Scalable Analysis Techniques for Microprocessor Performance Counter Metrics

Dong H. Ahn Jeffrey S. Vetter

Lawrence Livermore National Laboratory
Livermore, CA, USA 94551
{ahn1,vetter3}@llnl.gov

Contemporary microprocessors provide a rich set of integrated performance counters that allow application developers and system architects alike the opportunity to gather important information about workload behaviors. Current techniques for analyzing data produced from these counters use raw counts, ratios, and visualization techniques help users make decisions about their application performance. While these techniques are appropriate for analyzing data from one process, they do not scale easily to new levels demanded by contemporary computing systems. Very simply, this paper addresses these concerns by evaluating several multivariate statistical techniques on these datasets. We find that several techniques, such as statistical clustering, can automatically extract important features from the data. These derived results can, in turn, be fed directly back to an application developer, or used as input to a more comprehensive performance analysis environment, such as a visualization or an expert system.

1 Introduction

Contemporary microprocessors provide a rich set of integrated performance counters that allow application developers and system architects alike the opportunity to gather important information about workload behaviors. These counters can capture instruction, memory, and operating system behaviors. Current techniques for analyzing the data produced from these counters use raw counts, ratios, and visualization techniques to help users make decisions about their application source code.

While these techniques are appropriate for analyzing data from one process, they do not scale easily to new levels demanded by contemporary computing systems. Indeed, the amount of data generated by these experiments is on the order of tens of thousands of data points. Furthermore, if users execute multiple experiments, then we add yet another dimension to this already complex picture. This flood of multidimensional data can swamp efforts to glean important ideas from these counters.

With the trend toward larger systems, users will have no choice but to rely on automated performance analysis tools to sort through these massive data sets, recognize important features, identify parts of the application that are underutilizing the platform, and prescribe possible solutions. Figure 1 shows the major components of such a system. At step ①, the performance instrumentation captures and sorts the data at runtime or offline. Although users can examine the raw data immediately with a visualization tool like VGV [7] or Paraver [4], feature extraction tools and rule-based recommender systems [10, 16] can support the visualization process. For example, at step ⑤, a decision tree algorithm could identify those messages that are performing abnormally [19] and identify them in the visualization with a special glyph or color.

Very simply, this paper addresses these concerns by evaluating several multivariate statistical techniques on these datasets. We find that techniques such as statistical clustering offer promise for automatically extract important features from this performance counter data. These derived results can, in turn, be feed directly back to an

0-7695-1524-X/02 $17.00 (c) 2002 IEEE

application developer, or used as input to a more comprehensive performance analysis environment, such as a visualization [7] or an expert system [10].

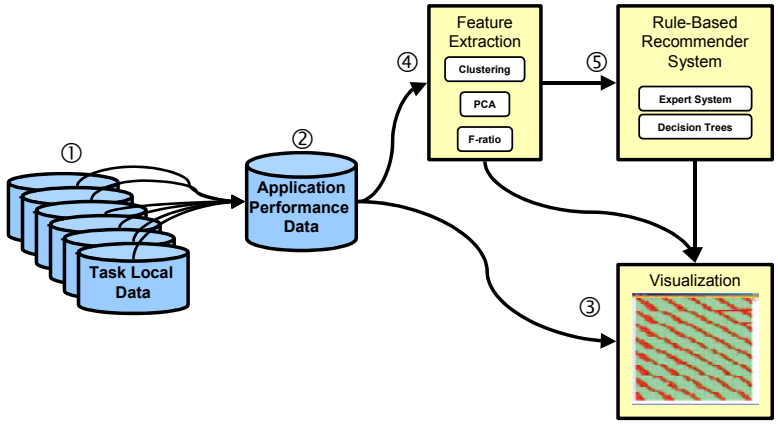

Figure 1: Components of a comprehensive performance analysis environment.

2 Microprocessor Hardware Performance Counters

Modern microprocessors include integrated hardware support for non-intrusive monitoring of a variety of processor and memory system events. Commonly referred to as hardware counters [3, 14], this capability is very useful to both computer architects [2] and applications developers [23]. Detailed software instrumentation can introduce perturbation into an application and the measurement process itself. On the other hand, simulation can become impractical for large, complex applications. These counters fill a gap that lies between detailed microprocessor simulation and software instrumentation because they have relatively low perturbation and can provide insightful information about processor and memory-system behavior [20, 22]. Even though this information is statistical in nature, it does provide a window into certain behaviors that are realistically impossible to harvest otherwise. For instance, on IBM's POWER3 microprocessor, these events include various counts of instructions, cache activity, branch predictions, memory coherence operations, and functional unit utilization.

Several tools and microprocessors have extended this functionality beyond simple event counting. Intel's Itanium processors [9] have features that allow monitoring based on an instruction address range, a specific instruction opcode, a data address range, and/or the privilege level. In addition, the Itanium supplies event address registers that record the instruction and data addresses of data cache misses for loads, the instruction and data addresses of data TLB misses, and the instruction addresses of instruction TLB and cache misses.

As another example, DEC implemented a useful strategy for hardware counters: instruction sampling within the microprocessor. Using this approach, a performance-monitoring tool, such as ProfileMe [5] or DCPI [1], could randomly elect to measure performance characteristics of individual instructions as they flowed through the processor pipeline. The tool could, then, gather this information over the execution of an application and attribute performance problems to certain instructions statistically.

2.1 Counting Hardware Events

Our approach to using hardware counters rests on bracketing targeted code regions with directives that program the counters to capture events of interest, start and stop the counters, and read and store the counter values. Users can insert these directives several ways including by hand, or with a compiler, a binary editor, or dynamic instrumentation. Hardware counters do require the appropriate operating system and library support to accredit counts appropriately to the proper processes and threads.

```
call f_start_section(1,0,ierr)
call hydxy( ddd, ddd1, ithread)
call deltat(" Finished X sweep",2)
call f_end_section(rank, 1,0,ierr)
BARRIER
call flag_clear
BARRIER
call f_start_section(2,0,ierr)
call hydyz( ddd1, ddd, ithread)
call deltat(" Finished Y sweep",2)
call f_end_section(rank, 2,0,ierr)
BARRIER
call flag_clear
```

Table 1: Sample code segment from function runhyd3 of sPPM.

Table 1 shows a code segment from sPPM [17] that has been instrumented with high level library routines written on top of MPX [15] and PAPI [3] in order to capture eight hardware counter values: total processor cycles, total instructions, cycles stalled waiting for memory accesses, floating point divide instructions, L1 cache misses, floating point instructions, load instructions, and store instructions.

As Figure 1 illustrates, every execution of this sequential code segment will generate one instance of counter values for each MPI task. Therefore, applications that execute this code segment millions of times will generate millions of instances of counter values. Table 1 shows the raw counter value table that is generated from the code segment in Figure 1 using two MPI tasks. The *G* column lists the instrumentation identifiers that represent different regions of the code. The *S* column lists instances of these regions. Clearly, in real experiments, this data management problem can become intractable!

G: Instrumentation ID	P: MPI Task	S: Instance	Counter value							
			1	2	3	4	5	6	7	8
1	1	1	8305760504	7795651387	2265349817	14689488	72993923	3744267304	2123235784	1253921843
1	1	2	8233114700	7587713598	2257442295	8987587	72816919	3612932116	2100752913	1260163691
1	2	1	8197360363	7701750765	2233070347	14695959	73425197	3736956914	2075722824	1237231534
1	2	2	8135138668	7593760051	2207456335	9172755	73699055	3590374684	2060311042	1230463969
2	1	1	8326329304	7559198564	2401195595	14583869	72382972	3717326869	2078653081	1233604083
2	1	2	8291791110	7421248463	2334628952	8509892	72074918	3540521698	2060023498	1230670879
2	2	1	8405106757	7645055689	2415396992	14655896	72785214	3708798229	2104739801	1248538508
2	2	2	8381061956	7523753702	2377276028	8606055	72608329	3553288776	2084495857	1256915516

Table 2: Counter values from code segment.

In this situation, the sheer volume of information quickly eclipses useful characteristics of the performance data. Simple questions are difficult to answer: which counters appear to be providing similar information; are the same counters for each task performing similarly; which counters account for most of the variation across all the tasks in the application; which tasks cause this variation?

Certainly, simple statistics, such as the minimum, maximum, and the average help here, but since these statistics apply to only one counter at a time, they reveal neither relationships among multiple counter values nor relationships across multiple instances or tasks.

3 Multivariate Statistical Techniques for Performance Data

As we illustrated in Section 2, each instrumentation point within an application can generate a vast number of hardware counter values. Multiple experiments can aggravate this issue even further. To analyze this data, we turn to multivariate statistical techniques to help focus the user's attention on the important metrics and the distribution of those metrics across parallel tasks.

3.1 Performance Metric Spaces

For further analysis, we model these values as points in a multidimensional space. To make this notion more formal, consider a set of k dynamic performance metrics, hardware counters in our case, measured on a set of P parallel tasks, on a set of g instrumentation regions, and on s samples. Abstractly, one can then view these events as defining a collection of these points that describe parallel system characteristics. Following [21], if R_i denotes the range of metric k, we call the Cartesian product

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

a performance metric space. Thus, the ordered k-tuples

$$(v_1 \in R_1; v_2 \in R_2; ... ; v_k \in R_k) \quad (1)$$

are points in M. It is important to note that this definition of the metric space does not include the dimensions of instrumentation identifier, parallel task identifier, or measurement instance. Furthermore, this model assumes that this higher-dimension data can be down-sampled into this space as appropriate. For instance, we collect all the points for one instrumentation region across all tasks and across all measurements and then project it into this metric space. This situation would generate $k \times P \times s$ points. While this trivial example illustrates our formalization, we expect to use our techniques on much larger systems where $k > 10$, $g > 10$, $P >> 10$, and $s >> 10$.

The goal of our analysis technique is now apparent; we must reduce this massive number of measurement points and the dimensionality of the metric space to a comprehendible scale. Traditional multivariate statistical techniques warrant investigation as vehicles for understanding this data. In fact, projection pursuit [21] and clustering [18] have been applied to understanding real-time performance data; this previous work strongly suggests that such techniques will be useful for managing hardware counter data. These multivariate statistical techniques allow users to draw inferences from observations with multiple variables (dimensions) and they include dimension reduction and classification.

3.2 Data Preparation

Raw data as generated by reading the hardware counters directly can provide useful information; however, in the context of performance analysis, derived metrics are equally important. For example, the raw metric for number of cycles supplies a useful estimate of how long a code region executed; however, the derived metric of number of instructions divided by the number of cycles (IPC or instructions per cycle) can directly emphasize

how well code regions are utilizing system resources. On the other hand, raw metrics are necessary to help gauge the overall importance of code regions per se. For instance, the IPC of a code region that accounts for only minuscule numbers of cycles during the application execution is irrelevant.

3.3 Clustering

Clustering is a rudimentary, exploratory technique that is helpful in understanding the complex nature of multivariate relationships [11, 12]. It provides a familiar means for assessing dimensionality, detecting outliers, and suggesting attractive hypotheses about relationships between the data. Cluster analysis makes no assumptions about the number of clusters or the cluster structure. It relies only on a metric that calculates the similarities or distances between data points. There have been a wide variety of clustering algorithms proposed. Major differences are whether particular methods simply partition data points into a given number of groups or build more complicated cluster (or data point) hierarchies.

In the context of hardware counter data, we propose both hierarchical and non-hierarchical methods will help users identify equivalence classes of data points and an 'important' subset of entire performance metrics that make more contribution to the existence of those classes.

We demonstrate how hierarchical algorithms give users insights about overall cluster structure of a data set by means of dendrogram, while nonhierarchical methods, such as the k-means algorithm, provide an efficient method to explain the importance of each metric on a cluster configuration by using F-ratio of each metric (Section 4.4). F-ratio is a technique for univariate analysis of variance that is defined as $\frac{Between-ClusterVariablity}{Within-ClusterVariablity}$. Hence, metrics that vary greatly among different clusters and remain the same in the same cluster yields higher F-ratio. K-means and F-ratio can also be employed when the decision on number of clusters is not obvious. This situation happens often when users do not have reasonable prior knowledge about target application's behavior. K-mean and F-ratio methods provide a means by which a system can automatically partition data points into a number of clusters as to maximize the between-cluster variability relative to the within-cluster variability.

3.4 Factor Analysis

Factor analysis is a multivariate technique that describes the covariance relationships among many variables in terms of a few underlying quantity, factors. In the context of hardware counter space, we propose it will reduce the dimensionality of our performance metric space, M = [$R_1 \times R_2 \times ... \times R_k$], by assembling highly correlated metrics in a peer group while separating uncorrelated ones into the other groups. (e.g. [R_a, R_b, R_c], x[R_d]x…x [$R_{i,} R_j \; R_k$]). This grouping can guide users to choose a right set of metrics for refining their code optimization efforts.

In the factor analysis model, our metrics space M can be rewritten as

$$R_1 - \upsilon_1 = l_{11}F_1 + l_{12}F_2 + \ldots + l_{1m}Fm + \varepsilon_1$$
$$R_2 - \upsilon_2 = l_{21}F_1 + l_{22}F_2 + \ldots + l_{2m}Fm + \varepsilon_2$$
$$\vdots$$
$$R_p - \upsilon_p = l_{p1}F_1 + l_{p2}F_2 + \ldots + l_{pm}Fm + \varepsilon_p$$

Where F_i is the i^{th} common factor, R_j j^{th} metrics, v_k mean of R_j, and coefficient l_{ji} is the loading of R_j on the factor F_i. As this notation suggests, grouping R's that have higher loadings for a particular F will yield a group whose R's are highly correlated.

3.5 Principal Component Analysis (PCA)

Principal component analysis (PCA) [11] explains the variance-covariance organization of a set of variables using a few linear combinations of these variables. The primary goals of PCA are data reduction and interpretation. Intuitively, PCA attempts to find a subset of the original variables that accounts for almost as much variability as all of the original variables. This subset can then replace the original variables, and thereby achieve a data reduction.

4 Evaluation

We empirically evaluated our techniques with three applications. As Table 1 illustrates, we first instrument the application and collect hardware counter data on the target platform. We then clean, merge, and prepare this data for statistical analysis. Next, we apply several statistical techniques to the prepared data.

4.1 Instrumentation and Data Collection

We manually instrument our target applications with source code annotations. Each instrumentation point identifies a code region to capture hardware counter metrics as Table 1 illustrates. Hence, each application has g instrumented code regions as defined in Section 3.1. For these experiments, we assume that each region captures the same set of k hardware metrics.

In this framework, our tool can either write the each sample to a tracefile during execution or accumulate the samples for each region and write the accumulated metrics to a file at termination. In the former context, tracefiles would grow at a rate proportional to $k \times g \times s$ for each parallel task. We implemented both modes, but still, we use the latter technique, which generates only $k \times g$ measurement points for each parallel task, to prevent an explosion of data and measurement overhead. Our statistical techniques remain valid for accumulated data; however, this selection has the drawback that accumulated measurements can hide certain performance phenomena.

At termination of the application experiment, each parallel task P generates a local file. Our prototype merges these P local files into one global file, containing all accumulated measurements for an application, and having size proportional to $k \times g \times P$. With all these raw metrics for one application now in one file, we can easily apply our statistical techniques to this file with a filter. This filter also manipulates the raw metrics for data cleaning and generating useful derived metrics as described in Section 3.2.

4.2 Platform

We ran our tests on two IBM SP systems, located at Lawrence Livermore National Laboratory. The first machine is composed of sixteen 222 MHz IBM Power3 8-way SMP nodes, totaling 128 CPUs. Each processor has three integer units, two floating-point units, and two load/store units. At the time of our tests, the batch partition had 15 nodes and the operating system was AIX 4.3.3. Each SMP node contains 4GB main memory for a total of 64 GB system memory. A Colony SPSwitch--a proprietary IBM interconnect--connects the nodes.

The second system is composed of 68 IBM RS/6000 NightHawk-2 16-way SMP nodes using 375 MHz IBM 64-bit POWER3-II CPUs. The system has a peak performance rating of 1.6 TeraOps, 1088 GB of global memory, and 20.6 TB of global disk. At the time of our tests, the batch partition had 63 nodes and the operating system was AIX 5.1. A Colony SPSwitch2--a proprietary IBM interconnect--connects the nodes.

4.3 Applications

We evaluate our proposed techniques on two scalable applications. Each application has different computational and communication characteristics [20, 22]. SPPM, for example, has large blocks of floating point computation with infrequent, large messages, while Sweep3D has frequent, small messages with smaller blocks of computation.

sPPM [17] solves a 3-D gas dynamics problem on a uniform Cartesian mesh, using a simplified version of the Piecewise Parabolic Method. The algorithm makes use of a split scheme of X, Y, and Z Lagrangian and remap steps, which are computed as three separate sweeps through the mesh per timestep. Message passing provides updates to ghost cells from neighboring domains three times per timestep. OpenMP provides thread-level parallelism within MPI tasks.

Sweep3D [8, 13] is a solver for the 3-D, time-independent, particle transport equation on an orthogonal mesh and it uses a multidimensional wavefront algorithm for "discrete ordinates" deterministic particle transport simulation. Sweep3D benefits from multiple wavefronts in multiple dimensions, which are partitioned and pipelined on a distributed memory system. The three dimensional space is decomposed onto a two-dimensional orthogonal mesh, where each processor is assigned one columnar domain. Sweep3D exchanges messages between processors as wavefronts propagate diagonally across this 3-D space in eight directions.

UMT is a 3D, deterministic, multigroup, photon transport code for unstructured meshes. The algorithm solves the first-order form of the steady-state Boltzmann transport equation. The equation's energy dependence is modeled using multiple photon energy groups. The angular dependence is modeled using a collocation of discrete directions. The spatial variable is modeled with an upstream corner balance finite volume differencing technique. The solution proceeds by tracking through the mesh in the direction of each ordinate. For each ordinate direction all energy groups are transported, accumulating the desired solution on each zone in the mesh. The code works on unstructured meshes, which it generates at run-time using a two-dimensional unstructured mesh and extruding it in the third dimension a user-specified amount.

4.4 Scatterplot/Correlation Matrix

A scatterplot matrix is a convenient mechanism to display the variance relationships among the multiple dimensions of counter metrics. The scatterplot matrix contains all the pairwise scatter plots of the variables on a single plot in a matrix format. Therefore, if

there are k variables, the scatterplot matrix will have k rows and k columns and the i^{th} row and j^{th} column of this matrix is a plot of variable i versus variable j.

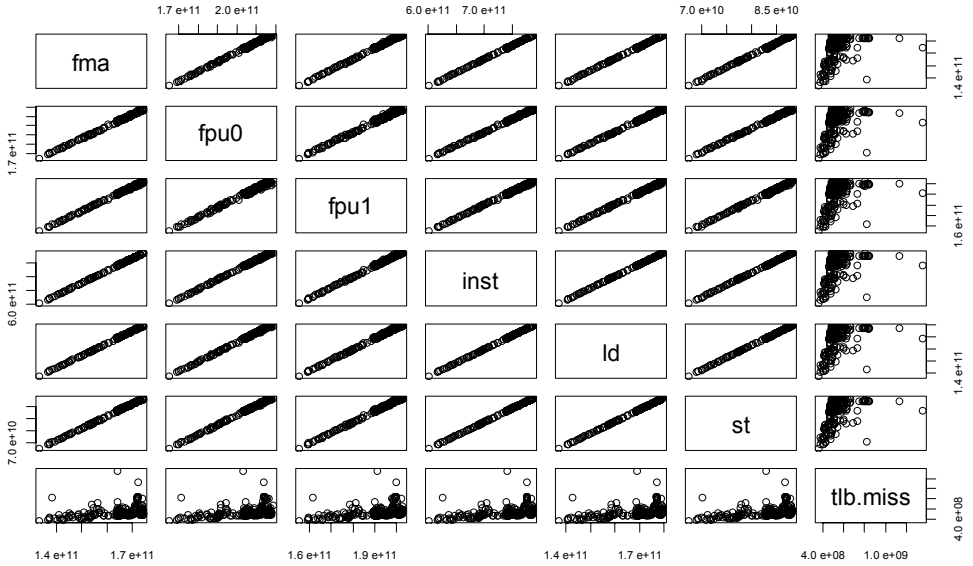

Table 3: Scatterplot matrix for UMT on 288 tasks with raw data from 7 counters.

Consider the presentation in Table 3 for UMT on 288 tasks. This figure quickly illustrates the relationships between each of the seven counter data points for all 288 tasks. Noticeably, six counters (fma, fpu0, fpu1, inst, ld, st) are highly correlated as Table 4 confirms with values greater than 0.99.

	fma	**fpu0**	**fpu1**	**inst**	**ld**	**st**	**tlb.miss**
fma	1.0000000	0.9986839	0.9985139	0.9997846	0.9997262	0.9992478	0.3598753
fpu0	0.9986839	1.0000000	0.9946780	0.9987720	0.9987540	0.9984938	0.3576571
fpu1	0.9985139	0.9946780	1.0000000	0.9984684	0.9984306	0.9980825	0.3621886
inst	0.9997846	0.9987720	0.9984684	1.0000000	0.9999964	0.9998370	0.3612941
ld	0.9997262	0.9987540	0.9984306	0.9999964	1.0000000	0.9998803	0.3614241
st	0.9992478	0.9984938	0.9980825	0.9998370	0.9998803	1.0000000	0.3625394
tlb.miss	0.3598753	0.3576571	0.3621886	0.3612941	0.3614241	0.3625394	1.0000000

Table 4: Correlation matrix for UMT on 288 tasks with raw data from 7 counters.

Using this result, we can quickly prune counters from our set of measurements because we can select one of the six counters (fma, fpu0, fpu1, inst, ld, st) as a representative and use that counter as a predictor for the others. Any task with a higher counter for one of these six implies a higher count for the other five. The tlb.miss counter, on the other hand, is only slightly positively correlated with these other six counters.

Naturally, this result assists user in determining which counters are counting redundant information. Since many microprocessors have only a limited number of counters on which to count many events, users must choose events to count wisely. This straightforward analysis helps with this decision.

4.5 Clustering

4.5.1 Agglomerative Hierarchical Method (AHM)

This method gives users insights about overall cluster structure that exist in a data space by constructing dendrograms. Figure 2 shows the dendrogram for one instrumented section of an sPPM experiment with 8 MPI tasks and 8 OpenMP threads per task. Since sPPM exploits parallelism with message passing for inter-node communication and OpenMP within shared memory for thread level parallelism, it is expected to have at least two natural clusters when using the raw counter data. AHM clearly identifies in Figure 2 the existence of two classes; one housing all 56 slave threads and the other cluster containing the 8 master threads.

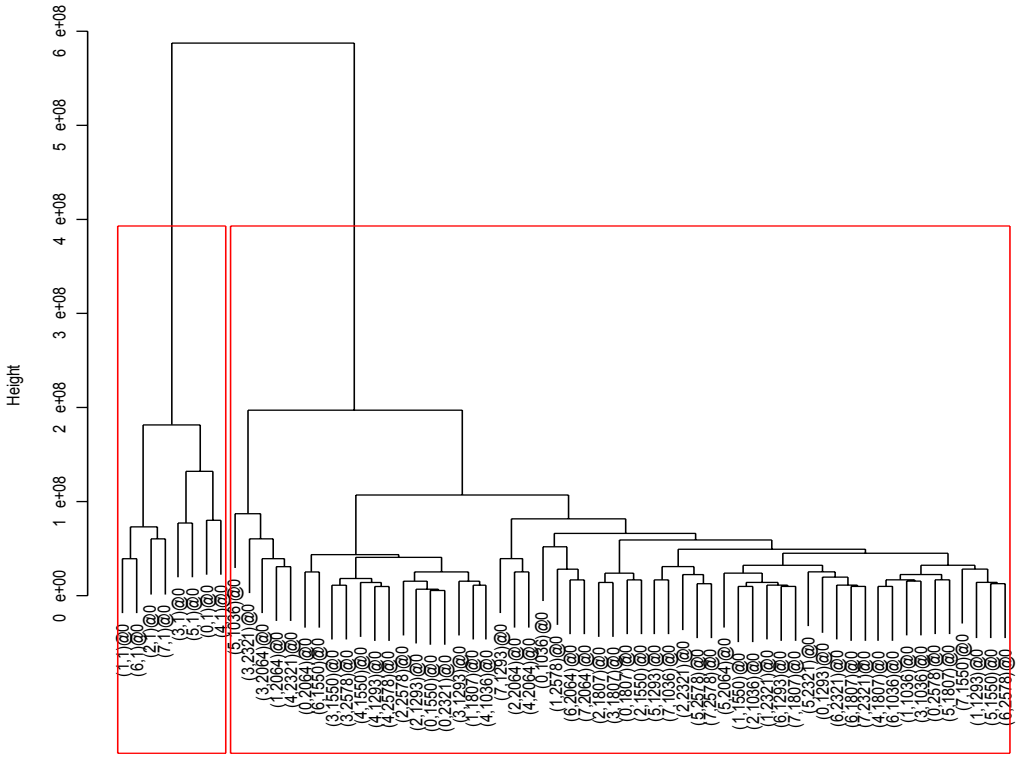

Figure 2: Dendrogram for a section of sPPM using raw metrics.

Figure 3 illustrates the dendrogram of the same section of sPPM using derived metrics. As expected, the configuration does not change much from Figure 2, suggesting that the hardware counter performance counters show that these tasks are performing similarly and any changes to code for either the master thread or the slave thread will propagate to its peers. Statistical techniques with raw metrics alone would not provide this perspective immediately. That is, the optimizations to one of the representatives in this group will most likely propagate to its peers in the same cluster.

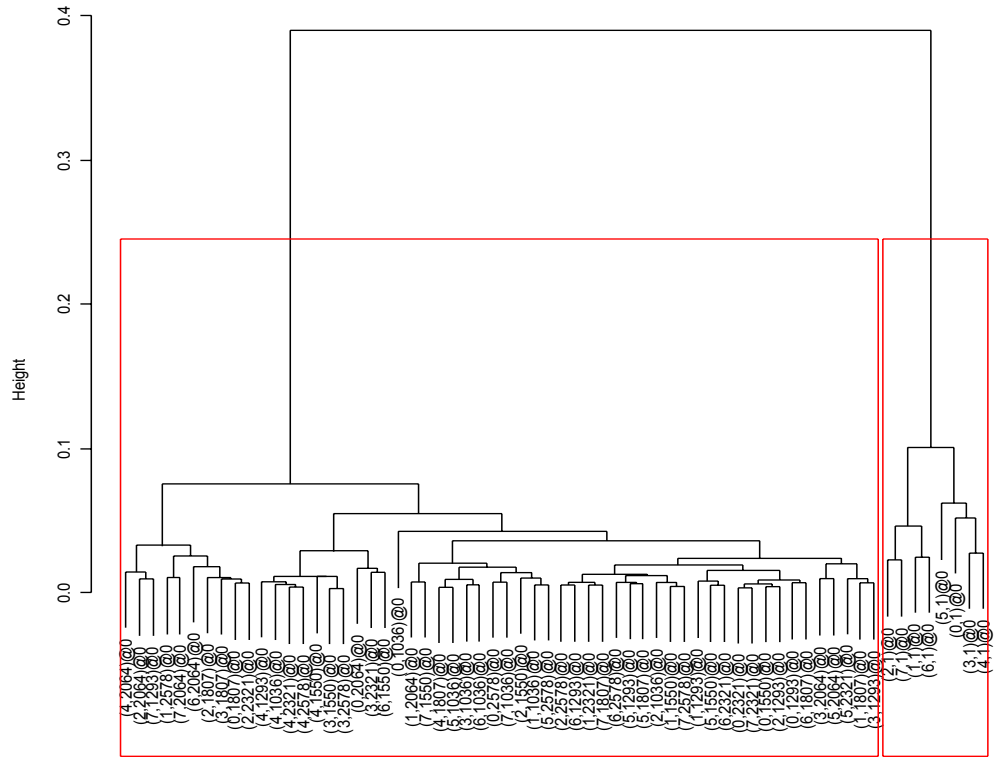

Figure 3: Dendrogram for a section of sPPM using derived metrics.

Figure 5 shows the dendrogram for the raw metrics of a section of Sweep3d for an experiment using 64 MPI tasks. In this case, three clusters are sufficiently different. Our initial results show that because Sweep3d decomposes the global 3-D problem onto a 2-D orthogonal mesh for processor assignment, the assignment creates two different equivalence classes: one for internal processors (right), and one for corner and edge processors (center and left).

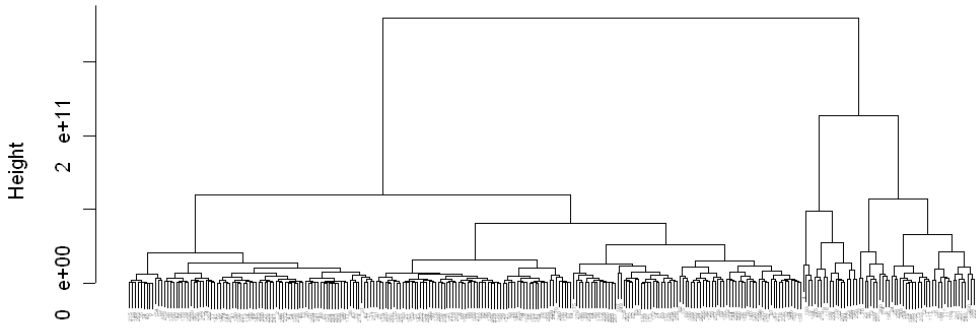

Figure 4: UMT hierarchical clustering results on raw metrics (task numbers elided).

Although there are slight differences in the amount of computation performed on each task for sPPM and Sweep3D, UMT is an unstructured grid application that partitions and assigns work to processors based on this partitioning. It is very likely that all of the tasks will have a slightly different amount of work and it illustrates the advantages of these techniques on both raw and derived metrics. Our results on raw data for UMT in Figure 4 show that the clusters are separated by greater distances when compared to

similar figures for sPPM and Sweep3d. The derived metrics, which included instructions per cycle, L1 and L2 cache miss ratios, and TLB miss ratios, yielded clusters that while distinct were very close as they were for sPPM derived metrics, even though the workload distribution for UMT is quite varied when compared to sPPM's regular workload distribution.

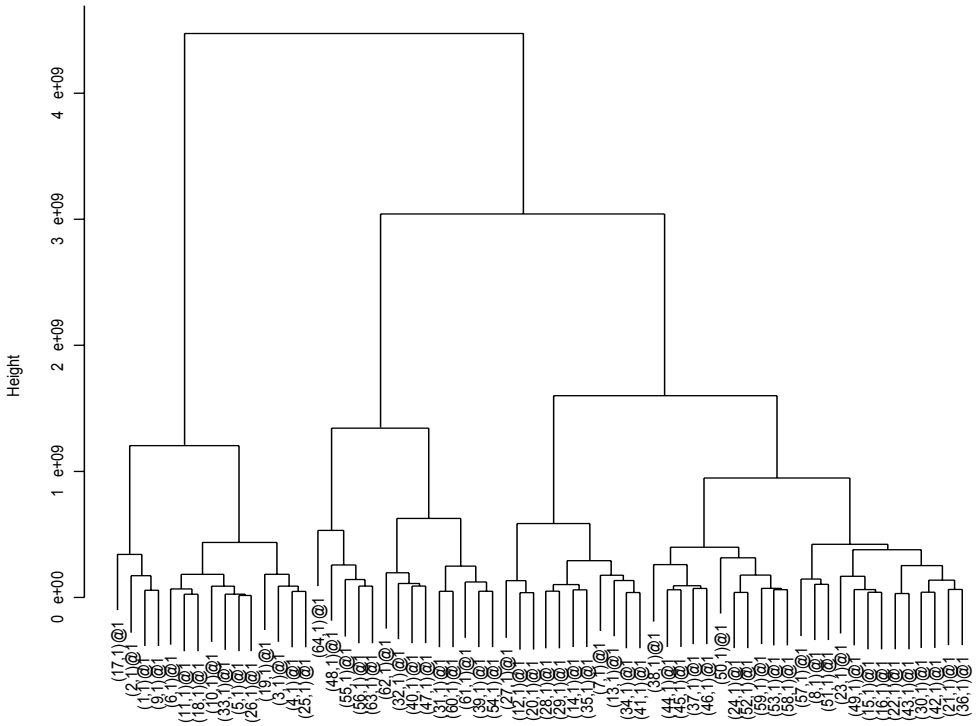

Figure 5: Dendrogram for sweep section of sweep3d using raw metrics.

4.5.2 k-means clustering and F-ratio

While AHM gives a general idea about cluster structure, it is not entirely convenient to compare clusters and compute the importance of an individual metrics that yield the particular cluster configuration. Using k-means clustering and F-ratios, we ordered metrics for the same section on sPPM by their F-ratio in Table 5. It suggests that PAPI_MEM_SCY (Cycles Stalled Waiting for Memory Access), PAPI_SR_INS (Store instructions executed), and PAPI_L1_TCM (L1 total cache misses), are the three major reasons that we have two distinct clusters.

Metrics	F-ratio
PAPI.MEM.SCY	**7.914255**
PAPI.SR.INS	**5.124050**
PAPI.L1.TCM	**5.097693**
PAPI.TOT.IIS	3.488066
PAPI.LD.INS	3.473341
PAPI.FP.INS	2.498332
PAPI.TOT.CYC	1.673567
PAPI.FDV.INS	0.161107

Table 5: Metrics ordered by F-ratio size for a section of sPPM.

On further investigation of this F-ratio result, we found that the hybrid MPI/OpenMP version of sPPM had significant differences between the master and worker threads. Since the master thread must manage the remaining worker threads, it incurs more copying of data and synchronization as evidenced by our F-ratio and k-means clustering results.

4.6 Factor Analysis

Table 6 shows the result of factor analysis on the same section of sPPM. Each column represents loadings of metrics for each factor. As it suggests, it is reasonable to group together those metrics with larger loadings per column.

Metric	Factor1	Factor2	Factor3
PAPI.LD.INS	**-0.658**	**0.519**	**0.541**
PAPI.BR.PRC	0.000	0.000	**0.792**
PAPI.L1.DCM	**0.969**	0.13	-0.161
PAPI.FXU.IDL	-0.244	**0.883**	-0.290
PAPI.BTAC.M	**0.799**	0.000	0.324
PAPI.FP.INS	**-0.974**	0.199	0.000
PAPI.MEM.WCY	**0.957**	0.000	0.000
PAPI.SR.INS	0.000	**0.690**	0.000

Table 6: Factor Analysis of sPPM (with three factors assumed).

By setting a somewhat arbitrary threshold, 0.5, we are able to make four groups: {PAPI.L1.DCM, PAPI.BTAC.M, PAPI.FP.INS, PAPI.MEM.WCY}, {PAPI.FXU.IDL, PAPI.SR.INS}, {PAPI.BR.PRC} and PAPI.LD.INS. We verified the metrics in the same group are highly correlated by scanning correlation matrix. Users can now select a representative metric per group for the further analysis.

4.7 PCA

We primarily use PCA as an input filter to clustering. That is, we first run PCA on our multidimensional counter data to extract those dimensions that contribute most to variability. Then, we invoke hierarchical clustering on the resulting dimensions.

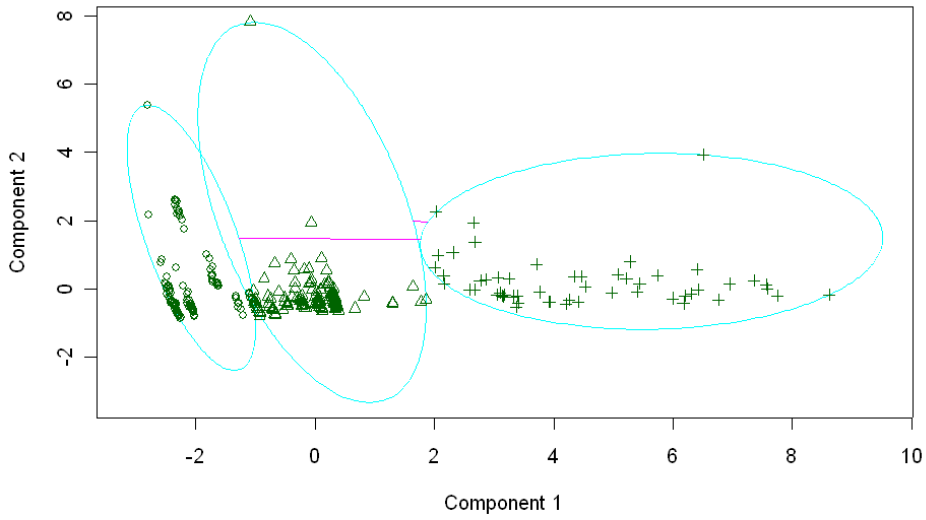

Figure 6: Hierarchical clustering results after PCA on UMT raw metrics at 288 tasks.

Figure 6 illustrates the effectiveness of hierarchical cluster after PCA on the raw counter data from UMT at 288 tasks. Component 1 is loaded to the inst metric while Component 2 is loaded to the tlb.miss metric. These two components capture 99.91% of the variability in this dataset. The benefit of using PCA before clustering is now clear. By using PCA to select metrics with the maximum variability, clustering can easily group tasks that are performing similarly.

5 Observations

Our experiments revealed several important points. First, most of the multivariate statistical techniques that we evaluated helped us answer some question that would be a burdensome task otherwise. Clustering, for example, improves the ability to identify which tasks in the application have similar performance counter metrics. The F-ratio test, discovers which metrics vary across tasks and why they form separate clusters. Second, these techniques are a means to an end. The output of these statistical methods is quite valuable, but they also require additional interpretation and integration with other methods, such as rule-based systems, to actually prescribe performance optimizations to the user. Third, raw performance counter data supplies information on load balance and correlation across metrics while derived performance data helps to identify regions of code that are performing in the same way. Fourth, although most of our applications and experiment platforms are similar, resulting in well-behaved performance and workloads, heterogeneous platforms or grid environments [6] offer new challenges in understanding performance data.

6 Conclusions

Scalable computing platforms generate tremendous volumes of performance data, especially when monitoring low-level, frequent events like those produced by microprocessor performance counters. Developers need new techniques to help them gain insight into these massive datasets. Traditional multivariate statistical techniques can play a prominent role in this effort by reducing the dataset dimensionality and classifying similar data points. Our experiments on several applications demonstrate the feasibility of this approach and highlight several useful implementation strategies. For example, our experiments with sPPM, Sweep3d, and UMT clearly confirmed that clustering on both raw and derived metrics can allow a user to understand the performance implications across all tasks in an application. Factor analysis is another technique that helps to correlate hardware counter data that appears closely correlated.

We are beginning to use these results from statistical analysis techniques in our environment to drive more advanced performance analysis systems as motivated in Section 1.

Acknowledgments

This work was performed under the auspices of the U.S. Department of Energy by the University of California, Lawrence Livermore National Laboratory under contract No. W-7405-Eng-48. This paper is available as LLNL Technical Report UCRL-JC-148058.

Appendix A: PAPI Power3 Hardware Events

Name	Description
PAPI_L1_DCM	Level 1 data cache misses (PM_LD_MISS_L1,PM_ST_L1MISS)
PAPI_L1_ICM	Level 1 instruction cache misses (PM_IC_MISS)
PAPI_L1_TCM	Level 1 cache misses (PM_IC_MISS,PM_LD_MISS_L1,PM_ST_L1MISS)
PAPI_CA_SNP	Requests for a snoop (PM_SNOOP)
PAPI_CA_SHR	Requests for exclusive access to shared cache line (PM_SNOOP_E_TO_S)
PAPI_CA_ITV	Requests for cache line intervention (PM_SNOOP_PUSH_INT)
PAPI_BRU_IDL	Cycles branch units are idle (PM_BRU_IDLE)
PAPI_FXU_IDL	Cycles integer units are idle (PM_FXU_IDLE)
PAPI_FPU_IDL	Cycles floating point units are idle (PM_FPU_IDLE)
PAPI_LSU_IDL	Cycles load/store units are idle (PM_LSU_IDLE)
PAPI_TLB_TL	Total translation lookaside buffer misses (PM_TLB_MISS)
PAPI_L1_LDM	Level 1 load misses (PM_LD_MISS_L1)
PAPI_L1_STM	Level 1 store misses (PM_ST_L1MISS)
PAPI_L2_LDM	Level 2 load misses (PM_LD_MISS_EXCEED_L2)
PAPI_L2_STM	Level 2 store misses (PM_ST_MISS_EXCEED_L2)
PAPI_BTAC_M	Branch target address cache misses (PM_BTAC_MISS)
PAPI_PRF_DM	Data prefetch cache misses (PM_PREF_MATCH_DEM_MISS)
PAPI_TLB_SD	Translation lookaside buffer shootdowns (PM_TLBSYNC_RERUN)
PAPI_CSR_FAL	Failed store conditional instructions (PM_RESRV_CMPL)
PAPI_CSR_SUC	Successful store conditional instructions (PM_ST_COND_FAIL)
PAPI_CSR_TOT	Total store conditional instructions (PM_RESRV_RQ)
PAPI_MEM_SCY	Cycles Stalled Waiting for memory accesses (PM_CMPLU_WT_LD,PM_CMPLU_WT_ST)
PAPI_MEM_RCY	Cycles Stalled Waiting for memory Reads (PM_CMPLU_WT_LD)
PAPI_MEM_WCY	Cycles Stalled Waiting for memory writes (PM_CMPLU_WT_ST)
PAPI_STL_ICY	Cycles with no instruction issue (PM_0INST_DISP)
PAPI_STL_CCY	Cycles with no instructions completed (PM_0INST_CMPL)
PAPI_BR_CN	Conditional branch instructions (PM_CBR_DISP)
PAPI_BR_MSP	Conditional branch instructions mispredicted (PM_MPRED_BR_CAUSED_GC)
PAPI_BR_PRC	Conditional branch instructions correctly predicted (PM_BR_PRED)
PAPI_FMA_INS	FMA instructions completed (PM_EXEC_FMA)
PAPI_TOT_IIS	Instructions issued (PM_INST_DISP)
PAPI_TOT_INS	Instructions completed (PM_INST_CMPL)
PAPI_INT_INS	Integer instructions (PM_FXU0_PROD_RESULT,PM_FXU1_PROD_RESULT,PM_FXU2_PROD_RESULT)
PAPI_FP_INS	Floating point instructions (PM_FPU0_CMPL,PM_FPU1_CMPL)
PAPI_LD_INS	Load instructions (PM_LD_CMPL)
PAPI_SR_INS	Store instructions (PM_ST_CMPL)
PAPI_BR_INS	Branch instructions (PM_BR_CMPL)
PAPI_FLOPS	Floating point instructions per second (PM_CYC,PM_FPU0_CMPL,PM_FPU1_CMPL)
PAPI_TOT_CYC	Total cycles (PM_CYC)
PAPI_IPS	Instructions per second (PM_CYC,PM_INST_CMPL)
PAPI_LST_INS	Load/store instructions completed (PM_LD_CMPL,PM_ST_CMPL)
PAPI_SYC_INS	Synchronization instructions completed (PM_SYNC)
PAPI_FDV_INS	Floating point divide instructions (PM_FPU_FDIV)
PAPI_FSQ_INS	Floating point square root instructions (PM_FPU_FSQRT)

References

[1] J.M. Anderson, L.M. Berc, J. Dean, S. Ghemawat, M.R. Henzinger, S.-T.A. Leung, R.L. Sites, M.T. Vandevoorde, C.A. Waldspurger, and W.E. Weihl, "Continuous profiling: where have all the cycles gone?," *ACM Trans. Computer Systems*, 15(4):357-90, 1997.

[2] P. Bose and T.M. Conte, "Performance analysis and its impact on design," *Computer*, 31(5):41-9, 1998.

[3] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci, "A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters," Proc. SC2000: High Performance Networking and Computing Conf. (electronic publication), 2000.

[4] J. Caubet, J. Gimenez, J. Labarta, L. DeRose, and J.S. Vetter, "A Dynamic Tracing Mechanism for Performance Analysis of OpenMP Applications," Proc. Workshop on OpenMP Applications and Tools (WOMPAT), 2001.

[5] J. Dean, J.E. Hicks, C.A. Waldspurger, W.E. Weihl, and G. Chrysos, "ProfileMe: hardware support for instruction-level profiling on out-of-order processors," Proc. Thirtieth Annual IEEE/ACM International Symposium on Microarchitecture (MICRO), 1997, pp. 292-302.

[6] I. Foster and C. Kesselman, Eds., *The Grid: blueprint for a new computing infrastructure*. San Francisco: Morgan Kaufmann Publishers, 1999, pp. xxiv, 677.

[7] J. Hoeflinger, B. Kuhn, P. Petersen, R. Hrabri, S. Shah, J.S. Vetter, M. Voss, and R. Woo, "An Integrated Performance Visualizer for OpenMP/MPI Programs," Proc. Workshop on OpenMP Applications and Tools (WOMPAT), 2001.

[8] A. Hoisie, O. Lubeck, H. Wasserman, F. Petrini, and H. Alme, "A General Predictive Performance Model for Wavefront Algorithms on Clusters of SMPs," Proc. ICPP 2000, 2000.

[9] Intel, "Intel IA-64 Architecture Software Developer's Manual, Volume 4: Itanium Processor Programmer's Guide," Intel 2000.

[10] Intel, *VTune Performance Analyzer*, http://www.intel.com/software/products/vtune, 2002.

[11] R.A. Johnson and D.W. Wichern, *Applied Multivariate Statistical Analysis*, 4 ed. Englewood Cliffs, New Jersey, USA: Prentice-Hall, 1998.

[12] L. Kaufman and P.J. Rousseeuw, *Finding groups in data: an introduction to cluster analysis*. New York: Wiley, 1990.

[13] K.R. Koch, R.S. Baker, and R.E. Alcouffe, "Solution of the First-Order Form of the 3-D Discrete Ordinates Equation on a Massively Parallel Processor," *Trans. Amer. Nuc. Soc.*, 65(198), 1992.

[14] K. London, J. Dongarra, S. Moore, P. Mucci, K. Seymour, and T. Spencer, "End-user Tools for Application Performance Analysis Using Hardware Counters," Proc. International Conference on Parallel and Distributed Computing Systems, 2001.

[15] J.M. May, "MPX: Software for Multiplexing Hardware Performance Counters in Multithreaded Programs," Proc. International Parallel and Distributed Processing Symposium (IPDPS) (electronic publication), 2001.

[16] 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, 1995.

[17] A.A. Mirin, R.H. Cohen, B.C. Curtis, W.P. Dannevik, A.M. Dimits, M.A. Duchaineau, D.E. Eliason, D.R. Schikore, S.E. Anderson, D.H. Porter, P.R.

Woodward, L.J. Shieh, and S.W. White, “Very High Resolution Simulation of Compressible Turbulence on the IBM-SP System,” Proc. SC99: High Performance Networking and Computing Conf. (electronic publication), 1999.

[18] D.A. Reed, O.Y. Nickolayev, and P.C. Roth, “Real-Time Statistical Clustering and for Event Trace Reduction,” *J. Supercomputing Applications and High-Performance Computing*, 11(2):144-59, 1997.

[19] J.S. Vetter, “Performance Analysis of Distributed Applications using Automatic Classification of Communication Inefficiencies,” Proc. ACM Int'l Conf. Supercomputing (ICS), 2000, pp. 245 - 54.

[20] J.S. Vetter and F. Mueller, “Communication Characteristics of Large-Scale Scientific Applications for Contemporary Cluster Architectures,” Proc. International Parallel and Distributed Processing Symposium (IPDPS), 2002.

[21] J.S. Vetter and D. Reed, “Managing Performance Analysis with Dynamic Statistical Projection Pursuit,” Proc. SC99: High Performance Networking and Computing Conf. (electronic publication), 1999.

[22] J.S. Vetter and A. Yoo, “An Empirical Performance Evaluation of Scalable Scientific Applications,” Proc. SC 2002, 2002.

[23] M. Zagha, B. Larson, S. Turner, and M. Itzkowitz, “Performance Analysis Using the MIPS R10000 Performance Counters,” Proc. Supercomputing (electronic publication), 1996.