Navigating Complexity to Achieve High Performance

KATHLEEN NICHOLS, Apple Computer; PAUL W. OMAN, University of Idaho

Judging from the traffic on electronic networks, software performance analysis is a hot topic. Are systems "fast enough?" "Reliable" enough? What effect does making changes have on performance? These questions are important in four key areas:

- Graphics workstations have made sophisticated interfaces possible, but response time is critical.
- Real-time embedded systems have strict timing and reliability requirements.
- High-speed networks, designed to support distributed computing on ultrafast CPUs, must be found to be fast enough and cost-effective before significant resources can be devoted to their development.
- Parallel computing poses a similar challenge: Significant speedups and/or larger problem-solving must be achieved to make the additional software, hardware, and programming overhead it requires cost-effective.

These four widely disparate areas all require performance analysis. In the past, performance has either been ignored or been carried out by running discrete, repetitious experiments that produce metrics that were then analyzed by hand.

Today we know that software performance is not a discrete entity that can be observed or measured directly. It is a complex, multidimensional assessment dependent on operational constraints that may or may not have been specified prior to the
analysis. We study today’s complex systems with more than simple measurements: We use instrumenting, modeling, and (software and hardware) simulation.

The overwhelming amount of data produced by today’s analysis systems require that it be digested before it’s presented to the user, possibly with some analysis done or important data highlighted. This issue presents several ways to do this.

DEFINITIONS

Performance analysis is the evaluation of a deed, feat, process, or presentation — software performance analysis is the evaluation of a software event. But evaluation against what standard? Of what characteristics? And by whom? The IEEE’s software-engineering glossary provides some key definitions:

**Performance:** (1) The ability of a computer system or subsystem to perform its functions. (2) A measure of the ability of a computer system or subsystem to perform its functions, for example, response time, throughput, number of transactions.

**Performance evaluation:** The technical assessment of a system or a system component to determine how effectively operating objectives have been achieved.

**Performance specification:** (1) A specification that sets forth the performance requirements for a system or system component. (2) Synonymous with requirements specification.

Clearly, software performance is critical to system performance. Software’s most obvious feature is correctness, so we tend to think of correctness as a first criterion for performance.

**Correctness:** (1) The extent to which software is free from design defects and from coding defects; that is, fault free. (2) The extent to which software meets its specified requirements. (3) The extent to which software meets user expectations.

**Proof of correctness:** (1) A formal technique used to prove mathematically that a program satisfies its specifications. (2) A program proof that results from applying this technique.

Correctness is not a binary decision: Is it correct, yes or no? Correctness from a software-engineering point of view is a twofold assessment of accuracy: (1) Is it logically accurate (Does the computation model what we want to model?); and (2) is it physically accurate (Are the computed values accurate enough to suit our needs?)? In this sense, correctness is more like Barry Boehm’s definitions of verification (Did we build the product right?) and validation (Did we build the right product?) than it is to most formal definitions of correctness.

**MEETING REQUIREMENTS**

In this issue, we focus on the part of the correctness definition that pertains to the extent to which software meets its specified requirements and user expectations. What good is a correct solution if it is not reached quickly enough or if it is not maintainable?

With this view of correctness, we define software efficiency as the degree of fit between the requirements and the discrete confines of time and space, with an eye toward accuracy and some measurable criteria. Performance analysis requires a realistic assessment of a system’s efficiency.

In a nutshell, performance analysis is the measuring, modeling, and subsequent tuning of software’s time, space, efficiency, and accuracy. Software tuning is the iterative application of measurement and modeling interspersed with adjustments. Tuning repositions the work within the performance landscape to fit the contours of the host system, the requirements specification, and users’ expectations.

**PERFORMANCE LANDSCAPE**

The software-performance landscape has a complexity that is manifested in two major ways: First, there are many indicators of system performance to choose from. We call the approach that seeks to reduce or combine the number of performance indicators a coalescing technique.

These techniques are analogous to using a compass to get your bearings and negotiating the landscape of the program on the host system.

Second, the increasing intricacy of systems makes it difficult not only to model them, but also to solve or instrument the model and discover what the solution means. We call the approach that renders complexity palatable an interpretive technique. Interpretive techniques present as much data as possible for human information processing and interpretation, just like a topological map or aerial view.

So there are two major tools for negotiating the performance landscape: measuring and modeling. Measurement, embodied in hardware and software instrumentation and tuning, quantifies a real system. Modeling lets you simulate and analyze a real or proposed system. These tools are often used together — measurement being used to validate a model. Trace-driven simulation is an example of using these methods together.

**THIS ISSUE**

This article collection is a good sampling of the performance landscape. The first four articles deal with measurement, the last two with modeling.

Two articles describe a coalescing technique:

- “Measuring and Analyzing Real-Time Performance” explains a time-complexity function derived from program timing and programmer input about expected timing complexity that fits curve using sound statistical methods.
- “Finite-Element Analysis on a PC” is a case study of how to improve performance based on a program’s timing measurements.

All four articles that describe interpretive techniques present systems that use color graphics under X Windows:
“Traceview: A Trace Visualization Tool” describes a graphical way to study large event trace files that follow a specific, simple format.

“Visualizing the Performance of Parallel Programs” describes ParaGraph, which uses the widely available PICL facility to get the measurement data that drives its displays, thus making a heroic effort at standardization in message-passing multicomputer performance.

“Interactive Visual Modeling from Performance Analysis” describes Q+, descended from one of the earliest visual performance tools and based on queueing networks, probably the most popular performance-modeling technique.

“Performability Modeling with UltraSAN,” also describes visual, hierarchical modeling like Q+, but it uses stochastic activity networks instead of queueing networks. UltraSAN embodies many innovations to keep model size small, thus speeding simulation and aiding analysis.

ACKNOWLEDGMENTS
We thank Rebecca Hammaker of Apple Computer for handling the voluminous photocopying and mailings involved in editing this special issue.

REFERENCES

Kathleen Nichols is a staff engineer at Apple Computer. Her research interests include performance evaluation of networks and multiprocessor interconnects, modeling multimedia network loads, and performance tools.

Nichols received a BS in electrical engineering from the University of Pittsburgh and an MS and PhD in electrical engineering from the University of California at Berkeley. She is a member of ACM and the IEEE.

Paul W. Oman is an assistant professor of computer science at the University of Idaho. Previously, he served as systems analyst and computer center director at various locations. He has written several articles on software tools and programming.

Oman received a PhD in computer science from Oregon State University. He is a member of ACM, IEEE, IEEE Computer Society, and Phi Kappa Phi.

Address questions about this issue to Nichols at Apple Computer, Advanced Technology Group, 20450 Stevens Creek Blvd., Cupertino, CA 95014; Internet nichols@apple.com.
Symposium on
Assessment of Quality Software Development Tools
May 27 - 29, 1992 - New Orleans, Louisiana

In the last few years, there has been a major renaissance in the availability, kinds, and scope of software development tools. Modern tools come in large number and large variety, creating a new challenge to software engineers: how to choose the right tools. There is no clear and simple way today to go about assessing tools and matching them to the needs of development organizations. This Symposium will review the problems of assessing software development tools, solicit case studies of tool applications and their impact on productivity, and examine strategies for the evaluation of future tools. A particular focus will be on the assessment of tools to assist with productivity and quality in software development.

Papers appropriate to the symposium are: studies of existing tools automating some task in software design, analysis, implementation, testing or maintenance. The program committee invites submission of completed original papers, not submitted to any other meeting or publication, addressing (but not limited to) the following areas:

- Quality Development/Design tools, CASE environments
- Quality Analysis tools
- Test/Verification/Validation tools
- Design Automation tools
- Prototyping tools
- Knowledge-based / Expert Systems
- Program Understanding and Reverse Engineering tools
- Experience with tool introduction and practical use

Please submit five (5) copies of full papers in English by November 27, 1991 to the General Conference Chair:

Ez Nahourall, IBM (718)892-8898, 3321 San Ignacio Avenue, San Jose, CA 95119 USA
(408) 281-5741 eznah@stlm71nus1.ibm.com

For information and registration, contact:
Judy Lee, IBM, 1000 NW 51 St., Boca Raton, FL 33432 USA (407) 962-1048

Important Dates:
Paper deadline: November 27, 1991; Authors notification: February 14, 1992

Sponsored by: Tulane University
In Cooperation: IEEE Computer Society TCSE

Call for Papers

Papers due: November 27, 1991

General Conference Chair:
Dr. J. Browne, U. Texas Austin & Dr. J. Hassell, Tulane Univ.

Program Chair: Ez Nahourall, IBM

Program Committee:
D. Belanger, AT&T Bell Labs. C. Richter, MCC
J. Cameron, LBMS U.K. F. Petry, Tulane Univ.
F. Chikinskay, Progress Software S. Shatz, U. Illinois - Chicago
Y. Fujiwara, Univ Taubikou Japan D. Sorn, Siemens
A. L. Goel, Syracuse Univ. L. Tripp, Boeing
J. Jenkins, City Univ, London C. Lamy, IBM France
A. L. Goel, Syracuse Univ. others...

With Assistance: IBM Systems & Software Education