Performance Contracts: Predicting and Monitoring Grid Application Behavior

Fredrik Vraalsen*
SINTEF Telecom and Informatics, Norway
E-mail: fredrik.vraalsen@sintef.no

October 15, 2002

1 Introduction

The Internet and the growing availability of inexpensive yet powerful commodity hardware has changed our view of computing. Computational grids have emerged as a new paradigm for high-performance computing. Grids consist of geographically distributed computational resources, such as supercomputers, clusters of workstations, scientific instruments, storage systems, visualization systems and so on. These resources are connected via high-capacity networks [5].

Given the dynamic nature of computational grids, both the applications running on the grid and the underlying infrastructure must be able to adapt to changing resource availability to sustain a predictable and desired level of application performance.

In this paper we present an approach to predicting application performance and enabling the detection of unexpected execution behavior, typically caused by unanticipated load on shared grid resources. We introduce a *performance contract* where resource, application, and performance commitments are specified. The performance contract should not be viewed as a binding document, but rather as an estimate of expected behavior in a shared environment lacking support for reservations or quality of service guarantees.

Further, we describe an *application signature model* used for predicting application performance on a given set of grid resources. In this model, we introduce the notion of *application intrinsic behavior* to separate the performance effects of the runtime system from the behavior inherent in the application itself. The signature model is used as the basis for the performance predictions. Sensors monitor runtime application and system performance, and fuzzy logic decision procedures are used to verify whether the contract commitments are upheld. This paper is based on the results from [18] and [19].

The remainder of this paper is organized as follows. First, in §2, related work is summarized. We then introduce performance contracts in §3 and the application signature model in §4. The infrastructure for application and resource monitoring is described in §5, followed by contract specification and verification in §6. The experimental results are described in §7, and finally, §8 summarizes the conclusions and outlines directions for future work.

2 Related Work

Optimising grid application performance involves many factors, from compiler technology and distributed algorithms via resource scheduling to application and system monitoring and simulation. The work described in this paper was performed as part of the GrADS

*The original work for this paper was performed as part of an M.Sc. degree at the Department of Computer Science of the University of Illinois at Urbana-Champaign, USA

project [10], which aims to integrate efforts in all of these areas in order to simplify grid application development.

Predicting application performance on a given parallel system has been widely studied in the past [3, 11, 16]. More recently those studies have been extended to distributed systems [9, 13].

Traditional performance prediction techniques often render performance models that are specific to a single architecture or a static set of resources. However, computational grid environments consist of a collection of dynamic, heterogeneous resources. Our approach seeks to separate the influences of the runtime system from the behavior *intrinsic* to the application. Similar to [9], we use information about application input parameters when predicting application performance. However, rather than relying on application performance data from previous runs on the same system, our approach combines knowledge of application intrinsic behavior with run-time predictions of resource availability from systems such as the Network Weather Service (NWS) [20] to derive acceptable application performance ranges.

The focus of traditional performance prediction methods is often prediction accuracy. However, given the variability in a grid environment, we derive ranges of acceptable performance rather than a single estimate. In addition, traditional performance prediction methods often assume that a dedicated set of resources will be available for the entire application run [6], which may not be true given the dynamic nature of computational grids. Our approach is instead based on monitoring application and system resource performance to verify that observed performance falls within the acceptable ranges. If the performance contract is violated, we signal other agents in the runtime environment and rely upon them to take corrective action, such as migrating the application to another system.

Traditional performance measurement uses application instrumentation to capture performance data during execution for post-mortem analysis [14]. Our work however requires real-time application performance data to verify the performance contracts. *Autopilot* is a toolkit for dynamic, run-time performance measurement and tuning on heterogeneous computational grids through instrumentation on the application source code level [15]. It is described in more detail in §5.1. Paradyn [12] and Dyninst [8] offer similar run-time performance measurements through binary patching of running application executables.

3 Performance Contracts

An application runs on a set of *resources* (e.g., processors and network links), each of which have certain *capabilities* (e.g., maximum floating point rate or available bandwidth). The application is executed with certain *problem parameters* (e.g., matrix size or image resolution), and it achieves some measurable and desired *performance* (e.g., render *r* frames per second or finish iteration *i* in *t* seconds) during its execution.

A *performance contract* states that given a set of *resources* with certain *capabilities*, for particular *problem parameters*, the application will exhibit a specified, measurable *performance*. We use the term *contract specifications* to refer collectively to the resources, capabilities, problem parameters, and performance.

A performance model generates a performance prediction for an application based on specified resources, capabilities, and problem parameters. To verify a contract, one must continually monitor the application and system during runtime to verify that the contract specifications are being met. The contract can be violated both by the application and the system, e.g. if the system resources don't deliver the expected performance.

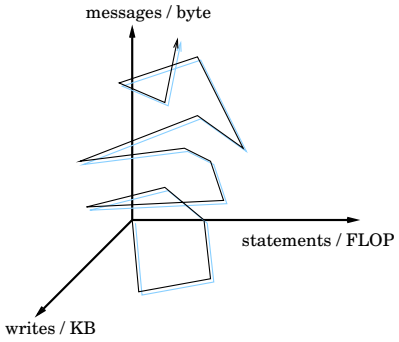

Figure 1: *Hypothetical 3-dimensional Signature*

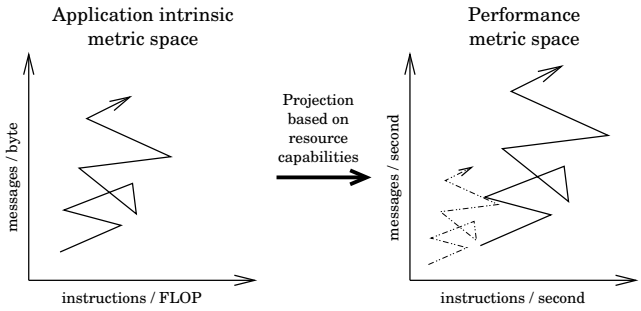

Figure 2: *Performance Projections*

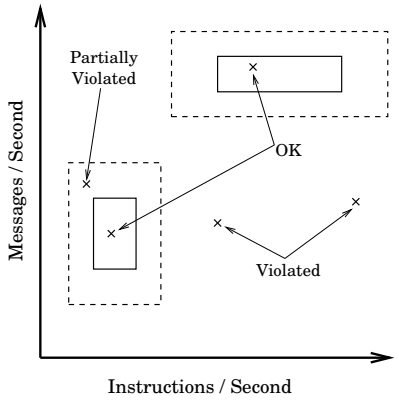

Figure 3: *Equivalence Classes and Violations*

```
var distance(0, 10) {
  set trapez SHORT (0,  3, 0, 3);
  set trapez LONG  (6, 10, 3, 0);
}
var contract(-1, 2) {
  set triangle OK       (0, 1, 1);
  set triangle VIOLATED (1, 1, 1);
}
if ( distance == SHORT )
   { contract = OK; }
if ( distance == LONG  )
   { contract = VIOLATED; }
```

Figure 4: *Fuzzy Logic Rulebase*

4 Signature Model

Achieved application performance reflects a complex interplay of application demands on system resources and the response of the resources to those demands. In this section, we outline a new approach to decomposing this interdependence, enabling us to separate specification of application demands on resources from specification of resource response. We then combine information about application demands and resource capabilities to generate a performance prediction for the application running on a specific system.

A *metric space* is a multidimensional space where each axis represents a single metric (e.g., FLOPS or I/O request size). Consider an application where we measure N different metrics. These metrics span out an N-dimensional metric space.

Each metric is measured at regular intervals, and the measured values specify a trajectory through the N-dimensional metric space. We call this trajectory, illustrated in Figure 1, a *signature*. As the application executes, it traces a trajectory in the metric space. Contract verification consists of comparing the predicted signatures with the signatures based on runtime measurements.

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 are independent of the capabilities of the system on which the application is running, such as processor speed or network bandwidth. These metrics express the demands the application places on the resources,

or the resource *stimuli*. Examples of application intrinsic metrics include number of bytes transferred per communication message and average number of source code statements per floating point operation.

As the application executes, it traces a trajectory through the *application intrinsic metric space*. We call this trajectory the *application intrinsic signature*. By selecting application intrinsic metrics that capture important resource demands, we can understand the load the application places on the execution environment. The ability of the resources to service the load determines the overall performance of the application. Timing issues or data dependencies may cause the application intrinsic signatures to vary between runs. However, we believe that for a large number of important applications, the application intrinsic signatures are not affected by this [18].

4.2 Performance Metrics

We refer to achieved application performance via *performance metrics* that express rates of progress. Examples of performance metrics are instructions per second and messages per second. In contrast to the application intrinsic metrics, the performance metrics are affected by the performance of the system resources.

Similar to the application intrinsic case, an application traces a trajectory through the *performance metric space* as it runs. This *performance signature* reflects both the application demands on the resources, as well as the *response* of the resources to those demands. In contrast to the application intrinsic signature, the performance signature may vary across application executions, even on the same resources. This variation results from resource sharing with other applications.

Our goal is to generate an expected performance signature for a given application on a specific set of resources during a particular period of time. In effect, this means predicting application performance.

4.3 Expected Signatures

To verify performance contracts, we need both the expected application intrinsic signature and expected performance signature for the application execution as well as runtime measurements of the actual signatures. To generate the expected performance signature, we *project* the expected application intrinsic signature into the performance metric space. This process is described below.

Because, by assumption, the application intrinsic signature depends only on the problem parameters, we can use the application intrinsic signature recorded from a previous run with the same or similar problem parameters. A database can be created with a set of recorded application intrinsic signatures for an application, and the appropriate signature can be looked up at runtime.

4.3.1 Performance Projections

We *project* the application intrinsic signature into performance metric space using a *single figure of merit* scaling factor for each dimension. The projection consists of scaling each point on the application intrinsic signature by multiplying its coordinates with the corresponding projection factor for that dimension. The resulting scaled points are the points on the projected signature.

The projection scaling factors correspond to the capabilities of the resources where the application will execute. Possible sources of capability information include peak perfor-

mance numbers, benchmark values, predictions of future resource availability (e.g., via NWS [20]), and observed resource performance from previous executions on the same system with similar problem parameters.

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

$$\frac{instructions}{FLOP} \times \frac{FLOPs}{second} = \frac{instructions}{second} \tag{1}$$

$$\frac{messages}{byte} \times \frac{bytes}{second} = \frac{messages}{second} \tag{2}$$

Figure 2 shows an example of how an application intrinsic signature could be projected into the performance metric space using two distinct sets of projection factors, reflecting different resource capabilities.

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

5 Monitoring Infrastructure

To verify performance contracts, one must measure both application intrinsic signatures and performance signatures during application execution. This may involve collecting data from multiple execution sites at differing granularities (microseconds to hours) and multiple system levels (i.e., from hardware performance counters through communication and I/O libraries to application stimuli). Hence, the infrastructure for collecting this data must be efficient, scalable and adaptable.

In this section, we describe our infrastructure for monitoring runtime behavior and verifying performance contracts. Although we use the application signature model to predict performance in our presentation, the infrastructure can also be used with other types of prediction models.

5.1 Autopilot Toolkit

Autopilot is a toolkit for dynamic, run-time performance measurement and tuning on heterogeneous computational grids [15]. Through the use of distributed sensors, actuators and decision procedures, applications and resources can be monitored in real-time to dynamically adapt to changing application resource demands and system resource availability.

Sensors extract performance data during the application execution, providing the data required for performance contract verification. Sensors can be inserted into application code either automatically (e.g., by the compiler) or manually by inserting calls to the sensor into the application code. Sensors can also be part of external programs or utilities to measure resource availability.

Actuators allow remote clients to modify the values of application variables or to call functions in the application. The actuators can be used to implement the policy decisions made based on the contract verification.

Sensor data can be processed by a data transformation function before being reported by the sensors. These *attached functions* accept raw sensor data as input and produce the transformed data as output. Attached functions can perform simple statistical computations (e.g., sliding window averaging) or more complex transformations.

There are several motivations for performing such transformations. First, some data are point measurements that must be converted into time varying metrics. The second is data smoothing; the processing function can act as a filter, mitigating the effect of data outliers. Third, it can reduce the data volume sent by the sensor to the remote sensor client. This may be important for efficiency, so that monitoring does not unduly perturb the application behavior and performance.

There exits a wide variety of classical techniques for decision making, ranging from decision tables and trees to standard control theory. However, many of the these techniques require both a deep understanding of the problem space as well as development of an exhaustive and consistent problem description.

Fuzzy logic specifically targets the attributes of performance optimization problems where the classical techniques fall short, namely conflicting goals (e.g., minimize response time and maximize throughput) and poorly understood optimization spaces. The highly dynamic, uncertain nature of computational grids makes fuzzy logic an ideal decision making technique for this environment. Furthermore, the decision procedures can easily be adjusted or even retargeted to a new domain by modifying the fuzzy logic rulebase, with little or no application development necessary.

The decision procedures in the Autopilot toolkit use data from Autopilot sensors as input values to the fuzzy logic rulebases. The fuzzy logic rules are then evaluated, and Autopilot actuators are used to enact the policy decisions made by the rulebases.

5.2 Performance Data Sources

To collect data about application and system performance, we need to gather information about what the application is doing and how the system responds to the application stimuli. We use the PAPI toolkit [1] to read the processor hardware performance counters. The information gathered by this toolkit is used both to determine the application instruction mix for the application intrinsic signature and the achieved floating point operation performance on the processor. Similarly, the MPI [17] profiling library is used to capture information about the application's communication behavior and the achieved communication bandwidth on the various processors of the virtual machine.

6 Contract Specification and Verification

In essence, contract verification consists of comparing the runtime application behavior with the expected application behavior to determine if the contract has been violated. Predicted application intrinsic signatures and projected performance signatures define *expected* application and system behavior, whereas runtime measurements capture *actual* behavior. Given these, we must specify (a) a policy to determine when a measured value is sufficiently different from the expected value to cause a contract violation and (b) a mechanism for implementing this contract verification policy.

Our implementation of performance contracts consist of two parts; (1) a set of equivalence classes representing acceptable ranges of application intrinsic behavior and application performance, and (2) a fuzzy logic rulebase that is used in conjunction with the equivalence classes to determine if the performance contract is violated. The equivalence classes represent our policy, and we use fuzzy logic as the mechanism for implementing the policy.

For some applications, the application intrinsic signature consists of sets of tightly clustered points in the metric space rather than points spread throughout the entire space. These

clusters reflect equivalence classes of demands that the application puts on the resources during different phases of the application execution.

In the applications we studied in §7, this effect was caused by the applications performing more or less the same type of work during their entire execution lifetimes. Both applications looped over the data, each loop iteration performing essentially the same operations on the different subsets of the data. Thus the sensors reported very similar values every time, leading to the tightly clustered points. We believe this type of behavior leading to a set of clusters is common for many types of grid applications.

When such equivalence classes exist, contract verification can be thought of as determining whether the metrics measured at runtime fall within an acceptable range of a cluster centroid. Figure 3 illustrates this concept of contract verification based on equivalence classes and degrees of tolerance. 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 smooth transitions between areas of acceptance and violation. Figure 4 shows an example of such a rulebase.

Our contract monitor receives measurement data from Autopilot sensors. We then compute the distance between the measured point and the centroid of the closest cluster. The rulebase is then evaluated, giving us a violation level for each individual metric and the contract as a whole. Given the violation levels, the contract monitor could decide to mark the individual metrics or overallcontracts as violated based on some threshold values. Currently we report the violation levels directly as the contract output.

The variability in Grid performance may necessitate the use of relatively large error bounds to avoid continously signaling contract violations. Accepting large deviations from expected performance without signaling a contract violation may seem counter-intuitive. However, unless the anticipated gain in execution time is substantially larger than the duration of the contract violation, there is no benefit to acting on violations. This reflects the common sense reality of legal contracts as well.

One reason for accepting relatively large deviations is that the cost of the remedy may be very high. For example, rescheduling the application to run on a different part of the grid requires checkpointing the application's state, moving all the associated data to a new set of machines, and restarting the computation. The relatively slow network links of a distributed grid make this operation very costly. Hence, it is often preferable to complete execution on the current virtual machine even if there are some contract violations.

Another reason is that many contract violations may be short-lived, caused by transient changes in resource availability. This is especially true in a dynamic grid environment. One may choose to not report an initial violation, but if the condition persists long enough, action may be warranted. Again, this reflects common sense reality.

7 Experimental Results

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

We performed experiments with two different applications. The first was a synthetic Master/Worker application, the second was an existing parallel linear algebra application. These applications were chosen because they represent important classes of grid applications.

In a grid environment, an application will often be executing on a collection of geographically distributed systems, connected by a relatively high-latency and low-bandwidth

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*

network (e.g., the Internet). Applications suitable for such an environment are therefore generally applications that perform a lot of computation relative to communication, and applications with few interdependencies between subtasks.

For our first group of experiments, we therefore created a program that executes a series of "jobs" according to a master/worker paradigm. For our second group of experiments, we used an application from the *ScaLAPACK* package. ScaLAPACK implements a set of linear algebra routines for heterogeneous, distributed memory parallel systems [2]. The application we chose solves a linear system by calling ScaLAPACK library routines. This is a suitable application for grid environments, because of the high ratio between computation ($O(n^3)$) and communication ($O(n^2)$).

We conducted experiments on a variety of systems in a grid environment. All systems supported the Globus software [4] for process management and authentication, Autopilot performance monitoring tools, and PAPI for hardware performance counter access. The systems ran Linux on the x86 processor architecture, and were grouped into a set of clusters as shown in Table 1.

All the experiments used the same fuzzy logic rulebase for their performance contracts. The rulebase is described in more detail in [18]. The basic criterion used for the contract verification is the distance between the measured data point and the closest equivalence class.

For each application, we evaluated both a set of local performance contracts for each individual processor and a global performance contract for the application as a whole. To test whether the contracts would detect a performance violation, we simulated two kinds of load external to the applications. A purely computational load was introduced by executing a floating point intensive task on one of the processors, and a communication load was introduced by repeated bursts of UDP packets from the Netperf tool [7] between two processors on the network.

This paper only discusses the results from running ScaLAPACK on several clusters in a local area network. However, we were able to show similar results for ScaLAPACK on clusters connected by a wide area network and for the master/worker application. The complete results are available in [18].

7.1 ScaLAPACK Application - Local Area

We executed the instrumented version of the ScaLAPACK application on a distributed system consisting of twelve processors, all from clusters at UIUC. The first four processors were from the *major* cluster, the next four were from the *opus* cluster, and the last four were from the *rhap* cluster. All runs were done with a matrix size of 10000x10000 and a sensor sampling period of 60 seconds. This problem size was sufficiently large to highlight interesting behavior, instead of all the data drowning in the noise from the variability of

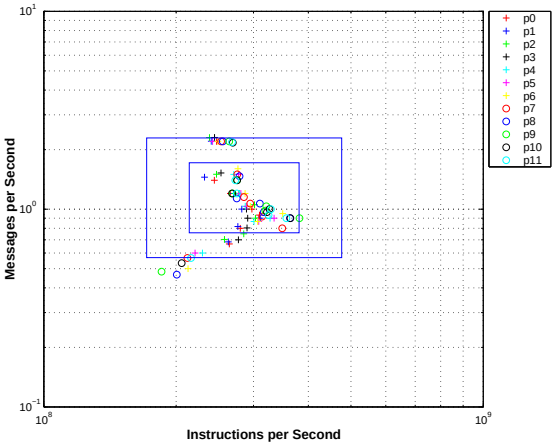

Figure 5: *ScaLAPACK Observed Performance Signature During Baseline Execution*

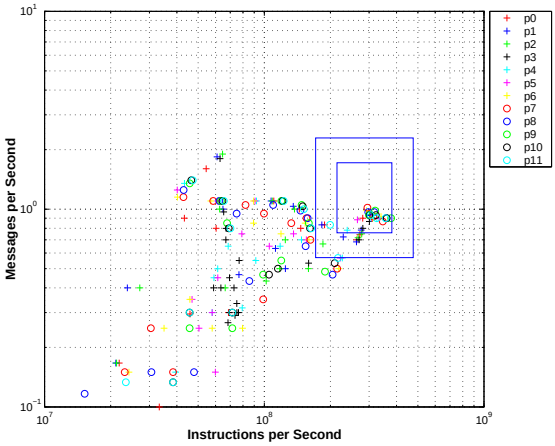

Figure 6: *ScaLAPACK Performance Signature Under Computation Load*

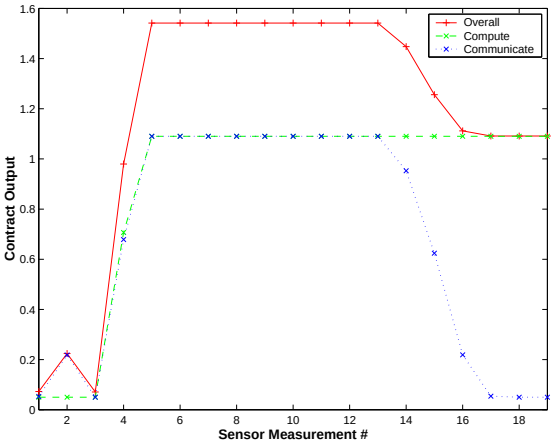

Figure 7: *ScaLAPACK Performance Contract Output Under Computation Load - Processor 3*

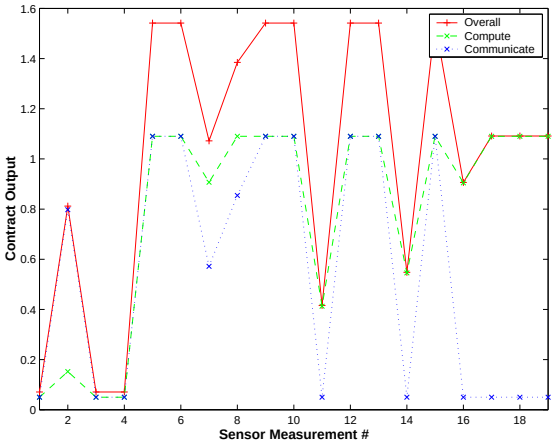

Figure 8: *ScaLAPACK Performance Contract Output Under Computation Load - Processor 9*

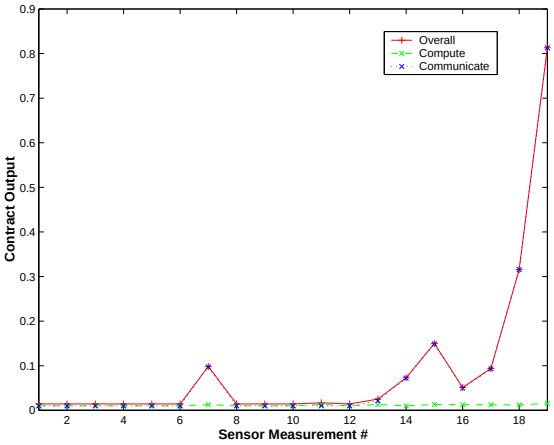

Figure 9: *ScaLAPACK Application Intrinsic Contract Output Under Computation Load - Processor 3*

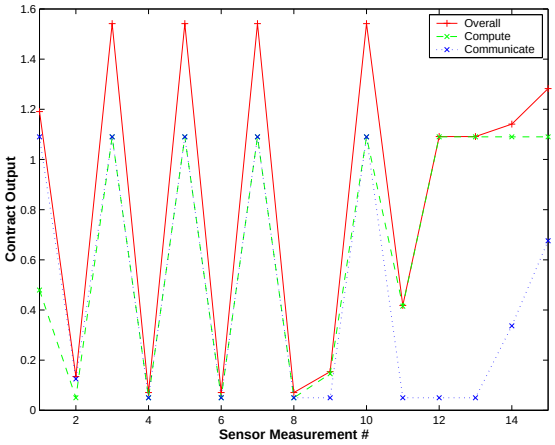

Figure 10: *ScaLAPACK Contract Output Under Network Load - Processor 9*

the network speed and processor load. Furthermore, it was small enough to make a series of parametric experiments practical, as each experiment only took on the order of 15-20 minutes. The sensor sampling period was determined by manually running tests.

We first conducted a baseline execution of the ScaLAPACK application on an unloaded system to capture the application intrinsic signature and the projection factors. With the application intrinsic signature and the projection factors computed for this execution, we then derived the projected performance signature and acceptable performance bounds for the different processors. The projection factors were derived from the observed FLOPs/second and bytes/second values from the baseline execution by computing the mean value for the individual processors. Figure 5 shows the actual application performance from the baseline execution, with the bounds derived for processor 8 in the previous step.

In addition to the baseline execution described above, we conducted two other executions on this system. First, we added a computational load on processor 3. Figure 6 shows the observed performance signatures for all processors in this run together with the acceptable performance range for processor 8.[1]

The performance violations were detected by the contracts for each processor, as shown in Figures 7 and 8 for processor 3 and processor 9, respectively. Note that the violations were more consistent for the processor with the additional load (processor 3), indicating that it was never able to achieve the expected level of performance when sharing the computational resources. This is because processor 3 was computing the entire time, but because of the external load it was computing at a lower rate than expected, thus constantly violating the contract. The other processors on the other hand were able to meet their expected level of performance when they had data to work with, but part of the time they were idle waiting for data from processor 3, causing the contract to oscillate between OK and violated as shown in the contract output for processor 9.

To check whether the contract monitor was able to detect that the violation was caused by a system resource violation, we also evaluated the contract for the application intrinsic metrics. Figure 9 shows the application intrinsic contract output from the processor with the external computation load (processor 3). The application intrinsic metrics are within the acceptable ranges for almost the entire execution, as is expected since the application intrinsic behavior of the application has not changed, only the speed at which it executes. Comparing this with the performance metric contract output in Figure 7 shows that the contract monitor is able to correctly identify the violation as a system violation.

Our second experiment was run while a heavy communication load was active between the machines *amajor* and *rhap6*. These machines were *not* participating in the application execution, but share the network with the machines that were. The contract outputs for the various processors executing the ScaLAPACK application indicated that performance was impacted by the additional network load. The contract output for processor 9, presented in Figure 10, shows violations throughout the execution. Thus the contract monitor was able to detect the contract violation.

We also evaluated the global contract for ScaLAPACK, using the global mean of the metrics across all the processes. During the run with load on a single processor, the performance of the entire application is affected even though only one of the processors is heavily loaded. This is because the individual processes are loosely synchronized, and the other processes eventually end up waiting for the slow process. Similar behavior can be seen when the network is loaded.

[1] Note that the scale used in this figure differs from that used in the previous figure.

8 Conclusions and Future Work

In this paper we described performance contracts and a performance model based on application signatures. We demonstrated the use of these concepts with our real-time monitoring infrastructure, and showed that we were able to detect when grid applications did not meet expected behavior. Our results to date show that this is a fruitful approach, however, much work remains.

Other choices of application intrinsic and execution metrics need to be investigated to represent the salient features of computation, communication, and I/O for a wider set of grid applications. In addition, more studies of the potential of global contracts are needed. For example, other statistical metrics can be computed as a basis for the global contract, or one could use a voting scheme to determine the global contract output based on the outcome of the local contracts. Work is also needed to extend our contract violation policy to tolerate transient periods of unexpected behavior without signaling a violation.

To handle application signatures that do not naturally exhibit equivalence classes when viewed from the perspective of the entire application, one can explore periodic projection and clustering throughout the course of the execution. Finally, analysis is needed on how to extend our technique to cases where baseline executions are not available. One possibility is using data collected during the current execution as a source for predictions of application behavior later in the same run.

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* (November 2000).

[2] CHOI, J., CLEARY, A., DEMMEL, J., DHILLON, I., DONGARRA, J., HAMMARLING, S., HENRY, G., OSTROUCHOV, S., PETITET, A., STANLEY, K., WALKER, D., AND WHALEY, R. C. ScaLAPACK: A Portable Linear Algebra Library for Distributed Memory Computers - Design Issues and Performance. In *Proceedings of Supercomputing 96* (November 1996).

[3] FAHRINGER, T. *Automatic Performance Prediction of Parallel Programs*. Kluwer Academic Publishers, Norwell, Massachusetts, 1996.

[4] FOSTER, I., AND KESSELMAN, C. Globus: A Metacomputing Infrastructure Toolkit. *International Journal of Supercomputer Applications 11*, 2 (1997), 115–128.

[5] FOSTER, I., AND KESSELMAN, C., Eds. *The Grid: Blueprint for a New Computing Infrastructure*. Morgan Kaufmann Publishers, July 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] HEWLETT-PACKARD COMPANY. *Netperf: A Network Performance Benchmark*, February 1996. Available from www.netperf.org.

[8] HOLLINGSWORTH, J. K., MILLER, B. P., AND CARGILLE, J. Dynamic Program Instrumentation for Scalable Performance Tools. Tech. Rep. CS-TR-1994-1207, 1994.

[9] 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* (Redondo Beach, California, August 1999), pp. 47–54.

[10] KENNEDY, K., AYDT, R., CHIEN, A., BERMAN, F., DONGARRA, J., FOSTER, I., JOHNSSON, L., KESSELMAN, C., REED, D., AND WOLSKI, R. Grid Application Development Software. www.hipersoft.rice.edu/projects/grads.html.

[11] MEHRA, P., SCHULBACH, C. H., AND YAN, J. C. 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 - SIGMETRICS'94* (Nashville, May 1994), pp. 181–190.

[12] MILLER, B. P., CARGILLE, J., IRVIN, R. B., NEWHALL, T., CALLAGHAN, M. D., HOLLINGSWORTH, J. K., KARAVANIC, K. L., AND KUNCHITHAPADAM, K. The Paradyn Parallel Performance Measurement Tools. Tech. Rep. CS-TR-1994-1256, 1994.

[13] PETITET, A., BLACKFORD, S., J.DONGARRA, ELLIS, B., FAGG, G., ROCHE, K., AND VADHIYAR, S. Numerical Libraries and The Grid: The GrADS Experiments with ScaLAPACK. Tech. Rep. UT-CS-01-460, University of Tennessee, April 2001.

[14] REED, D. A., AYDT, R. A., NOE, R. J., ROTH, P. C., SHIELDS, K. A., SCHWARTZ, B. W., AND TAVERA, L. F. Scalable Performance Analysis: The Pablo Performance Analysis Environment. In *Proc. Scalable Parallel Libraries Conf.* (1993), IEEE Computer Society, pp. 104–113.

[15] RIBLER, R. L., VETTER, J. S., SIMITCI, H., AND REED, D. A. Autopilot: Adaptive Control of Distributed Applications. In *Proceedings of the Seventh IEEE Symposium on High-Performance Distributed Computing* (Chicago, Illinois, July 1998).

[16] SAAVEDRA-BARRERA, R. H., SMITH, A. J., AND MIYA, E. Performance prediction by benchmark and machine characterization. *IEEE Transactions on Computers 38*, 12 (December 1989), 1659–1679.

[17] SNIR, M., OTTO, S., HUSS-LEDERMAN, S., WALKER, D., AND DONGARRA, J. *MPI: The Complete Reference, Volume 1, The MPI Core, Second Edition*. MIT Press, Boston, Massachusetts, 1998.

[18] VRAALSEN, F. Performance contracts: Predicting and monitoring grid application behavior. Master's thesis, University of Illinois, Urbana-Champaign, August 2001.

[19] VRAALSEN, F., AYDT, R. A., MENDES, C. L., AND REED, D. A. Performance contracts: Predicting and monitoring grid application behavior. In *Grid Computing - GRID 2001 Second International Workshop* (Denver, CO, USA, November 2001), pp. 154–165.

[20] 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).