# Performance Issues in Parallel Processing Systems

Luiz A. DeRose, Mario Pantano, Daniel A. Reed, and Jeffrey S. Vetter*

{derose,pantano,reed,jsv}@cs.uiuc.edu
WWW: http://www-pablo.cs.uiuc.edu/

Department of Computer Science
University of Illinois
Urbana, Illinois 61801 USA

**Abstract.** Computer system performance measurement and code optimization have a long and illustrious history that can be traced to the very first computers. With high-performance parallel systems, the problems become more complex and the need to achieve high fractions of peak performance more critical. In this chapter, we describe key points in the historical development of performance tools for parallel systems, as well as sketching current research challenges.

## 1 Introduction

Simply put, the goal of performance analysis is to provide the data and insights required to optimize the execution behavior of application or system components. Using such data and insights, application and system developers can choose to optimize software and execution environments along many axes, including execution time, memory requirements, and resource use. Given the diversity of performance optimization goals and the wide range of possible problems, a complete performance analysis toolkit necessarily includes a broad range of techniques. These range from mechanisms for simple code timings to multi-level hardware/software measurement and correlation across networks, system software, runtime libraries, compile-time code transformations, and adaptive execution.

High-performance parallel systems exacerbate already difficult performance-optimization problems. Although performance is the *raison detre* for parallel computing, subtle interactions among parallel software components can easily lead to unexpected bottlenecks. More perniciously, Amdahl's law shows that serialization of even a small fraction of a code can have dramatic effects on overall

* This work was supported in part by the Defense Advanced Research Projects Agency under DARPA contracts DABT63-94-C0049 (SIO Initiative), F30602-96-C-0161, DABT63-96-C-0027, and N66001-97-C-8532 by the National Science Foundation under grants NSF CDA 94-01124 and ASC 97-20202, and by the Department of Energy under contracts DOE B-341494, W-7405-ENG-48, and 1-B-333164.

performance. Consequently, the history of parallel computing is replete with stories of application and system designs that failed to meet expected performance goals.

For example, preliminary experiences with terascale systems (i.e., thousand processor systems capable of more than $10^{12}$ operations per second), suggest that the fraction of achieved performance can actually decline as one adds processors. These systems are so complex that the resource demands of each large-scale, multidisciplinary application can expose portions of the behavioral-execution space never before tested or explored. It is these unanticipated patterns of hardware, system software, and application interaction that cause poor performance.

To understand and master the complexity of large-scale parallel systems, a plethora of performance-analysis tools and techniques have been proposed and developed. Despite their number and diversity, most tools and techniques lie in one of three broad categories: analytic modeling, simulation, or empirical performance measurement.

The attraction of analytic models (e.g., Petri nets, queueing networks, or Markov models) lies in their ready parameterization and rapid solution, allowing one to quickly predict performance over a broad range. Their major liability is that tractability requirements often dictate restrictive assumptions (e.g., service times with negative exponential distributions) that are violated in practice, lessening accuracy.

In contrast, the flexibility of discrete event simulation enables construction of detailed, highly accurate models of system components. Furthermore these models can be evaluated using realistic workloads. This flexibility is not without price — simulations must be carefully validated against empirical data, lest generalizations from parametric simulation studies be based on faulty models. Moreover, the time and computational cost demanded by detailed simulations, particularly for large-scale systems, often precludes their use.

Given these constraints, experimental performance analysis remains the method of choice for tuning large-scale, parallel systems. In this chapter, we begin by discussing the typical cycle of performance analysis, emphasizing measurement, data analysis and visualization, and system tuning.

The remainder of this chapter is structured as follows. In §3, we begin with a detailed description of the range of basic instrumentation and measurement techniques for parallel systems, followed in §4 by an assessment of more sophisticated approaches to reducing and analyzing performance data sets. In §5, we discuss a suite of performance visualization and presentation techniques for parallel systems. This is followed in §6 by an assessment of integrated methodologies for instrumentation, analysis, and visualization, including analysis of the costs for each. Finally, we conclude in §7 with a few observations on the performance analysis process.

## 2 The Performance Analysis Cycle

The most common approach to experimental performance analysis relies on a five phase cycle of instrumentation, measurement, analysis, presentation, and optimization. At present, the phases of this cycle span code preparation, execution, and post-mortem analysis and optimization.

*Instrumentation.* Performance instrumentation can be either implicit or explicit. Implicit instrumentation relies on monitoring daemons to capture performance data; most profiling tools [20] rely on some variation of this approach. In contrast, explicit instrumentation modifies software by directly inserting measurement code. For application or system software, this explicit instrumentation can be inserted automatically (e.g., via object code patching [32] or via instrumenting parsers or compilers [1]) or manually by including calls to instrumentation library routines. For hardware instrumentation, one needs either specialized probes or software accessible counters or registers (e.g., the hardware counters now embedded on most commodity microprocessors [48]).

*Measurement.* To capture performance data, one then observes the instrumented system during execution, while the instrumentation software and hardware records pertinent performance data. For application software instrumentation, this typically includes procedure, loop, and basic block execution counts and times, communication (i.e., message passing or shared memory references) and synchronization costs, I/O patterns and system calls. System software measures often include network traffic, virtual memory overhead, context switches and task scheduling patterns, and disk traffic. Finally, common hardware measures include instruction mixes and stalls, as well as cache misses.

*Data Reduction and Correlation.* Following measurement, the resulting performance data must be processed to yield compact and easily interpretable results. If event tracing [36] was used to capture the pattern of software component interactions (e.g., message passing, I/O, or procedure calls), statistical summarization can transform large, unwieldy masses of raw data into succinct descriptions. Alternatively, it may be necessary to relate measurements of executable code to source code by inverting key compile-time code transformations (e.g., when high-level data parallel code was translated to message passing code during compilation [1]).

*Analysis and Presentation.* The goal of instrumentation and measurement is insight. Hence, one must present data in meaningful and intuitive ways. For simple profiles, standard tabular mechanisms suffice. For more complex measurements and subtle performance problems where one must trace problems from proximate to root causes, more powerful data analysis and presentation techniques are required. Static color graphics, workstation animations, and immersive virtual environments have all been used to visualize performance data [19].

*System Optimization.* Given an understanding of the causes of performance problems, the final step is remediation. This requires either modifying the application or system software, reconfiguring or adding hardware, or changing application or system parameters. Realizing these changes can be either trivial or extraordinarily difficult. Moreover, one must weigh the intellectual and labor costs of optimization against the perceived rewards. Rarely is *optimizing* execution time or system resource use the true goal. Instead, one seeks to reduce execution time or resource use subject to some constraints.

### 2.1 Measurement Constraints

Within this five step cycle, users and performance analysts are free to choose instrumentation, measurement, analysis, and presentation techniques that balance

- *resolution*, the level of detail (e.g., profiling or tracing) and granularity (e.g., loop nest or procedure call) of measurements,
- *perturbation*, the change in observed behavior from that when no instrumentation is present,
- *flexibility*, the ability to apply performance tools in multiple contexts and applications in configurable ways, and
- *accuracy*, capturing the true behavior of the observed system, subject to errors induced by perturbation.

Because many of these attributes are inimical (.e.g., high resolution is rarely possible without substantial perturbation), no single set of options is appropriate for all performance optimization scenarios. For example, some techniques minimize measurement perturbation by dynamically enabling instrumentation only when needed [32], while other techniques attempt to estimate and remove perturbation from measurements during analysis [29]. Both of these perturbation choices affect accuracy and resolution.

### 2.2 *In Vivo* Optimization

The classic cycle of instrumentation, measurement, reduction, analysis, and optimization described above presumes that observed behaviors are repeatable, namely that subsequent executions based on post-mortem optimization will yield performance improvement because similar behavior will accrue. For the common case of application program tuning on an otherwise quiescent sequential or parallel system, this assumption holds.

However, emerging applications are increasingly dynamic and adaptive and execute on heterogeneous collections of distributed, shared resources. In such an environment, the execution context is rarely repeatable, making post-mortem tuning problematic. Instead, the conventional *in vitro* cycle of instrumentation, measurement, assessment, and tuning must be replaced with *in vivo* optimization techniques that instrument, measure, and tune software during its execution. We will return to this topic in §4.

## 3 Instrumentation and Measurement Techniques

One can instrument systems at many levels, though the four most common are hardware, system software, runtime systems, and application code. Given the complexity of parallel systems and their sensitivity to small changes in component interactions, correlating performance data across two or more of these levels is normally required to understand the actual causes of performance problems.

Software instrumentation, for example, can capture the interaction of compiler-synthesized code with runtime libraries and system software. However, to understand the effects of superscalar instruction scheduling or cache hierarchies in a distributed shared memory (DSM) system, one must capture and correlate both software and hardware performance metrics.

Fortunately, new microprocessors commonly provide a set of performance registers for low-overhead access to hardware performance data. For example, the MIPS R10000 [33] includes registers that count hardware events, including cycles, instruction and data cache misses, floating point instructions, and branch mispredictions. Similar counters exist on other architectures.

These different levels of instrumentation can be realized via several methods. First, instrumentation can be inserted either statically, during code development or compilation, or dynamically, during execution. Second, instrumentation can be inserted manually or automatically. Finally, the nature of the data gathered (e.g., hardware, application, or system software) depends on the measurement technique.

### 3.1 Static and Dynamic Instrumentation

The most common instrumentation approach augments source code with calls to specific instrumentation libraries. During execution, these library routines collect behavioral data. Examples of static instrumentation systems include the widely used UNIX `prof` and `gprof` [20], the Automated Instrumentation and Monitoring System (AIMS) [47], the Executable Editing Library (EEL) [27], and the Pablo performance environment toolkit [38].

The primary drawback of static instrumentation is its lack of flexibility — it cannot be modified during program execution. Thus, changing the types of data collected typically requires application re-instrumentation, recompilation, and a new execution.

In contrast, dynamic instrumentation allows users to interactively change instrumentation points, focusing measurements on code regions where performance problems have been detected. Paradyn [32] is the exemplar of such dynamic instrumentation systems. The drawback of dynamic instrumentation is that the application must execute long enough for users or performance runtime systems to find performance problems by incrementally modifying the instrumentation.

### 3.2 Interactive and Automatic Insertion

As we noted earlier, performance analysis tools can insert instrumentation either automatically or interactively. The strength of interactive instrumentation is its

flexibility; users can insert instrumentation library calls at arbitrary code points. However, naive or excessive instrumentation can excessively perturb execution behavior (e.g., by inhibiting compiler optimizations or by instrumenting frequently executed inner loops). Conversely, automatic instrumentation relies on the compiler or the runtime system to insert measurement code. This approach reduces the risk of perturbation, but sacrifices user control over instrumentation points.

### 3.3 Instrumentation Methods

Regardless of the instrumentation level, there are four main approaches to performance data capture: sampling, counting, timing, and tracing. Each reflects a different balance among data volume, potential instrumentation perturbation, accuracy, and implementation complexity. In addition, each can be implemented in a variety of ways, ranging from extrinsic (e.g., an external hardware monitor that counts data cache misses) to intrinsic (e.g., inserted code that computes a histogram of procedure activation lifetimes).

**Program Counter Sampling.** By far the most common instrumentation method is program counter sampling. The `prof` and `gprof` utilities [20] periodically sample the program counter and compute a histogram of program counter locations. Profiling depends on an external sampling task, leading to coarse granularity and requiring the total program execution time to be long enough to accumulate a statistically meaningful sample set. Moreover, as discussed in §6.1, profilers often assume a simple mapping from object code to the original source code, which is rarely true on parallel systems with aggressive restructuring compilers.

**Event Counting.** Event counting eliminates some of sampling's limitations, albeit with possibly increased perturbation for frequent events. Because counting is not a statistical measure, the observed frequencies are accurate even for short executions.

Some performance data capture toolkits combine sampling and counting. For instance, the MIPS R10000 microprocessor includes two hardware performance counters, each able to track one of 16 events per cycle. To capture more than one event per counter, the operating system maintains a set of 32 virtual counters, multiplexing (sampling) the physical counters across these. This multiplexing mechanism sacrifices accuracy but increases coverage, allowing a single program execution to acquire data for all the desired counters.

**Interval Timing.** Interval timing combines counting with elapsed-time measurements. Rather than simply sampling the program counter, interval timing surrounds code fragments with calls to timing routines, accumulating cumulative data and counts of code fragment execution time. This requires software access to high-resolution, low overhead clock.

With increasing processor speeds, the standard UNIX timing facility, often with 50 or 60 Hz resolution, is unacceptably coarse. Only with microsecond resolution (or better) clocks and low overhead access routines can one accurately measure small code fragments.

**Event Tracing.** Event tracing is the most general of instrumentation techniques — from an event trace, one can compute counts or times; the converse is not true. The primary disadvantages of tracing are the large data volume and the potential instrumentation perturbation. For example, instrumenting procedures to record entry and exit can easily generate 16 KB/second on a single processor if the mean procedure activation lifetime is 500 microseconds and the data associated with each event includes only a four-byte event identifier and a four-byte time stamp. On a parallel system with hundreds or thousands of processors, the data volume can be many megabytes/second.

Runtime summarization can reduce event tracing's data volume by trading computation overhead for data volume. Rather than generating trace files, the instrumentation system computes metric values during execution, generating periodic statistical summaries. In §4, we discuss summarization techniques in more detail.

In general, perturbation, resolution, and accuracy are dependent on the instrumentation method; no single instrumentation approach is appropriate for all cases. The choice of a particular method is dictated by the desired information and the constraints of the available instrumentation — some measurements may not be possible in a particular environment.

## 4 Performance Data Analysis

The main goal of performance analysis is identifying bottlenecks or sources of poor performance. However, with increased software and parallel system complexity, identifying these bottlenecks becomes commensurately more difficult.

Performance bottlenecks may lie in instruction schedulers, memory-hierarchy management, wide-area or local communication protocols, scheduling algorithms, or application load balance. Consequently, the metrics that highlight performance problems are highly dependent on the execution context; determining appropriate metrics and assessing the resulting data is the goal of performance analysis.

### 4.1 Data Management Techniques

The simplest form of analysis filters data that do not meet user-specified, criteria, retaining and computing metrics from only the pertinent data (e.g., standard statistical metrics such as the mean, minimum, maximum, percentiles, variance, and coefficient of variation). Often, these metrics suffice — the causes of performance problems are obvious once even simple measurements are examined.

For more complex performance problems, detailed analysis based on event traces may be required. However, the potentially large volume of event trace data, the large number of possible performance metrics, and consequent behavioral perturbations make this approach impractical for large, long-running applications. To retain the advantages of event tracing while minimizing data volume and intrusion, analysts must reduce both the number of metrics that are needed to identify bottlenecks (metric dimensionality) and the number of locations where data must be captured (metric plurality).

As a basis for discussing event trace reduction, consider a set of $n$ dynamic performance metrics, each measured on a set of $P$ parallel tasks. Conceptually, an event trace defines a set of $n$ dynamic performance metrics, $m_i(t)$, on each of $P$ tasks

$$(m_1(t), m_2(t), ..., m_n(t))_p \qquad p \in [1..P]$$

that describe parallel system characteristics as a function of time $t$. Following [35], if $R_i$ denotes the range of metric $m_i(t)$, we call the Cartesian product

$$M = R_1 \times R_2 \times ... \times R_n$$

a performance metric space. Thus, the ordered $n$-tuples

$$(m_1(t) \in R_1; m_2(t) \in R_2; ...; m_n(t) \in R_n)$$

are points in $M(t)$, and the event trace defines the temporal evolution of these $P$ points in an $n$ dimensional space.

There are several possible approaches to reducing metric dimensionality and plurality. Below, we describe two — statistical clustering and projection pursuit. Statistical clustering reduces the number of measurement points (i.e., $P$) by eliding data from groups of processors with similar behavior. Conversely, projection pursuit reduces the dimensionality of the metric space (i.e., $n$) by identifying "important" metrics.

**Statistical Clustering.** Programs for parallel systems generally use the Single Process Multiple Data (SPMD) model (e.g., using MPI), the data-parallel or object-parallel models (e.g., using HPF or parallel C++ ), or simple functional decompositions. In the first three models, the same code executes on all processors, with behavior differentiated by data-dependent control flow. In the functional decompositions, similar code executes on several processor groups.

Regardless of the programming model, tasks executing the same code with similar data form behavioral equivalence classes with comparable trajectories in the metric space. Statistical clustering groups processors (or tasks) in equivalence classes whose performance metrics trace similar trajectories in the metric space and chooses one member of each grouping as a representative of the others.

As an example, consider an SPMD code that relies on a master task to read initialization data and allocate work to a set of $N$ worker tasks. If all workers behave similarly, clustering identifies two clusters, the first with cardinality one,

the master, and a second with cardinality $N$, the workers. Only two tasks, the master and any one of the remaining tasks, fully represent the range of application behavior. By periodically computing clusters using real-time data, an event tracing system need record data only for the representatives of each cluster, dramatically reducing the total data volume [35].

**Projection Pursuit.** Statistical clustering reduces the number of processors (or tasks) $P$ from which event data must be recorded, but it does not reduce the number of metrics $n$, the dimensionality of the metric space. Projection pursuit [25], a statistical variant of principal component analysis, identifies a subset of the metrics that captures most of the statistical variation.

Conceptually, at any time $t$, many of the $n$ metrics are highly correlated. Multidimensional projection pursuit selects the least correlated $k$ metrics from the multidimensional metric space; typically, $k$ is two or three (i.e., two or three-dimensional projection pursuit). Using projection pursuit, real-time event tracing need record only that subset of the metrics deemed "important" by the projection.

As Fig. 1 suggests, combining statistical clustering and projection pursuit can potentially reduce both the number of processors and the number of metrics that an event tracing system must record during program execution [41]. However, the possible data reduction from clustering and projection pursuit must be balanced against the additional computational overhead and decreased measurement accuracy.

Between successive application of clustering and projection pursuit, only data from the representative processors and metrics is recorded. If the algorithms are applied too infrequently, application phase changes may be missed (i.e., the recorded data may no longer be representative). Conversely, if applied too frequently, the computational overhead may excessively perturb application behavior. Hence, developing adaptive windowing algorithms for triggering clustering and projection pursuit remains a key research problem.

### 4.2 Performance Prediction

In standard practice, performance analysis is remedial, identifying problems created earlier, during design of hardware, application or system software. However, if it were possible to estimate application or system performance *during* rather than after design, performance analysis tools could lessen the high intellectual cost of developing and optimizing parallel applications. This would allow developers to understand the interactions of architectural parameters, problem size, application structure, and number of processors, identifying and eliminating potential bottlenecks during design.

Because performance bottlenecks may shift as these values change, a broad characterization of the parameter space would both enable creation of applications with portable performance and place system design on a solid performance

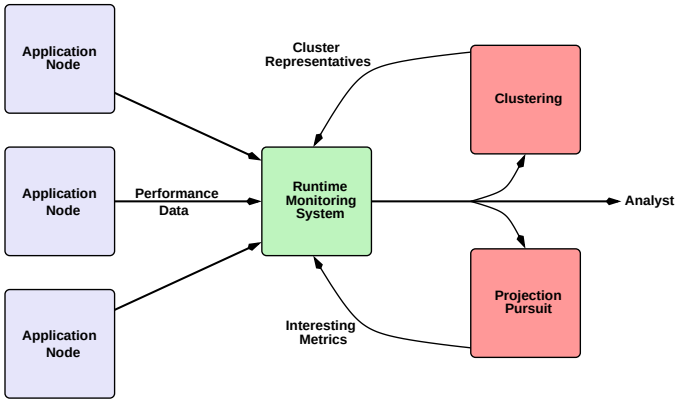


**Fig. 1.** Integrated Clustering and Projection Pursuit

engineering foundation. These characterizations need not be quantitatively exact, only qualitatively correct. Thus, large performance prediction errors are acceptable if the predictions accurately identify bottlenecks and track their movement with changing assumptions and parameters.

Several groups have developed tools and techniques to support performance prediction, including estimating the effectiveness of data partitioning strategies [3] and the execution time of parallel applications [22]. Most of these tools estimate only total execution time, rather than identifying bottlenecks by estimating the behavior of individual code fragments; $P^3T$ [16] is one notable exception. Moreover, they assume regular program structure with repeatable behavior.

To truly understand the interplay of complex program structure and observed performance, one must tightly integrate compile-time symbolic analysis [17] and performance measurement to analyze both aggregate application behavior and individual code fragments Using a symbolic manipulator, a compiler could create cost models for multiple code variants, evaluate the resulting expressions for specific numbers of processors and problem sizes, and synthesize the the code variants with the highest predicted performance. By augmenting and validating these symbolic models with dynamic performance data from program executions, the compiler could incrementally refine the quality of its predictions and code generation choices.

In this model, compilation and performance analysis are a closed loop, with both symbolic performance predictions and quantitative measurements guiding the compilation process [30]. However, for symbolic control flow analysis to be broadly effective, it must be extended to include both linear and non-linear expressions, more powerful simplification mechanisms, and relations between input data and output information.

### 4.3 Scalability Analysis

Assessing the scalability of algorithms, applications, and architectures is a specific aspect of performance prediction. Like symbolic analysis, most automated scalability techniques require deep compiler/performance tool integration.

As a feasibility test of performance scalability analysis, Mendes extended the Rice Fortran D95 compiler to predict the scalability of regular data parallel codes [31, 1]. The extended compiler translated data parallel code and generated a symbolic cost model for the execution time of each code fragment in the original data parallel source code. The scalability of these fragments could then be evaluated as a function of program size and number of processors.

In a similar effort, Sun *et al* extended the Vienna Fortran Compilation System (VFCS) [4] for scalability prediction [45]. The integrated system combines performance data from measurements on a small number of processors with a cost model of the application. The scalability is then automatically predicted as the number of processors varies in proportion to the problem size.

These experiments have shown that the automatic scalability prediction is feasible for small, regular codes. However, general techniques are needed that can predict performance scaling for large, irregular applications.

## 5 Performance Visualization

Because it allows users to grasp essential features of large, complex data sets, visualization is now a standard part of the scientific process. These same qualities also make visualization highly attractive for performance data analysis [24, 26, 44, 39], especially on parallel systems where data collections grow very large.

Scientific and performance data visualization share many features and design goals, though they also differ in important, though subtle ways.

- *Flexibility.* Given the diversity of performance data, visualization systems must allow users to bind data to those graphical representations the users find useful and intuitive (e.g., it should be possible to represent the same data multiple ways and different data using the same representations). Further, when a binding is not obvious, visualizations should allow users to explore a variety of bindings and data perspectives [40].
- *Multiple levels of detail.* Because performance problems can arise at every hardware and software level, visualizations must enable users to focus on interesting yet complex behavior while eliding irrelevant or tedious details. Consider a distributed computation that spans several geographically distributed sites. A performance analyst might wish to view the wide-area communication traffic and aggregate performance at each site, examine the task scheduling on one node, or, explore the data locality of a code fragment. Visualization techniques should encourage users to freely navigate among the many levels of performance data associated with such a distributed computation.

- *Data management.* As we have discussed, detailed event tracing of large parallel or distributed systems can quickly produce large volumes of data. Unlike traditional scientific data, performance data is often of high dimension, and very sparse — it contains multiple, time varying metrics captured on each of hundreds or thousands of processors. Performance data management systems must filter, extract, and correlate data subsets in response to user queries. Clustering, statistical summarization, projection pursuit, binning, surface contouring, and volume visualization [43] all reduce the data management burden by extracting important features from the data set while discarding repetitive or uninteresting data.
- *User interfaces and interaction.* Visualizations should provide natural and intuitive user interfaces and methods for users to manipulate large data collections. Although static visualizations of data collections are useful and common, comprehension generally improves when users can interactively explore their data [42].
- *Scalability.* Given the volume of performance data generated by parallel systems, visualizations must scale to large numbers of processors and support analysis of codes that run hours or days. For very large executions, this mandates real-time measurement and display.

Attempts to satisfy these conflicting goals have resulted in large numbers of performance analysis and visualization systems for high-end computers. Notable examples include the seminal Seecube toolkit [14], ParaGraph [23], Medea [9], and Pablo [38]. Despite the diversity of tools, their displays can be broadly classified along five axes.

- *Statistical visualizations.* Because many performance metrics are statistical measures, a large class of performance displays are variants of common statistical data visualizations [12] (e.g., scatterplots or histograms). The generality of these displays is both a strength and a weakness. They can represent virtually any performance metric. However, they do not easily scale to large numbers of metrics nor do these simple animations readily convey evolutionary behavior.
- *Specialized performance visualizations.* In contrast to generic statistical data displays, several research groups have developed domain-specific performance displays that directly represent some important feature of parallel hardware or software. Among the best known of these are procedure call graph displays [37], network topologies [23], and geographic traffic displays [19]. Many of these domain-specific displays are tightly coupled to specific programming models (e.g., message passing with PVM or MPI). For example, ParaGraph [23] includes a wide variety of hypercube topology projections, each animated with message traffic patterns.
- *Animations.* Because the majority of performance data consists of time series, animation is often used to represent evolutionary behavior. For large data volumes and long-running computations, animation is often the only practical method to create scalable displays — displays lack sufficient resolution to display entire execution lifetimes without abstracting detail.

- *Source code browsers.* As noted in §6, with increasingly sophisticated compilers, the behavior of compiler-synthesized code may differ dramatically from that written by the user (e.g., by transforming data parallel code into message passing code for a distributed memory parallel system). For users to optimize their code, performance measurement and display systems must relate dynamic performance data to the original source code. Not only does this require deep integration of compilers and performance tools, it encourages creation of source code browsers that display both source code and performance data. Source code browsers like SvPablo [15] display hardware metrics (e.g., cache behavior and instruction mixes) beside lines of source code, allowing users to explore performance within the context of their source code.
- *Immersive environments.* With high-end systems emerging with thousands of processors, performance presentation systems must either elide increasing detail to represent aggregate performance on workstation displays or adopt new, high resolution, high modality systems that can display more data. Immersive virtual environments not only have multi-megapixel resolution, they exploit sound and haptics to create immersive experiences — one is surrounded by dynamic performance data. As an example, Virtue [37] supports collaborative, immersive performance analysis and real-time display. Real-time data streams are mapped to a hierarchy of displays, ranging from geographic representations of wide-area traffic, through "time tunnel" representations of parallel system behavior, to dynamic call graphs of execution on a single tasks. By touching graph components, users can expand or contract subgraphs and change the behavior of executing software.

With a brief description of performance instrumentation, analysis, and display techniques, we turn now to integration of the components.

## 6 Deeply Integrated Performance Environments

For performance evaluation tools to be truly effective, they must integrate measurement, data analysis, visualization, and performance remediation. Ideally, they should also scale to thousands of processors, support analysis of distributed computations, and be portable across a wide variety of parallel systems. As we noted earlier, scalability means that the tools must not only capture and analyze data from large numbers of processors or tasks but also present the data in ways that are intuitive and instructive.

Early performance visualization systems focused on single architectural and programming models (e.g., message passing) and represented the states of individual processors (e.g., by a colored square for each processor) and communication links (e.g., by communication network animations). In consequence, workstation screen real estate limited these displays to a few hundred processors.

Complex applications now involve code written in a variety of languages and programming models, all executing on multiple parallel architectures. Without

performance tool portability and flexibility, users must learn a new tool each time the programming model or parallel architecture changes.

Today, the *de facto* standard for performance instrumentation and analysis remains application profiling with post-mortem analysis of performance bottlenecks. This model was adequate for parallel codes written in sequential languages for homogeneous parallel systems and for compiler-generated object code that directly reflects source code control. However, sophisticated parallelizing compilers, high-level languages like HPF, and object-parallel models like parallel C++ mean that an application software developer's mental model of a program and the actual code that executes on a particular parallel system can be quite different. Concurrently, execution environments have expanded to include heterogeneous collections of parallel systems with time-varying resource availability.

New performance tools must guide optimization at the semantic level where code is developed. Thus, they must integrate dynamic performance data with compile-time data that describes the mapping from the high-level source to the low-level, explicitly parallel code, minimizing total performance data volume while still enabling users to interactively drill down to identify performance problems on remote systems. Moreover, they must accommodate execution heterogeneity and non-reproducible behavior, replacing *post-mortem* analysis with real-time analysis and optimization. Below, we outline the challenges inherent in integrating compilers and performance tools.

### 6.1 Integrated Compiler Support

To support source-level performance analysis of transformed programs, compilers must share data on the mapping from the low-level, executable code to the original source [1]. Conversely, a restructuring compiler should exploit dynamic performance data code generation during subsequent recompilation.

**Compiler Data Sharing.** Several academic and vendor groups have developed integrated compilation and performance analysis systems. For example, we integrated Pablo with the Rice Fortran D restructuring compiler to create an integrated system that could relate the performance of measured object code to regular data parallel source [1, 2]. During compilation, the Fortran D compiler emitted both instrumented code and the sequence of source code transformations applied, tying each synthesized code fragment to a portion of the original data parallel source code. Later, during program execution, the instrumentation system captured data from the compiler-synthesized code. Finally, a post-processing phase exploited the program transformation data to map performance metrics to the original data parallel source code. These metrics included both execution times and array reference data locality.

In a related effort, Calzarossa *et al* [10] integrated Medea [9] with the VFC compiler's instrumentation system [6]. This integration provides a comprehensive view of the performance of irregular applications written in HPF+ [5]. For independent loops, each individual phase (work distribution, inspector, gather,

executor, and scatter) is automatically instrumented by the compiler during the code transformation. Medea analyzes the collected performance data and presents a detailed assessment of the performance in each phase.

Finally, Malony *et al* [34] have developed complementary approaches for parallel, object-oriented pC++ [8], capturing data on object method invocations. Building on the Sage's [7] support for pC++, TAU includes routine and data access profile and parallel execution extrapolation displays.

Although promising, all of these are research efforts. Robust integration of performance measurement and compilers will require new committments from vendors to expose software interfaces and exchange performance data.

**Performance-Directed Code Optimization.** The compiler-performance tool partnership can aid compilers as well as performance tools by providing compilers with data not available from code analysis (e.g., likely loop bounds or branch directions). For example, trace scheduling for very long instruction word (VLIW) processors [18, 13] can use profile data to optimize execution of the most probable code execution paths. Commercially, Silicon Graphics compilers use *prof* or *gprof* data to order procedure object code, based on procedure invocation frequencies, to reduce page faults and instruction cache misses.

More recently, Hwu *et al* [11] have developed even more aggressive profile-based optimization techniques for superscalar and VLIW code generation. These approaches are an integral part of the code generator for the new HP/Intel Merced processor.

For data parallel and object parallel codes, it is also possible to exploit data on interprocessor communication frequencies and method invocations to choose or optimize data distributions [3]. However, this and other performance-directed optimizations require concurrent, coordinated development of compilers and performance tools; market pressures often make this impractical. Simply put, despite the great promise of performance-directed code generation, much work remains to balance ease of use and possible performance gains.

### 6.2 Dynamic Resource Management

Although post-mortem performance analysis remains the method of choice, it is ill-suited to emerging multidisciplinary applications with time-varying resource demands that execute on heterogeneous collections of geographically distributed computing resources. Not only may the execution context not be repeatable *across* program executions, resource availability may change *during* execution. In such chaotic environments, only real-time measurement and dynamic optimization can adapt to changing application resource demands and system resource availability.

Dynamic optimization can be either interactive, based on real-time visualization and human-in-the-loop adaptation, or software-mediated by an intelligent decision support system. In both cases, the performance analyst must formulate general-purpose, performance-optimization rules that can be applied by the adaptive infrastructure.

Several groups have built interactive or software-mediated adaptive control systems [46]. Notable examples include Leblanc's [28] adaptive real-time system for robotic control, Schwan *et al*'s Falcon adaptive control toolkit [21], and our own Autopilot distributed control system [41].

Autopilot, built atop the Globus wide-area communication system [19], includes a set of configurable performance sensors and resource policy actuators. These sensors and actuators are managed by a fuzzy logic control system whose rules are triggered by sensor data. Rule outputs activate remote policy actuators to adjust application or system behavior for maximal performance. For interactive visualization and control, the fuzzy logic rule system can be replaced with our Virtue immersive visualization system [37].

## 7 Conclusions

As the space of high-performance computing evolves to include ever more complex, distributed collections of parallel systems, analyzing and tuning application, system software, and architecture designs becomes increasingly difficult. To understand and master the complexity of these systems, a plethora of performance analysis tools and techniques have been proposed and developed. Although great progress has been made, many open problems remain, including accurate performance prediction for large-scale applications, real-time adaptive control for wide-area computations, and visualizations that highlight important performance problems.

## References


1. Adve, V., Mellor-Crummey, J., Wang, J.-C., and Reed, D. Integrating Compilation and Performance Analysis for Data-Parallel Programs. In *Proceedings of Supercomputing'95* (November 1995).
2. Adve, V. S., Mellor-Crummey, J., Anderson, M., Kennedy, K., Wang, J., and Reed, D. A. Integrating Compilation and Performance Analysis for Data-Parallel Programs. In *Proceedings of the Workshop on Debugging and Performance Tuning for Parallel Computing Systems*, M. L. Simmons, A. H. Hayes, D. A. Reed, and J. Brown, Eds. IEEE Computer Society Press, 1994.
3. Balasundaram, V., Fox, G., Kennedy, K., and Kremer, U. A Static Performance Estimator to Guide Data Partitioning Decisions. In *3rd ACM Sigplan Symposium on Principles and Practice of Parallel Programming (PPoPP)* (April 1991).
4. Benkner, S. Vienna Fortran Compilation System - Version 2.0 - User's Guide. Tech. rep., University of Vienna, 1995.
5. Benkner, S., and Pantano, M. HPF+: Optimizing HPF for Advanced Applications. *Supercomputer 13*, 2 (1997), 31–43.
6. Benkner, S., Sanjari, K., Sipkova, V., and Velkov, B. Parallelizing Irregular Applications with the Vienna HPF+ Compiler VFC. In *HPCN Europe* (April 1998), Lecture Notes in Computer Science, Springer-Verlag.

7. Bodin, F., Beckman, P., Gannon, D., Gotwals, J., Narayana, S., Srinivas, S., and Winnicka, B. Sage++: An Object-Oriented Toolkit and Class Library for Building Fortran and C++ Restructuring Tools. In *OON-SKI'94 Proceedings of the First Annual Object-Oriented Numerics Conference* (April 1993), pp. 122–138.
8. Bodin, F., Beckman, P., Gannon, D., Narayana, S., and Yang, S. Distributed pC++: Basic Ideas for an Object Parallel Language. In *OON-SKI'93 Proceedings of the First Annual Object-Oriented Numerics Conference* (April 1993), pp. 1–24.
9. Calzarossa, M., Massari, L., Merlo, A., Pantano, M., and Tessera, D. Medea: A Tool for Workload Characterization of Parallel Systems. *IEEE Parallel and Distributed Technology 3*, 4 (November 1995), 72–80.
10. Calzarossa, M., Massari, L., Merlo, A., Pantano, M., and Tessera, D. Integration of a Compilation System and a Performance Tool: The HPF+ Approach. In *HPCN Europe* (April 1998), Lecture Notes in Computer Science, Springer-Verlag.
11. Chang, P. P., Mahlke, S. A., and Hwu, W. W. Using Profile Information to Assist Classic code Optimization. *Software - Practice & Experience* (to appear).
12. Cleveland, W. S., and MiGill, M. E., Eds. *Dynamic Graphics for Statistics*. Wadsworth & Brooks/Cole, 1988.
13. Colwell, R., Nix, R., O'Donnell, J., Papwoth, D., and Rodman, P. A VLIW Architecture for a Trace Scheduling Compiler. In *Proceedings of the Second International Conference on Architectural Support for Programming Languages and Operating Systems* (October 1987).
14. Couch, A. *Graphical Representations of Program Performance on Hypercube Message-Passing Multiprocessors*. PhD thesis, Tufts University, Department of Computer Science, 1988.
15. DeRose, L., Zhang, Y., and Reed, D. Svpablo: A Multi-Language Performance Analysis System. In *Computer Performance Evaluation Modelling Techniques and Tools* (September 1998), R. Puigjaner, N. Savino, and B. Serra, Eds., Lecture Notes in Computer Science, vol. 1469, Springer-Verlag, pp. 352–355.
16. Fahringer, T. Estimating and Optimizing Performance for Parallel Programs. *IEEE Computer 28*, 11 (November 1995), 47–56.
17. Fahringer, T. Effective Symbolic Analysis to Support Parallelizing Compilers and Performance Analysis. In *HPCN Europe* (April 1997), Lecture Notes in Computer Science, Springer-Verlag.
18. Fisher, J. Trace Scheduling: A Technique for Global Microcode Compactation. *IEEE Tranactions on Computers* (July 1981), 478–490.
19. Foster, I., and Kesselman, C., Eds. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan-Kaufmann, 1998.
20. Graham, S., Kessler, P., and McKusick, M. gprof: A Call Graph Execution Profiler. In *Proceedings of the SIGPLAN '82 Symposium on Compiler Construction* (Boston, MA, June 1982), Association for Computing Machinery, pp. 120–126.
21. Gu, W., Eisenhauer, G., Schwan, K., and Vetter, J. Falcon: On-line monitoring and steering of parallel programs. *Concurrency: Practice and Experience 10*, 9 (1998), 699–736.
22. Hartleb, F., and Mertsiotakis, V. Bounds for the Mean Runtime of Parallel Programs. In *Computer Performance Evaluation '92: Modeling Techniques and Tools* (1992), R. Pooley and J. Hillston, Eds., pp. 197–210.
23. Heath, M. T., and Etheridge, J. A. Visualizing the Performance of Parallel Programs. *IEEE Software* (Sept. 1991), 29–39.

24. Heath, M. T., Malony, A. D., and Rover, D. T. The Visual Display of Parallel Performance Data. *Computer 28*, 11 (1995), 21–8.
25. Hurley, C., and Buja, A. Analyzing High-dimensional Data with Motion Graphics. *SIAM Journal of Scientific and Statistical Computing 11*, 6 (Nov. 1990), 1193–1211.
26. Kraemer, E., and Stasko, J. T. The Visualization of Parallel Systems: An Overview. *Jour. Parallel and Distributed Computing 18*, 2 (1993), 105–17.
27. Larus, J. R., and Schnarr, E. EEL: Machine-Independent Executable Editing. In *Proceedings of the SIGPLAN '95 Conference on Programming Languages Design and Impelementation (PLDI)* (June 1995).
28. LeBlanc, T. J., and Markatos, E. P. Operating System Support for Adaptive Real-time Systems. In *Proceedings of the Seventh IEEE Workshop on Real-Time Operating Systems and Software* (May 1990), pp. 1–10.
29. Malony, A. D., Reed, D. A., and Wijshoff, H. Performance Measurement Intrusion and Perturbation Analysis. *IEEE Transactions on Parallel and Distributed Systems 3*, 4 (July 1992), 433–450.
30. Mendes, C. L. *Performance Scalability Prediction on Multicomputers*. PhD thesis, University of Illinois at Urbana-Champaign, May 1997.
31. Mendes, C. L., Wang, J.-C., and Reed, D. A. Automatic Performance Prediction and Scalability Analysis for Data Parallel Programs. In *Proceedings of the CRPC Workshop on Data Layout and Performance Prediction* (Houston, April 1995).
32. Miller, B. P., Callaghan, M. D., Cargille, J. M., Hollingsworth, J. K., Irvin, R. B., Karavanic, K. L., Kunchithapadam, K., and Newhall, T. The Paradyn Parallel Performance Measurement Tools. *IEEE Computer 28*, 11 (November 1995), 37–46.
33. MIPS Technologies Inc. *MIPS R10000 Microprocessor User's Manual*, 2.0 ed., 1996.
34. Mohr, B., Malony, A., and Cuny, J. Tau Tuning and Analysis Utilities for Portable Parallel Programming. In *Parallel Programming using C++*, G. Wilson, Ed. M.I.T. Press, 1996.
35. Nickolayev, O. Y., Roth, P. C., and Reed, D. A. Real-time Statistical Clustering for Event Trace Reduction. *International Journal of Supercomputer Applications and High Performance Computing* (1997).
36. Reed, D. A. Experimental Performance Analysis of Parallel Systems: Techniques and Open Problems. In *Proceedings of the 7th International Conference on Modelling Techniques and Tools for Computer Performance Evaluation* (May 1994), pp. 25–51.
37. Reed, D. A., Aydt, R. A., DeRose, L., Mendes, C. L., Ribler, R. L., Shaffer, E., Simitci, H., Vetter, J. S., Wells, D. R., Whitmore, S., and Zhang, Y. Performance Analysis of Parallel Systems: Approaches and Open Problems. In *Proceedings of the Joint Symposium on Parallel Processing (JSPP)* (June 1998), pp. 239–256.
38. Reed, D. A., Aydt, R. A., Noe, R. J., Roth, P. C., Shields, K. A., Schwartz, B., and Tavera, L. F. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proceedings of the Scalable Parallel Libraries Conference* (1993), A. Skjellum, Ed., IEEE Computer Society.
39. Reed, D. A., Elford, C. L., Madhyastha, T., Scullin, W. H., Aydt, R. A., and Smirni, E. I/O, Performance Analysis, and Performance Data Immersion. In *Proceedings of MASCOTS '96* (Feb. 1996), pp. 1–12.

40. Ribarsky, W., Ayers, E., Eble, J., and Mukherjea, S. Using Glyphmaker to Create Customized Visualizations of Complex Data. *IEEE Computer*, July (1994), 57–64.
41. Ribler, R., Vetter, J., Simitci, H., and Reed, D. Autopilot: Adaptive Control of Distributed Applications. In *Proc. Seventh IEEE Int'l Symp. High Performance Distributed Computing (HPDC)* (1998), pp. 172–9.
42. Robertson, G. G., Card, S. K., and Mackinlay, J. D. Information Visualization using 3D Interactive Animation. *Communications of the ACM 36*, 4 (1993), 56–71.
43. Rosenblum, L. J. Research Issues in Scientific Visualization. *IEEE Computer Graphics and Applications 14*, 2 (1994), 61–3.
44. Stasko, J., Domingue, J., Brown, M. H., and Price, B. A., Eds. *Software Visualization: Programming as a Multimedia Experience,*. MIT Press, Cambridge, MA, 1998.
45. Sun, X. H., Pantano, M., and Fahringer, T. Performance Range Comparison for Restructuring Compilation. In *IEEE International Conference on Parallel Processing* (Minneapolis, August 1998), pp. 595–602.
46. Vetter, J. S. Computational Steering Annotated Bibliography. *SIGPLAN Notices 32*, 6 (1997), 40–4.
47. Yan, J. C., Sarukkai, S. R., and Mehra, P. Performance Measurement, Visualization and Modeling of Parallel and Distributed Programs using the AIMS Toolkit. *Software Practice & Experience 25*, 4 (April 1995), 429–461.
48. Zagha, M., Larson, B., Turner, S., and Itzkowitz, M. Performance Analysis Using the MIPS R10000 Performance Counters. In *Proceedings of Supercomputing'96* (November 1996).