Experiment Management Support for Performance Tuning[1]

Karen L. Karavanic and Barton P. Miller

{karavan,bart}@cs.wisc.edu

Computer Sciences Department
University of Wisconsin
Madison, WI 53706-1685

Abstract

The development of a high-performance parallel system or application is an evolutionary process -- both the code and the environment go through many changes during a program's lifetime -- and at each change, a key question for developers is: how and how much did the performance change? No existing performance tool provides the necessary functionality to answer this question. This paper reports on the design and preliminary implementation of a tool which views each execution as a scientific experiment and provides the functionality to answer questions about a program's performance which span more than a single execution or environment. We report results of using our tool with an actual performance tuning study and with a scientific application run in changing environments. Our goal is to use historic program performance data to develop techniques for parallel program performance diagnosis.

1 INTRODUCTION

> "An experimental science is supposed to do experiments that find generalities. It's not just supposed to tally up a long list of individual cases and their unique life histories. That's butterfly collecting."
>
> - Richard C. Lewontin [1]

The development of a high-performance parallel system or application is an evolutionary process. It may begin with models or simulations, followed by an initial implementation of the program. The code is then incrementally modified to tune its performance and continues to evolve throughout the applications's life span. At each step, the key question for developers is: *how and how much did the performance change?* This question arises comparing an implementation to models or simulations; considering versions of an implementation that use a different algorithm, communication or numeric library, or language; studying code behavior by varying number or type of processors, type of network, type of processes, input data set or work load, or scheduling algorithm; and in benchmarking or regression testing. Despite the broad utility of this type of comparison, no existing performance tool provides the necessary functionality to answer it; even state of the art research tools such as Paradyn[2] and Pablo[3] focus instead on measuring the performance of a single program execution.

We describe an infrastructure for answering this question at all stages of the life of an application. We view each program run, simulation result, or program model as an *experiment*, and provide this functionality in an *Experiment Management* system. Our project has three parts: (1) a representation for the space of executions, (2) techniques for quantitatively and automatically comparing two or more executions, and (3) enhanced performance diagnosis abili-

1. This work is supported in part by Wright Laboratory Avionics Directorate, Air Force Material Command, USAF, under grant F33615-94-1-1525 (DARPA order no. B550), NSF grants CDA-9024618 and CDA-9623632, and Department of Energy Grant DE-FG02-93ER25176. The U.S. Government is authorized to reproduce and distribute reprints for Governmental purposes notwithstanding any copyright notation thereon. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either expressed or implied, of the Wright Laboratory Avionics Directorate or the U.S. Government.

.

ties based on historic performance data. In this paper we present initial results on the first two parts. The measure of success of this project is that an activity that was complex and cumbersome to do manually, we can automate.

The first part is a concise representation for the set of executions collected over the life of an application. We store information about each experiment in a *Program Event*, which enumerates the components of the code executed and the execution environment, and stores the performance data collected. The possible combinations of code and execution environment form the multi-dimensional *Program Space*, with one dimension for each axis of variation and one point for each Program Event. We enable exploration of this space with a simple naming mechanism, a selection and query facility, and a set of interactive visualizations. Queries on a Program Space may be made both on the contents of the performance data and on the metadata that describes the multi-dimensional program space. A graphical representation of the Program Space serves as the user interface to the Experiment Management system.

The second part of the project is to develop techniques for automating comparison between experiments. Performance tuning across multiple executions must answer the deceptively simple question: what changed in this run of the program? We have developed techniques for determining the "difference" between two or more program runs, automatically describing both the structural differences (differences in program execution structure and resources used), and the performance variation (how were the resources used and how did this change from one run to the next). We apply our technique to compare an actual execution with a predicted or desired performance measure for the application, and to compare distinct time intervals of a single program execution. Uses for this include performance tuning efforts, automated scalability studies, resource allocation for metacomputing [4], performance model validation studies, and dynamic execution models where processes are created, destroyed, migrated [5], communication patterns and use of distributed shared memory may be optimized [6,9], or data values or code may be changed by steering [7,8]. The difference information is not necessarily a simple measure such as total execution time, but may be a more complex measure derived from details of the program structure, an analytical performance prediction, an actual previous execution of the code, a set of performance thresholds that the application is required to meet or exceed, or an incomplete set of data from selected intervals of an execution.

The third part of this research is to investigate the use of the predicted, summary, and historical data contained in the Program Events and Program Space for performance diagnosis. We are exploring novel opportunities for exploiting this collection of data to focus data gathering and analysis efforts to the critical sections of a large application, and for isolating spurious effects from interesting performance variations. Details of this are outside of the scope of this paper.

2 PROGRAM EVENTS AND AUTOMATED COMPARISON

The *Program Event* is a representation of a program run or execution. Associated with each Program Event is a collection of performance data. We automatically compare the changes between two (or more) Program Events with our structural difference operator.

2.1 The Program Event

We represent a program as a collection of discrete program resources. Possible resources include the program code, application processes, machine nodes, synchronization points, data structures, and files. Each group of resources provides a unique view of the application. We organize the program resources into classes according to the aspect of the application they represent, and structure each class as a tree, called a *resource hierarchy.*

Definition 2.1. A *Program Event*, $\mathcal{E}$, is a forest composed of zero or more unique resource hierarchies:

$$\mathcal{E} = R_0 \cup R_1 \cup \ldots \cup R_n, \qquad n \geq 0. \qquad ❑$$

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

In the `Code` resource hierarchy of Figure 1, the root level (level 0) is the program-level view of an application, which represents the cumulative behavior of the whole program. Level 1 is the module-level view of the source code, and level 2 is its function-level view. Each level of a resource hierarchy is a set of resources, and each level above the leaf level is a partition of the set of nodes in the next lower level. For example, the module level of Figure 1, which

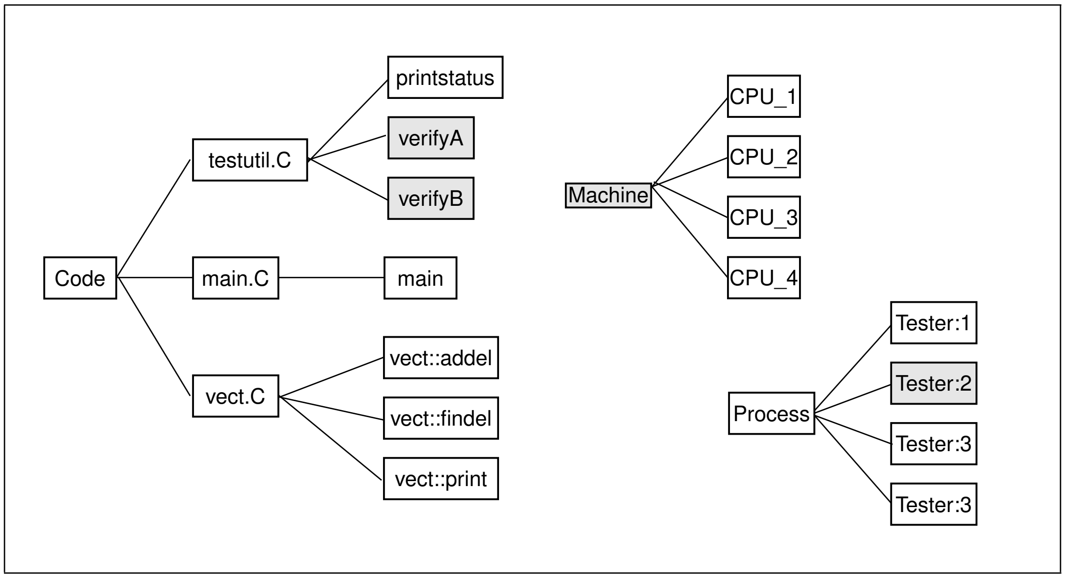

Figure 1: Program Event for Program *Tester,* with Three Resource Hierarchies

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

$R_0^0 = \{\text{Code}\}$ $\qquad R_0^1 = \{\text{testutil.C, main.C, vect.C}\}$

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

Definition 2.2. A *resource hierarchy R* is a tree of the form

$R = (r, T)$

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

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

Figure 1 shows a sample Program Event for a parallel application called *Tester.* The Program Event is the set {Code, Machine, Process}. The Code hierarchy contains nodes that represent the program's modules and functions; the Machine hierarchy contains one node for each CPU on which Tester executed; and the Process hierarchy contains one node for each process.

A resource is a representation of a logical or physical component of a program execution. A single resource might be used to represent a particular aspect of the program or the environment in which it executes: a process, a function, a CPU, or a variable. In Figure 1, the leaf node labeled "verifyA" represents the resource <Code/testutil.C/verifyA>. The semantic meaning attached to a particular resource is relevant only to the programmer and does not affect the model functionality; however, a program execution component must be uniquely represented by a single resource. Each internal node of a resource hierarchy tree represents a set of one or more resources; for example, <Code/testutil.C> is a single resource that represents the aggregation of the set {printstatus, verifyA, verifyB}.

A *resource name* is formed by concatenating the labels along the unique path within the resource hierarchy from the root to the node representing the resource. For example, the resource name that represents function verifyA (shaded) in Figure 1 is <Code/testutil.C/verifyA>.

2.2 Representing Performance Information in a Program Event

For a particular performance measurement, we may wish to specify certain part or parts of a program. For example, we may be interested in measuring CPU time as the average for all executions, as the total for one entire execution, or as the total for a single function. The *focus* constrains our view of the program to a selected part. Selecting the root node of a hierarchy represents the unconstrained view, the whole program. Selecting any other node narrows the view to include only those leaf nodes that are descendents of the selected node. For example, the shaded nodes in

Figure 1 represent the constraint: functions verifyA and verifyB of process Tester:2 running on any CPU.

Definition 2.3. A *focus F* is formed by selecting one resource node from each of the resource hierarchies R:

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

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

The Program Event provides an intuitive naming scheme for program components called *resource normal form*. We convert the selected set to *resource normal form* by concatenating the selections from each resource hierarchy. For example, the shaded selection of Figure 1 is represented as:

< /Code/testutil.C/(verifyA,verifyB), /Machine, /Process >.

Performance data is represented by a function of the form $P(\mathcal{E}, m, f, t)$:

- $\mathcal{E}$: a program execution as defined in Definition 2.1.
- *m:* the name of the *metric*, which is a measurable execution characteristic, such as CPU time.
- *f:* the *focus* for this performance data, for example CPU time for focus < /Code/Main.C, /Machine, /Process >.
- *t:* the *time interval*, specifies when during an execution the data was collected.

P returns a *performance result (r),* which may be a simple scalar or more complex object. Because data is uniquely identified using a focus, we say the performance data is stored in *resource normal form*.

2.3 The Structural Difference Operator

When comparing the performance of two program executions, a natural first question is, how different were the code and environments used in the two tests, and where did they differ? If we perform two test runs using identical code running on identical, dedicated platforms, every resource hierarchy of the first execution has an identical counterpart in the second execution; performance data may be meaningfully compared for every focus that is valid for one of the individual executions. The comparison becomes more complex if we consider cases in which either the code or the run time environment (or both) differ between our two test runs.We need to determine the common set of valid resources to determine the new set of valid foci for the program.

Given two Program Events, $\mathcal{E}_1$ and $\mathcal{E}_2$ (see Figure 2) , we can calculate their structural difference. To perform a

$S \leftarrow \{\ \}$

$\forall (r_i, T_i) \in \mathcal{E}_1$

  if $\exists (r_j, T_j) \in \mathcal{E}_2$, s.t. match (r_i, r_j) **then**

    $S \leftarrow S \cup \{ r_i \oplus r_j, T_i \oplus T_j\}$

    $\mathcal{E}_2 \leftarrow \mathcal{E}_2 - (r_j, T_j)$

  else $S \leftarrow S \cup \{(r_i, T_i)\}$

$S \leftarrow S \cup \mathcal{E}_2$

Figure 2: Algorithm to find the Structural Difference of two Program Events, $\mathcal{E}_1 \oplus \mathcal{E}_2$

structural comparison of two program executions, we compare the two sets of resource hierarchies in a top-down manner. The *Structural Difference Operator,* $\oplus$, takes two Program Events as operands and yields a Program Event Group (PEG). The result contains all resources from the original two Program Events, so the structural difference operation works as a hierarchical set union operation. This operator can be applied iteratively to build up a single set of resource hierarchies that characterizes any number of distinct program runs.

Figure 3 shows an application of the structural difference operator to two program events $\mathcal{E}_1$ and $\mathcal{E}_2$. The top set of resource hierarchies describes event $\mathcal{E}_1$, and the middle set describes $\mathcal{E}_2$. The bottom of the figure shows the result. Resources common to both executions are outlined with mixed dash/dot; <Code> and <Code/Module2> are examples of such resources. Resources unique to $\mathcal{E}_1$ or $\mathcal{E}_2$ are outlined with dot or solid boxes respectively.

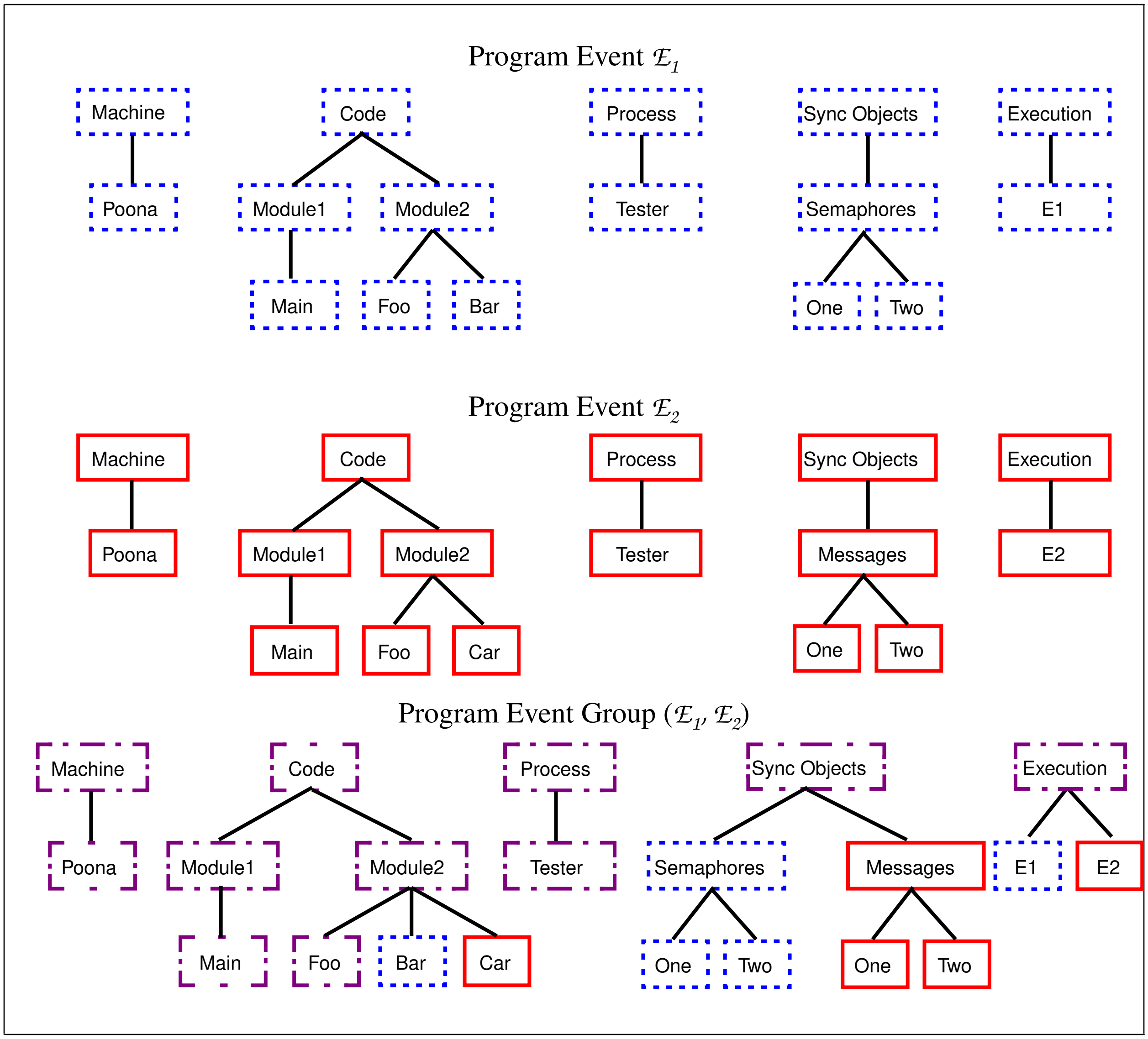

Figure 3: An Example of the Structural Difference Operator

2.4 Moving Beyond the Single Execution Model

A collection of two or more Program Events is stored in a Program Event Group (PEG). The interface to the PEG is centered around a single set of resource hierarchies, constructed using the Structural Difference Operator, which represents the components of all of its Program Events. The collection of Program Events is stored in a multi-dimensional Program Space. Users can navigate this space, visualize its form and contents, and make queries on the structure and contents. We developed two aggregation operators of particular use in viewing PEG performance data: the *list* aggregation operator, when applied to a hierarchy that contains more than one selected node, yields a performance result that is a list of values rather than a single value; and the *cluster* aggregation operator groups the requested performance data by value – each group contains values that were within a specified Δ of each other (represented by the average of the group).

We have defined several new metrics particular to the multi-execution PEG. The *discrete distance* metric is a binary function that indicates whether or not two specified performance results differ by more than a specified interval. The metric *dd* is calculated as follows:

$$dd(x,y) = \begin{cases} 1 & if \quad |x-y| \geq \delta \\ 0 & if \quad |x-y| < \delta \end{cases}$$

where x and y are two different performance results. We use the discrete distance metric as a building block for clustering and differencing of performance results.

The *performance difference* operator takes as inputs two executions and returns a list of all foci for which the discrete distance metric yields an answer of true. The list is computed hierarchically as shown in Figure 4. We iterate through the resource nodes of the Program Event, starting with the focus which represents the entire program execution, "currentFocus." If there is a performance difference noted, we then check more specific foci for the same metric. We provide a display of performance difference results as part of our prototype (see Figure 7). This metric is useful to focus attention immediately to performance changes from one version to the next.

```
answer ← {}
enqueue (pendingQueue, wholeProgram)
while ! (isEmpty (pendingQueue))
        currentFocus ← dequeue (pendingQueue)
        pr1 ← P(ℰ1, m, currentFocus, )
        pr2 ← P(ℰ2, m, currentFocus, )
        If dd(pr1, pr2) = true
                answer ← answer ∪ {currentFocus)
                for each f in magnify(currentFocus)
                        enqueue (pendingQueue, f)
return answer
```

Figure 4: Algorithm for the Performance Difference Operator perfDiff ($\mathcal{E}_1$, $\mathcal{E}_2$, m). The performance difference operator searches through the performance results for the specified metric *m* and the requested Program Events $\mathcal{E}1$ and $\mathcal{E}2$, returning a list of all foci for which the results are different.

We construct more specific foci using an operation we call magnify (see Figure 5). As specified in Definition 2.3, a focus contains one node from each resource hierarchy. For each resource hierarchy in the original focus, we form a set of new foci by replacing the given resource with the each of its children in the next lower level of the hierarchy. The result of the magnify operation applied to a focus is a set of foci.

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

```
answer ← {}
for i = 1 to n
        f ← f - ri
        for each j in children(ri)
                answer ← answer ∪ (f + j)
return answer
```

Figure 5: Algorithm for magnify(*f*). Magnify returns the set of foci constructed by making all possible one step descents down the resource hierarchy from the given starting focus.

The previous two metrics aim to describe *what* has changed. We would also like a metric to describe *how much* performance has changed. We are currently investigating two approaches to a *performance distance* metric: a Euclidean distance, and a weighted average of performance result values. The goal is a quantitative measure of how much performance differs between two or more executions, which can be used to weigh or rank performance bottlenecks.

3 CASE STUDIES

To test out our model on existing scientific applications, we have implemented a preliminary prototype of an experiment management performance tool. The prototype was written using C++ and Tcl/Tk [19] extended with a tree widget [20]. Resource hierarchy loading, display, and differencing were all implemented directly in Tcl/Tk. To test our design we have examined data collected while tuning several parallel programs. We provide examples of using our prototype to (1) compare implementations based on alternate communication libraries, (2) evaluate performance as a program evolves through versions, and (3) track data for a scalability analysis. In each case, the use of an experiment management system simplifies and speeds the programmer's task.

3.1 Comparing Alternate Implementations: Porting a PVM Application to MPI

In this study we compared the structure of two versions of a parallel message-passing FFT code called ns, which was ported from the PVM to the MPI message passing libraries. The application solves the Navier-Stokes equation in three dimensions. A scientist porting an application wants directed feedback about the resulting changes in performance to the application, and hopefully some idea of the cause of any performance degradation. The first step is to provide feedback on the structural differences between the old and new versions. In Figure 6 we show the resource

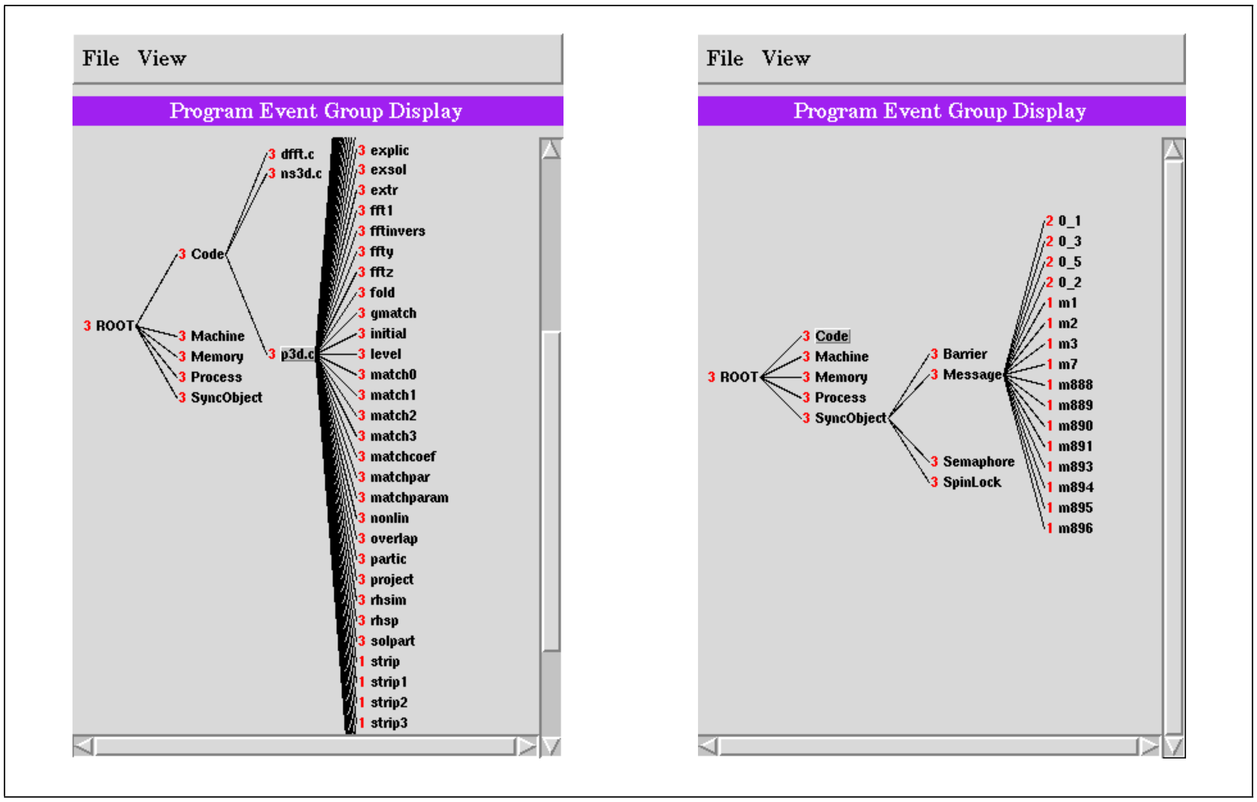

Figure 6: Resource Hierarchies for Program Event Group: nspvm, nsmpif. The PEG Display allows the developer to navigate resource hierarchies and quickly see what differed structurally between two (or more) program runs. The display is organized like the resource hierarchies from Figure 1, with the addition of an integer run identifier (RID). Each run is labeled with a value 1, 2, 4, 8, etc. Resources that appeared in only one run, are labeled with the RID of that run (1 or 2 in this example). Resources that appear in more run are labeled with the sum of the RIDs; the resources labeled with 3 appeared in both runs 1 and 2.. We show here two views of the PEG: the display on the left has the module "p3d.c" expanded, while the display on the right results from expanding the Message node to see individual message tags.

hierarchies that resulted from applying the Structural Difference Operator to two Program Events, a 4-node run of nspvm (PVM version) and a 4-node run of nsmpif (MPI version), both on the IBM SP-2. This display provides a quick way to see what differed in both the code and the environment between the two runs. The left in Figure 6 shows that Code, Machine, Memory, Process, and SyncObject resources appear in both Program Events. In the Code hierarchy, three modules (dfft.c, ns3d.c, and p3d.c) appear in both runs. At the leaf, we can see four procedures (strip, and strip1-3) appeared only in Program Event 1. The Message hierarchy on the right shows the changes in message tags: tags 0_1, 0_3, 0_5, and 0_2 represent MPI message tags, and the rest represent the message tags for the PVM version. By selecting any of these resources, we can display the performance data for the Program Event(s) that include that resource.

3.2 An Evolving Application: Performance Tuning a Shared Memory Application

A protein-folding application, called Fold4, was recently developed in our Chemical Engineering Department. The tuning effort was reported in detail elsewhere [10]. We analyzed the program versions and performance data from this study and were able to automate the identification of changes from one version to the next.

We ran three versions of Fold4, taken from different steps in the performance tuning study. The researchers conducting the study had located several problems. Version 1 was a simple port to the Wisconsin Cluster of Workstations (COW) from the SGI PowerChallenge. A problem was identified with a serial portion of the code that consumed 40% of the execution time on 8 nodes. Version 2 changed to data partitioning to try to relieve the bottleneck. A problem was identified with false sharing of data blocks. Version 3 padded and aligned data to improve the cache behavior. We present selected results here to demonstrate the benefit of the experiment management approach in navigating the large space of resources and data involved in a complete performance tuning study.

First we examined the structural differences between versions. To consider the changes from version 1 to version 2, we merged these two Program Events (by applying the Structural Difference Operator) and generated a Program Event Group (PEG). The PEG display distinguishes foci valid for both Program Events. The Memory hierarchy shows that a data change of some kind occurred, since there are more data structures in the memory hierarchy for Version 2.

Next we examined the performance differences, by applying the performance difference operator to metric memoryBlockingTime. The perfDiff operator pairwise compares performance data available for both Program Events. The results are presented in the perfDiff display (shown in Figure 7). This display shows that memory blocking behavior differed overall in the two runs; further it differed in each process, and that those differences were localized to five data structures (GM, GM->part0-3).

3.3 A Scalability Study

In this study we compared 4 executions of a parallel global circulation message-passing code called Ocean. We ran the same code on 2, 4, 8, and 16 nodes of the Wisconsin COW, and used the merge, cluster, and performance difference features to explore the behavior of the application while scaling. Figure 8 shows the Program Event Group Display. This Program Event Group was built up by applying the structural difference operator twice. The set of resource nodes it contains is the union of all resource hierarchies in the three original Program Events. The two views of Figure 8 demonstrate our prototype's ability to represent an application's different environments. In this particular study, the different runs re-used a common set of machine nodes, which is clearly shown in the first display. Nodes labelled "cow03," "cow04," "cow05," and "cow06" were common to all three executions, and are tagged with a "5." Nodes "cow07," "cow08," "cow17," and "cow18" are all unique to the 16-node program run, and are tagged with a "1." The remaining machine nodes were used in both the 8- and the 16-node runs, and are tagged with a "3." The display on the right shows the different processes. A string comparison here yields limited information, since each process name is unique. The display demonstrates our prototype's ability to provide an interface and naming mechanism for the various types of program resources found in a multi-execution performance tuning environment.

We show an example of the results of a multi-execution query in Figure 9. Here we requested the metrics CPUtime, SyncWaitTime, and IOWaitTime for three executions of the application, on 2, 4, and 8 nodes of a SUN SPARCstation cluster. The data used here was collected using Paradyn, and is stored in the form of Paradyn histograms. The result of the query is 9 Paradyn histograms. Here we display the result using xmgr.

4 RELATED WORK

There has been extensive work on parallel profiling tools [11,12,3,2,13], and a more limited amount of work on tools that allow displays of performance data from multiple runs [14,21]. In the debugging area, work has been done to find bugs by comparing the execution of a new program version to an old one by comparing program output and user-selected variables[16]. GUARD compares the program currently being debugged with a reference version of the same program, to detect differences in variable values at user-defined points during the execution. Comparisons are made of simple or complex data types; the programs compared may be written in different languages or run across heterogeneous environments. They have defined difference operators for simple and complex data types which can compare complex structures and factor out machine dependent data representations, returning a boolean result which indicates whether the two variable values differed at the matching points in each computation. Unlike our approach, the programmer must separately determine points of comparison and items of comparison for each version of the pro-

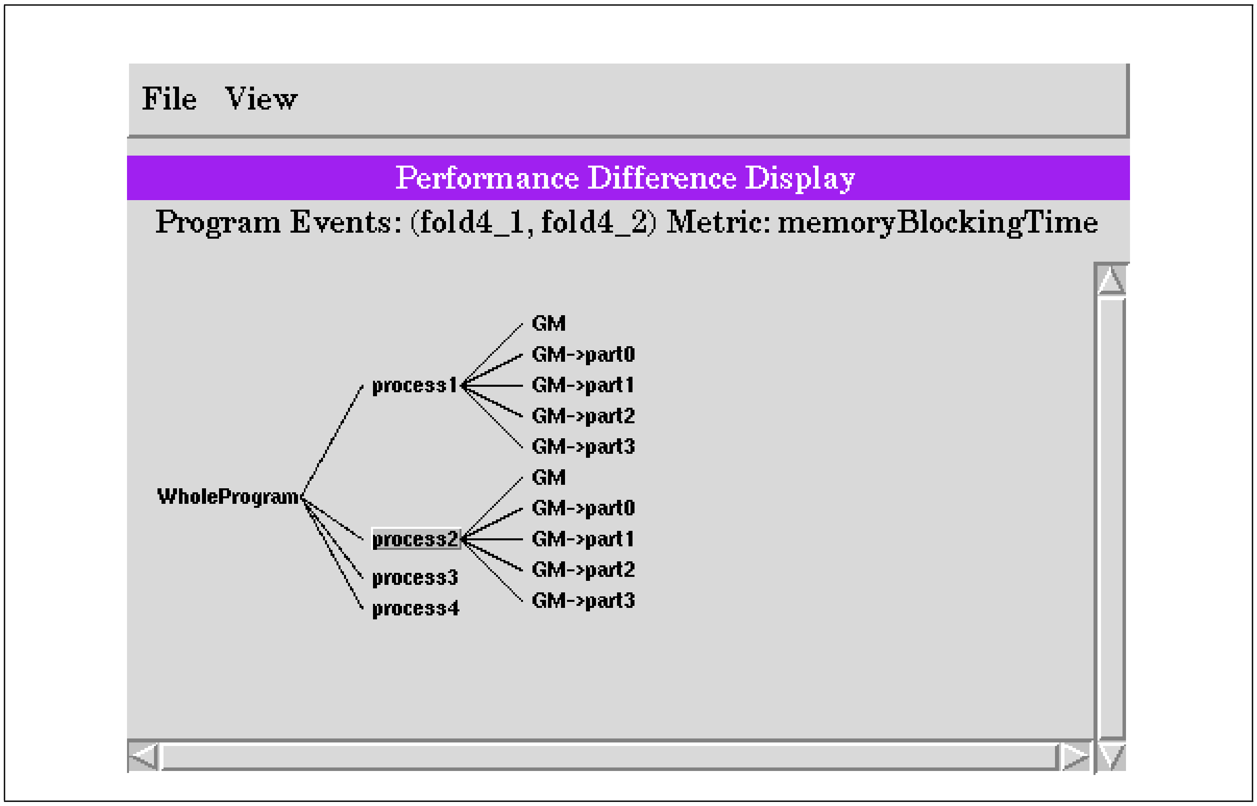

Figure 7: Results of the Performance Difference Operator for metric MemoryBlockingTime. The nodes shown represent resource combinations for which there is a performance difference detected between the Program Events. Starting at the left, the node WholeProgram means that there is some performance change between the two runs. The process nodes indicate that performance changed in some way for each of the four processes. The next level details individual shared data structures: GM is a shared index structure, and GM->part0, GM->part1, GM->part2, and GM->part3 are the shared data structures listed in the index. The data structures listed are those common to both runs for which memoryBlockingTime changed. This snapshot was taken after selecting two nodes, process1 and process2, to see a detailed display of their children. A visualization of the performance data for any node, showing the plots of the values for each run, can be launched by selecting the node on this display.

gram.

Several recent efforts have defined program similarity. MTOOL [22] includes a metric that compares observed and predicted execution times to the granularity of individual basic blocks. Their approach, useful only for memory tuning, defines the difference between predicted and observed execution time as a memory bottleneck. Mendes [23] defines similarity for execution graphs used in his trace transformation approach to performance prediction. They define similarity s between two graphs as the degree of the largest isomorphic graphs that are induced subgraphs of each of the two original graphs. The distance d between the graphs is defined as d = n - s, where n is the number of vertices of each of the original graphs. Saavedra and Smith [24] use two metrics for program similarity: program characteristic similarity, which is a normalized squared Euclidean distance, and execution time similarity. These are addressing a similar problem to our structural difference and performance difference operators, although our approach is to provide an extensible framework for both structural and execution comparison, and to allow performance difference to be calculated at many different levels of detail and for many different metrics.

The Wisconsin Program Slicing Project at UW-Madison [25] investigates methods for determining syntactic and semantic differences between two versions of a sequential source code. Their focus is on precise characterization and identification of the differences between two program versions. It is currently limited to sequential Pascal codes, and because it examines source code only, is not useful in identifying differences between executions of the same code in differing environments.

In the physical and life sciences, there has been work to automate and store experimental data [15]. Interactive data exploration and scientific data visualization is the subject of much ongoing research. We restrict the scope of our research to identifying issues particular to a multi-execution environment. We combined Paradyn and Devise [14] to enable side-by-side, linked visualization of data from multiple executions of a parallel application. However, this ini-

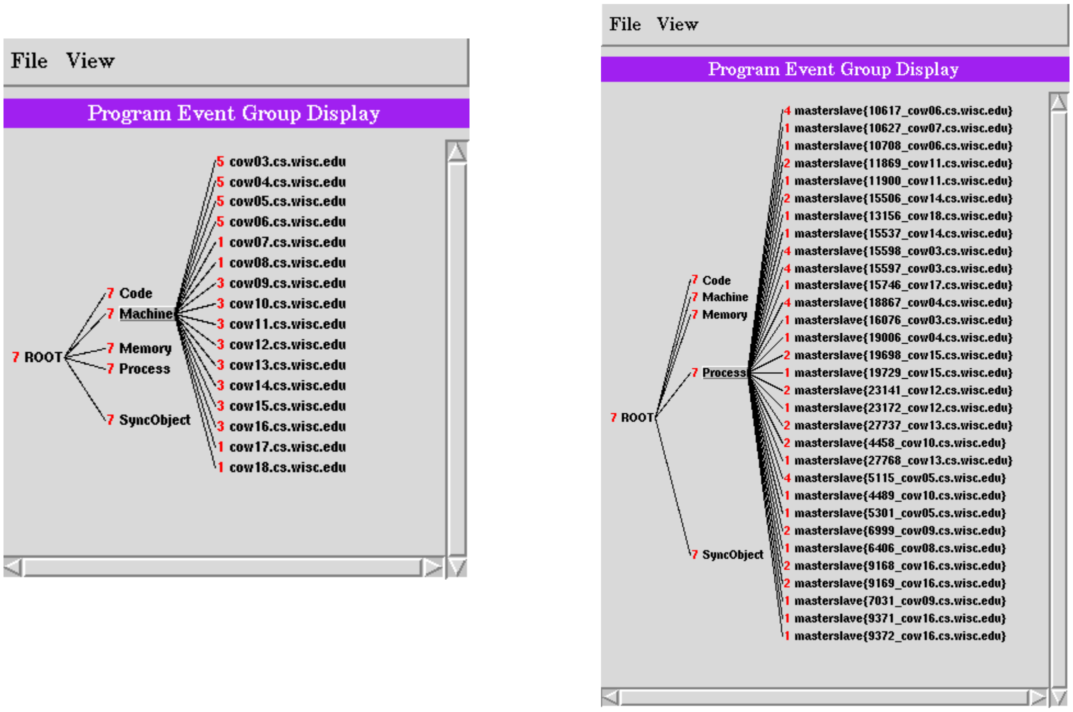

Figure 8: Resource Hierarchies for Program Event Group: ocean. This Program Event Group represents three executions of the PVM application "ocean" run on 4, 8, and 16 nodes, and is the result of performing two consecutive structural difference operations. We show here two views of the PEG: the display on the left has the module "machine" expanded, while the display on the right results from expanding the process node.

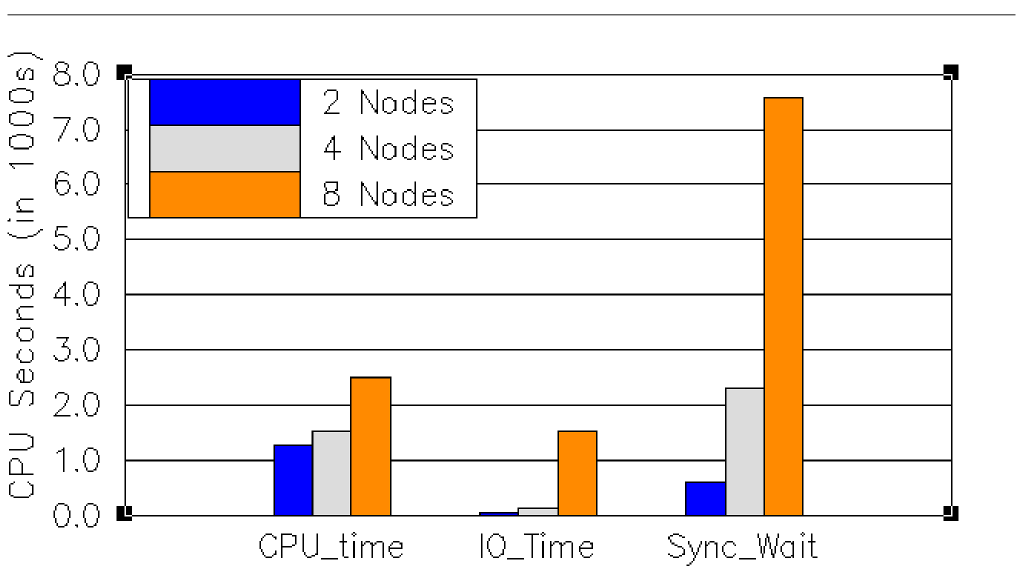

Figure 9: Results of the metrics CPUtime, IO_Time, and Sync_Wait for a Program Event Group.

tial effort did not address the problems in trying to match phases of the program across two runs of different execution time, it simply displayed the two runs without any ability to correlate a point in one graph with the corresponding point of execution in the other graph. IPS-2 [21] allowed linked visualization of data from different runs, matching them by normalizing to total execution time. Ribler, Abrams et al Virginia [26] have developed alternative methods for visualizing categorical trace data, including recent work on visualizing multiple traces simultaneously.

Our representation of a Program Space, use of an experiment management infrastructure, and automated comparison of program runs represents a new direction in defining a performance tool that more closely maps to the actual process programmers must go through to tune their codes, and that uses automation to focus the user's attention into small critical areas of change.

5 CONCLUSIONS

Taking an experiment management view of performance tuning allows us to describe an application's behavior throughout its lifetime and across a variety of environments and code ports. Our multi-execution view provides much valuable feedback for the scientist developing or maintaining a parallel code. We are currently working to expand the prototype described here, with a richer set of data types, a web-based interface to service developers engaged in geographically distributed cooperative program development and use, and additional performance displays. We are investigating the use of an object oriented database for data storage, and exploring techniques to improve Paradyn's on-the-fly diagnosis capabilities using feedback from our experiment management system.

6 REFERENCES

[1] D. L. Wheeler. Top population geneticist delivers scathing critique of field. *The Chronicle of Higher Education*, February 14, 1997.

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

[3] D. Reed *et al*. Scalable performance analysis: The Pablo performance analysis environment. I. C. Press, editor, *Proc. Scalable Parallel Libraries Conference*, pages 135–142, Los Alamitos CA, 1993.

[4] I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *International Journal of Supercomputer Applications*, 1997, to appear.

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

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

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

[8] K. Kunchithapadam and B. P. Miller. Integrating a Debugger and a Performance Tool for Steering, in **Debugging and Performance Tools for Parallel Computing Systems**. IEEE Computer Society Press, 1996. ed. by M.L Simmons, A.H. Hayes, J.S. Brown, and D.A. Reed.

[9] S. Fink, S. Kohn, and S. Baden. Flexible communication mechanisms for dynamic structured applications. *IRREGULAR '96*, Santa Barbara, CA, August 1996.

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

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

[12] W. Williams, T. Hoel, and D. Pase. The MPP Apprentice performance tool: Delivering the performance of the Cray T3D. *Programming Environments for Massively Parallel Distributed Systems*, Monte Verita, 1994.

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

[14] K. L. Karavanic, J. Myllymaki, M. Livny, and B. P. Miller. Integrated visualization of parallel program perfor-

mance data. *Parallel Computing*, 23(1997) 181-198.

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

[16] R. Sosic and D. Abramson. Guard: A relative debugger. *Software Practice and Experience*, to appear.

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

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

[19] J.K. Ousterhout. *Tcl and the Tk Toolkit*. Addison-Wesley Publishing Company, Reading, MA, 1994.

[20] A. Brighton. “The Tree Widget” in **Tcl/Tk Tools**. O-Reilly and Associates, Inc. *to appear*, 1997.

[21] R. Bruce Irvin and Barton P. Miller. Multi-Application Support in a Parallel Program Performance Tool. Technical Report #CS-TR-93-1135, February 1993, Computer Sciences Department, University of Wisconsin - Madison.

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

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

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

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

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

7 AUTHOR BIOGRAPHIES

Karen L. Karavanic *is working toward a doctoral degree in the Computer Sciences Department at the University of Wisconsin, Madison. She received BA and MS degrees in computer science from New York University and the University of Wisconsin, respectively. Karavanic holds a NASA Graduate Student Research Fellowship.*

Barton P. Miller *received his B.A degree in Computer Science from the University of California, San Diego in 1977, and M.S. and Ph.D. degrees in Computer Science from the University of California, Berkeley in 1979 and 1984, respectively. Since 1984, he has been at the University of Wisconsin–Madison, where he is a Professor in the Computer Sciences Department. His research interests include parallel and distributed program measurement and debugging, extensible operating systems, network management and naming services, and user interfaces. He currently directs the Paradyn Parallel Performance Tool project, developing next-generation performance tools for both massively parallel computers and workstation clusters. Miller is Program co-Chair of the 1998 ACM/SIGMETRICS Symposium on Parallel and Distributed Tools, and was General Chair of the 1996 ACM/SIGMETRICS Symposium on Parallel and Distributed Tools. He also twice chaired the ACM/ONR Workshop on Parallel and Distributed Debugging, and is on the editorial boards of IEEE Transactions on Parallel and Distributed Systems, Computing Systems and the International Journal of Parallel Processing.*