Performance Contracts: Predicting and Monitoring Grid Application Behavior

Fredrik Vraalsen, Ruth A. Aydt, Celso L. Mendes, and Daniel A. Reed
{vraalsen,aydt,cmendes,reed}@cs.uiuc.edu

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

Abstract. [1] Given the dynamic nature of grid resources, adaptation is required to sustain a predictable level of application performance. A prerequisite of adaptation is the recognition of changing conditions. In this paper we introduce an *application signature model* and *performance contracts* to specify expected grid application behavior, and discuss our monitoring infrastructure that detects when actual behavior does not meet expectations. Experimental results are given for several scenarios.

1 Introduction

Computational grids have emerged as a new paradigm for high-performance computing [5]. Given the dynamic nature of grid resources, both distributed applications and the underlying infrastructure must adapt to changing resource availability to sustain a predictable level of application performance. A prerequisite of adaptation is the recognition of changing conditions.

In this paper we present an approach to predicting application performance and enabling the detection of unexpected execution behavior. We summarize related work, introduce a *performance contract* where commitments are specified, and put forward an *application signature model* for predicting application performance. We describe a *monitoring infrastructure* that detects when behavior does not fall within the expected range, and show experimental results for several execution scenarios.

2 Related Work

Predicting application performance on a given parallel system has been widely studied [12, 9, 3]. Recently the studies have been extended to distributed systems [7, 10]. The traditional focus of performance prediction methods is accuracy. Given the variability in a grid environment, we believe it is not practical

[1] This work was supported in part by the National Science Foundation under grants ASC 97-20202, EIA-997502, and the PACI Computational Science Alliance Cooperative Agreement, and by the Department of Energy under contract W-7405-ENG-36.

to pursue maximal predictive accuracy. Our techniques derive acceptable performance bounds and evaluate whether observed performance falls within those bounds.

Quality of service approaches seek to guarantee a minimum level of resource availability by reserving resources [6]. Our work does not rely on reservations or make guarantees. Instead, we use monitoring to determine when performance expectations are violated, signaling other agents to take appropriate action.

3 Performance Contracts

A *performance contract* states that given a set of *resources* (e.g., processors or networks), with certain *capabilities* (e.g., floating point rate or bandwidth), for particular *problem parameters* (e.g., matrix size or image resolution), the application will exhibit a specified, measurable, and desired *performance* (e.g., render r frames per second or finish iteration i in t seconds). We use the term *contract specifications* to refer collectively to the resources, capabilities, problem parameters, and performance.

To validate a contract, one must continually verify that the contract specifications are being met. Strictly speaking, a contract is *violated* if any contract specification does not hold during application execution. In practice, some degree of imprecision in the specifications is expected, reflecting variability in the environment and imperfections in the models. For these reasons, the monitoring system should tolerate "minor" deviations from the contract specifications.

4 Application Signature Model

A *performance model* generates a performance prediction for an application based on specified resources, capabilities, and problem parameters. Possible sources of models include application and library developers, compile time code analysis, and historical observations. Achieved application performance reflects a complex interplay of application demands on execution resources and the response of those resources to the demands. We outline a new approach to decomposing this interdependence, enabling separate specification of application stimuli, resource capabilities, and the composition of these into a performance prediction reflecting resource response.

4.1 Application Intrinsic Metrics

We define *application intrinsic metrics* as metrics that are solely dependent on the application code and problem parameters. These metrics express the demands the application places on the resources, and are independent of the execution environment. Examples of such metrics include messages per byte and average number of source code statements per floating point operation.

An *application intrinsic metric space* is a multidimensional space where each axis represents a single application intrinsic metric. Consider N metrics, each

measured at regular intervals, for each of P parallel tasks. For each task, the measured values specify a trajectory through the N-dimensional metric space. We call this trajectory, illustrated in Figure 1, the *application signature*.

Some applications exhibit timing-dependent behavior and the application signature varies across runs. For others, values computed during runtime control the execution path and the application signature cannot be known in advance. However, we believe that, for a large number of important applications, the application signatures are not affected by timing or runtime dependencies.

By selecting metrics that capture critical resource demands, we can characterize the load an application places on the environment. The ability of the resources to service the load determines the performance of the application.

4.2 Execution Metrics

An application signature describes application stimuli to execution resources but does not quantify expected performance. We now consider the performance of the application on a given set of resources, and refer to achieved application performance in terms of *execution metrics* that express rates of progress. Examples of execution metrics are instructions per second and messages per second. An application traces a trajectory through the *execution metric space* as it runs, and we call this the *execution signature*.

The execution signature reflects both the application demands on the resources and the response of the resources to those demands. The execution signature may vary, even on the same resources, due to resource sharing with other applications. Our goal is to generate an expected execution signature for a given application on a specific set of resources during a particular period of time.

4.3 Performance Projections

To generate the expected execution signature, we *project* the application signature into execution metric space using a scaling factor for each dimension. The projection scaling factors correspond to the capabilities of the resources on which the application will execute. Possible sources of capability information include peak performance numbers, benchmark values, predictions of future availability (e.g., via NWS [13]), and observed levels of service for previous executions.

In the following projection equations, the first terms represent application intrinsic metrics, the second represent projection factors based on resource capabilities, and the results represent performance expressed as execution metrics.

$$\frac{instructions}{FLOP} \times \frac{FLOPs}{second} = \frac{instructions}{second} \qquad \frac{messages}{byte} \times \frac{bytes}{second} = \frac{messages}{second}$$

An application signature projected into two execution signatures, reflecting different resource capabilities, is shown in Figure 2. In the projection shown as the solid line, the bandwidth is higher than in the projection shown as the dotted line. The computing capability is the same for both.

We use the term *application signature model* to refer to our method of predicting application behavior by projecting an application signature into execution space via resource capability scaling factors.

4.4 Performance Contracts

Relating the application signature model to the performance contract from Section 3, the contract problem parameters are encompassed in the application signature; resources and capabilities are reflected in the projection factors; specified measurable performance is defined by the model output as points in the execution metric space. Contract validation consists of determining whether application intrinsic and execution metrics, measured at runtime, lie within an acceptable range of the expected values stated in the contract.

For some applications, the signatures consist of tightly grouped points in the metric spaces rather than far-reaching trajectories. Standard clustering techniques can be used to partition the points into equivalence classes, yielding cluster centroids and variance factors in the N dimensions. Our clustering algorithm takes a single parameter that specifies the maximum distance between a point and a cluster centroid, effectively controlling the size and number of clusters it identifies in a given set of points.

When such equivalence classes exist, contract validation can be thought of as determining whether runtime metrics fall within an acceptable range of a cluster centroid. The "acceptable range" controls the level of violation that will be tolerated. Ranges can be tuned independently for each dimension based on a number of factors including computed cluster variance, confidence in resource capability values, and estimates of model accuracy. Figure 3 illustrates this concept of contract validation based on equivalence classes and degrees of tolerance.

5 Monitoring Infrastructure

Our monitoring infrastructure is based on *Autopilot*, a toolkit for application and resource monitoring and control [11]. When an application executes, Autopilot *sensors*, embedded in application or library code, register with an *Autopilot Manager*. Sensor *clients* query the manager to locate sensors with desired properties and attach to those sensors to receive measurement data. The sensors, managers, and sensor clients can be located anywhere on the grid.

Application signatures and projections define *expected* application behavior, and runtime measurements capture *actual* behavior. Given these, we must specify (a) a policy to determine when a measured value is sufficiently different from an expected value to raise a violation and (b) a mechanism for implementing this policy. We base our policy on the equivalence classes and ranges of tolerance described in Section 4.4, and use *fuzzy logic* as the implementation mechanism.

Fuzzy logic allows one to linguistically state contract violation conditions. Unlike boolean logic values that are either true or false, fuzzy variables can assume a range of values, allowing for smooth transitions between areas of acceptance and violation. Using Autopilot decision procedures, which are based on fuzzy logic, it is possible to change the rule set describing contract violation conditions to accommodate different equivalence classes and levels of tolerance.

A rulebase defining two fuzzy variables, *distance* and *violation*, and linguistic rules for setting the *violation* based on the *distance* are shown in Figure 4. Space

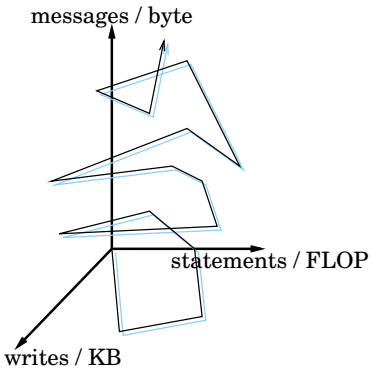

Fig. 1. Hypothetical Application Signature

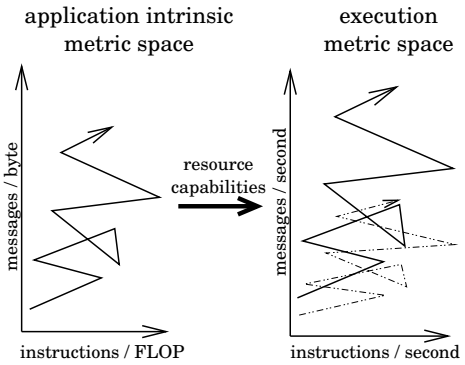

Fig. 2. Performance Projections

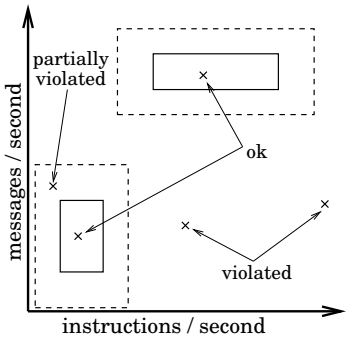

Fig. 3. Equivalence Classes and Violations

```
var distance(0, 2) {
    set trapez LOW  ( 0, .5, 0, .5 );
    set trapez HIGH ( 1, 2, .5, 0 );  }
var violation(-1, 2) {
    set triangle NONE  ( 1, 1, 1 );
    set triangle TOTAL ( 0, 1, 1 );   }

if ( distance == LOW  )
   { violation = NONE; }
if ( distance == HIGH )
   { violation = TOTAL; }
```

Fig. 4. Contract Rulebase

constraints prohibit a full explanation, but intuitively each fuzzy variable has an associated transition function that maps "crisp" values to a degree of truth for the corresponding fuzzy members. The range of crisp values and the shape of the transition functions are controlled by the numeric values in the rulebase. We adjust the numeric values to implement the desired degrees of tolerance.

Our contract monitor receives sensor measurements and evaluates the fuzzy logic rulebase using these as crisp input. The distances for each dimension in the metric space are evaluated individually, and an overall violation level is obtained by combining the individual results. Given the violation levels, contracts could be marked as violated based on threshold values for individual metrics and overall behavior. Currently we report the violation levels directly as contract output.

6 Experimental Results

To verify the feasibility of the performance projection approach and contract monitoring infrastructure presented in Sections 4 and 5, we conducted a series of experiments using distributed applications on a grid testbed. All testbed sys-

Cluster Name	Location	Nodes	Processor	Network
major	Illinois	8	266 MHz Pentium II	Fast Ethernet
opus	Illinois	4	450 MHz Pentium II	Myrinet
rhap	Illinois	32	933 MHz Pentium III	Fast Ethernet
torc	Tennessee	8	550 MHz Pentium III	Myrinet
ucsd	San Diego	4	400 MHz Pentium II	Fast Ethernet

Table 1. Experimental System Environment

tems ran Globus [4] and Linux on the x86 processor architecture, as shown in Table 1. We inserted a call in each application to create Autopilot sensors that periodically captured data from the PAPI [1] interface (number of instructions and floating point operations) and from MPI event wrappers (number of messages and bytes transferred).

To simulate environments where applications compete for resources, we introduced two kinds of load external to the applications. A purely computational load was added by executing a floating point intensive task on one of the nodes, and a communication load was generated via bursts of UDP packets.

6.1 Master/Worker Application

For our first group of experiments, we created a program that executes a series of "jobs" following a master/worker paradigm. Each worker repeatedly requests a job, receives and executes the job, and sends the result back to the master. Parameters control the number and size of jobs, and their floating point intensity.

We executed the master/worker application on a virtual machine comprised of twelve nodes from the *major*, *ucsd*, and *torc* clusters, connected by a wide-area network. We captured application intrinsic metrics using sensors with a 30 second sampling period, and built the application signature shown in Figure 5a. This figure shows two types of node behavior. The master, represented by points to the right, executes few floating point operations as reflected in a higher number of instructions/FLOP. The workers, represented by points to the left, execute the bulk of the floating point operations. The messages/byte metric is constant for all nodes, reflecting the consistent size of the messages between each master/worker pair.

Based on the derived application intrinsic signature, we projected the performance behavior of the application for our particular virtual machine. Using computation and communication projection factors specific to the *major*/*ucsd*/*torc* clusters, we obtained the predicted execution signature shown in Figure 5b. Hardware differences among the clusters caused the workers to have different projected performance. We derived our projection factors by computing mean values for observed FLOPs/second and bytes/second, representing the computation and communication characteristics of each node.

The application and projected execution signatures define the expected application behavior on the selected resources. To obtain the parameters used for contract validation, we first clustered the points in each metric space to create sets of equivalence classes. In application intrinsic space, we used the cluster

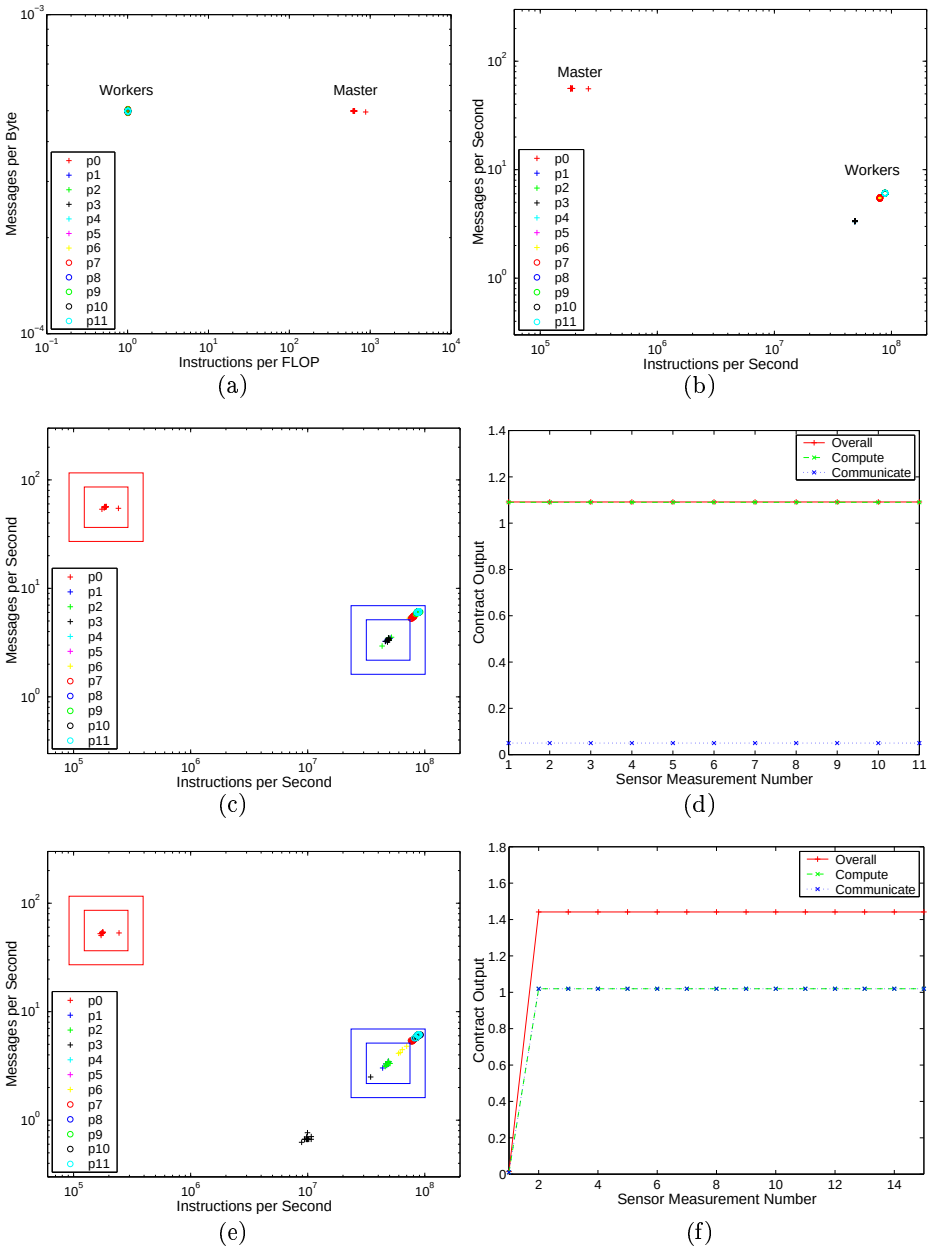

Fig. 5. Master/Worker Results: (a) application intrinsic signature (b) projected execution signature (c) baseline observed execution signature (d) intrinsic contract output for processor 9; plots of *overall* and *compute* overlap (e) measured execution signature under load (f) performance contract output for processor 3; plots of *compute* and *communicate* overlap

variance, reported by the clustering algorithm, to define the acceptable performance ranges. In execution space, we based the tolerance around the centroids on the variance in the projection factors.

Figure 5c shows the regions of acceptable performance for the master and for one of the workers, and the observed points captured from all nodes during the actual execution. The inner border for each region corresponds to no violation and the outer border to a total violation. Points that fall between the two borders partially violate the contract for that equivalence class. If an observed point lies outside the regions of all equivalence classes, a contract violation is reported. Since the parameters for the acceptable ranges of Figure 5c were derived from the same baseline execution, it is not surprising that most observed points satisfy the contract.

To determine whether the contracts we developed were able to detect contract violations, we repeated the master/worker execution under different operating conditions. We executed the program on the same machines, but configured the jobs assigned to nodes 6 through 11 to perform fewer floating point operations, testing our ability to detect violations in the application intrinsic space.

We applied the application intrinsic contracts and consistently detected violations on nodes 6 through 11. Figure 5d shows the contract output for processor 9. Computation, communication, and overall violation levels are shown separately, with higher values corresponding to more severe violations. The computation and overall contracts were consistently violated, while the communication contract was always satisfied.

In our next experiment, we reran the master/worker program with the original job mix and introduced an external floating point load on processor 3 of our virtual machine shortly after execution began. Because there was not any communication between workers, we expected the external load to affect only the performance of the worker on the node where the load was placed. This was confirmed by the measured execution signatures, presented in Figure 5e. While the signatures for most workers were the same as in Figure 5c, the performance of processor 3 no longer fell within the predicted range. Our monitor correctly detected the violation, as indicated by the contract output shown in Figure 5f.

6.2 ScaLAPACK Application

For our second group of experiments, we used the *pdscaex* application from the *ScaLAPACK* package [2], which solves a linear system by calling the *PDGESV* library routine. We executed an instrumented version of *pdscaex* with matrices of size 10000 and 20000, which were sufficiently large to highlight interesting behavior, but small enough to make parametric experiments practical.

We began our ScaLAPACK tests by executing on a virtual machine with four nodes from each of the *major*, *opus*, and *rhap* clusters at Illinois, using a 10000x10000 matrix and a sensor sampling period of 60 seconds. Following the same procedure described for the master/worker application, we conducted a baseline execution to obtain the application intrinsic signature. With that signature and the mean projection factors computed for the execution, we derived

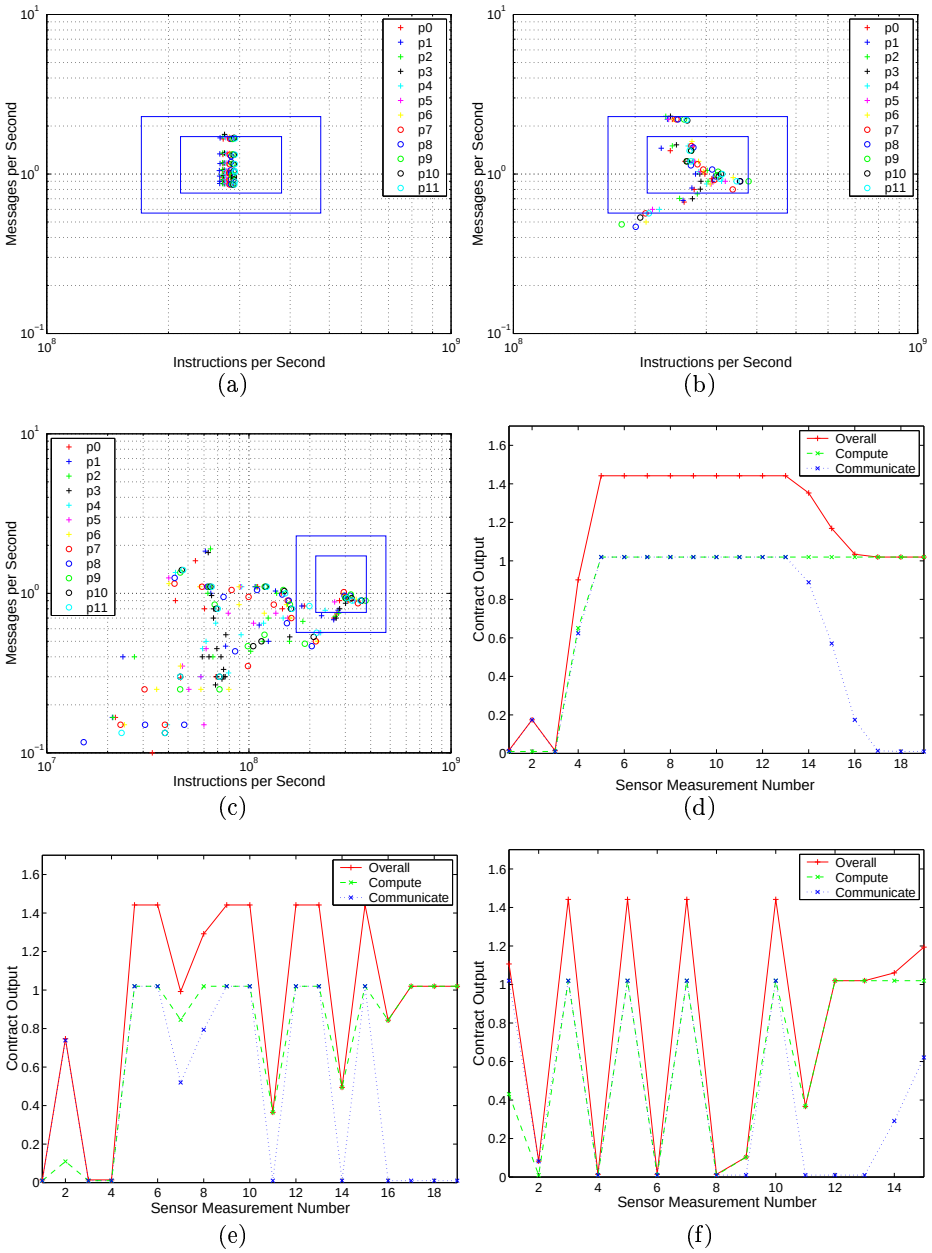

Fig. 6. ScaLAPACK Local Area Results: (a) projected execution signature (b) baseline observed execution signature (c) observed execution signature under load (d) contract output for processor 3 with computational load on processor 3 (e) contract output for processor 9 with computational load on processor 3 (f) contract output for processor 9 with network load

the projected signature and bounds of Figure 6a. Figure 6b shows the actual application performance, with the bounds derived in the previous step. Note that even in the baseline execution a few observed points lie outside the expected performance range indicated by the rectangles. In cases like this, one could adjust the fuzzy logic rulebase to increase the degree of tolerance.

In addition to the baseline described above, we conducted two other *pdscaex* executions in this environment. First, we added a computational load to processor 3. Figure 6c shows the observed execution signatures for all nodes in this run. Because of implicit synchronization in the algorithm, the performance of all nodes decreased while the additional load was present. Furthermore, both the communicate and compute performance metrics were affected, as the synchronization effectively links the two dimensions. The performance violations were detected by the contracts for each node, as shown in Figures 6d and 6e for node 3 and node 9, respectively. Violations were very consistent for processor 3, the node with the additional load, indicating that it never was able to achieve the expected level of performance when sharing compute resources. Although delayed when unable to get input, processor 9 did occasionally execute sufficient instructions per second to avoid severe levels of violation.

Our last *pdscaex* local-area test was run while a heavy communication load was active between nodes *amajor* and *rhap6*. These nodes were *not* participating in the execution, but shared the network with nodes that were. The contract output for the various *pdscaex* nodes indicated that performance was degraded by the network load. The contract output for processor 9, presented in Figure 6f, shows periodic violations throughout the execution.

We extended our ScaLAPACK tests to a wide-area grid environment consisting of machines at Illinois and Tennessee. We selected six nodes from each of the *rhap* and *torc* clusters, used a matrix size of 20000x20000, and set our sensor sampling period to 4 minutes to capture both compute and communicate activity in most periods. We conducted our baseline *pdscaex* execution under these conditions.

To simulate an external perturbation in our execution environment, we repeated this *pdscaex* execution and introduced a floating point load on processor 8 shortly after the beginning of the execution. This load ran concurrently with *pdscaex* for 50 minutes before being removed. The measured performance metrics for this run are shown in Figure 7a.

Figures 7b and 7c show the contract results for nodes 4 and 8, respectively. As with the earlier local-area run with an added computation load, the contract output for the loaded node (processor 8) showed constant violations while the load was in place. The performance of other nodes (e.g. processor 4) was also affected, but not as consistently.

Figure 7d shows the observed FLOP rates for a subset of the processors and offers insight into the behavior of *pdscaex* when one node is a bottleneck. The FLOPs/second for the loaded processor plateaued below its expected performance level when the load was in place. In contrast, the FLOPs/second measurements for other processors oscillated between expected values and zero, in-

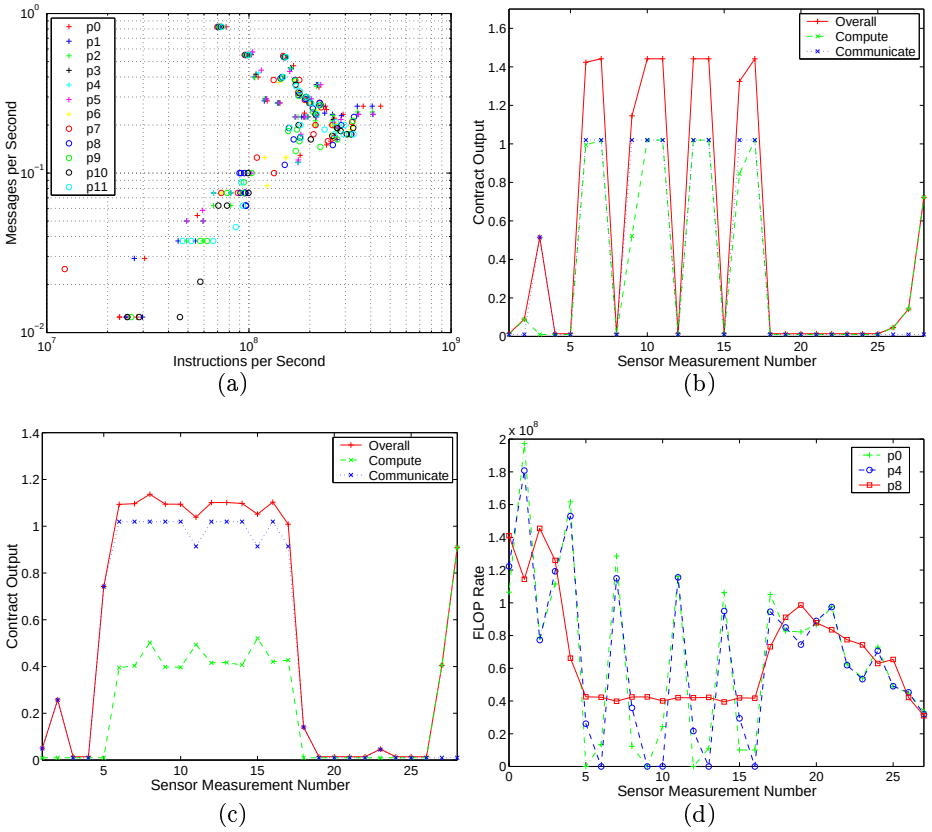

Fig. 7. ScaLAPACK Wide Area Results: (a) execution signature with load on processor 8 (b) contract output for processor 4 with load on processor 8 (c) contract output for processor 8 with load on processor 8 (d) FLOP rate for selected processors

dicating that the processors were either computing at the projected rate or idle while waiting for data from another processor. These FLOP rate behaviors are reflected in the contract violations shown in Figures 7b and 7c.

7 Conclusions and Future Work

In this paper we described performance contracts and a performance model based on application signatures, demonstrated the use of these with our monitoring infrastructure, and showed that we could detect unexpected grid application behavior. Our results indicate that this is a promising approach, however much work remains.

We plan to investigate other application intrinsic and execution metrics to capture computation, communication, and I/O characteristics for a wider set of applications. We are studying "global" contracts based on overall application

behavior, rather than on per-process commitments, and will extend our violation policy to tolerate transient unexpected behavior. To handle application signatures with no equivalence classes over the entire application, we will explore periodic projection and clustering. Finally, we are considering how to extend our technique to cases for which baseline executions are not available, possibly using data collected during the current execution as a source for later predictions.

This work is the result of fruitful collaborations with the GrADS [8] team, notably Fran Berman, Andrew Chien, Jack Dongarra, Ian Foster, Lennart Johnsson, Ken Kennedy, Carl Kesselman, Rich Wolski, and many staff and students.

References

1. Browne, S., Dongarra, J., Garner, N., London, K., and Mucci, P. A Scalable Cross-Platform Infrastructure for Application Performance Tuning Using Hardware Counters. In *Proceedings of Supercomputing 2000* (2000).
2. Choi, J., et al. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. In *Proceedings of Supercomputing 96* (1996).
3. Fahringer, T. *Automatic Performance Prediction of Parallel Programs.* Kluwer Academic Publishers, 1996.
4. Foster, I., and Kesselman, C. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications 11*, 2 (1997).
5. Foster, I., and Kesselman, C., Eds. *The Grid: Blueprint for a New Computing Infrastructure.* Morgan Kaufmann Publishers, 1998.
6. Foster, I., Roy, A., Sander, V., and Winkler, L. End-to-End Quality of Service for High-End Applications. *IEEE Journal on Selected Areas in Communications Special Issue on QoS in the Internet* (1999).
7. Kapadia, N., Fortes, J., and Brodley, C. Predictive application-performance modeling in a computational grid environment. In *Proceedings of the Eight IEEE Symposium on High-Performance Distributed Computing* (1999).
8. Kennedy, K., et al. Grid Application Development Software. http://hipersoft.cs.rice.edu/grads/.
9. Mehra, P., et al. A Comparison of Two Model-Based Performance-Prediction Techniques for Message-Passing Parallel Programs. In *Proceedings of the ACM Conference on Measurement & Modeling of Computer Systems* (1994).
10. Petitet, A., et al. Numerical Libraries and The Grid: The GrADS Experiments with ScaLAPACK. Tech. Rep. UT-CS-01-460, University of Tennessee, 2001.
11. Ribler, R., et al. Autopilot: Adaptive Control of Distributed Applications. In *Proceedings of the Seventh IEEE Symposium on High-Performance Distributed Computing* (1998).
12. Saavedra-Barrera, R. H., Smith, A. J., and Miya, E. Performance prediction by benchmark and machine characterization. *IEEE Transactions on Computers 38*, 12 (1989).
13. Wolski, R., Spring, N. T., and Hayes, J. The Network Weather Service: A Distributed Resource Performance Forecasting Service for Metacomputing. *The Journal of Future Generation Computing Systems* (1999).