Skeleton Based Performance Prediction on Shared Networks

Sukhdeep Sodhi
Microsoft Corp.
One Microsoft Way
Redmond, WA - 98052
ssodhi@microsoft.com

Jaspal Subhlok
Department of Computer Science
University of Houston
Houston, TX 77204
jaspal@uh.edu

Abstract

The performance skeleton of an application is a short running program whose performance in any scenario reflects the performance of the application it represents. Such a skeleton can be employed to quickly estimate the performance of a large application under existing network and node sharing. This paper presents and validates a framework for automatic construction of performance skeletons of parallel applications. The approach is based on capturing the compute and communication behavior of an executing application, summarizing this behavior and then generating a synthetic skeleton program based on the summarized information. We demonstrate that automatically generated performance skeletons take an order of magnitude less time to execute than the application they represent, yet predict the application execution time with reasonable accuracy. For the NAS benchmark suite, we observed that the average error in predicting the execution time was 6%. This research is motivated by the problem of performance driven resource selection in shared network and grid environments.

1. Introduction

Shared networks, varying from workstation clusters to computational grids, are an increasingly important platform for high performance computing. Performance of an application strongly depends on the dynamically changing availability of resources in such distributed computing environments. Estimating the performance of an application on a given set of resources under varying network and node sharing conditions is a challenge that must be addressed for selecting the best set of execution nodes for an application. This paper introduces *performance skeletons*, which are customized short running programs whose performance mirrors the performance of the application they represent.

The performance skeleton of an application is a synthetically generated program that has the same fundamental execution characteristics as the application it represents. The execution time of the performance skeleton program on a given set of nodes is expected to equal the execution time of the corresponding application under the same conditions scaled down by a fixed large constant factor. Therefore, performance skeletons can be used for predicting application performance. Since this approach involves actual execution of skeleton code at the point in time when a performance related decision has to be made, the estimated performance is for current conditions and no system or network monitoring is required.

This paper introduces a simple methodology for automatically constructing performance skeletons for message passing parallel programs. The primary focus in this paper is on generating skeletons to estimate performance of compute and communication bound applications under CPU and network sharing, rather than on predicting performance across nodes with different system architectures. The main consideration is that the performance skeleton have the same basic computation and communication behavior as the application it represents. The basic approach is to monitor the application message exchanges and CPU utilization pattern, summarize this execution information, and generate a synthetic skeleton program that will reproduce

Appears in the Grids and Advanced Networks (GAN'04) Workshop at CCGRID 2004: The IEEE International Symposium on Cluster Computing and the Grid, April 2004, Chicago, IL
This research was performed while Sukhdeep Sodhi was a graduate student at the University of Houston.

the summarized computation and communication behavior. The procedure does not require access to the source code or any other information about the application. We have implemented this approach and validated it with NAS benchmark programs running on a shared test-bed. We present results that demonstrate the ability of performance skeletons to predict application performance accurately.

2 Related Work

This research is motivated by the problem of resource selection in the emerging field of grid computing [8, 9]. Condor [10] and LSF [28] address node selection based on CPU and software considerations effectively. Research on node selection based on broader system and network resources has been in the context of getting the best general group of execution nodes [4, 20, 26] or customized procedures to select execution nodes for a particular application or application class [5, 6, 7, 14]. Recent work on performance prediction in our group [17, 21, 24, 25] uses a general application characterization framework to create a customized performance model for each application that can predict application performance when the current availability of system resources is provided as input. Hence this model depends on the accuracy of network status information provided by tools such as NWS [22, 27] and Remos [11, 12], which can be expensive to maintain and may be inaccurate or outdated. We build on the general application characterization framework mentioned above to create customized skeletons for each application. Then we employ these skeletons for performance estimation and node selection thereby eliminating the dependence on network monitoring tools.

Reed et.al. [13] generate compact application signatures using a curve-fitting approach to reduce event-tracing overheads. Their goal is to use application signatures for online performance monitoring and tuning. Snavely et.al. [18] create application and machine signatures to simulate application behavior across different system or processor architectures. Calder et.al. [15, 16] exploit periodic application behavior to identify portions of the program that are representative of an application for the purpose of architectural simulations. While all these research efforts have their own different goals, we have borrowed ideas from them to drive this project.

3 Performance Skeletons

A performance skeleton of an application is defined as a short running program that has the same fundamental execution characteristics as the application it represents, although it has no semantic relevance. The execution time of the performance skeleton is designed to be proportional to the execution time of the corresponding application under any execution condition. That is, if the performance skeleton executes in 1/20th of the time it takes to execute an application on a dedicated testbed, the execution time of the skeleton should always be 1/20th of the application execution time on any computation platform under any resource sharing scenario. The goal is to infer the expected performance of an application in a new or changed environment by simply executing the application's performance skeleton. We would like to point out that the skeleton execution is completely different from actually executing the application for a short time. The skeleton should capture the total execution of an application in a short time while the beginning part of an application is typically not representative of the entire application in terms of execution characteristics.

The challenge of this research is to develop a procedure to construct performance skeletons automatically. Ideally a performance skeleton and the corresponding application should generate similar resource usage and system activity, which implies similar computation activity, memory access pattern, communication behavior and synchronization behavior. In this paper we have focused on performance prediction when the application always runs on the same types of nodes but network and CPU resources are shared and hence their availability changes dynamically. The skeletons we construct are not expected to predict performance accurately on systems with different node architecture or memory hierarchy from the one employed for skeleton construction. The skeletons discussed in this paper mimic the compute/communication phases of the applications. The computation phases in the skeletons consist of generic computation which may not be effective for prediction on different system and memory architectures. The skeletons employ the same kind of communication calls as the application.

The skeleton based approach to performance prediction is designed for distributed scientific applications with a relatively static computation structure. Applications with dynamic load balancing, for instance, cannot employ this approach to performance prediction effectively. The approach can be used for many applications outside the domain of scientific computing also but a discussion is beyond the scope of this paper. Our implementation of the skeleton construction framework is for MPI message passing programs.

4 Skeleton Creation Framework

We first outline the main steps involved in the creation of a performance skeleton from a distributed memory message passing MPI application.

1. Application is executed on a controlled testbed and information about CPU and communication behavior is recorded.
2. Application execution information collected in the previous step is summarized into a compact representation.
3. Source code for the corresponding performance skeleton program is generated from the compact execution representation.

Note that this skeleton construction procedure is based on measured execution properties and does not involve access to the source code. We now discuss each of the above steps in more detail.

4.1 Execution data collection

To generate the execution trace of an MPI application we link it with a profiling library developed for this purpose and run it on a dedicated testbed without any competing processes or traffic. The profiling library records information for each process in a separate trace file. Each MPI operation called, parameters passed to it, and its start time and end time, are recorded. Timing measurement is done to microsecond granularity using the Linux *gettimeofday* system call [1]. Time for computation operations is recorded as the time spent between the end of one MPI operation and the start of the next MPI operation. This information is sufficient to identify the communication, computation and synchronization behavior of the application. Generation of the trace file requires no modification of the application source code. We ran MPI applications with and without recording of events with the profiling library and verified that it does not add any significant overheads.

4.2 Create compact representation

The goal of this step is to generate a compact representation of the execution trace. This compact representation should capture the high-level computation, communication and synchronization patterns of the application. This step is divided into two stages. In the first stage we read the trace log for each process and represent it as a sequence of events. We then group similar events and replace them by a new event which represents the average of this group. As an example, suppose we encounter the following two operations:

MPI_Send(Node 3, 2000 bytes), and
MPI_Send(Node 3, 1800 bytes)

If both these events occur only once, they are both replaced by the following operation:

MPI_Send(Node 3, 1900 bytes)

Grouping similar events helps in generating a more compact representation. Events that are grouped together are execution phases of approximately equal duration and message calls with similar parameters. Different type of MPI calls or identical calls to exchange messages with different nodes are never grouped together.

This stage converts the trace log into a sequence of execution events, which can be represented as a string of symbols such as:

$$\alpha\ \beta\ \gamma\ \beta\ \gamma\ \beta\ \gamma\ \kappa\ \alpha\ \alpha\ ....$$

representing the sequence of execution events with different occurrences of the same symbol referring to functionally identical execution events.

In the second stage we identify and mark repeated execution behavior as loops. The problem of identifying repeating application behavior is now represented as the problem of finding repeating sub-strings within a string. We have developed an algorithm [19] which recursively identifies all the repeating sub-strings, starting with the largest matches and working down to substring matches of length 1. The repeating sub-strings are then organized as recursive loop nests with substrings of symbols as loop bodies and the number of repetitions as the number of loop iterations. This is the compact representation of the trace log.

4.3 Generating performance skeleton

The next step is to generate the performance skeleton of the application from the compact representation discussed above. It is desirable that the performance skeletons be short running since execution of the performance skeleton is an overhead in performance estimation. However, the prediction accuracy is likely to be lower for shorter running skeletons. In our framework, the desired ratio K between the execution time of the application and the execution time of the corresponding performance skeleton is provided as a parameter. The fraction K essentially places an upper-bound on the execution time of the skeleton.

The construction of performance skeleton begins with generating a program by converting the symbols in the compact representation to actual 'C' programming language code that creates similar system activity with synthetic computation code and synthetic MPI calls. This is the initial uncompressed performance skeleton program. Next, this skeleton code is scanned and the numbers of iterations in loops with more than K iterations are reduced by a factor K. Additional loops with less than K iterations may have to be generated to maintain the execution activity sequence.

At this point the skeleton consists of compressed loops, loops that are not compressed because the number of iterations is less than K, and uncompressed code outside of loops. The uncompressed loops are now unrolled, that is, they are replaced by the individual operations within the loop repeated for each iteration of the loop. Groups of K occurrences of execution operations in the uncompressed part of skeleton are identified and replaced by a single occurrence. Finally all remaining uncompressed operations with less than K occurrences are *scaled down* by a factor K by adjusting their parameters. For a compute operation we reduce the duration by a factor of K, for communication operations we reduce the bytes exchanged by a factor of K. This yields the final performance skeleton consisting of synthetic 'C' code with MPI calls.

One weakness of this approach is that scaling down a communication operation by reducing the number of bytes exchanged is not accurate. Execution time of the reduced operation would be higher than expected because communication operations have two time components; latency, which is fixed for all message sizes, and message transfer time, which can be scaled down linearly. By reducing the number of bytes exchanged we only reduce the message transfer time, leaving the latency component intact. A more accurate scaling down cannot be achieved without making some assumptions about the execution environments. However, we point out that this kind of reduction is a "last resort" that is employed only for iterations that remain after division by K and operations not in loops. In practice, the impact on overall performance estimation is minimal for most applications.

5 Experiments and Results

A prototype framework for automatic construction of performance skeletons was implemented. Automatically constructed skeletons were then used to predict application execution time in a variety of scenarios and the predictions were compared to the measured application execution time for validation. Additional experiments were performed to compare the prediction accuracy of this approach to other simple approaches. This section describes the validation experiments and discusses the results.

5.1 Experimental setup

The testbed for the experiments is a compute cluster composed of 10 Intel Xeon dual CPU 1.7 GHz machines connected by Gigabit Ethernet links and a full crossbar switch. Results are presented for experiments conducted on 4 nodes. All experimental results are based on the MPI implementation of the NAS Parallel Benchmarks [3, 23]. The codes used are BT (Block Tridiagonal solver), CG (Conjugate Gradient), IS (Integer Sort), LU (LU Solver), MG (Multigrid) and SP (Pentadiagonal solver). All programs are compiled using GNU `g77`, (Fortran) compiler except IS, which is compiled with the `gcc` (C) compiler. The MPICH implementation of MPI is used. The bandwidth between computation nodes was managed with the Linux advanced networking *iproute2* [2] in order to simulate limited bandwidth availability due to competing network traffic. *iproute2* works by intercepting the network packets and passing them through artificial queues to simulate bandwidth limitations.

5.2 Accuracy of skeleton based prediction

To evaluate the performance skeleton approach we first constructed performance skeletons for class B NAS benhmark codes. The desired ratio between the execution time of the skeleton and the application (K in earlier discussion) was selected to be 10. The *actual* ratio L between the execution time of the application and the corresponding performance skeleton is then computed by measuring their execution times on a testbed with no other competing process or network traffic. We have:

$$L = \frac{ApplicationExecutionTime}{SkeletonExecutionTime}$$

Note that L and K are close to each other but not identical because of the approximations inherent in skeleton construction. For our experiments, K was set to 10, while L varied between 11 and 13. Employing L over K for prediction is expected to yield better accuracy since some of the inaccuracies in skeleton construction are filtered out as they affect all executions of a skeleton equally and this was verified by actual experiments.

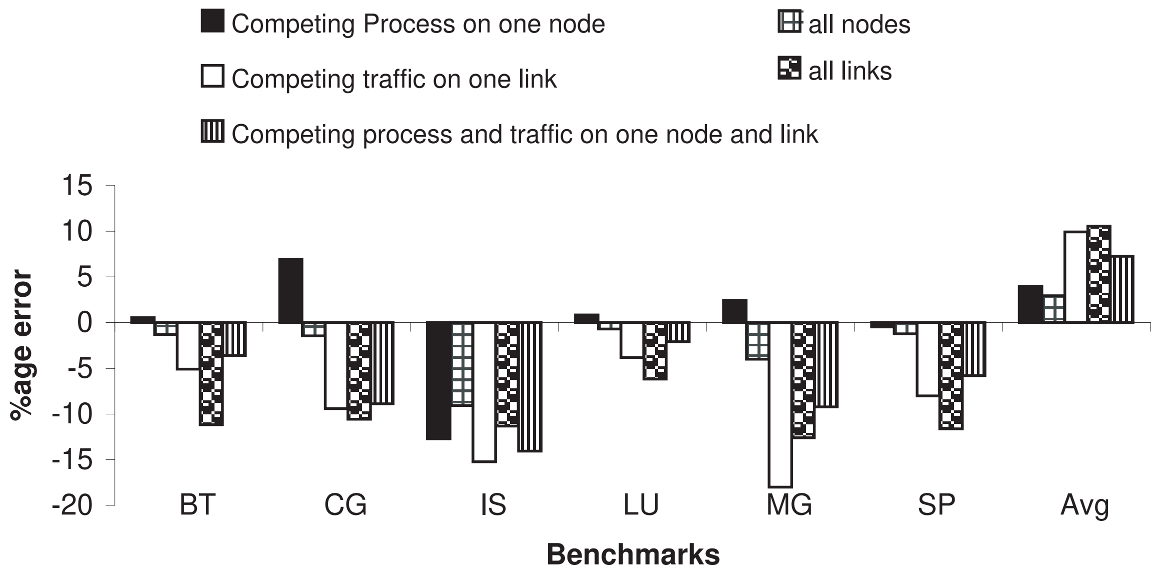

Figure 1. Percentage difference between predicted execution time based on performance skeletons and measured execution time for NAS benchmarks for 5 resource sharing scenarios. A negative error value implies that the execution time was overestimated while a positive value implies that it was underestimated. The average error over all benchmarks is computed with absolute error values.

Subsequently, the benchmarks and corresponding performance skeletons are run on the same testbed under the following five resource sharing scenarios:

1. Two competing compute intensive processes are run on one node.

2. Above two processes are run on all four nodes.

3. Available bandwidth on one of the links was reduced to 10Mbps using *iproute2*.

4. Bandwidth on all links was reduced to 10Mbps.

5. Competing processes as above on one node and bandwidth reduced on one link.

In each case we have

$$PredictedApplicationExecutionTime = L * SkeletonExecutionTime$$

The $ActualApplicationExecutionTime$ is obtained by direct measurement during actual application execution in the same scenario. We then determine the accuracy of prediction by comparing this predicted application execution time with the measured application execution time. The percentage difference between measured and predicted execution times are plotted in Figure 1.

We observe that prediction error is relatively small for all scenarios spanning a range between 0 and 18%. The average prediction error is 6%. The conclusion is that performance skeletons can effectively predict execution time, at least under these scenarios.

We also observe from Figure 1 that prediction error is generally higher for execution scenarios with competing traffic. We speculate that one of the reasons is the non-linear reduction in execution time of communication operations in some situations discussed earlier. Finally, different benchmark programs exhibit different levels of prediction error but no clear pattern emerges. Further analysis of the relationship between application characteristics and prediction error is beyond the scope of this paper but is discussed in [19].

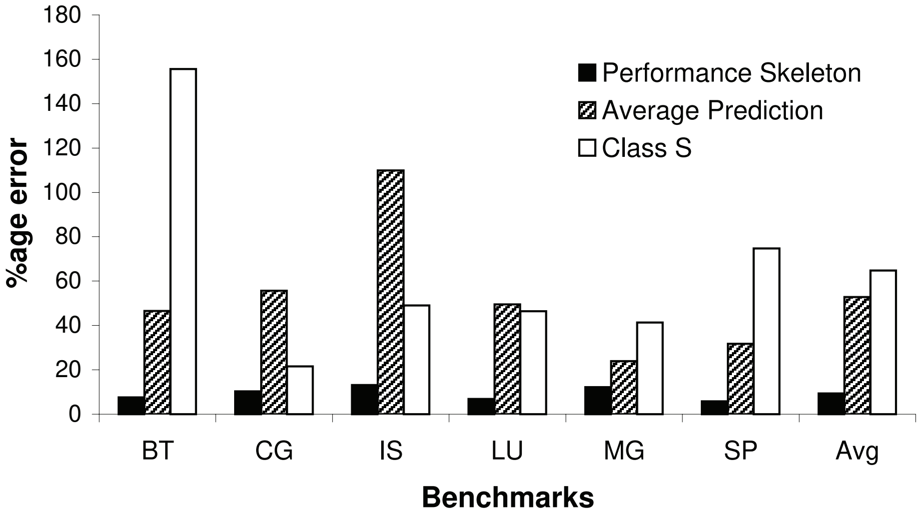

Figure 2. Performance prediction error with performance skeletons and other simple methods. Results are averaged over the 5 scenarios of node and/or link sharing stated earlier.

5.3 Comparison with other prediction approaches

We performed additional experiments to compare prediction accuracy of such performance skeletons versus two other "reasonable" approaches to performance prediction listed as follows:

Average Prediction: The average slowdown of the entire benchmark suite under a given resource sharing scenario was used to predict the execution time for every program in the same scenario. The reasoning is that, if all programs slow down roughly equally under resource competition, there is no need for customized performance skeletons for applications discussed in this paper. A generic short running program could be run to predict the execution time for any application under resource sharing.

Class S Prediction: The experiments described in this paper were performed on Class B NAS benchmarks, which run in 10s to 100s of seconds without load on 4 machines in our cluster. Each NAS benchmark also has a Class S version which runs in a few seconds. In this case, the Class S benchmarks were used as the performance skeletons for the Class B benchmarks for performance prediction. The reasoning is that since both classes of benchmarks perform the same fundamental calculations but on different data sizes and scales, the short running class S benchmarks could be considered good manually generated performance skeletons.

The performance prediction error for each of these approaches is plotted in Figure 2. Clearly, the performance skeleton approach to performance prediction performs much better than the others. The conclusion is that monitoring system activity as the basis for constructing performance skeletons is superior to other simple minded approaches. We would like to point out here that the Class S benchmarks execute in a few seconds which is generally significantly less time than the skeletons that are used for results presented in this paper. For a fairer comparison we should use skeletons that run as fast as the Class S benchmarks, which is a subject of ongoing experiments.

6 Discussion

We outline the limitations of the skeleton based approach to performance prediction as well as the limitations of our implementation.

Execution on an architecture different from the testbed used for skeleton creation: While creating a performance skeleton, we make no effort to reproduce the instructions in the original application or the memory access pattern. Hence the skeletons are not expected to predict performance accurately if used on nodes with a different memory or CPU architecture. This is being addressed in ongoing research.

Different number of nodes and data sets than the prototyping testbed: A skeleton models the application behavior for a certain number of executing nodes and hence cannot be employed directly for execution on a different number of nodes. The results presented in this paper use the same data sets for the prototyping run and performance estimation experiments. Construction of skeletons that can be *scaled* across number of nodes or data set sizes is a topic for future research.

Synchronization behavior: The current implementation of our framework does not accurately model the synchronization behavior of the corresponding full application. While constructing the skeleton we set the duration of a compute operation within a loop to its average duration across all iterations of the loop. This may cause the skeleton to display less synchronization delays as compared to the full application. We believe that this may be a cause for the high prediction error in some scenarios. A better approach would be to use the frequency distribution of compute durations instead of taking the average. Further experimentation is required to test this reasoning.

Wide area networks: All results presented in this paper are for a local cluster. Bandwidth sharing in a small cluster is simulated using artificial queues and is more uniform when compared with a wide area network, which may have several application streams competing for available bandwidth. However, we believe that performance skeleton based prediction approach will work for wide area networks if the sharing conditions are same (or similar) during skeleton and application execution. This is because the performance skeleton reproduces corresponding application's communication behavior and hence its performance will get affected in the same manner as that of the full application. In any case, extensive experimentation is needed for validation of this approach on wide area network and computation grids.

Adaptive applications: We assume that each application node performs the same amount of work independent of CPU and network conditions. Hence, if the application modified the work allotted to different nodes or changed its execution behavior due to changes in network or node conditions, e.g. a dynamically adjusting master-slave application, then prediction using the current implementation of our framework will not work. Clearly significant additional investigation is required to adapt this approach to dynamic load balancing and other adaptive applications.

7 Conclusions

This paper makes a case for automatically employing the knowledge of an application's periodic behavior to performance estimation in shared cluster and grid environments. A major problem in performance prediction and automatic node selection for shared network environments is the cost and accuracy of network and node usage information. We describe a performance estimation framework that does not directly require network or node usage information, thus eliminating the potential for estimation errors due to inaccurate network information. In our experiments the framework was effective in predicting the performance of programs in the NAS parallel benchmark suite under a variety of resource sharing conditions. Our system currently works for MPI message passing applications but is not fundamentally limited to any programming model since it analyzes the execution events of an application and requires no source code modification.

Clearly more work needs to establish and generalize this approach to performance prediction and its application to resource management and, this has been discussed in the previous section. However, we believe that this paper makes a compelling case that this new approach to performance prediction has clear advantages in shared network and grid computing.

8 Acknowledgments

This research was supported, in part, by the National Science Foundation under award number NSF ACI-0234328, the Department of Energy through Los Alamos National Laboratory (LANL) contract number 03891-99-23 , and by University of Houston's Texas Learning and Computation Center. We wish to thank numerous current and former members of our research group for their contribution to this work. Finally, the paper is much improved as a result of the comments and suggestions made by the anonymous reviewers.

References

[1] Linux man pages.

[2] W. Almesberger. Linux network traffic control — implementation overview.

White Paper, April 1999. Available at ftp://lrcftp.epfl.ch/pub/people/almesber/pub/tcio-current.ps.

[3] D. Bailey, T. Harris, W. Saphir, R. van der Wijngaart, A. Woo, and M. Yarrow. The NAS Parallel Benchmarks 2.0. Technical Report 95-020, NASA Ames Research Center, December 1995.

[4] F. Berman, R. Wolski, S. Figueira, J. Schopf, and G. Shao. Application-level scheduling on distributed heterogeneous networks. In *Proceedings of Supercomputing '96*, Pittsburgh, PA, November 1996.

[5] P. Bhatt, V. Prasanna, and C. Raghavendra. Adaptive communication algorithms for distributed heterogeneous systems. In *Seventh IEEE Symposium on High-Performance Distributed Computing*, Chicago, IL, July 1998.

[6] J. Bolliger and T. Gross. A framework-based approach to the development of network-aware applications. *IEEE Trans. Softw. Eng.*, 24(5):376 – 390, May 1998.

[7] H. Casanova, G. Obertelli, F. Berman, and R. Wolski. The AppLeS Parameter Sweep Template: User-level middleware for the grid. In *Supercomputing 2000*, pages 75–76, 2000.

[8] I. Foster and K. Kesselman. Globus: A metacomputing infrastructure toolkit. *Journal of Supercomputer Applications*, 11(2):115–128, 1997.

[9] A. Grimshaw and W. Wulf. The Legion vision of a worldwide virtual computer. *Communications of the ACM*, 40(1), January 1997.

[10] M. Litzkow, M. Livny, and M. Mutka. Condor — A hunter of idle workstations. In *Proceedings of the Eighth Conference on Distributed Computing Systems*, San Jose, California, June 1988.

[11] B. Lowekamp, N. Miller, D. Sutherland, T. Gross, P. Steenkiste, and J. Subhlok. A resource query interface for network-aware applications. In *Seventh IEEE Symposium on High-Performance Distributed Computing*, Chicago, IL, July 1998.

[12] B. Lowekamp, D. O'Hallaron, and T. Gross. Direct queries for discovering network resource properties in a distributed environment. *Cluster Computing*, 3(4):281–291, 2000.

[13] C. Lu and D. A. Reed. Compact application signatures for parallel and distributed scientific codes. In *Proceedings of Supercomputing 2002*, Baltimore,MD, Nov 2002.

[14] G. Shao, F. Berman, and R. Wolski. Master/slave computing on the grid. In *9th Heterogeneous Computing Workshop*, pages 3–16, 2000.

[15] T. Sherwood, E. Perelman, and B. Calder. Basic block-dsitribution analysis to find periodic behavior and simulation points in applications. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, Sep 2001.

[16] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder. Automatically characterizing large scale program behavior. In *10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS-X)*, San Jose, CA, October 2002.

[17] A. Singh and J. Subhlok. Reconstruction of application layer message sequences by network monitoring. In *IASTED International Conference on Communications and Computer Networks*, Boston, MA, November 2002.

[18] A. Snavely, N. Wolter, and L. Carrington. Modeling application performance by convolving machine signatures with application profiles. In *IEEE Workshop on Workload Characterization*, Austin, TX, 2001.

[19] S. Sodhi. Automatically constructing performance skeletons for use in grid resource selection and performance estimation frameworks. Master's thesis, University of Houston, Jan 2004.

[20] J. Subhlok, P. Lieu, and B. Lowekamp. Automatic node selection for high performance applications on networks. In *Proceedings of the Seventh ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, pages 163–172, Atlanta, GA, May 1999.

[21] J. Subhlok, S. Venkataramaiah, and A. Singh. Characterizing NAS benchmark performance on shared heterogeneous networks. In *11th International Heterogeneous Computing Workshop*, Fort Lauderdale, FL, April 2002.

[22] M. Swany and R. Wolski. Multivariate resource performance forecasting in the network weather service. In *Supercomputing 2002*, November 2002.

[23] T. Tabe and Q. Stout. The use of the MPI communication library in the NAS Parallel Benchmark. Technical Report CSE-TR-386-99, Department of Computer Science, University of Michigan, Nov 1999.

[24] S. Venkataramaiah and J. Subhlok. Performance prediction for simple CPU and network sharing. In *LACSI Symposium 2002*, Santa Fe, NM, October 2002.

[25] S. Venkataramaiah and J. Subhlok. Performance estimation for scheduling on shared networks. In *9th Workshop on Job Scheduling Strategies for Parallel Processing*, Seattle, WA, June 2003.

[26] J. Weismann. Metascheduling: A scheduling model for metacomputing systems. In *Seventh IEEE Symposium on High-Performance Distributed Computing*, Chicago, IL, July 1998.

[27] R. Wolski, N. Spring, and C. Peterson. Implementing a performance forecasting system for metacomputing: The Network Weather Service. In *Proceedings of Supercomputing '97*, San Jose, CA, Nov 1997.

[28] S. Zhou. LSF: load sharing in large-scale heterogeneous distributed systems. In *Proceedings of the Workshop on Cluster Computing*, Orlando, FL, April 1992.