrent processing and a progressive evolution of tool implementations, targeting current and future parallel computing goals. The state of tools for parallel computing is discussed from a perspective of performance evaluation. Many of the challenges that arise in parallel performance tools are common to other tool areas. I look at four such challenges: modeling, observability, diagnosis, and perturbation. The need for tools will always be present in parallel and distributed systems, but the emphasis on tool support may change. The discussion given is intentionally high-level, so as not to exclude the many important ideas that have come from parallel tool projects. Rather, I attempt to present viewpoints on the challenges that I think would be of concern in any performance tool design.

Keywords: parallel performance environments, performance evaluation, performance diagnosis, perturbation, observability, measurement, prediction, parallel tools

1. Introduction

Computer systems are arguably the most complex machines ever invented, and parallel computers and distributed computers are the most complex computer systems. In simple terms, parallel and distributed systems are designed to support concurrent computer operations. Although concurrent actions are a common phenomenon in the natural world, encoding concurrency in a computer system such that the computation is "correct" is not a simple task, even for seemingly trivial problems. Parallel systems also have a more specific aim

to support the simultaneous execution of concurrent operations for achieving high performance. Maintaining high efficiency in parallel execution further complicates how parallel systems are programmed and used.

Parallel systems are important as computing platforms because they offer the potential to solve problems requiring multiple computing resources and high-end performance. However, this potential cannot be actualized without the support of *tools*, particularly tools for *performance analysis* and *debugging*. Designing and developing tools for parallel systems is intrinsically difficult due to the complexity, both architecturally and operationally, of the computing space represented. In general, a tool should

- Incorporate a *model* of the system and its operation in order to reduce problem complexity;
- Be sensitive to *observability* constraints that limit the scope of what is knowable of and about the system;
- *Diagnose* important system states so as to aid the user in analysis; and
- Account for possible *perturbation* of the system caused by instrumentation intrusion or perturbation of the model results cause by model abstractions.

A tool's utility is determined partly by the sophistication of the system model on which it is based, and this sophistication requires knowledge of system operation and behavior. Given the complexity of parallel platforms, this knowledge may be difficult to obtain. Utility is also affected by the ability to capture the requisite information about the system under certain access, accuracy, and granularity constraints. Certain information may be unobtainable and, hence, unavailable to the tool. Perhaps the most important aspect of a tool is its benefit to problem solving. A tool can be a tremendous aid in discovering and avoiding parallel computing problems if it supported the ability to diagnose system states. However, tools can also influence the system when making measurements for purposes of analysis. In the worst case, system behavior can be perturbed to the point that observations are unreliable and the models that use the data lead to misleading conclusions.

There is a rich research history in the field of parallel and distributed tools. Many important contributions have been made to understand concurrency, control program behavior, debug program correctness, evaluate performance, and present results to users. Rather than attempt a comprehensive summary of these contributions, the reader is directed to the conference proceedings, journals, and bibliographic databases given in the references for the extensive background in the field. In particular, the reader can find excellent recent research surveys of the field in [HM98, RB98, RWM+98, HML95, RC98]. This chapter instead presents a higher-level view of parallel tools than what might be appear in tool surveys. Out of respect for the many important tools that have been developed, only a few will be cited as examples of more general themes. The perspective presented is based on a consideration of the four challenging problems for tools listed above — modeling, observability,

diagnosis, and perturbation — specifically as they concern tools for parallel performance evaluation. It is my hope that this more abstract discussion of parallel performance evaluation tools will provide some insight into the parallel tools field as a whole.

In the remainder of the chapter, I first introduce (Section 2) the general problem of parallel performance evaluation as a motivation for tools. In Section 3, a performance environment is advocated as a general guiding framework for tool development. Section 4 discusses the use of models in tool design and how, given a model, performance measurement and analysis techniques are implemented. The problem of performance observability is discussed in Section 5. In Section 6, the concept of a performance diagnosis system is introduced. Parallel performance can be perturbed by several factors. The challenge of performance perturbation analysis is considered in Section 7. Finally, concluding remarks are given in Section 8.

2. Motivation

Two years after Scherr's classic Ph.D. dissertation [Sch65], considered by some to be the seminal work in computer systems performance evaluation [Fer78], Amdahl published his now famous paper on the limits of parallel performance speedup [Amd67]. Although there have been significant advancements in performance evaluation techniques since Scherr's thesis (particularly in the areas of monitoring, simulation, analytic modeling, and bottleneck analysis), "Amdahl's Law"[1] has arguably remained the most fundamental (and the most controversial) result in parallel systems performance evaluation:

> "For over a decade prophets have voiced the contention that the organization of a single computer has reached its limits and that truly significant advances can be made only by interconnection of a multiplicity of computers in such a manner as to permit cooperative solution. ... Demonstration is made of the continued validity of the single processor approach. ... A fairly obvious conclusion which can be drawn at this point is that the effort expended on achieving high parallel processing rates is wasted unless it is accompanied by achievements in sequential processing rates of very nearly the same magnitude. ... At any point in time it is difficult to foresee how the previous bottlenecks in a sequential computer will be effectively overcome." [Amd67]

[1] Amdahl's Law states that if s is the fraction of a computation that must be executed serially, then the speedup of the computation is bounded above by $\frac{1}{s+(1-s)/n}$, where n is the number of processors used. Note, $\lim_{n\to\infty} \frac{1}{s+(1-s)/n} = \frac{1}{s}$.

Amdahl's Law is fundamental in its simplicity and its generality: it defines an upper bound on the performance of a parallel computation, relative to its sequential execution time, in terms of a single software parameter (the fraction of sequential computation) and a single hardware parameter (the number of processors). Amdahl's Law is controversial because this speedup bound places severe limits on the performance benefits of parallel computer systems; in general, it implies that achieving good parallel performance will be exceedingly difficult.

The last thirty years attest to the veracity of Amdahl's arguments. Several studies have extended his simple speedup model to further quantify parallel execution overheads, effects of execution partitioning strategies, and tradeoffs in speedup versus efficiency. The principal issue is one of *parallel performance scalability*: how does the performance of a parallel system change relative to the hardware and software effects of increasing the number of processors used to execute a program and/or increasing the size of a program's input? Various *scalability metrics* have been defined to evaluate whether parallel computers can deliver their performance potential. The most recognized of these, "scaled speedup," has even been used to refute the suitability of Amdahl's Law for evaluating the performance of large-scale parallel systems [Gus88]. However, regardless of the metric used, the critical performance question remains: how is the performance potential offered by parallel computer systems achieved by general-purpose parallel applications?

Ferrari characterized a *performance evaluator* as one that tries to solve computer systems problems and uses the most appropriate techniques and tools at hand (a process Ferrari calls *applied performance evaluation*) [Fer78]. In the context of parallel computer systems, two questions are of importance:

- What is the role of the performance evaluator (and, in general, performance evaluation)?
- What are the performance problems and the appropriate techniques and tools used to solve them?

The discussion of Amdahl's Law gives us a point of reference for addressing these questions in parallel computing.

First, delivered performance is the *raison d'être* of parallel computer systems: if the purpose of a sequential computer system is to execute a program to perform a computation, the purpose of a parallel computer system is to execute a program faster than a sequential computer system. Amdahl presents this purpose in the form of a single performance metric, *speedup*, which can be use to evaluate the effectiveness of a parallel program's execution on a parallel machine. Although Amdahl's Law was used to downplay the importance of parallel systems, it equally represents a challenge: good performance is possible, but it will be difficult to obtain. In this respect, the role of performance evaluation in parallel systems is to understand the causes of actual performance behavior for purposes of performance optimization.

Second, parallel performance is an inherently complex metric. Although the limits of parallel performance (both offered potential and speedup bounds) are relatively simple to define (e.g., Amdahl's Law), the *performance space* is large, ranging from the performance achieved on one processor to the peak performance on all processors. Furthermore, the difference between potential and delivered performance on a parallel machine can be significant. Amdahl's Law describes these variations in terms of a single parameter, but, in general, many factors can contribute to performance variability. These factors are interdependent, and seemingly minor changes in their relationship can often induce large changes in the performance achieved. Hence, the performance space is multi-dimensional and can be highly irregular.

Third, parallel performance is difficult to measure, characterize, and understand. It is known that Amdahl's Law is an oversimplification of the cause of parallel performance degradation (i.e., sequential execution); clearly, other overheads limit performance. Even so, determining the amount of time a parallel computation spends in sequential execution can be nontrivial. In general, parallel performance factors are dynamic in time, distributed in location and state, and parallel in occurrence. Although a parallel system is a deterministic automaton, and, in principle, one could envision having full knowledge of system activities, the complexity of hardware and software restricts performance observation: any measurement will be incomplete and any characterization will be an abstraction of true performance behavior. Moreover, performance behavior (the interaction and importance of performance factors) can be highly sensitive to changes in execution context.

Finally, parallel performance is the product of a specific combination of parallel system (hardware and system software) and application program. The performance evaluation requirements are therefore dictated by the specific needs of the problem context and the user. In contrast to sequential computers, the performance evaluation of parallel systems is more specialized in its role and more personalized in its application; in fact, the "performance evaluator" is most often the parallel program developer, because intimate knowledge of the program is usually required to hunt down "performance bugs." Although the advances in performance evaluation technology for sequential systems can be leveraged in the parallel domain, the individuality and complexity of the parallel performance problem mandates that the techniques be uniquely and carefully applied. New parallel performance evaluation techniques must also be developed, with an orientation towards performance optimization.

Since Amdahl's paper was published, there has been a growing crisis in parallel performance evaluation: the technological advances in parallel computer systems (hardware and software) are increasing the complexity of the computational environment, progressively diminishing the general user's ability to operate these systems near the high-end of their performance range. Presently, the crisis is acute. There are scalable parallel machines being intro-

duced today whose performance characteristics are reported only as unachievable peak performance numbers. Furthermore, the system support for obtaining performance data and the integration of this data into the overall system environment are woefully inadequate. Although the growing acceptance of massively parallel computing and the arguments for performance scalability continue to uphold the promise of parallelism, the intellectual challenge to achieve good parallel performance, as originally articulated by Amdahl over thirty years ago, remains.

The development of performance evaluation environments for parallel computer systems is one approach to overcoming this crisis. The idea is to develop an environment for solving performance problems based on a methodology of applied parallel performance evaluation and an integrated set of tools for performance modeling, measurement, analysis, presentation, and prediction. The goal is to relieve the user of the manual effort of performance investigation while reducing the intellectual burden of understanding complex performance behavior. The above discussion supports the need for environments for parallel performance evaluation as a way to reduce the complexities of the performance problem for the user. However, to be effective, performance evaluation environments must be carefully developed to be an integral component of a parallel system's design and use.

3. Environment Design

The *scientific method* — the systematic testing of hypotheses through controlled measurement of observable phenomena, analysis of collected data, and modeling of empirical results — has been advocated as the working definition of "experimental (computer) science" [Den80] and as the basis of the "quantitative philosophy of performance evaluation" [Fer78]. Denning remarked that "science classifies knowledge", and that "experimental science classifies knowledge derived from observations" [Den80]. The advancement of computer science knowledge will increasingly require an experimental approach — the building of experimental apparatus to understand new ideas and to validate their usefulness in practice. Denning commented that the experimental apparatus is not usually the subject of such research and that unless the apparatus is used to obtain significant new knowledge, the research is not substantive. However, in any field (and performance evaluation, in particular), progress in experimental science is inextricably coupled with advances in observational technology; the ability to test hypotheses that predict the existence of heretofore undetected phenomena intimately depends on the requisite tools to more accurately measure and analyze known phenomena. In performance evaluation, the new "scientific" knowledge sought is the understanding of and solution to computer systems problems. The quantitative tools used are the experimental apparatuses of applied performance evaluation. Better tools to

observe and model performance will lead to better solutions to performance problems.

Although the scientific method's systematic measurement and hypothesis testing is both necessary and desirable for parallel performance evaluation, the limited understanding of parallel execution and the complexities of performance observation make the construction of parallel performance environments based on the scientific approach especially difficult. In general, the environment design must meet two basic requirements:

- The need to specify new parallel performance problems in terms of the characteristics of the parallel system, the structure and parameters of the application program, the stored *performance knowledge*, and the current, empirical performance data (*performance hypothesis formulation*).
- The need to conduct performance experiments (including measurement, analysis, presentation, and modeling) to assess performance behavior (*performance observation*).

The first requirement reflects the notion that effective parallel performance evaluation will involve the application of a cyclic (scientific) methodology of designing new performance experiments based on cumulative system and performance information. This includes the initial targeting of performance hypotheses from experiences with other performance problems and the progressive refinement of the hypotheses as a result of performance experiments. The second requirement focuses on the issues concerned with building and applying tools to test performance hypotheses. In particular, the need for performance data to validate a hypothesis must be balanced against the observational capabilities of the performance tools as constrained by the parallel system hardware and software.

Fig. 3.1 shows a general design framework for a parallel performance evaluation environment based on the scientific approach. This framework is more a reflection of an idealized environment than one that might be realized in practice, due to the design complexities and implementation tradeoffs involved. However, our belief is that this design view serves as a useful basis to discuss some of the challenges that arise when attempting to develop parallel performance tools. Because the development of any set of performance tools will involve tradeoffs between feasibility, functionality, accuracy, and cost, considering these issues in a general context will provide us with a foundation to evaluate the capabilities of present environments and to describe the requirements of parallel performance environments of the future.

The Pablo[2] project [RAN+93] best exemplifies our environment design model as a methodological and experimental framework for developing a suite of parallel performance tools driven by the evolving requirements for performance analysis in parallel systems and by the types of performance problems these systems present. The Pablo research has explored all aspects of the

[2] The Pablo project homepage is *http://www-pablo.cs.uiuc.edu*.

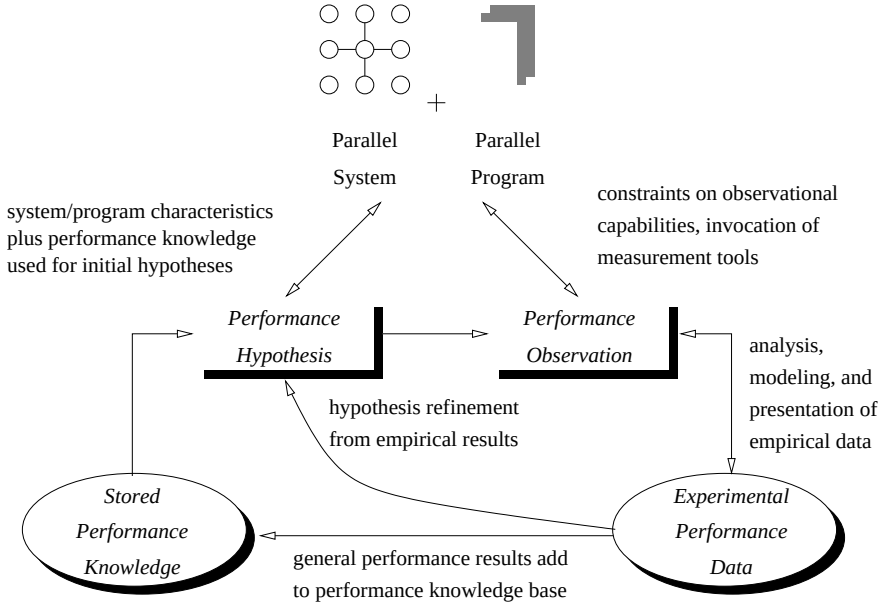

Fig. 3.1. Parallel Performance Evaluation Environment Design

model throughout the last ten years, demonstrating a range of techniques for modeling, measurement, analysis, and visualization. Its current use in the Delphi system [RPF99] demonstrates the logical extension of a performance evaluation model into an integrated environment that includes knowledge of the parallel computing platform for modeling at multiple system levels, parallelizing compiler technology for language-level performance instrumentation, mapping, and prediction, and distributed computing support for tool interoperation.

4. Parallel Performance Paradigms

As suggested in Fig. 3.1, the characteristics of the parallel system (hardware and software) and the application program will be important determinants in the development and use of a parallel performance environment. Although performance problems are often addressed in a specific system/program context, the ability to apply conceptual abstractions of parallel performance to guide performance investigation and to generalize results from performance experiments will be important for effective environment use. Here we use the term *parallel performance paradigm* to represent the combination of an abstract model of performance and the processes (measurement and analysis)

needed to apply and integrate the model in performance problem solving. To the extent that parallel performance paradigms can be realized in actual performance environments, they will serve to help reduce the intellectual complexity of performance evaluation for the user.

What counts as a useful parallel performance paradigm? On a basic level, this question implies that there is (are) accepted definition(s) of parallel performance. There are three classes of quantitative "performance indices" for evaluating computer systems: *productivity* (i.e., throughput), *responsiveness* (i.e., turnaround or response time), and *utilization*. Of these, responsiveness is the index of merit for parallel performance. Thus, for our purposes, good parallel performance paradigms will be those that can express, in some general manner, the influence of the most important parallel system and application factors on response time performance.

The execution time speedup model, represented in Amdahl's Law, is an example of a simple, universal parallel performance paradigm. It is simple because the number of processors and sequential execution time are the only performance factors that matter in the model. It is universal because the paradigm can be used for any parallel environment or program both for performance experimentation — the measurement of speedup as function of the number of processors — and for performance prediction — the estimation of the performance on n processors based on the sequential execution time measurements on m processors. However, execution speedup is a poor paradigm for investigating performance problems (i.e., performance diagnosis), serving only an indicator of good or bad parallel performance. The performance scalability extensions to the basic speedup models help to quantify the influence of additional performance factors, but are still too general to explain performance behavior.

The power of a parallel performance paradigm comes from both its ability to represent performance abstractly, for comparative and predictive purpose, and its ability to characterize performance specifically, for reasons of diagnosis and tuning. The generality of the underlying model can be at odds with the specificity needed in performance measurement and execution analysis. Paradigms can either be extended to add performance metrics while keeping the analysis models simple, or be made more specific with greater model detail and analysis resolution, but at the risk of less general application.

A paradigm based on the parallel execution profile of a particular performance metric (e.g., execution time or degree of parallelism) is important because it expresses a procedure for evaluating performance limiting behavior. The profile might be coarse-grained, describing parallel performance by a set of summary statistics, or fine-grained, representing parallel performance as a time sequence of metric values. For instance, the common execution time profile orders code segments according to their impact on total execution time; code segments representing a higher percentage of the total time might be candidates for performance optimization. A parallelism profile, on the

other hand, reflects a history of parallelism behavior and highlights regions where there is the potential for parallelism improvement. However, parallel performance paradigms based on profiles alone are insufficient as a basis for formulating performance hypotheses because they offer no explanation as to why the performance behavior occurred.

Alternatively, parallel performance paradigms based on the properties of the parallel execution environment and the program's computation have a greater potential for investigating performance problems. For instance, performance models can be defined with respect to computational structures for parallel workflow (e.g., *pipelined*, *master-slave*, or *work queue*), or with respect to parallel work synchronization mechanisms (e.g., *fork-join*, *barrier*, or *message passing*) or scheduling algorithms (e.g., *task level* or *loop level*; *self scheduling* or *block scheduling*). A parallel performance paradigm based on such models will specify the required measurements for the type of structures, mechanisms, and scheduling algorithms used and will designate the associated types of analysis to be undertaken. Higher level performance models are based on computational abstractions (e.g., *control flow* versus *data flow*; *control parallel* versus *data parallel*; or *single program, multiple data* versus *bulk synchronous parallel*), which can be used to refine and to prioritize lower level measurements to performance problems in the computational domain. The important point here is that the performance paradigm is founded on the characteristics of the parallel execution of interest, thereby providing a means for expressing and evaluating observed performance behavior.

Parallel performance paradigms provide a framework for defining performance measurements, aiding in performance diagnosis, and supporting performance prediction. During the performance evaluation process the paradigms should be modified and refined as new performance knowledge is gained through observation. For this reason, multi-paradigm approaches are common. A paradigm might employ *resource usage models* to identify performance anomalies and then *event models* to identify computational states that lead to the anomalies. The *program activity graph* is a well-known multi-paradigm representation that uses nodes in the graph to signify significant events in the program's execution and arcs to show the ordering of events within a process or the synchronization dependencies between processes. By overlaying parallel program performance metrics one can see how the inter-event, inter-process dependencies in a parallel program influence which procedures are important to a program's execution time.

Many parallel tool researchers have naturally applied paradigms in their work. In their paper "Analyzing Parallel Program Execution Using Multiple Views" [LMF90], LeBlanc, Mellor-Crummey, and Fowler emphasize the need to develop a (general) unified approach to parallel program analysis that supports the creation and integration of multiple views of an execution and allows the user to tailor views to specific analysis. The Paradyn[3] project [MCC95]

[3] The Paradyn project homepage is *http://www.cs.wisc.edu/ paradyn*.

is an excellent example of this approach in practice. Paradyn is based on a flexible model of performance instrumentation and a well-defined notion of performance bottlenecks and program structure, so that measurements can be made for investigating bottlenecks associated with specific causes and specific parts of a program. Measurements are possible at different levels of the parallel system and open interfaces for performance analysis and visualization are provided for constructing alternative performance tools.

5. Performance Observability

In order to evaluate the performance of a parallel application executing on a parallel computer system, certain aspects of application and system behavior must be made observable. Whereas a performance paradigm provides a conceptual foundation for investigating and understanding performance problems, an environment must also support a means for performance experimentation — the measurement, analysis, and presentation of parallel performance phenomena. *Parallel performance observability* is the ability to accurately capture, analyze, and present (collectively, to *observe*) information about the performance of a parallel computer system [Mal90]. Tools for performance observability must balance the *need* for performance data against the *cost* of obtaining it (environment complexity and performance intrusion). Too little performance data makes performance evaluation difficult; too much data can be complex to analyze and might perturb the measured system. What combination of tools for performance observation is appropriate for parallel computer systems? How do the architecture, hardware, and system software affect how performance data is collected? What performance events can and cannot be observed? How do the performance evaluation tools affect the performance being measured? How should performance information be conveyed to the performance analyst? Unfortunately, there is no formal approach to determine, given a parallel performance evaluation problem, how to accurately "observe" parallel execution in order to produce the required performance results. Furthermore, any parallel performance experiment will ultimately be constrained by the capabilities of the available tools for performance observation.

Performance measurement is the foundation of performance observability. If an experiment cannot be constructed, even in principle, to measure a phenomenon, it cannot operationally be said to exist. If a phenomenon cannot be measured in practice, it cannot be observed. The complexity of parallel computer systems makes *a priori* performance prediction difficult and experimental performance measurement crucial. A complete characterization of software and hardware dynamics is needed to understand the performance of parallel execution and requires efficient techniques for runtime performance instrumentation and data collection. Although performance measurement is a necessary component of parallel performance environments, the degree and

type of performance measurement support depends on its intended purpose, and the nature of the performance experiments to be conducted defines the needed capabilities of the performance monitoring system, its observational detail, and acceptable cost.

The diversity of parallel performance problems makes it difficult to develop a single set of performance monitoring techniques. For every performance experiment, there nonetheless exists a minimal set of required events that must be captured. In general, a parallel execution can be regarded as a sequence of *actions* representing the computational activities one wishes to observe. The execution of an action generates an *event*, an encoded instance of the action. A "performance measurement" can be viewed as the collection of a (possibly infinite) set of events. Indeed, event-based models have been widely used to describe program behavior and to define techniques for performance measurement. A system can be represented in terms of the observable effects and interactions of system components as represented by a stream of characteristic atomic behaviors (i.e., events), giving an abstract view of program behavior in terms of a sequence of hierarchically defined events. In general, the more detailed the measurement, the more data can be provided to a performance model, allowing for more detailed analysis.

However, before events can be analyzed, they must be detected and captured by a monitoring system. The selection of instrumentation and data collection tools defines both the granularity and detail of performance data that can be measured. Events of interest can occur at different observation points (hardware and software), which may or may not be accessible. Furthermore, depending on the type of measurement desired, the amount of performance data that must be collected and stored can vary. In practice, the need to observe time-dependent parallel performance behavior and the problems associated with the specification of complex performance events and their detection often necessitates measurement solutions that capture a large volume of time-based event data (e.g., tracing) for later analysis.

The design and development of tools for detailed performance instrumentation and data capture on parallel machines is non-trivial, often requiring significant engineering effort for their implementation. Monitoring solutions based on tracing must solve several implementation problems, including event timestamp consistency (both in accuracy and synchronization), trace buffer allocation, tracing overhead, and trace I/O. Although software recording of performance data suffices for low frequency events, capture of detailed, high-frequency performance data ultimately requires hardware support if the performance instrumentation is to remain efficient and unobtrusive. Alternatively, techniques to control monitoring overhead dynamically by changing instrumentation during execution have been successful in reducing significantly the amount of performance data captured.

The lesson of measurement detail versus accuracy is that because parallel programs are composed of multiple threads of control, the accuracy of perfor-

mance characterization depends on some global knowledge of program state. Although behavioral models of parallel program execution allow events to be measured independently for each thread of execution and then combined to determine global states, certain measurements must additionally be made to preserve global performance data integrity (e.g., "global" time measurement). That is, parallel program measurement must not only capture thread actions that reflect logical, operational behavior, but also data that will be used to establish an accurate reference for performance analysis (e.g., global time reference).

There have been many research studies on the different aspects of the performance observability problem discussed above, particularly in respect to the problem of instrumentation and monitoring. Modeling and evaluating design alternatives for performance observability will always remain a challenge. Rover, Waheed, and Hollingsworth [RWH98] took on that challenge in their study of design alternatives for on-line instrumentation systems based on different criteria for effectiveness, intrusion, and complexity of implementation. Their results establish models based on metrics derived from these criteria as they applied in different system architecture contexts: network of workstations (NOW), symmetric multiprocessors (SMP), and massively parallel processing (MPP) systems. These models are intended to be used to provide early feedback to tool developers regarding instrumentation overhead and performance.

6. Performance Diagnosis

Given a foundation for performance modeling and a means for performance measurement, a parallel performance environment can support an process commonly known as *performance debugging*. When a performance problem (i.e., a *performance bug*) is present, tools in the environment can be used to investigate the problem, identify its source, and provide data for performance improvement. Performance debugging is the process of applying these tools. How performance bugs are identified and how they are explained is the problem of *performance diagnosis* [MH99]. Expert parallel programmers often improve program performance enormously by experimenting with their programs on a parallel computer, then interpreting the results of these experiments to suggest changes. This expertise has had difficulty finding its way into performance environments for two reasons. First, researchers lack a theory of what diagnosis methods work, and why. There is no formal way to describe or compare how expert programmers solve their performance diagnosis problems in particular contexts. There is no standard theory for understanding diagnosis system features and fitting them to the programmer's particular needs. As a result, researchers cannot easily compare and evaluate the performance debugging tools they produce, and many potential users do not find systems that are applicable to their performance diagnosis

problems. Second, performance debugging tools are not easily adaptable to new requirements. Highly automated systems, while providing considerable help to the programmer, are hard to change, hard to extend, and hard to combine with other systems.

In simple terms, performance diagnosis guides the programmer in identifying poor decisions made in parallel programming or in configuring parallel execution. By finding and explaining the chief performance problems of the program, diagnosis helps the programmer determine which decisions had the worst performance effects and how those effects might be repaired. During performance diagnosis, the programmer decides which performance data to collect, which features to judge significant, which hypotheses to pursue, and what confirmation to seek. A *performance diagnosis method* can be defined as the policies used to make such decisions, and a *performance diagnosis system* as a suite of programs that supports some diagnosis method, ideally in an automatic way. The research problem is to define a theory of performance diagnosis methods and to use that theory to create more automated, adaptable performance diagnosis systems.

To attack the first obstacle to performance diagnosis systems, lack of theoretical justification, a "knowledge-level" theory of performance diagnosis must be developed. In particular, a knowledge-level theory must answer the question, What knowledge does a programmer use to choose actions to meet performance diagnosis objectives? The theory breaks the question down into two parts: What methods do expert programmers use?, and How can we rationalize the programmer's choice of methods? Underlying the challenge of developing a knowledge base is the fact that different performance metrics provide useful information for different types of performance bottlenecks (bugs). This is one reason for the emphasis on an underlying parallel performance paradigm: it provides a context for performance data interpretation. The use of multiple paradigms help to address different performance issues. Since every parallel application may have a different set of possible performance problems, the user is often left to select the appropriate application; a comprehensive pre-enumeration of possible performance diagnoses (hypotheses) is difficult. However, recent research has tried first to provide better guidance to the user by treating the problem of finding a performance bottleneck as a search problem, and second to define this space by describing "fault taxonomies" for the performance problems that commonly arise [MH99].

The forgoing discussion suggests one reason why performance diagnosis systems are not widely used: they are not adaptable to a wide variety of contexts. To help arrive at an initial diagnosis, performance diagnosis systems define a limited fault taxonomy, a finite set of performance problems to look for. To date, systems have derived this set from the workings of the programming language and runtime system they support. It follows that the diagnosis systems are limited to a particular class of target machines and

environments (more abstractly, parallel performance paradigms). However, if we could find methods and rationales that cut across a substantial number of diagnosis systems, then we might be able to identify general methods, and differences among systems could then be studied to extract rationale.

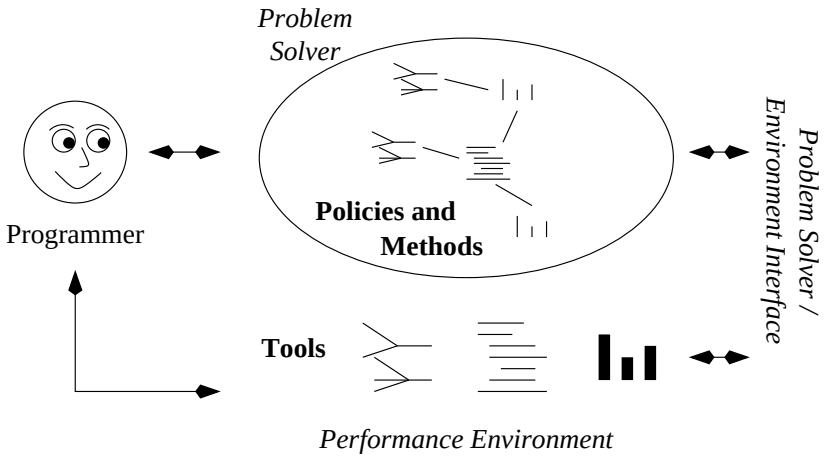

Fig. 6.1. Framework of a Parallel Performance Diagnosis System

The second obstacle to the acceptance of diagnosis systems — poor automation and adaptability — can be addressed by a new diagnosis system framework (Fig. 6.1). Here, policies would be interpreted by a goal-oriented problem solver to choose methods to pursue. The methods would in turn interface with the programming environment to apply tools to carry out experiments. The problem solver is based on knowledge-level theory of expert performance diagnosis and is able to perform actions to accomplish a method's diagnostic goal, often instructing a tool to perform some measurement or analysis experiment that will add new information to the performance database. The purpose of the environment interface is to support adaptable diagnosis by separating diagnosis methods from the software tools that support those methods. It specifies diagnosis actions in terms of their effects on a high-level performance database. Methods can thus execute these actions and track their effects without knowing what commands are sent to tools, or how data and programs are stored in files. As a result, general methods can be adapted unchanged to new tools. One can reuse knowledge about which steps to take in performance diagnosis in contexts where the manner in which those steps are taken differs significantly.

The ideas above have been captured in our Poirot performance diagnosis research [MH99]. One of the unique aspects of this work is that we have recon-

structed or "reverse engineered" some answers to the above questions (i.e., the knowledge-level theory) from a survey of research papers on performance diagnosis systems, and from the case studies that appeared in those papers. The goal was to find methods and rationale that cut across a substantial number of diagnosis systems. Each performance diagnosis system was viewed as a collection of methods for heuristic classification. Similarities among systems were analyzed to identify general methods, and differences among systems were studied to extract rationale. The result of the survey is a rationalized classification of performance diagnosis systems, a systematic description of what methods performance diagnosis systems use, and why they use them. These results can be found in [MH99].

Hollingsworth's W3 search model [HML95] is an excellent representative of a diagnosis fault taxonomy that has been actualized in a working tool, the Paradyn Performance Consultant [MCC95]. The W3 search model looks for performance problems through an iterative process of refining the answers to three questions: *why* is the application performing poorly, *where* is the bottleneck, and *when* does the problem occur. To answer the why question, tests are conducted to identify the type of bottleneck (e.g., synchronization, I/O, computation). Answering the where question isolates a performance bottleneck to a specific resource used by the program (e.g., a disk system, a synchronization variable, or a procedure). Answering when a problem occurs, tries to isolate a bottleneck to a specific phase of the program's execution. The Performance Consultant uses the W3 search model to automatically guide it in instrumentation and analysis as the program is executing.

7. Performance Perturbation

Computer system performance evaluation is subject to the same instrumentation pitfalls facing any experimental science; notably, uncertainty and instrumentation perturbation. Instrumentation, no matter how unobtrusive, introduces performance perturbations, and the degree of perturbation is proportional to the fraction of the system state that is captured: excessive instrumentation perturbs the measured system, but limited instrumentation reduces measurement detail. Simply put, performance instrumentation manifests an *Instrumentation Uncertainty Principle* [Mal90]:

- Instrumentation perturbs the system state.
- Execution phenomena and instrumentation are coupled logically.
- Volume and accuracy are antithetical.

The terms "Heisenberg Uncertainty" and "probe effect" have been used to describe the error introduced in the performance measurement due to a monitor's intrusion on computer system behavior. The primary source of instrumentation perturbations is the execution of additional instructions. However,

ancillary perturbations can result from disabled compiler optimizations and additional operating system overhead. These perturbations manifest themselves in several ways: execution slowdown, changes in memory reference patterns, event reordering, and even register interlock stalls. Perturbation due to instrumentation has two effects on the events occurring during parallel execution: temporal effects and resource assignment effects. In addition to the slowdown caused by instrumentation overhead, temporal effects include possible event re-orderings as the measurement changes the likelihood of different partial order executions. Resource assignment effects occur because the instrumentation changes the dynamic resource demands. In instances where the computation dynamically adapts to resource availability, instrumentation can perturb resource allocation and utilization.

Performance measurements can differ significantly from actual execution (where measurements are disabled) unless the perturbation effects are taken into account by the performance environment during performance analysis. The goal of *performance perturbation analysis* is the recovery of actual runtime performance behavior from perturbed performance measurements. Formal models of performance perturbation are needed that permit quantitative evaluation of perturbations given instrumentation costs, measured event frequency, and desired instrumentation detail. Techniques based on timing and event models have been applied with positive results [Mal90]. Because actual performance behavior is inferred (approximated) by these models from the performance measurements, however, no absolute means for testing the accuracy of perturbation analysis is available. Rather, performance approximations were empirically validated with respect to two measures: total program execution time and selected even timings.

It is not uncommon that execution time is degraded many-fold when a program is measured. If total program execution time is accurately approximated after perturbation analysis is applied, the implication is that perturbation analysis errors are not accumulating. On the other hand, the reason detail performance measurements are made is to observe events of finer granularity. Perturbation analysis must also accurately resolve individual event timings. To determine the accuracy of trace events, one needs a standard of reference. No such standard exists, because the actual event trace is unknown. Instead, a sequence of event traces, each with successively smaller subsets of the detailed trace measurement, must be produced and the approximated event timings of correlated events compared. From the measurement uncertainty principle, as the number of trace events decreases, the presumed accuracy of the event timing approximations increases. If the approximated times of events correlated across the traces correspond, then it follows that the timing of other events in the detailed trace should also be accurate.

However, this validation approach is not wholely satisfying, because it lacks a theoretical basis. In general, concurrent execution involves data dependent behavior. The states of parallel programs inherently form a partial

order that must be followed during execution. If dependency control is spread across threads of execution, instrumentation can perturb the timing relationships of events and, thus, their actual execution ordering. If performance instrumentation is designed correctly, an un-instrumented parallel execution that satisfies Lamport's *sequential consistency* criterion[4] [Lam79] implies that the performance measurement will be *non-interfering* and *safe*. If the performance measurements involve only the detection and recording of event occurrence (i.e., tracing), the partial order relationships will be unaffected and the set of *feasible* executions[5] will remain unchanged. Beginning with a total ordering of measured events consistent with the *happened before* relation [Lam78] defined by the original partial order execution, time-based and event-based perturbation analysis can be applied to thread events that occurred either during independent execution to remove the instrumentation overhead or in dependent execution to enforce the semantics of operations that implement inter-thread synchronization. As long as the total ordering of dependent events present in the measured execution is maintained during this analysis, the final approximated execution will also be a feasible execution.

But is the final approximated execution a "likely" execution? That is, would the approximated execution ever actually occur, and with what expectancy would it occur? Any perturbation analysis approximation must be safe (i.e., must not violate partial ordering relationships) and, therefore, must be provided sufficient measurements that capture the operations that enforce ordering during execution. However, the accuracy of perturbation analysis depends not only on more precise synchronization measurements, but also on additional knowledge of actual (likely) execution behavior, which is unattainable from measurements alone. The set of *likely* executions is the subset of the *feasible executions* that are most probable. In many cases, the complete range of feasible executions will be restricted to a smaller set of likely executions due to the computational environment. If instrumentation is added, the set of likely executions can change. Computing the likelihood distribution of feasible executions is an extremely difficult problem, requiring an execution time model of concurrent operation. Thus, the inability to predict likely executions makes it difficult to bound the error of measurement-only perturbation analysis.

Simply put, performance measurement alone is insufficient to solve the perturbation analysis problem. If additional information were provided to the perturbation analysis process that describes certain behavioral properties and resource allocation and usage of the parallel computation (e.g., data dependency information, loop scheduling algorithms, processor allocation, memory usage), the perturbation analysis could use this information to make

[4] A parallel execution is sequentially consistent if the result is the same as if the operations were executed in some sequential order obtained by arbitrarily interleaving the thread execution streams.

[5] The set of program executions that could result from the partial order of program events is known as the *partially ordered set* of (feasible) executions.

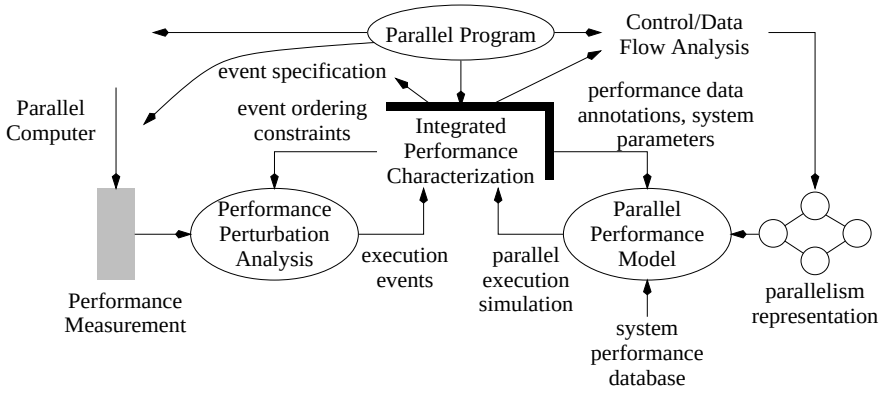

Fig. 7.1. Unifying Framework for Measurement-Based Experimental Performance Analysis

more accurate approximations by modeling the effects of nondeterministic execution in the presence of instrumentation.

This observation suggests a strong relationship between parallel performance paradigms which are used to define methodologies for performance measurement and diagnosis, and performance perturbation analysis.

- The accuracy of parallel performance models depend on the validity of the performance data used.
- Performance perturbation analysis depends on knowledge of context-dependent execution control and system performance information, which is provided in the parallel performance models, to resolve perturbation errors.

This relationship can be captured in a framework for measurement-based experimental performance analysis, unifying performance perturbation analysis and parallel performance modeling research; see Figure 7.1. The interesting parts of the framework concern the feedback paths:

- to event specification and program analysis, for changing the granularity of performance observation;
- to perturbation analysis, for preventing execution ordering violations; and
- to parallel performance modeling, for annotating the representational form of the parallel program with measured performance data and system parameters.

One can consider performance perturbation more generally as a change to "real", unperturbed performance of a parallel computation as a result of some change to the parallel execution environment. This change could be the result of performance measurement, as we have discussed here, or the result of performance analysis abstraction. Because we are trying to discover the "real" performance by an analysis process, whether it is based on measurement, simulation, or analytical modeling, there is always a question about the accuracy

of the performance approximation. That accuracy can be perturbed not only by instrumentation intrusion, but also by inaccurate or incorrect modeling assumptions. The important insight is that perturbation analysis can be more fully regarded as a general performance prediction problem. The goal is to estimate (predict) performance based on stored performance knowledge coupled with abstractions of parallel program and system behavior. Only by understanding the interplay of performance knowledge with parallel models (actual or abstract) can high confidence approximations be achieved. The natural tension between the complexity of measurement and modeling makes this an interesting challenge.

Our work on perturbation analysis [MR91, MRW92] demonstrates the effectiveness of perturbation models in approximating aggregate performance data from traces gathered using intrusive monitoring, even in cases of behavioral changes due to event ordering influences. We show that even in cases of very high intrusion, accurate analysis is possible. The effect of perturbation of an execution environment is considered in our work on performance extrapolation [SM95]. Here we use measurements of a multithreaded program running on a single processor to estimate the performance of the program on a target parallel machine, substituting performance models for architectural and system components (e.g., network and scheduling) that are being varied (i.e., perturbed).

The trace recovery research of Gannon et al. [GWA+94] combines perturbation analysis with software modeling. They develop a tool that generates timed Petri net (TPN) models of intrusively monitored software; reinstruments the software as needed; and then uses the TPN model to recover, from the corrupted trace, the approximate trace that would have been observed had monitoring-induced timing and event order changes not been present. The amount and type of trace information provided by this approach is often sufficient to resimulate a system accurately to some known point. This allows the user to not only determine when behavioral changes occur due to intrusion, but also determine the sensitivity of program behavior to intrusion (i.e.. program robustness). The correct trace can also be fed to a deterministic TPN simulator to visualize the process, which may allow the programmer to determine and alleviate bottlenecks that limit program performance.

8. Summary

The changing nature of the parallel computing platform extends the bounds of how these systems are programmed and used, further increasing computational and performance complexity. Tools must adapt to this change. Designing and building tools for parallel performance evaluation is one of the most challenging research areas in computer science. Not only are there fundamental issues associated with modeling and observing concurrent, parallel operations, but the self-referential and self-diagnostic notions of computer-based

tools trying to understand computational behavior are extraordinary. This chapter has presented a performance evaluation perspective on the general research area of parallel tools. The views presented on modeling, observability, diagnosis, and perturbation are applicable to the more general field as a whole. They are also useful as guideposts for understanding how tools should evolve to meet the requirements of next-generation systems.

References

[Amd67] Amdahl, D., Validity of the single-processor approach to achieving large-scale computer capabilities, *Proc. of the AFIPS Conference*, 1967, 483-485.

[Che93] Cheng, D., A survey of parallel programming languages and tools, Technical Report RND-93-005, NASA Ames Research Laboratory, 1993.

[Den80] Denning, P., What is experimental computer sciene, *Communications of the ACM* **23**, 1980, 543-544.

[DT98] Dongarra, J., Tourancheau, B., (eds.), *Workshop on Environments and Tools for Scientific Parallel Computing*, 1992, 1994, 1996, 1998.

[Eur] Eurotools working group, (See `http://www.irisa.fr/EuroTools/`).

[Fah95] Fahringer, T., Estimating and optimizing performance for parallel programs, *IEEE Computer* **28**, *(Special issue on Performance Evaluation Tools for Parallel and Distributed Computer Systems)*, 1995, 47-56.

[Fer78] Ferrari, D., *Computer Systems Performance Evaluation*, Prentice-Hall, Englewood Cliffs, 1978.

[GWA+94] Gannon, J., Williams, K., Andersland, M., Casavant, T., Lump, J., Trace recovery in multiprocessing systems: Architectural considerations, *Proc. of the 1994 International Conference on Parallel Processing*, 1994, 97-101.

[Gus88] Gustafson, J., Reevaluating amdahl's law, *Communications of the ACM* **31**, 1988, 532-533.

[HM95] Heath, M., Malony, A., Rover, D., Parallel performance visualization: From practice to theory, *IEEE Parallel and Distributed Technology* **3**, 1995, 44-60.

[HMR95] Heath, M., Malony, A., Rover, D., The visual display of parallel performance data, *IEEE Computer* **28**, *(Special issue on Performance Evaluation Tools for Parallel and Distributed Computer Systems)*, 1995.

[HMF95] Helm, B., Malony, A., Fickas, S., Capturing and automating performance diagnosis: the poirot approach, *Proc. of the International Parallel Processing Symposium*, 1995, 606-613.

[Hol94] Hollingsworth, J., *Finding Bottlenecks in Large-scale Parallel Programs*, PhD thesis, University of Wisconsin, 1994.

[HM98] Hollingsworth, J., Miller, B., Instrumentation and measurement, in Foster, I., Kesselman, C., (eds.), *The GRID: Blueprint for a New Computing Infrastructure*, Morgan Kaufman Publishers, San Francisco, 1998, 339-366.

[HML95] Hollingsworth, J., Miller, B., Lumpp, J., Techniques for performance measurement of parallel programs, in Casavant, T., Tvrdik, P., Plasil, F., (eds.), *Parallel Computers: Theory and Practice*, *IEEE Computer Society Press*, 1995.

[Int98] *International Conference on Modelling Techniques and Tools for Computer Performance Evaluation*, 1991-1998.

[Jou90] *Journal of Parallel and Distributed Computing* **9**, *(Special issue on Sofware Tools for Parallel Programming and Visualization)*, 1990.

[Jou93] *Journal of Parallel and Distributed Computing* **18**, *(Special issue on Tools and Methods for Visualization of Parallel Systems and Computations)*, 1993.

[Lam78] Lamport, L., Time, clocks, and the ordering of events in a distributed system, *Communications of the ACM* **21**, 1978, 558-565.

[Lam79] Lamport, L., How to make a multiprocessor computer that correctly executes multiprocess programs, *IEEE Transactions on Computers* **28**, 1979, 690-691.

[LMF90] LeBlanc, T., Mellor-Crummey, J., Fowler, R., Analyzing parallel program executions using multiple views, *Journal of Parallel and Distributed Computing* **9**, 1990, 203-217.

[Mal90] Malony, A., *Performance Observability*, PhD thesis, University of Illinois, Urbana-Champaign, 1990.

[MH99] Malony, A., Helm, R., A theory and architecture for automating performance diagnosis, *Fifth Generation Computing Systems, Special Issue on Performance Data-mining in Parallel and Distributed Computing*, 1999.

[MR91] Malony, A., Reed, D., Models for performance perturbation analysis, *Proc. of the Workshop on Parallel and Distributed Debugging*, 1991, 1-12.

[MRW92] Malony, A., Reed, D., Wijshoff, H., Performance measurement intrusion and perturbation analysis, *IEEE Transactions on Parallel and Distributed Computing* **3**, 1992, 433-450.

[MCC95] Miller, B., Callaghan, B., Cargille, J., Hollingsworth, J., Irvin, R., Karavanic, K., Kunchitkapadam, K., Newhall, T., The paradyn parallel performance measurement tools, *IEEE Computer* **28**, *(Special Issue on Performance Evaluation Tools for Parallel and Distributed Computer Systems)*, 1995, 37-46.

[Pan] Pancake, C., Parallel debugger bibliography, (See `http://www.cs.orst.edu/~pancake/papers/biblio.html/`).

[Par] The parallel tools consortium, (See `http://www.ptools.org/`).

[Pas] *Pasadena Workshop on System Software and Tools for High Performance Computing Environments*, 1992, 1995, (See `http://cesdis.gsfc.nasa.gov/PAS2/index.html/`).

[RC98] Rajamony, R., Cox, A., Parallel programming tools, Technical Report, Rice University, 1998,

[RAD98] Reed, D., Aydt, R., DeRose, L., Mendes, C., Ribler, R., Shaffer, E., Simitci, H., Vetter, J., Wells, D., Whitmore, S., Zhang, Y., Performance analysis of parallel systems: Approaches and open problems, *Proc. of the Joint Symposium on Parallel Processing (JSPP)*, 1998, 239-256.

[RAN+93] Reed, D., Aydt, R., Noe, R., Roth, P., Shields, K., Schwartz, B., Tavera, L., Scalable performance analysis: The pablo performance analysis environment, in Skjellum, A., (ed.), *Proc. of the Scalable Parallel Libraries Conference*, 1993, 104-113.

[RPF99] Reed, D., Padua, D., Foster, I., Gannon, D., Miller, B., Delphi: An integrated, language-directed performance prediction, measurement, and analysis environment, *Frontiers '99: The 9th Symposium on the Frontiers of Massively Parallel Computation*, 1999.

[RB98] Reed, D., Ribler, R., Performance analysis and visualization, in Foster, I., Kesselman, C., (eds.), *The GRID: Blueprint for a New Computing Infrastructure*, Morgan Kaufman Publishers, San Francisco, 1998, 367-394.

[RS99] Rover, D., Shanblatt, M., (eds.), *International Journal of Parallel and Distributed Systems and Networks*, 1999.

[RWH98] Rover, D., Waheed, A., Hollingsworth, J., Modeling and evaluating design alternatives for an on-line instrumentation system: A case study, *IEEE Transactions on Software Engineering* **24**, 1998, 451-470.

[RWM+98] Rover, D., Waheed, A., Mutka, M., Bakic, A., Software tools for complex distributed systems: Toward integrated tool environments, *IEEE Concurrency* **6**, *(Special Issue on Engineering of Complex Distributed Computing Systems)*, 1998, 40-54.

[Sch65] Scherr, A., *An Analysis of Time Shared Computer Systems*, PhD thesis, Massachusetts Institute of Technology, Cambridge, 1965.

[SM95] Shanmugam, K., Malony, A., Performance extrapolation of parallel programs, *Proc. of the International Conference on Parallel Processing*, 1995, 117-120.

[SHB+94] Simmons, M., Hayes, A., Brown, J., Reed, D., (eds). *Debugging and Performance Tuning for Parallel Computing Systems: Toward a Unified Environment*, 1994.

[SHB+96] Simmons, M., Hayes, A., Brown, J., Reed, D., (eds.), *Debugging and Performance Tuning for Parallel Computing Systems*, IEEE Computer Society Press, 1996.

[SKB89] Simmons, M., Koskela, R., Bucher, I., (eds.), *Instrumentation for Future Parallel Computing Systems*, ACM Press, 1989.

[SKB90] Simmons, M., Koskela, R., Bucher, I., (eds.), *Parallel Computer Systems: Performance Instrumentation and Visualization*, ACM Press, 1990.

[Sym98] *Symposium on Parallel and Distributed Tools*, ACM Press, ACM SIGMETRICS, 1996, 1998.

[Wor93] *Workshop on Parallel and Distributed Debugging*, ACM SIGPLAN/SIGOPS and Office of Naval Research, 1988, 1991, 1993.

[YP95] Yan, J., Pancake, C., Simmons, M., (eds.), *IEEE Computer* **28**, *(Special issue on Performance Evaluation Tools for Parallel and Distributed Computer Systems)*, 1995.

[YPS95] Yan, J., Pancake, C., Simmons, M., (eds.), *IEEE Parallel and Distributed Technology, Special issue on Performance Evaluation Tools for Parallel and Distributed Computer Systems*, 1995.