Detecting Runtime Environment Interference with Parallel Application Behavior

Rashawn L. Knapp[1], Karen L. Karavanic[1], and Douglas M. Pase[2]

[1]Portland State University
Department of Computer Science
P.O. Box 751
Portland, OR 97207-0751
{knappr, karavan}@cs.pdx.edu

[2]IBM System x HPC Portfolio Development
P.O. Box 12195, Dept. E1WA/205/EE154
3039 Cornwallis Rd.
Research Triangle Park, NC 27709-2195
pase@us.ibm.com

Abstract

Many performance problems observed in high end systems are actually caused by the runtime system and not the application code. Detecting these cases will require parallel performance tools to incorporate information about the runtime system; however many current tools do not. We present a test suite for evaluating the ability of performance tools to reach a correct diagnosis in cases where a problem is caused by the runtime environment. We include a set of results for one of the tests, which measures application performance as NFS server load is increased. We also present a model for performance diagnosis that combines system and application level information.

1. Introduction

Developing high end applications requires solving a wide range of challenges: hardware issues of power and speed; shortcomings of commodity operating systems; developing libraries that scale up to computation and communication across hundreds or thousands of processors; and of course, development of algorithms to simulate complex real world systems. Many of today's high performance systems use a combination of shared memory and distributed memory, in which each node comprises several processors and a shared memory, and a collection of nodes are interconnected via a communication network. Only applications that are considered important enough for such resources, and complex enough to need them, are developed for these platforms, and this drives the need for manageable and accurate performance analysis of applications as they are developed, improved, ported, and scaled. There are notable difficulties for conducting performance analysis in high end computing. There are challenges due to factors of scale, including system size, application size, and the large volumes of analysis data. Performance analysis often involves the use of several tools and many manual steps, both of which add delay to the performance analysis process. Even for a simple application, it would be extremely difficult to conduct a performance analysis study using only manual methods. At most, one might be able to gather wall-clock timing information; and this would be at the granularity of the stop watch used, a granularity much coarser than that of the computing system. Therefore, fast and accurate analysis requires automated or semi-automated performance tools.

Many causes of poor application performance are actually found in the other layers of the runtime system: hardware, the network/interconnect, the operating system, or third-party library code. For example, Tsafrir characterizes interference from the systems software stack as "an increasingly important factor in parallel cluster applications."[1] Hensbergen states "The impact of this so-called 'OS noise' creates problems synchronizing barriers across large clusters and creates efficiency problems along with low-utilization of system resources."[2] However, the tools used to diagnose parallel application performance do not include analysis about the status and behavior of the runtime environment. They are only able to report the causes of poor performance in terms of the application. This makes performance problems related to the runtime system much more difficult to detect, and greatly increases the time to reach a correct diagnosis.

One category of possible interference scenarios is poor or unexpected system software behavior. An example comes from a recent study of trace tool performance [3]. Measurements were collected for executions of SMG2000,

1-4244-0910-1/07/$20.00 ©2007 IEEE

an ASC Purple benchmark, on MCR, a Linux cluster at Lawrence Livermore National Laboratory (LLNL) running the CHAOS operating system and the Lustre parallel file system. The researchers were surprised to discover that execution times for experiments with large trace buffers were greater than execution times for experiments with small trace buffers. After several weeks of testing, the researchers obtained unexpected results from an experiment set: the large buffer executions performed better than the small buffer executions, as originally expected. All experiment data had been stored with PerfTrack [4], an experiment management system that automatically collects information about execution environments. The PerfTrack data showed that the operating system and file system had been upgraded. The researchers concluded that the file system was the likely cause of the initial poor performance.

The research literature includes several examples of problems caused by a mismatch between normal operating system behavior and the common structure of parallel applications [5, 6, 7, 8]. One illustrative example involves ASCI Q, a 2,048-node high performance computer at Los Alamos National Laboratory (LANL) [6]. At the time of ASCI Q's initial deployment, an analytical model was used to predict the application's execution time using the data from previous runs on other platforms and hardware measurement data. The first runs of the application on the full system performed much worse than predicted; at 4096 processors, the time to completion of one application cycle was twice the predicted value. The model was accurate if one to three processors per node were utilized, and quite inaccurate for runs using all four processors. Each cycle in the application was configured to perform a consistent amount of work, and so the expectation was that each executed cycle would complete in the same amount of time; however, the results showed high variance. Synthetic benchmarks showed that in each 32-node cluster of ASCI Q, nodes 0, 1, and 31 showed consistently longer execution times than the other nodes. Ultimately, it was determined that system activities were causing application processes to be switched out on some nodes and not others, causing those nodes to lag behind due to context switch overhead.

Our goal is to develop diagnostic techniques for automated performance tools that incorporate measurements and knowledge of the runtime environment. This paper describes two initial steps towards this goal: development of a test suite, and a model for automated diagnosis. We have designed a test suite for characterizing the ability of existing performance tools to detect performance problems rooted in the runtime system; we include a full set of experimental results for one test from this suite. Then we describe a model for combining measurable metrics to reach accurate diagnoses related to runtime system interference.

2. Related Work

We are not aware of any existing techniques for automated diagnosis of parallel applications that take into account problems originating outside of the application.

A number of tools support collecting measurement data related to both application performance and system related metrics. Often this is done by the inclusion of hardware counter data in traces (Vampir [9], SCALEA [10], PerfSuite [11], and TAU [12]) or by user-defined metrics (IPS-2 [13] and Paradyn [14]). Several tools integrate system metrics more directly. SCALEA-G [15] is a system monitoring and performance analysis package designed for Grid applications that provides integrated visualizations of application and system level analysis results. Active Harmony [16] performs automatic runtime adaptation of applications and workloads in parallel and distributed computing environments. Active Harmony considers the application and the resources utilized by the application as tunable parameters of the execution environment. Continuous Program Optimization (CPO) combined with vertical profiling [17, 18] involves cyclic phases of monitoring and optimization. In the monitoring phase, data collected across system layers is analyzed. In the optimization phase, the analysis is used to improve application performance by adapting an application to its execution environment and/or changing aspects of execution environment. The monitoring facility is presented as a vertical set of layers; this is similar to the notion of layers presented in this work. The CPO research, to date, focuses on dual processor workstations and does not extend to high end systems.

Paradyn [14, 19] conducts online performance analysis of parallel applications. The search strategy for identifying performance bottlenecks, as implemented by the Paradyn Performance Consultant, incorporates a top-down approach of testing hypotheses. If a high-level hypothesis evaluates to true then subsequent hypotheses along that branch of inquiry will be tested; and, conversely, while a hypothesis is not true, lower-level hypotheses are not tested. In this manner, the search forms a directed acyclic graph, where a child node is only investigated if its parent hypothesis has evaluated to true. Since evaluation always causes additional instrumentation of the running application, this method bounds instrumentation overhead and perturbation by substantially reducing the search space. Paradyn may run into an instrumentation cost limit before the bottleneck search has been sufficiently refined. When this predetermined limit is reached, further downward searching is halted, presenting diagnostic results which may be too general for correctly identifying the cause of a performance problem. In our model for determining a

correct diagnosis, we propose a bottom-up approach where low-level tests trigger the testing of more complex diagnoses. We also incorporate a flexible and dynamic scheme for assigning priorities to diagnoses that allows us to specify the order of evaluation for diagnoses.

3. A Test Suite for Performance Diagnosis Tools

To survey current tool capabilities, we have developed a test suite for parallel performance tools. Our specific goal was to measure each tool's ability to determine causes of poor application performance. The framework for our test suite includes a set of four layers encompassing all aspects of a running application: application, library, operating system, and physical. Each test simulates or creates a condition existing in a particular layer that will negatively affect application performance. The results are interpreted by comparing the diagnosis reported by the tool to the real diagnosis. To fully pass a test, a tool must correctly identify the root cause layer and correctly identify at least one condition within the layer responsible for the performance problem. In the remainder of this section we briefly describe the individual tests.

The Naive Broadcast Test. The Naive Broadcast Test is designed to simulate performance problems originating in the application. This test is included in the suite to check a tool's ability to distinguish application level and system level problems. For these kinds of problems the most reasonable approach to optimize the application performance is by directly modifying the application source code. The test uses an MPI application in which one sending process uses a point to point communications operation to send the same message content to every other participating MPI process, rather than using the collective operation `MPI_Bcast`. To pass this test, a tool must identify the application layer as the layer containing the root cause and identify that the `MPI_Send` could potentially be replaced by `MPI_Bcast`.

The MPI Library Bottleneck Test. The MPI Library Bottleneck test is designed to simulate performance problems originating in the Library Layer. This test involves running an MPI application that uses the `MPI_Allreduce` operation, linked to a modified profiling library with extra wait time inserted before the call to the `MPI_Allreduce` code. To pass this test, the tool must identify the library layer as the layer containing the root cause and identify that the `MPI_Allreduce` function is a costly function.

The NFS Server Test. The NFS Server Test is designed to simulate performance problems in the Operating Systems Layer. This experiment creates a scenario where an NFS server receives a high number of requests for the same file from various clients. In this test one machine that is separate from the application's cluster is designated as the NFS server. An MPI application is configured to run on one node in the cluster, and it does a series of file reads on a single file. Two other processes on separate nodes perform repeated writes to the same file that the MPI application is trying to read. To fully pass this test, the tool must identify the operating system layer as the layer containing the root cause and identify that the problem is file contention.

The Memory Difference Test. The Memory Difference Test is designed to simulate performance problems originating in the physical layer. This experiment requires a physical environment where one node in a cluster has less memory than the other nodes, or a simulation of such an environment via software. Other than the memory difference, the nodes are homogeneous. An MPI application is run on a set of nodes that includes the node with less memory, with one process assigned to each node. The application is constructed so that every MPI process does the same set of memory intensive tasks. The process assigned to the node with less memory takes longer to complete its work than the other processes. To pass this test, the tool must identify the physical layer as the layer containing the root cause and identify performance is constrained by the size of memory available on one of the participating nodes.

The Interconnect Test. The Interconnect Test is designed to simulate performance problems originating in the Physical, Library, or Operating System Layer. The experiment measures the performance of a communication intensive MPI application when available bandwidth is far below the theoretical capacity. In this test a very large message is passed back and forth between two processes running on separate nodes, using the blocking `MPI_Send` and `MPI_Recv` communication methods. Available network bandwidth for the participating processes is reduced by generating extra traffic on the links between the two nodes. To pass this test, the tool must identify the interconnect bandwidth as the root cause, and point to the particular cause for the bandwidth reduction.

4. Experimental Results

In this section we show results for the NFS Server Test from the test suite. For this experiment we constructed two environments: a normal environment, representing a baseline environment for application performance; and a suboptimal environment, in which some aspect of the normal environment was purposefully changed to affect application performance. Our goal was to determine the extent to which application based methods could correctly diagnose the root cause of the observed performance degradation in the suboptimal environment. We analyzed application performance using application level timing

routines, MPI profiling, and event tracing. The profiles and traces were collected with TAU [20], and we used ParaProf [20] and Vampir to visualize the profile and trace data.

We designed an MPI application in which the master, or rank zero process, reads a file and stores a work value based on the file contents. The time spent by the master in this I/O stage is measured using `MPI_Wtime`. The master assigns work to each of the participating MPI processes through `MPI_Bcast`. As a collective operation, all processes call this function: the master sends the work value to all worker processes, and each worker process, before returning from the function call, receives a work value. In the final phase, the application's work is performed by each of the participating processes. Each process knows how much work to perform based on the received work value. In addition to timing the I/O phase of rank zero, the execution time for all processes is measured using `MPI_Wtime`. These simple timing constructs were coded into the MPI application, and the same executable was used for all executions in both environments. LAM/MPI 7.0.6 [21] was the implementation of MPI used in this experiment.

All experiments were conducted on an eight node Linux cluster (See Figure 1). Each node was running kernel version 2.6.15-26-686 and was configured with a 3.2 GHz Intel Pentium 4 hyper-threaded CPU, 1 GB of memory, and one network interface card capable of 100 Mbps transmission. Each node was directly connected to a central switch. We designated one of the machines as an NFS server, four machines for the MPI application, and the remaining three for unrelated processes in the suboptimal environment. The MPI application was always executed with four processes on the same four machines, and the master process was always launched on the same machine. In the suboptimal environment processes issued frequent update requests to the NFS server for the file required by the MPI application.

We captured a baseline for the normal environment by executing 30 trials of the MPI application under three settings: without additional instrumentation, with profiling instrumentation, and with tracing instrumentation. In each setting, there was very little variation in execution wall-clock time among the trials, and there was negligible variation among the processes in a single trial. Additionally, rank zero's I/O times were consistent across the execution set. We collected similar measurements for the application in the suboptimal environment, under the three settings described above. In the suboptimal environment there was noticeable variation in total execution times, such that the total execution time for some executions was much greater than what was observed for the well-behaved executions. There was also variation in rank zero's I/O time. In Table 1 we summarize this data for the master process in the normal and suboptimal environments. The presented measurements are reported in seconds and were obtained from the execution sets that did not include additional instrumentation. We see large differences between the normal and suboptimal environments for the maximum, average, and variance values for wall-clock time and I/O time. Although these simple timing constructs identify the existence of a performance problem in the suboptimal environment, they are not sufficient for determining the root cause.

We analyzed the TAU profiling data in the two environments. Figure 2 contains two ParaProf screenshots showing the dominating functions in each environment. In the normal environment main is the dominating function

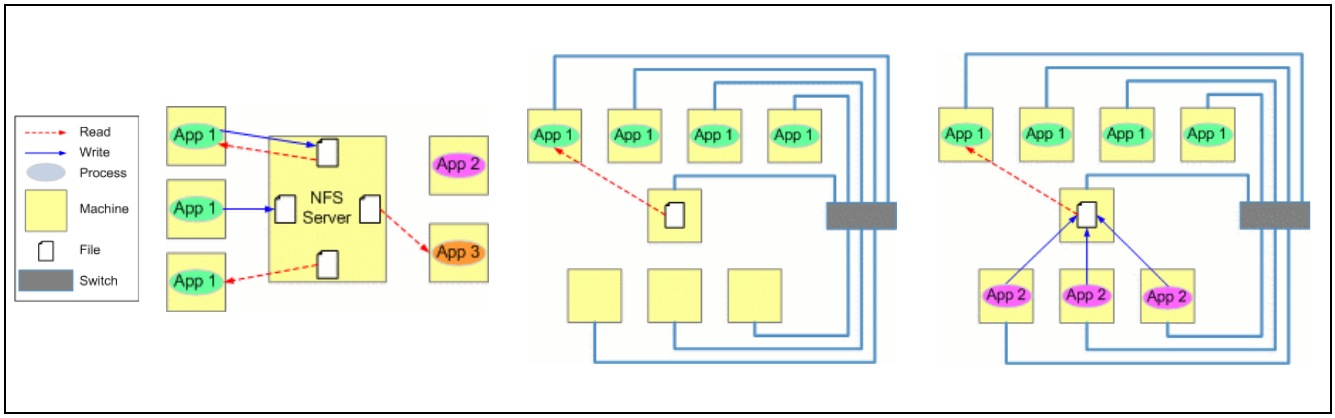

Figure 1. General and Experimental Scenarios. The left figure shows the typical scenario as it would be found in practice: several applications are running and accessing the same NFS server, from different machines. The middle and right figures show the experimental environments used to model the key characteristics of the real scenario, with the optimal environment in the middle and the suboptimal environment on the right.

Table 1: Execution and I/O Times for Master Process

	Wall-Clock		I/O	
	Normal	**Sub-optimal**	**Normal**	**Sub-optimal**
Min.	11.30	11.27	0.00038	0.00036
Max.	12.05	41.28	0.00096	30.00
Avg.	11.48	14.71	0.00051	3.43
Var.	0.03	73.15	<0.000001	73.14

for all processes, and the MPI functions are negligible. This is contrasted against the suboptimal environment in which ranks one through three are dominated by time spent in `MPI_Bcast` and the master process is dominated by time spent in `main`.

We then analyzed the TAU trace data. The left side of Figure 3 shows a screenshot of a Vampir timeline for a representative poor performing execution in the suboptimal environment. We see that `MPI_Bcast` is the dominating function for three processes and that these processes have matching intervals for MPI activities. Conversely, the master process is dominated by main and its MPI activities do not register at this scale. The second screenshot examines this timeline at a finer granularity. In this view, we see that the Master process does not begin execution of its call to `MPI_Bcast`, until 22.329 seconds into the overall execution. Knowing that the master process is responsible for placing data into the broadcast buffer, we readily see that the other processes are waiting for most of the time that they spend in `MPI_Bcast`.

We conclude that it is difficult to determine the root cause of the performance variability in the suboptimal environment using only techniques available in application based performance tools. In this experiment, the performance problems were caused by processes which were not part of the MPI set of processes and were executing on machines outside the scope of the executing MPI application. This type of situation presents a challenge to application based performance tools, as these tools are not designed to simultaneously track application events and events in the rest of the runtime environment. The types of additional data that would be of use in this problem scenario would capture the activities of the shared NFS server resource in relation to the MPI application's activities.

5. A Model for Automated Diagnosis

Detecting runtime interference with parallel application behavior will require online application monitoring to collect a combination of system and application level metrics. Due to hardware limitations and perturbation caused by unbounded software instrumentation, it is impossible to collect all possible performance data at all times. In this section we describe our model for determining a correct diagnosis.

Our model for automated diagnosis incorporates metrics, expectations, diagnoses, and triggers. Each high level condition or behavior is a *diagnosis*. Each diagnosis is either a comparison of a *metric* and an *expectation* or it

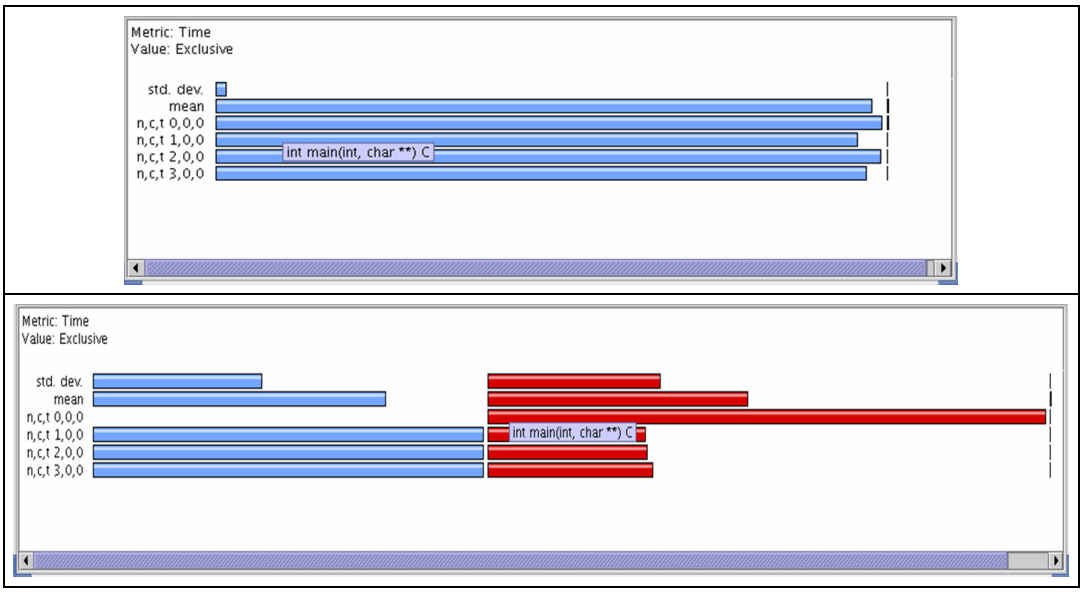

Figure 2. ParaProf Screenshots of TAU Profile Data for the NFS Server Test. The screenshots show the exclusive time for the dominating functions for each MPI process, for the normal environment (top) and the suboptimal environment (bottom).

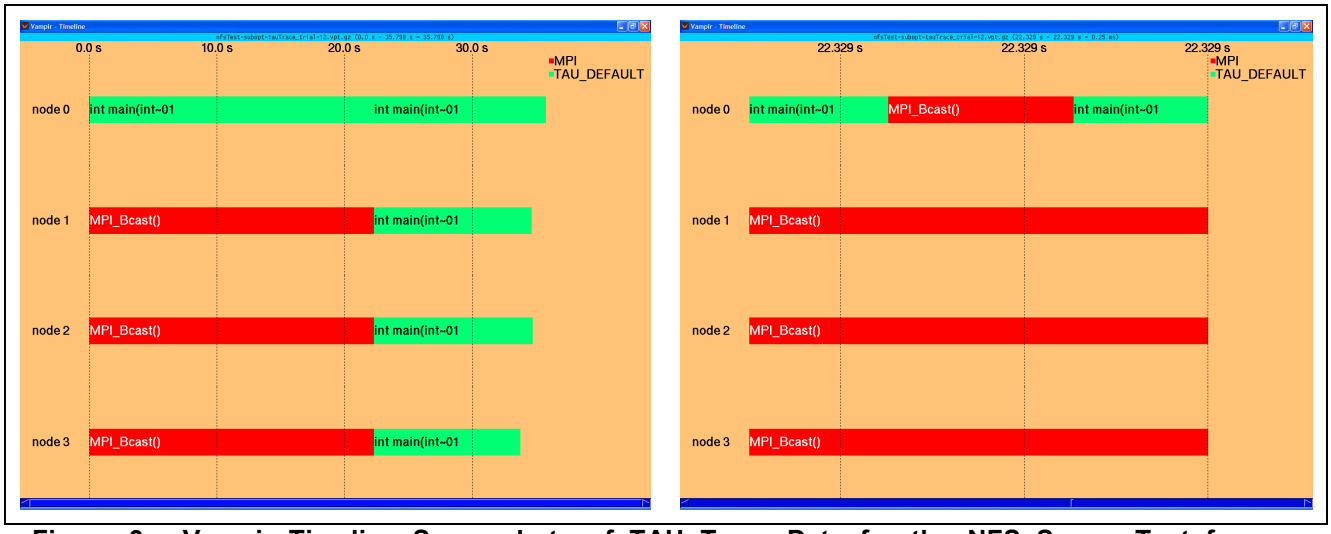

Figure 3. Vampir Timeline Screenshots of TAU Trace Data for the NFS Server Test for an Execution in the Suboptimal Environment. In the left screenshot, it is evident that the processes on nodes 1, 2, and 3 spend more than half of their execution time in MPI_Bcast**, while the MPI activities of the process on node 0 (the master) barely register at this scale. On the right, a zoomed in view of the timeline displays a granularity in which the** MPI_Bcast **execution time for the master process can be visualized, showing that the master process does not execute its** MPI_Bcast **function until 22.329 seconds into the overall execution.**

is a logical combination of diagnoses. A metric is any measurable value. An expectation is a particular value for a metric, used as a threshold in evaluating possible diagnoses. Note that each diagnosis may have more than one definition; a diagnosis is correct if any one of its valid definitions is true. Each diagnosis may be assigned a priority. A *trigger* is a unidirectional link from a *base diagnosis* to a *target diagnosis*. When a base diagnosis is true the trigger is *active*, and when the base diagnosis is false or has not been tested the trigger is *inactive*. When a trigger becomes active, its target diagnosis is placed on an evaluation queue.

Given the search space formed by all possible diagnoses, exhaustive search is not practical for reasons of scalability and perturbation. A key challenge to automated performance diagnosis is to develop an algorithm for ordering traversal of the search space, on the assumption that most of the measurements in the fully expanded space will never be made. Our approach is to navigate the search space in priority order, where priorities are assigned to diagnoses.

In the remainder of this section we present an illustrative example which corresponds to the NFS Server Test scenario. The definitions for the diagnoses used in this scenario are shown in Table 2. ***NFSs_Bottleneck*** represents the file server contention present in the suboptimal environment in the NFS Server Test. This diagnosis is assessed as true based on the values of three other diagnoses, each of which is defined as a combination of diagnoses or as a combination of metrics and expectations (see Table 3). The diagnosis of ***App_LowReadRate*** exists when the time required for accomplishing an application read operation exceeds an expectation for how long it should take. By evaluating the number of bytes read in relation to the length of time required to accomplish the reading task, an application read rate can be established. If this rate is less than the expectation for *minReadRate*, then the application is diagnosed as having a low read rate. ***NFSc_HighReadReqCount***, the second diagnosis in the top level composition, indicates that the number of NFS client read requests is high compared to the actual number of bytes read by the application. This diagnosis is a combination of two diagnoses: ***NFSc_LowReadRate*** and ***NFSc_ReadCount***. ***NFSc_LowReadRate*** is evaluated by comparing the number of bytes read per client read request to a specified expectation for the minimum number of bytes read per NFS client read request. The assessment for ***NFSc_ReadCount*** tests if the number of read requests, issued by an NFS client over an interval of time, is greater than one. The NFS client has a high read request count if both ***NFSc_LowReadRate*** and ***NFSc_ReadCount*** evaluate to true. ***NFSs_HighServerReqToClientReqs*** is the third diagnosis contained in the top level definition, and it indicates that the ratio of the number of requests served by an NFS server to the number of requests made by a

specific NFS client is high. In the simple case, where only one client interacts with an NFS server, we expect this ratio to be close to one; and this would indicate that the server is responding to the client's requests. In the general case, where the server interacts with multiple clients, this ratio will be greater than one; and if it exceeds some expectation we know that the ratio of server requests to client requests is high.

Continuing this example, we now illustrate the use of triggers and a priority scheme. The priorites used in this example are shown in Table 2. We represent priorities as numeric values, where lower numbers have higher priority. In Table 4 we list the triggers, and in this example we have two triggers. We have assigned a priority of one to ***App_LowReadRate***, and we have assigned all the other diagnoses a priority of two. We assume ***App_LowReadRate*** is the first diagnosis considered, If `App_readRate` < *minReadRate*, the target diagnosis ***NFSc_HighReadReqCount*** would be placed on the evaluation queue. By definition, it is evaluated by ***NFSc_LowReadRate*** and ***NFSc_ReadCount***. If both are true, the diagnosis ***NFSc_HighReadReqCount*** would change to true and the trigger pointing to ***NFSs_HighServerReqsToClientReqs*** would become active, causing this diagnosis to be placed on the evaluation queue. If this diagnosis evaluates to true, then the diagnosis for ***NFSs_Bottleneck*** would evaluate to true. In this example, we start with ***App_LowReadRate***, so that the limited resources available for monitoring and conducting online diagnosis are allocated efficiently, delaying tests for diagnoses related to NFS server performance problems if an application is not exhibiting a related symptom.

6. Conclusions

A large number of published and anecdotal examples describe instances of communications libraries, the Operating System, and hardware preventing well-developed parallel applications from achieving good performance in practice. This problem is exacerbated by the inability of most parallel performance tools to detect and report these root causes. Analysts tuning applications must try a number of different tools and approaches to determine the true cause for performance that does not meet expectations. Traditional performance analysis tools

Table 2: Diagnosis Definitions and Priorities for the NFS Server Test Example

Diagnosis	Definition	Priority
NFSs_Bottleneck	***App_LowReadRate*** $\wedge$ ***NFSc_HighReadReqCount*** $\wedge$ ***NFSs_HighServerReqsToClientReqs***	2
App_LowReadRate	`App_readRate` < *minReadRate*	1
NFSc_HighReadReqCount	***NFSc_LowReadRate*** $\wedge$ ***NFSc_ReadCount***	2
NFSc_LowReadRate	`AppNFSc_bytesReadPerReadReq` < *minBytesReadPerReadReq*	2
NFSc_ReadCount	$\texttt{NFSc\_readCount}_t > 1$	2
NFSs_HighServerReqsToClientReqs	`NFSs_reqRate` > *maxNFSsReqRate*	2

Table 3: Metrics and Expectations for the NFS Server Test Example

Metric	Definition
`App_readRate`	The number of bytes read per second by an application
$\texttt{App\_bytesRead}_t$	The total number of bytes read by an application over an interval of time, *t*
$\texttt{NFSc\_readCount}_t$	Total number of client read requests issued over an interval of time, *t*
`AppNFSc_bytesReadPerReadReq`	$\frac{\texttt{App\_bytesRead}_t}{\texttt{NFSc\_readCount}_t}$
`NFSs_reqRate`	The number of requests received by an NFS server per second
Expectation	**Definition**
minReadRate	A minimum value for the number of bytes that should be read per second
minBytesReadPerReadReq	A minimum number of bytes that should be read per client read request
maxNFSsReqRate	A maximum number of requests a NFS server can handle per second

Table 4. Triggers for the NFS Server Test Example

Base Diagnosis	Target Diagnosis
App_LowReadRate	***NFSc_HighReadReqCount***
NFSs_HighReadReqCount	***NFSs_HighServerReqsToClientReqs***

are unable to provide adequate guidance to analysts in determining the causes of performance problems rooted in the runtime environment.

We are developing a new approach for performance analysis called Environment Aware Performance Analysis. This approach seeks to identify root causes of observed performance in a selected set of common scenarios. Environment Aware Performance Analysis combines analysis of an application's execution behavior with analysis of the runtime environment, resulting in a more accurate performance diagnosis.

References

1. D. Tsafrir, Y. Etsion, et al. System noise, OS clock ticks, and fine-grained parallel applications. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS '05*), pages 303-312, Jun. 2005.
2. E. Van Hensbergen. The effect of virtualization on OS interference. In *Proceedings of the First Workshop on Operating System Interference in High Performance Applications (PACT '05)*, Sep. 2005.
3. K. Mohror and K.L. Karavanic, A study of tracing overhead on a high-performance Linux cluster. Portland State University , Computer Science Dept., Portland, OR, Tech. Rep. TR-06-06, Dec. 2006.
4. K. L. Karavanic, J. May, K. Mohror, B. Miller, K. Huck, R. Knapp, and B. Pugh. Integrating database technology with comparison-based parallel performance diagnosis: the PerfTrack performance experiment management tool. In *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC '05)*, Nov. 2005.
5. T. R. Jones, L. B. Brenner, and J. M. Fier. Impacts of operating systems on the scalability of parallel applications. Lawrence Livermore National Laboratory, Livermore, CA., Tech. Rep. UCRL-MI-202629, 2003.
6. F. Petrini, D. J. Kerbyson, and S. Pakin. The case of the missing supercomputer performance: achieving optimal performance on the 8,192 processors of ASCI Q. In *Proceedings of the 2003 ACM/IEEE Conference on Supercomputing (SC '03)*, Nov. 2003.
7. J. C. Phillips, G. Zheng, S. Kumar, and L. V. Kalé. NAMD: biomolecular simulation on thousands of processors. In *Proceedings of the 2002 ACM/IEEE Conference on Supercomputing (SC '02)*, Nov. 2002.
8. L. V. Kale, S. Kumar, G. Zheng, and C. W. Lee. Scaling molecular dynamics to 3000 processors with projections: a performance analysis case study. In *Proceedings of the Terascale Performance Analysis Workshop, International Conference on Computational Science (ICCS '03)*, Jun. 2003, pages 25-32.
9. H. Brunst, D. Kranzlmüller, and W. E. Nagel. Tools for scalable parallel program analysis - Vampir VNG and DeWiz. In *Distributed and Parallel Systems: Cluster and Grid Computing*, Vol. 777, Z. Juhasz, P. Kacsuk and D. Kranzlmüller, Eds. New York: Springer Kluwer International Series in Engineering and Computer Science, 2004, pages 93-102.
10. H-L. Truong, T. Fahringer, G. Madsen, A. D. Malony, H. Moritsch, and S. Shende. On using SCALEA for performance analysis of distributed and parallel programs. In *Proceedings of the 2001 ACM/IEEE Conference on Supercomputing (SC '01)*, Nov. 2001.
11. R. Kufrin. PerfSuite: An accessible, open source performance analysis environment for Linux. In *6th International Conference on Linux Clusters: The HPC Revolution 2005*. Apr. 2005.
12. S. Shende and A. D. Malony. The TAU parallel performance system. *International Journal of High Performance Computing Applications,* vol. 20, no. 2, 2006.
13. B. P. Miller, M. Clark, J. Hollingsworth, S. Kierstead, S.-S. Lim, and T. Torzewski. IPS-2: The second generation of a parallel program measurement system. *IEEE Transactions on Parallel and Distributed Systems*, vol. 1, no. 2, pages 206-217, Apr. 1990.
14. 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*, vol. 28, no. 11, pages 37-46, Nov. 1995.
15. H.-L. Truong and T. Fahringer. SCALEA-G: A unified monitoring and performance analysis system for the grid. *Scientific Programming*, vol. 12, no. 4, pages 225-237, 2004.
16. J. K. Hollingsworth and P. J. Keleher. Prediction and adaptation in Active Harmony. In *Proceedings of the Seventh IEEE International Symposium on High Performance Distributed Computing(HPDC-7 '98)*, pages 180-188, Jul. 1998.
17. C. Cascaval, E. Dusterwald, P. F. Sweeney, and R. W. Wisniewski. Performance and environment monitoring for continuous program optimization. *IBM Journal of Research and Development*, vol. 50, no. 2-3, pages 239-247, Mar./May 2006.
18. R. W. Wisniewski, *et al*. Performance and environment monitoring for whole-system characterization and optimization. *PAC2 Conference on Power/Performance Interaction with Architecture, Circuits, and Compilers*, Oct. 2004.
19. H. W. Cain, B. P. Miller, and B. J. N. Wylie. A callgraph-based search strategy for automated performance diagnosis. In *Proceedings of the 6th International Euro-Par Conference (Euro-Par '00)*, Aug./Sep., 2000.
20. *TAU User's Guide*, Department of Computer and Information Science, University of Oregon, LANL, and NM Research Centre Julich, Germany, 2005. [online]. Available: http://www.cs.uoregon.edu/research/tau. [Accessed: Jan 15, 2006].
21. The LAM/MPI Team, *LAM/MPI User's Guide, Version 7.0.6*, Pervasive Technology Labs, Indiana University, 2004. [online]. Available: http://www.lam-mpi.org. [Accessed: Mar. 17, 2006].