Automatic Construction and Evaluation of Performance Skeletons

Sukhdeep Sodhi and Jaspal Subhlok
University of Houston
Department of Computer Science
Houston, TX 77204
{ssodhi,jaspal}@uh.edu

Abstract

The performance skeleton of an application is a short running program whose execution time in any scenario reflects the estimated execution time 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 a framework for automatic construction of performance skeletons of a specified execution time and evaluates their use in performance prediction with CPU and network sharing. The approach is based on capturing the execution behavior of an application and automatically generating a synthetic skeleton program that reflects that execution behavior. The paper demonstrates that performance skeletons running for a few seconds can predict the application execution time fairly accurately. Relationship of skeleton execution time, application characteristics, and nature of resource sharing, to accuracy of skeleton based performance prediction, is analyzed in detail. The goal of this research is accurate performance estimation in heterogeneous and shared computation environments.

1. Introduction

Computational grids are emerging as the vehicle for future high performance scientific and commercial computing. Execution environments for grids have to address allocation of resources to applications, and that is driven by the expected performance of an application on different parts of a grid. Estimation of application performance has an important role to play in grid computing, and the problem is much more complex for a shared heterogeneous computation environment than for conventional high performance computing platforms.

The research community clearly recognizes the importance of performance estimation in grid environments and substantial research effort has been invested in measurement, modeling, and prediction of various system resources. Measurement and prediction of CPU availability has been studied in [11, 33]. Measurement and modeling of network bandwidth and latency is a very active area of research [7, 14, 18, 26]. NWS (Network Weather Service) [34] and REMOS (Resource Monitoring System) [16] are two systems that have been specifically designed for measurement of available CPU and network resources in grid environments. NWS, in particular, is in widespread use as a CPU and bandwidth monitoring and prediction tool.

Systems for resource management and scheduling for problem solving on grid environments include Netsolve [8], Nimrod/G [6], Gallop [32], AppLeS [4] and Condor [15, 19]. These systems rely on measured and predicted availability of CPU, bandwidth and other resources to make resource allocation and management decisions where applicable. AppLeS [4] pioneered application level scheduling, where resource selection is performed by agents associated with an application based on available resource information, rather than by a central resource manager. A number of algorithms and frameworks have been proposed for resource selection in networked environments based on system status information, a few examples being [4, 27]. Some of the recent research has emphasized the importance of application properties in resource allocation and addresses resource selection based on mapping application properties to the system status [5, 9, 20, 24, 29, 31].

While the research discussed above represents many different directions, the state of the art approach to resource selection for applications can be broadly summarized as consisting of the following steps:

1. *System characterization:* Measure and predict the status and availability of system resources such as CPU and network capacities.

2. *Application characterization:* Develop a model that captures the dependence of an application's performance on availability of resources.

3. *Mapping and scheduling:* Select the best nodes to execute the application based on available system status and application characteristics.

We argue that this state of the art has the following inherent limitations that motivate a different paradigm:

- *Maintaining accurate current system status information is inherently expensive*. In order to have recent CPU and network information whenever a resource assignment decision has to be made, available system resources have to be monitored continuously and status information has to be broadcast frequently. For network properties, measurements themselves consume bandwidth and the complexity increases quadratically with the size of the available grid. High speed backbone network links are particularly challenging, especially since it is not desirable to consume a critical shared resource for measurements.
- *Estimating application performance based on system status is inherently error prone.* Measurement tools provide resource availability and utilization information such as CPU load factor and unused bandwidth on various components of a grid. On the other hand, the key information of interest for resource management is how a particular application will perform on a set of resources under current system status. Predicting the performance of application tasks from system status information is very difficult. The following examples underline the complexity:
  - The amount of CPU time that a process is likely to get on a computation node cannot be determined even when the load average on the node is known since it partly depends on the synchronization structure of the parallel and distributed applications in the system.
  - The expected duration of a bulk transfer cannot be estimated even when accurate point to point unused bandwidth information is available since it depends on the transport protocols used by the application and other traffic on the network.

  Finally even if performance of individual node computations and data transfers can be determined, estimating collective communication and overall application performance is still challenging as it depends on the nature of sharing in the network and the application structure.

The conclusion is that it is virtually impossible to estimate application performance from network status in many scenarios. This has motivated us to follow a different approach to estimating performance in shared heterogeneous grid environments which is based on the claim:

The most effective and efficient way to estimate the performance of an application under the existing status of grid resources is brief monitored execution of code that mimics the application.

We refer to such code as the **performance skeleton** of the application. More formally, a performance skeleton is a synthetically generated short running program whose execution time always reflects the performance of the application it represents. Hence, simply executing the performance skeleton in a shared execution environment provides an estimate of application performance in that environment. The resource selection for an application is then addressed as follows. A group of candidate node sets is identified for execution (using existing approximate methods) and the final choice is made by comparing the execution time of the application skeleton on each node set.

The central contribution of this paper is a framework for automatic construction of accurate performance skeletons for distributed applications and evaluation of the capability of automatically generated skeletons to predict performance efficiently and accurately.

While we have used resource selection in shared grid environments for motivation of this research, it is important to point out that this approach to performance prediction has broad applicability. Another example is prediction of the performance of important applications on a future architecture under simulation. Since execution under simulation is multiple orders of magnitude slower than real execution, this skeleton based approach can be particularly appropriate. The real application does not have to be simulated at all as the skeleton can be built on existing machines.

The basic philosophy in construction of a performance skeleton is that if the skeleton executes operations that are representative of application execution, the performance of the skeleton and the application will change similarly in response to changes in the execution environment. Hence, a performance skeleton must capture the execution behavior of the application in terms of synchronization and message exchange patterns, CPU usage patterns, and memory access patterns, yet execute for a very short time only. Our approach is to measure the application performance behavior during execution, summarize it by identifying repeating phases, and then reproduce it as a synthetic skeleton program.

We briefly discuss other projects that summarize application behavior and their goals. Reed et.al. [23, 17] generate compact application signatures using a curve-fitting approach to reduce event-tracing overheads for online performance monitoring and tuning. Snavely et.al. [24] create application and machine signatures to simulate application behavior across different system or processor architectures. Duesterwald et.al. [12] identify phase behavior for kernel-level resource aware scheduling. Sherwood et.al. [21, 22]

exploit periodic application behavior to identify portions of the program that are representative of an application for the purpose of architectural simulations. Our approach is driven by many of the ideas and concepts developed in these projects. However, we have a very different goal, which is to develop an independent skeleton program. An alternate approach is explored in FAST [10], a tool that performs abstract simulations while completely avoiding execution of computation code. This approach ignores program control flow, which can impact the communication pattern and computation time. FAST also requires significant modification to the source program, while our approach does not require access to the source code.

2. Performance skeletons

A performance skeleton is defined as a program whose execution time is directly related to the execution time of the application it represents; if the execution time of a skeleton is 1/1000th of the application execution time on a dedicated cluster, then this relationship should hold in any execution environment even when nodes and links are shared with other applications. This definition is idealistic, and in practice, the goal is to build a skeleton that conforms to these conditions as closely as possible. The skeleton should also be as short-running as possible as skeleton execution is an overhead. We would like to point out that skeleton execution is very 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.

For the performance behavior of a skeleton to be similar to that of an application, the execution and resource usage patterns of the skeleton must be similar to the dominant corresponding patterns of the application. We have the following specific criteria:

1. *CPU activity:* The processing done by the CPU and CPU busy/idle phase pattern should be similar for the application and the skeleton.
2. *Memory activity:* The memory access pattern in the skeleton should be representative of the application. This is particularly important to get similar cache performance on nodes with different memory hierarchies.
3. *I/O activity:* The I/O pattern in the skeleton should be representative of the application.
4. *Communication and synchronization:* The data exchange patterns among processes should be similar for the application and skeleton to preserve the communication and synchronization performance. The sizes, types, frequencies and the global patterns of the network messages exchanged should be similar.
5. *Application phase transitions:* An application transitions between different phases of execution at multiple levels of granularity. The sequence of these phases, as well as the CPU, memory and communication activities in each phase, should be reflected in the skeleton.

Our long term project goal is to generate skeletons conforming to the above constructive definition but this paper is limited to performance skeletons which mimic the communication sequences and coarse computation behavior of the application. Such skeletons are sufficient for predicting performance of compute and communication bound applications under resource sharing. Reproduction of memory accesses and fine-grain instruction level computation behavior is critical for performance estimation across different processor and memory architectures, but not essential for simple CPU and network sharing scenarios. We discuss our efforts in reproducing memory behavior for performance prediction in [30].

3. Automatic construction of skeletons

We have developed a framework for automatic construction of performance skeletons and implemented it for message passing MPI programs. We outline the procedure in this section; details and algorithms employed are available in [25]. The main steps are as follows:

1. *Record application's execution trace:* The application is executed on a controlled testbed and its execution activity, specifically CPU usage and message exchanges, is recorded. This is the ***execution trace***.
2. *Compress execution trace into an execution signature:* The repeated patterns in the recorded execution trace are identified and used to generate a compact representation of the trace by introducing a "loop structure". The new compact representation is the ***execution signature.***
3. *Generate performance skeleton program from the execution signature:* The application execution signature is converted to a computer program which generates execution activity that is similar to the recorded execution signature but with execution time scaled down by a given factor K. This is the ***performance skeleton.***

This skeleton construction procedure is illustrated in Figure 1. This procedure does not involve source code analysis, modification or instrumentation and hence has broad applicability. The skeleton construction details are driven by the desired ratio between the execution time of the application and the corresponding performance skeleton, which we call the *scaling factor*. We now discuss the steps in more detail.

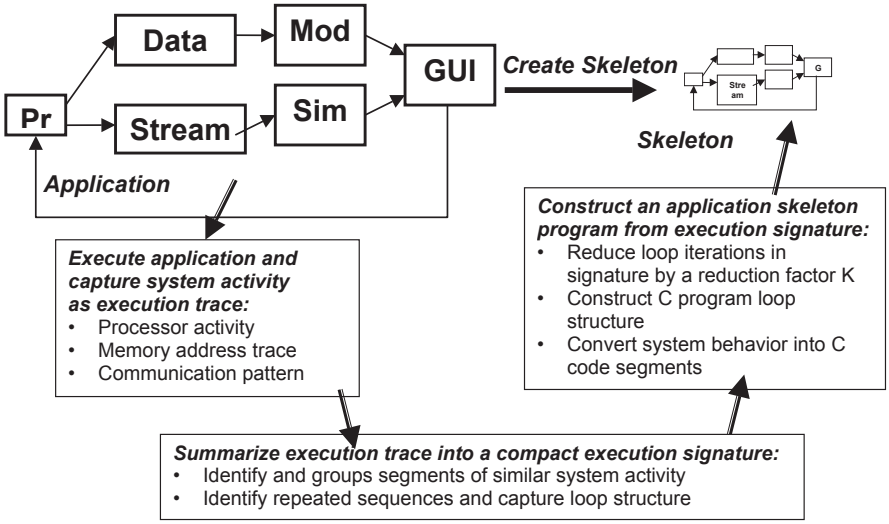

Figure 1. Construction of application performance skeletons.

3.1. Recording of execution trace

To generate the execution trace of an MPI application, it is linked with a profiling library developed for this purpose and executed on a dedicated testbed without any competing processes or network traffic. The profiling library records information for each application process in a separate trace file. Each MPI library call, along with the parameters passed to it and its start time and end time, are recorded. Timing measurement is done to microsecond granularity with 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. Generation of the trace file requires no modification of the application source code. We verified that the execution time overhead of trace generation is negligible, typically well under 1% of the execution time.

3.2. Compression of execution trace to execution signature

The application execution trace is a long record of message exchanges and interleaved compute operations of varying duration. Most of the time during application execution is typically spent in repeating loops as application execution activity tends to be cyclic. The goal of this step is to identify cyclic behavior in the execution trace to generate a compact execution signature. While establishing repeating behavior we look for segments of the execution trace with similar activity rather than exactly identical segments. This process consists of clustering similar execution events in the trace followed by conversion of repeated operation sequences into a loop structure.

Clustering similar execution events:

The objective of this stage is to replace the execution trace by a string of symbols where substantially similar execution events are placed in one cluster and assigned the same symbol.

As an example, suppose we encounter the following two operations in a trace:

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 or message calls with similar parameters. Our approach treats different MPI primitives and blocking and non-blocking calls as distinct events, thus ensuring that they are never grouped together. We identify the non blocking calls and associated *MPI_Wait()* to determine the corresponding overlapped region. This helps develop a faithful representation of the application's communication structure.

Formally we have developed a measure for dissimilarity of events in N– dimensional space based on [13], with one dimension for each parameter of an execution event. The extent of clustering is controlled by a *similarity threshold* which can be assigned a value between 0 and 1. A lower similarity threshold represents more strict rules for clustering, but will lead to less compression. A threshold of 0 implies that only identical events are clustered together.

This stage converts the trace log into a string of

symbols such as:

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

where each occurrence of a symbol represents an execution event with different occurrences of the same symbol referring to functionally identical execution events.

Clustering of similar events and representing them by an "average event" implies some loss of information but leads to significant compression, and subsequently, smaller skeletons. This tradeoff can be managed with the similarity threshold parameter.

Identification of cycles:

The objective of this step is to identify repeated execution segments and compress them as loops. Since the previous step converts the execution trace into a sequence of frequently repeating symbols, the problem of identifying repeating application behavior is now represented as the problem of finding repeating substrings within a string. As an example, the following string:

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

should be replaced by:

$\alpha[(\beta)^2\gamma]^3\kappa[\alpha]^2$

We have developed an algorithm [25] which recursively identifies all the repeating sub-strings, starting with the largest matches and working down to substring matches of a single symbol. The repeating substrings are then organized as recursive loop nests with sub-strings of symbols as loop bodies and the number of repetitions as the number of loop iterations.

We now address how a given value of similarity threshold translates to specific rules for compression and then discuss how the value of similarity threshold is determined. For message passing operations, the value of the similarity threshold linearly relates to the maximum difference in message sizes allowed for communication operations to be combined into a cluster. The above compression procedure is applied across communication operations without regard to interleaving computations. When two sequences of communication events with interspersed computation events are to be combined, an average value of execution time for the corresponding computation events in the sequence is used to build the compressed sequence. This approach represents maximum flexibility in combining computation events but was found to be effective in our experience.

An iterative process is employed to determine the optimal value of the similarity threshold based on the desired compression ratio Q between the length of the execution trace and the length of the compressed execution signature. Initially the similarity threshold is set to 0 and the clustering and compression procedure is applied. If the degree of compression is less than the desired ratio Q, the similarity threshold is increased gradually until the desired compression of Q (or higher) is achieved. Now, the question is how should Q be determined ? Based on our experience, we have used Q = K/2 where K is the scaling factor between the application execution time and the desired skeleton execution time. It is desirable to have an upper bound on similarity threshold so that very different execution events are not combined. In practice, this may not be a significant issue. The maximum similarity threshold that was required across the NAS benchmarks for meaningful experiments was always less than .20 which we consider acceptable.

3.3. Generation of performance skeleton program from execution signature

The previous stage gave us the execution signature which is a compressed record of the complete execution of the application. The execution signature compresses execution information by using a loop structure with loop bodies representing repeating execution behavior. Our goal in this step is to create a short running program in a programming language like C/C++ which reproduces the scaled down dominant execution behavior represented by the execution signature. The specific goal is to take the application's execution signature and the desired *scaling factor* K as inputs, and generate an appropriate performance skeleton. The skeleton construction procedure is outlined as follows:

1. The numbers of loop iterations in the application signature are reduced by a factor K. Loop iterations that form the remainder in this division process are unrolled and become a component of the unreduced part of the signature.
2. Groups of K occurrences of identical execution operations anywhere in the unreduced part of the skeleton are identified and replaced by a single occurrence.
3. All remaining unreduced operations are *scaled down* by a factor K by adjusting their parameters. For compute operations, the duration of execution is reduced by a factor K. For communication operations, the number of bytes exchanged is reduced by a factor K.
4. This modified application signature is converted to synthetic C code by generating corresponding synthetic loops, MPI calls, and compute operations.

More details are available in [25]. 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 typically 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 for operations not in loops. In practice, the impact on overall performance estimation is expected to be minimal for most applications.

3.4. Shortest running "good" skeleton

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. The framework we have developed is designed to construct skeletons for any scaling factor that is provided, and equivalently, for an arbitrary skeleton execution time. A key question in this research is as follows: How short running can a skeleton be and still generate reasonable performance estimates ?

To address this, the skeleton construction framework heuristically determines the shortest runtime skeleton that it believes can be constructed without significantly sacrificing prediction accuracy, and issues a warning if the requested scaling factor implies a smaller skeleton. To determine the shortest "good" skeleton, the framework identifies the *dominant sequence of execution events* in the application that comprise a significantly large percentage of application execution time. A skeleton is considered a good skeleton if at least one full iteration of the dominant sequence of execution events is included.

As an example consider the NAS IS (Integer Sort) benchmark whose main communication operation is a large all-all transfer. The accuracy of the skeleton is expected to be good if full all-all transfers are included. Hence the minimum size for a good skeleton will be the shortest skeleton that includes at least one full all-all transfer.

4. Experiments and results

A prototype framework for automatic construction of performance skeletons has been implemented. We used it to generate skeletons to predict the performance of the corresponding applications on a network testbed.

4.1. 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, 28]. 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.

4.2. Experiments conducted

Performance skeletons were constructed for each Class B NAS Benchmark program with an intended skeleton execution time of 10 seconds, 5 seconds, 2 seconds, 1 second and 0.5 second by defining the appropriate scaling factors.

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

1. Two competing compute intensive processes are run on one node.
2. Two competing compute intensive processes are run on each node.
3. Available bandwidth on one of the links was reduced to 10Mbps using *iproute2*.
4. Available bandwidth on each link was reduced to 10Mbps.
5. Competing processes as above on one node and reduced bandwidth as above on one link.

(Note that at least two processes are required to create significant CPU contention on dual processor nodes.)

For each application, the execution time was predicted for each resource sharing scenario and each skeleton as the product of the skeleton execution time and the corresponding *measured scaling ratio*. The measured scaling ratio is similar to the scaling factor except that actual skeleton execution time on a dedicated testbed is used which can be slightly different from the intended execution time for which the skeleton was constructed. The predicted execution time was compared to the actual measured application execution time for all scenarios. We now present the results.

4.3. Validation of skeleton properties

The performance skeletons are expected to have execution behavior that reflects the application. As a basic test,

we compared the percentage of time spent in the communication (MPI) operations versus other computations for the skeletons and the application. The results are illustrated in Figure 2.

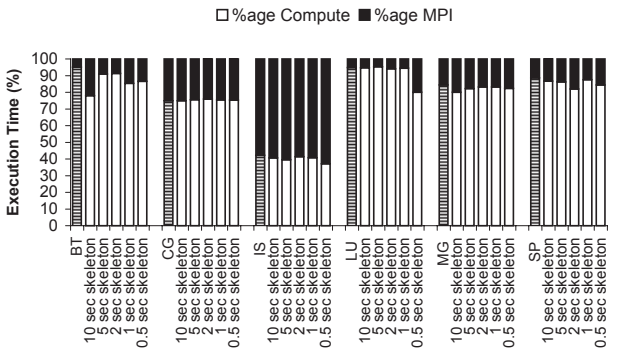

Figure 2. Time spent by NAS benchmarks and corresponding skeletons in different execution activities. The bar with horizontal lines is for the actual application.

We observe that the ratio between the computation and communication time is broadly similar for the skeletons and the corresponding application. The 0.5 second skeleton for the LU benchmark shows a somewhat larger communication time ratio than the other cases. We expect that very small skeletons will not represent the application as faithfully as larger skeletons as more approximations are involved in their construction. The ratios for the skeletons of BT benchmark show more variation than others. The conclusion is that moderate variations are possible because of the nature of skeleton construction process but most skeletons are fairly close to their application in this respect.

4.4. Validation of performance prediction

Average error in execution time predicted by the performance skeletons across applications and skeleton sizes is plotted in Figure 3. These results are averaged across resource sharing scenarios. We observe that the average prediction error across all benchmarks, scenarios and skeleton sizes is a relatively low 6.7% implying that the performance skeletons can predict execution time effectively. We now discuss the relationship of prediction accuracy to application characteristics, skeleton size and resource sharing scenarios.

Skeleton size and benchmarks: Our goal of "short running" performance skeletons is to reduce overheads but preserve prediction accuracy. From Figure 3 we observe that

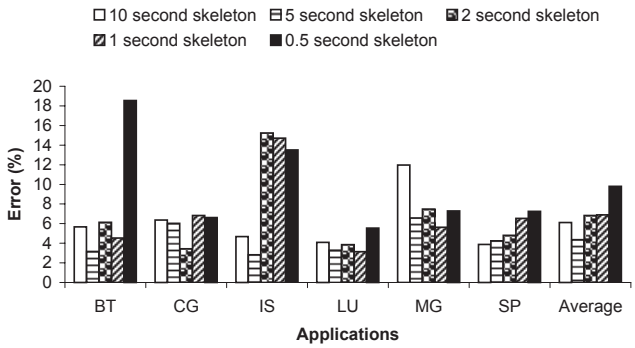

Figure 3. Prediction error for NAS benchmarks across skeletons sizes from 10 to 0.5 seconds. The error is averaged across all resource sharing scenarios.

the relationship between average prediction error and skeleton size shows no distinct pattern across benchmarks. For some benchmarks, prediction error does not change much when going from 10 second to 0.5 second skeletons. However, error is usually close to the highest for the smallest 0.5 second skeletons. The average error across all applications for 0.5 seconds skeletons is around 8% versus the range around 5% to 6% for other cases.

The minimum execution times of a "good" skeleton for each benchmark as determined by our framework, based on discussion in section 3.4, is listed in Figure 4. According to this table, the skeletons that are flagged as potentially "not good" are 0.5 and 1 second skeletons for BT, 0.5, 1, and 2 second skeletons for IS, and 0.5 and 1 second skeletons for LU. Indeed the 4 cases with the highest prediction error, i.e., the 0.5 second BT skeleton and 0.5,1, and 2 second IS skeletons, were flagged to have low prediction value by the skeleton construction framework.

Application	Smallest Skeleton
BT	1.01 sec
CG	0.13 sec
IS	3 sec
LU	1.97 sec
MG	0.34 sec
SP	0.34 sec

Figure 4. Estimated minimum execution time for the smallest good skeleton.

The prediction errors for each size skeleton are grouped together and displayed in Figure 5. While there is no uniform pattern again, the number of cases with a relatively

large prediction error increase with reduced skeleton sizes and are clearly higher for 0.5 second skeletons.

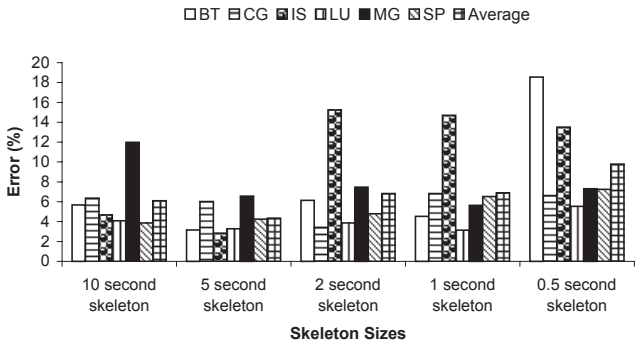

Figure 5. Prediction error for skeletons of different sizes for NAS benchmarks. The error is averaged across all resource sharing scenarios.

The main conclusion is that performance skeletons of a few seconds are normally adequate for reasonably accurate performance prediction, with a loose correlation between smaller skeletons and lower prediction accuracy. Also, the framework generates meaningful application specific lower bounds for skeleton sizes below which the prediction power of a skeleton is less reliable.

Sharing scenarios: We examine how the nature of sharing relates to accuracy of performance prediction. Our experiments have spanned sharing of one or all CPUs, one or all communication links, and a combination of one node and one link. Figure 6 shows prediction error under different sharing scenarios when employing representative 10 second skeletons. We observe that the prediction error is higher for scenarios that include competing traffic. In the case of CPU sharing only, the error is higher for the "unbalanced" sharing of a single node versus sharing of all nodes.

We believe that prediction error is higher for network sharing because communication operations cannot be scaled down linearly unlike compute operations, as discussed in section 3.3. We speculate that the error in unbalanced execution scenarios is higher because of potential inaccuracy in reproduction of synchronization behavior in performance skeletons. While constructing a skeleton we set the duration of compute operations within loops to their average duration across iterations of the loop. A more accurate approach that considers frequency distribution of the duration of compute events will be taken in the future.

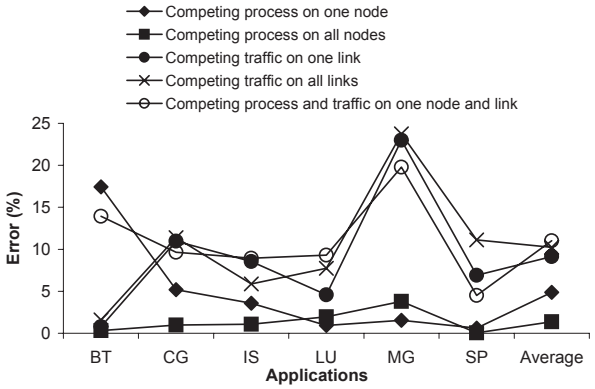

Figure 6. Prediction error for NAS benchmarks across five resource sharing scenarios. A 10 second skeleton was used.

4.5. Comparison with other prediction techniques

We performed additional experiments to compare prediction accuracy of such performance skeletons versus two other simple and "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; instead, 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 with Class B NAS benchmarks, which run in 30 to 900 seconds without load on 4 machines in our cluster. Each NAS benchmark also has a Class S version which typically runs in less than a second. 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 calculation 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 7. The performance skeleton approach based on the framework in this paper is clearly better than the other methods. Prediction with 0.5 second skeletons, which roughly take as long to run as Class

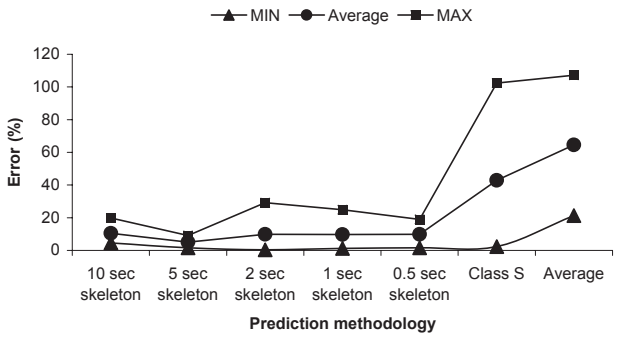

Figure 7. Minimum, maximum and average prediction error for the NAS benchmark suite for prediction with different size skeletons, with class S benchmarks as skeletons, and using average prediction. The execution scenario is one competing process on one node and traffic on one link

S benchmarks, is also clearly superior to other methods. Hence the overhead of our approach is also competitive.

The above results are significant for the following reasons. The fact that "average prediction" approach is relatively ineffective proves that applications have widely varying execution behavior and hence an approach that is customized to applications is required. The inability of Class S benchmarks to predict the behavior of larger Class B benchmarks shows that one cannot simply run an application with a very small input data set and expect it to have similar execution behavior as running with realistic data sets.

5. Limitations and extensions

Our current framework is only a step toward a comprehensive solution for execution driven performance prediction and has many limitations. Prediction across CPU and memory architectures cannot be made without better modeling of instruction level execution and memory access patterns. The current implementation is limited to MPI programs. The implementation can be improved to better manage scaling down of communication. More experimentation, particularly on wide area networks is needed for stronger validation. Additional work is needed to scale predictions across different numbers of processors and different size data sets. However, the current framework is effective for performance prediction in basic shared execution environments.

6. Concluding remarks

This research is in the direction of the broader topic of resource selection and performance estimation in grid computing environments. We believe that knowledge of an application's cyclic behavior can be effectively employed in grid performance estimation and resource selection frameworks. This paper introduces performance skeletons, which are short running programs that can be used for performance estimation with resource sharing. The main advantage of a performance skeleton based approach is that it avoids the cost and inaccuracy associated with determining up-to-date information regarding node and network usage and translating it to expected application performance.

We demonstrate that automatically generated performance skeletons that run in seconds can predict application performance accurately, and that our framework can effectively compute the size of the shortest possible "good" skeleton that can estimate performance accurately. The paper also offers insight into how application characteristics, skeleton size and nature of resource competition impact prediction accuracy. In summary, the paper presents a promising approach to performance estimation with resource sharing and provides convincing evidence that it is practical and effective.

7. Acknowledgments

This material is based upon work supported by the National Science Foundation under Grant No. ACI-0234328 and Grant No. CNS-0410797. Support was also provided by 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.

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] 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.

[6] R. Buyya, D. Abramson, and J. Giddy. Nimrod/g: An architecture for a resource management and scheduling system in

a global computational grid. In *The 4th International Conference on High Performance Computing in Asia-Pacific Region*, 2000.

[7] N. Cardwell, S. Savage, and T. Anderson. Modeling TCP latency. In *Proceedings of IEEE INFOCOM 2000*, pages 1742–1751, 2000.

[8] H. Casanova and J. Dongarra. NetSolve: A network-enabled server for solving computational science problems. *The International Journal of Supercomputer Applications and High Performance Computing*, 11(3):212–223, Fall 1997.

[9] 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.

[10] M. Dikaiakos, A. Rogers, and K. Steiglitz. Fast: A functional algorithm simulation testbed. In *International Conference On Parallel and Distributed Systems*, December 1993.

[11] P. Dinda and D. O'Hallaron. An evaluation of linear models for host load prediction. In *Proceedings of the 8th IEEE International Symposium on High Performance Distributed Computing*, August 1999.

[12] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior abd its variability. In *International Conference on Parallel Architectures and Compilation Techniques (PACT)*, New Orleans, LA, September 2003.

[13] J. Han and M. Kamber. *Data Mining: Concepts and techniques*. Morgan Kaufman Publishers, 2001.

[14] K. Lai and M. Baker. Nettimer: A tool for measuring bottleneck link bandwidth. In *USENIX Synposium on Internet Topology and Systems*, pages 123–134, March 1991.

[15] 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.

[16] 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.

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

[18] V. Paxson and S. Floyd. Wide-area traffic: The failure of Poisson modeling. *IEEE/ACM Transactions on Networking*, 3(3):226–244, June 1995.

[19] R. Raman, M. Livny, and M. Solomon. Matchmaking: Distributed resource management for high throughput computing. In *7th IEEE International Symposium on High Performance Distributed Computing*, july 1998.

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

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

[22] 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.

[23] A. Snavely, L. Carrington, N. Wolter, J. Labarta, R. Badia, and A. Purkayastha. A framework for performance modeling and prediction. In *Proceedings of Supercomputing 2002*, Baltimore,MD, Nov 2002.

[24] 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.

[25] S. Sodhi. Automatic construction of performance skeletons for grid resource selection and performance estimation frameworks. Master's thesis, University of Houston, Jan 2004.

[26] M. Stemm, S. Seshan, and R. Katz. Spand: Shared passive network performance discovery. In *USENIX Symposium on Internet Technologies and Systems*, Monterey, CA, June 1997.

[27] 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.

[28] 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.

[29] H. Tangmunarunkit and P. Steenkiste. Network-aware distributed computing: A case study. In *Second Workshop on Runtime Systems for Parallel Programming (RTSPP)*, Orlando, March 1998.

[30] A. Toomula and J. Subhlok. Replication memory behavior for performance prediction. In *LCR 2004: The 7th Workshop on Languages, Compilers, and Run-time Support for Scalable Systems*, Houston, TX, October 2004.

[31] 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.

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

[33] R. Wolski, N. Spring, and J. Hayes. Predicting the CPU availability of time-shared unix systems on the computational grid. *Cluster Computing*, 3(4):293–301, 2000.

[34] 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.