Cross-Platform Performance Prediction of Parallel Applications Using Partial Execution

Leo T. Yang[*] Xiaosong Ma[†*] Frank Mueller[*]

[*] Department of Computer Science, North Carolina State University, Raleigh, NC, 27695-7534
[†] Computer Science and Mathematics Division, Oak Ridge National Laboratory
{tyang2,ma,mueller}@csc.ncsu.edu

ABSTRACT

Performance prediction across platforms is increasingly important as developers can choose from a wide range of execution platforms. The main challenge remains to perform accurate predictions at a low-cost across different architectures.

In this paper, we derive an affordable method approaching **cross-platform performance translation** based on **relative performance** between two platforms. We argue that relative performance can be observed without running a parallel application in full. We show that it suffices to observe very short **partial executions** of an application since most parallel codes are iterative and behave predictably manner after a minimal startup period. This novel prediction approach is observation-based. It does not require program modeling, code analysis, or architectural simulation. Our performance results using real platforms and production codes demonstrate that prediction derived from partial executions can yield high accuracy at a low cost. We also assess the limitations of our model and identify future research directions on observation-based performance prediction.

1. INTRODUCTION

Problem Overview:

Studying application performance has been in the limelight of high-performance computing (HPC) for a long time. In this realm, formal performance modeling and simulation have played a central role [1, 2, 6, 8, 9, 13, 15, 18, 21, 24, 28, 31, 34, 36, 37]. While model-based prediction has become indispensable for future architectures, its accuracy is increasingly compromised for existing architectures due to their complexity and heterogeneity. In particular, HPC users need *cross-platform performance prediction* to compare the efficiency of hardware platforms, to guide acquisitions of new systems, and to steer the development of next-generation hardware. The growing variety, scale, and complexity for applications and platforms alike have made traditional model- or simulation-based approaches difficult and expensive to be deployed across platforms.

This paper contributes a low-cost, high-accuracy approach to cross-platform performance prediction that is highly appealing due to is stunning simplicity. While this approach is shown to succeed for a large number of HPC applications and hardware platforms, we also assess its limitations and identify future directions for our work. The strengths of these contributions are on the breadth of our experimentation and the simplicity, wide applicability, and portability of our methodology.

Motivation:

Scientists today have access to an increasing number of geographically distributed HPC resources. It is essential for them to determine the performance of their specific applications on a wide range of execution platforms in deciding which system best fits their needs (both for acquisition and account application purposes), especially for long production runs. If scientists were provided with performance estimations for their applications on several large clusters, their choosing and gaining accesses to suitable platforms would be considerably simplified. In addition, they could be guided in their quantitative request for service units on a specific cluster.

Accurate performance estimates are also instrumental in assisting a grid resource scheduler to efficiently schedule user jobs. A “meta-scheduler” would benefits from knowledge about the runtime of jobs on each participating site in improving the global load balance and overall job throughput. When a job is scheduled to run at a specific grid-participating site, the local job scheduling system there also requires an estimated upper bound on its execution time. This bound is then utilized in scheduling decisions [25, 30]. Overly inaccurate estimations can result in excessive wait times in queues or in forced premature job termination (cancellation) during execution for over- and under-estimations, respectively.

Performance prediction for parallel programs has a large body of prior work (see Section 2). These prior efforts have mostly focused on performance modeling or program simulation. However, the *size*, *diversity*, and *extensibility* of today’s HPC environments pose new challenges that traditional performance prediction approaches were not designed to address.

First, it is increasingly difficult to obtain knowledge regarding *both* the applications and the execution platforms. Application developers often lack sufficiently detailed knowledge about accessible hardware platforms. Similarly, a job scheduler cannot obtain application-specific knowledge without user intervention. As a result, traditional, model-based performance prediction has become a case-by-case effort per application and platform instead of maturing to an automated computing service. Second, most traditional performance prediction approaches incur high *costs* in time and manpower, which is prohibitive for everyday usage. For fairly accurate results, the cost of predictions may exceed the benefits from job scheduling and platform selection in the first place.

Main Approach:

Our main approach is, in contrast to previous work, based on *observation-based* performance prediction. We enable very short "test drives" of applications on multiple candidate platforms to quickly derive the execution time of much longer runs. The timing results of these test drives can be stored in a database for reuse in future predictions. This approach facilitates cross-platform performance estimation as an affordable *utility*, equally beneficial to HPC users and grid schedulers.

One of the key innovations of our work is its reliance on the novel concepts of *relative performance* and *partial execution*. We observe that HPC users often have one or more *reference computers* to develop and test applications. Consequently, our work investigates *inter-platform performance translation*. More specifically, this paper make the following contributions:

- We predict the overall execution time of a large-scale application on a *target system* using the combination of its known performance on a *reference system* and the *relative performance* between the two systems derived from a very short run of the application.

- We determine the an early point in execution that captures the cross-platform relative performance reasonably well, thereby providing a low-cost, high-accuracy approach to prediction.

Our results from four production-scale parallel simulation codes show very promising prediction accuracy and low prediction overhead: in most cases, with a short execution that takes 1% or less of the total execution time, we obtain an overall execution time prediction accuracy of 97% or higher. We also discuss the limitations and extensions of our methodology, namely:

- We show that such early observations do not fully capture dynamically changing computational behavior. We further discuss approaches based on hardware performance counters combined with micro-benchmarks to tighten predictions.

- We report reduced accuracy for reusing partial execution results in predictions across varying problem sizes and numbers of processors.

2. RELATED WORK

Grid Job Scheduling: There has been an increasing interest and efforts on grid scheduling (also called *meta-scheduling*) [17, 29, 32, 27], mostly built upon local job schedulers for executing jobs on distributed computing resources. It is recognized that execution time is an important job parameter to be translated across machines. However, existing or under-development grid schedulers either do not offer execution time translation, thereby implying that users are responsible for specifying a "safe" maximum wall time value across machines, or adopt simplified translation methods, such as stretching the execution time with the CPU frequency ratio between two machines [27], which is known to be very inaccurate. Our work can form a building block for future grid schedulers by offering affordable job execution time predictions for diverse applications and platforms.

Parallel Program Performance Prediction: There have been numerous previous studies of performance prediction for parallel programs. Many of these studies are built upon performance modeling techniques (e.g., [1, 9, 13, 18, 24, 31, 34]), requiring either in-depth knowledge of the applications to build analytical models (e.g., [2, 21, 28, 36, 37]) or special compiler/instrumentation tools to infer such knowledge from parallel codes (e.g., [6, 8, 15, 24]). With careful modeling of applications and platforms, many of these previous studies achieved high prediction accuracy. However, detailed modeling often compromises the portability of prediction tools. *E.g.*, some existing approaches are application-specific [2, 19] or language-specific [8, 15]. In addition, a number of prediction techniques are based on simulations (e.g., [3, 6]), where simulators are used to measure the execution time of applications.

Most of the work mentioned above targeted performance prediction for the purpose of *performance optimization*. In contrast, our method targets performance prediction as a means for making resource usage estimation to help application owners in their research planning and daily use of diverse computing resources. In such cases, users may not be able or willing to afford traditional performance prediction techniques, which require a fair amount of work due to model building or instrumentation plus simulation. Further, today's multi-component codes (such as the simulations discussed in this paper) are comprised of modules from many application domains with diverse computational models/algorithms, making analytical modeling very hard. Also, they often use external libraries (MPI, BLAS, PETSc, NetCDF, just to name a few), and/or multiple languages (*e.g.*, mixed C/Fortran/C++ programming). This greatly decreases the feasibility and effectiveness of both analytical and compiler-aided performance modeling. Finally, given the large number of application-platform combinations in future HPC environments, simulation-based prediction can consume excessive system resources themselves. In contrast, we develop *observation-based* performance prediction, which does not require in-depth knowledge of parallel codes or systems. This makes our approach application-independent, language-independent, and platform-independent. In our evaluation, we used multiple DOE production simulations and a wide selection of contemporary supercomputers, which makes a side-by-side comparison with prior work impractical due to the differences in platforms and codes studied. In most of our test cases, our prediction scheme achieves high accuracy that matches or exceeds the best results reported by prior methods, while our worst accuracy observed is comparable with accuracies reported by many existing studies.

Further, many multi-platform performance studies (e.g., [3, 6, 7, 18, 23, 24]) evaluated their approaches with data collected at multiple supercomputers. However, data from each machine are processed individually, so are predictions and evaluations performed. Our approach, instead, combines benchmarking results from multiple platforms for cross-platform prediction. In addition, work on cross-platform performance prediction (*e.g.*, Prophesy [33] or convolution methods to map hand-coded kernels to machine profiles [4]) was often based on modeling computational kernels instead of complex applications with diverse tasks.

Several recent studies addressed performance prediction in heterogeneous grid environments [16]. A few projects addressed relative performance [19] and performance portability [20, 26]. However, we are not aware of work on performance prediction *based on* relative performance.

Finally, the repetitive behavior of applications has been exploited in speeding up architectural simulators [22], predicting performance metrics based on history information [12] and synthesizing kernels that resemble applications using architectural simulation [5]. Such studies exploit repetition at "instruction block" level, while we exploit larger-scale and more explicit behavior repetition in high performance scientific codes, based on the iterative nature many of them possess. Our method may be extended to complement the above work in facilitating efficient simulation.

3. METHODOLOGY AND SYSTEM DESIGN

Scientific applications generally have computationally intensive kernels. The performance of such applications is often constrained by the floating point resources available, memory bandwidth, and the characteristics of inter-processor communication, *via* either shared memory or message passing. Due to these constraints, performance prediction is often challenging. At the same time, scientific applications are characterized by their regularity in the course of executions, specifically with regard to array reference patterns (at the micro level) and alternating phases of computation and communication or I/O (at the macro level). It is this regularity that we exploit for our observation-based performance prediction in contrast to traditional model- or simulation-based predictions.

The repetitive nature of scientific applications at the macro level is generally a property of their computational model, often based on the notion of convergence in different mathematical approximation methods. Many, if not most, parallel simulations are *timestep*-based. In such applications, a timestep is one step of computation followed by inter-processor communication to update data. Timestep computation is repeated until the results converge (*e.g.*, when the simulation object reaches a stable state), or the computation has completed a given number of timesteps.

Traditional model-based or simulation-based approaches to performance prediction generally consider the entire execution of an application and study the interplay between the program and the architecture in a case-by-case manner. For an observation-based approach, executing the entire application takes too long to be acceptable. Instead, our approach utilizes *partial execution* for a limited number of timesteps to capture the *relative performance* across platforms for an application. We argue that *due to the highly repetitive nature of scientific codes, the relative performance observed in this short partial execution is likely to sustain through the entire run*. When used in conjunction with known full execution time of a particular application on a reference platform, partial execution results can be utilized for very cost-effective cross-platform execution time predictions.

3.1 Application Model

As described above, many parallel scientific applications are iteration-based. The execution of these applications can be described by a regular expression:

$$I(C^*[W])^*F$$

Such an application typically starts its execution with a one-time *initialization phase* (I), where it reads the input data, gathers execution configuration with initial communication, performs data distribution if necessary, and prepares data structures. The execution then enters the main timestep loop, repeating the *timestep computation phase* (C) to compute the target problem(s). Once every k timestep, there is an optional *I/O phase* (W) for writing periodic output, such as intermediate snapshots or checkpoints. Finally, the execution is concluded with a one-time *finalization phase* (F).

Between two platforms, a given application is likely to have different relative performance during the above execution phases. Since the initialization phase I and the finalization phase F both occur only once, with overhead often negligible in long-running applications, we focus on the repeated computation (C) and I/O (W) phases[1]. The key challenges we face here are 1) to predict the overall relative performance for the entire execution in spite of unstable performance in the initial timesteps, and 2) to predict the correct mixture of relative performance from the C and W phases, when the periodic I/O frequency k is unknown. Our performance prediction models designed to solve these problems are presented in Section 3.3. Before we discuss these models, we first describe our partial execution scheme for observing relative performance in short application runs.

[1]In most studies on model-based performance prediction, efforts have focused on modeling and analyzing computation kernels only.

3.2 Partial Execution

We have devised an API in support of partial execution for arbitrary iteration-based applications. While the design was inspired by the properties of timesteps, the API can also be used in the absence of explicit timesteps — as long as the activities between two consecutive calls closely represent repetitive phases in the application's execution. In the rest of the paper, we use the term "timestep" when referring to these periodic phases. The API for partial execution is as follows:

- `init_timestep()`: This is an optional call to time-stamp the beginning of an execution. For applications with large start-up overhead, *e.g.*, due to reading large data sets from secondary storage, this call may be used to separate the initialization overhead from subsequent regular timesteps.
- `begin_timestep()`: This call identifies the beginning of a timestep and allows counters for metrics to be reset between timesteps.
- `end_timestep(maxsteps)`: This call indicates the end of a timestep and logs metrics pertinent to the timestep work. The parameter `maxsteps` specifies the total number of timesteps before partial execution prematurely terminates the program's execution.

The `init_timestep()` and `begin_timestep()` calls bracket the initialization phase I. Similarly, a neighboring pair of `begin_timestep()` and `end_timestep()` calls bracket the timestep computation phase C if this timestep does not perform periodic I/O, and a combined computation-I/O phase CW if otherwise. As mentioned earlier, initialization is a one-time overhead typically with negligible cost to long-running applications. Hence, the initialization call `init_timestep()` can often be omitted. Our implementation discards the first timestep in computing relative performance, assuming the ramp-up is finished by the second timestep. Excluding the first timestep also allows caches to warm up before timing results enter the performance model.

Partial execution utilizing this API allows one to obtain metrics on a per-timestep basis and limits the number of timesteps executed. Once this number is exceeded, the application will be terminated prematurely. Hence, the objective of partial execution is not to obtain numerical results from scientific codes but to quickly and cheaply capture their rudimentary execution behavior. Only requiring high-level knowledge about the application's control flow and as few as two extra lines of code inserted, partial execution *via* the above APIs is affordable, scalable and portable. The metrics obtained during partial execution can then be utilized to predict the performance of an application run across different platforms, as detailed below.

3.3 Cross-Platform Performance Prediction

Our approach of observation-based performance prediction is based on two sets of data, one from the *reference platform*, where we have more performance knowledge about the application in question (denoted as A), and one from the *target platform*, where we want to predict the full execution time.

We assume that T_{ref}, the full execution time of A on the reference platform, or num_steps, the total number of timesteps in the full execution, is available. This is reasonable considering that there is at least one "base platform" where the code is developed or tested, and researchers typically keep track of the overall statistics for long-running jobs. In addition, we perform the same partial executions of A on the reference platform as we do on the target platform (see below).

On the target platform, we carry out a set of partial executions of A for a limited number of timesteps. Both the number of partial executions and the number of timesteps per partial execution can be small, as will be demonstrated in the experimental section. The objective of the approach is to inflict minimal time overhead for any executions on both platforms so that performance predictions can be provided quickly. This is especially important when scientists need to select their preferred target from a large set of candidate platforms (based on the relative performance, *i.e.*, machine M_1 is x times faster than machine M_2). With known execution times on the reference platform, one can further estimate the absolute performance to supply tight, yet relatively safe bounds on wall-clock time for their submitted jobs, long before having observed a complete run on the target platform, which can take hours or days.

Base Prediction Model via *Cumulative Averages:*

The base model utilizes *cumulative averages* (aka. running averages) to predict performance. It assumes that the relative performance across platforms stabilizes early in the execution and remains stable throughout the run. This is true for many regular applications with fixed dataset sizes and algorithms. Relative performance observation is based on the average per-timestep execution time for a set of repeated partial executions to obtain the per-timestep execution time t_{step} and the initialization overhead t_{init}, using the aforementioned API.

Suppose on each platform, m partial executions are performed, each running the first n timesteps (counting from timestep 2). We denote the timestep execution time for the ith timestep in the jth partial execution as $t_{step\_i,j}$. The per-timestep cumulative average from multiple partial executions up to the lth timestep ($1 \leq l \leq n$) is calculated as

$$t_{avg\_step} = \frac{1}{lm} \sum_{1 \leq i \leq l, 1 \leq j \leq m} t_{step\_i,j}$$

Similarly, if t_{init} is benchmarked in m partial executions, the average initialization overhead is

$$t_{avg\_init} = \frac{1}{m} \sum_{j=1}^{m} t_{init\_j}$$

The observed relative performance $\mathbb{R}_{tar\_ref}$ between the target and the reference platform can then be calculated using the resulting average per-timestep overhead t_{avg_step} :

$$\mathbb{R}_{tar\_ref} = \frac{t_{tar\_avg\_step}}{t_{ref\_avg\_step}}$$

Equipped with the above observed relative performance and known execution time T_{ref} on the reference platform, we can estimate absolute performance of A on the target platform T_{tar_est} as:

$$T_{tar\_est} = t_{tar\_avg\_init} + \mathbb{R}_{tar\_ref} \times (T_{ref} - t_{ref\_init})$$

If T_{ref} is not available but the total number of timesteps, num_steps, is known, we can predict T_{tar_est} as:

$$T_{tar\_est} = t_{tar\_avg\_init} + t_{tar\_avg\_step} \times num\_steps$$

The accuracy of the above prediction can be assessed by comparing T_{tar_est}, the predicted absolute performance of A, with T_{tar}, the measured performance on the target platform:

$$accuracy = \frac{T_{tar\_est}}{T_{tar}}$$

Note that a full execution on the target platform for this metric is only required to assess the model. However, when this prediction technique is applied to a grid job scheduler, one of our target use cases, such full execution time may be obtained without additional cost from the batch job accounting system, after a job is eventually scheduled and executed on one of the platforms managed by the grid job scheduler. Such "free" information can be conveniently fed back to the prediction model for its self-evaluation and self-adaptation.

The observed relative performance *during short, partial executions* approaches the overall relative performance calculated from actual full executions on both platforms when the initialization time t_{init} is small relative to overall execution time, which results in high prediction accuracy.

Prediction via *Filter Model:*

As mentioned above, the base model utilizing cumulative averages of timestep overheads suffices for applications with regular computation and platforms without runtime/OS intervention that affects execution time. To generalize our prediction to compensate for fluctuations in execution time and answer the challenges listed at the end of Section 3.1, we designed two prediction models enhancing the base model. The first one, the filter model, predicts the relative performance more accurately in the presence of erratic fluctuations of timestep overheads.

Initial fluctuations may occur for multiple timesteps instead of just during the initialization phase or the first timestep computation phase, where cache warm-up typically occurs. Such fluctuations often originate from runtime/OS intervention, such as process and/or memory page migration to better utilize system resources.

These fluctuations are often highly platform dependent and unpredictable to application developers. As will be shown in more detail in Section 4, we encountered such fluctuation in one of our test applications, which is observed on only one of four test platforms and spans the initial 20 timesteps.

The enhanced filter model *captures* fluctuations and *filters out* initial fluctuations. With this model, our implementation performs online processing of collected per-timestep overheads during partial execution. The current fluctuation is considered significant if this ratio differs from 1 by a threshold δ or more. Both n and δ are tunable and will be quantified later.

If we detect initial fluctuations in a partial execution of n timesteps, we treat it as part of the one-time initialization cost. Suppose the detected fluctuation ends at timestep x. We then calculate the new initialization overhead and timestep average up to the lth timestep ($l \geq x$):

$$T'_{init} = T_{init} + \sum_{i=1}^{x-1} t_{step\_i}$$

$$T'_{avg\_step} = \frac{1}{l-x+1} \sum_{i=x}^{l} t_{step\_i}$$

This adjustment is made on both the reference and the target systems, even if only one of them observes a fluctuation. We then replace the initialization overhead and timestep cumulative average in our base prediction model with the above new values (both further averaged over m partial executions). This enhanced prediction filters out the impact of the initial fluctuation.

Prediction via *Sliding Window:*

In addition to initial spikes, periodic fluctuations due to I/O phases are common. One could augment the API to bracket I/O phases or to specify the frequency of periodic I/O at the expense of the users' burden. Furthermore, the I/O frequency is often a configurable parameter obtained from input files and may vary between application runs. Hence, we settled on a different approach.

We take a unified solution to capturing both one-time and periodic fluctuations, and further enhance the filter model to use a *sliding window* of timestep overhead averages. In addition to capturing fluctuations and filtering out initial spikes, the relative performance is adaptively *sampled* in the presence of periodic fluctuations.

As a result, we now need to distinguish between random anomalies and periodic fluctuations. We apply heuristics 1) to compare partial execution times from multiple platforms (it is more likely an anomaly if it only occurs on one system) and 2) to detect recurring spikes/dips (it is more likely an anomaly if it only occurs once, especially close to the beginning of execution). As before, a fluctuation is considered significant if this ratio differs from 1 by a threshold δ or more. In practice, choosing $n = 5$ and $\delta = 0.05$ allows us to identify both one-time fluctuations and periodic I/O activities.

If we detect recurring performance fluctuations (most likely due to periodic I/O activities) every k timesteps, we sample the relative performance with an appropriate mixture of the "regular" and "irregular" timesteps, no matter where a partial execution is terminated. This is done by computing a *sliding window* of averages instead of cumulative averages, as an enhancement on the filter model. Intuitively, this method uses the average of timestep times collected in a contiguous window of size w. To ensure that we include the correct proportion of computation and periodic activities such as I/O, we use a window size that is an integer multiple of the observed pattern length k. Again, we adjust the initialization overhead and timestep average used in the base prediction model to use the sliding window average up to the lth timestep with a window size w ($l \geq w$):

$$T'_{init} = T_{init} + \sum_{i=1}^{l-w} t_{step\_i}$$

$$T'_{avg\_step} = \frac{1}{w} \sum_{i=l-w+1}^{l} t_{step\_i}$$

Obviously, the sliding window average matches the cumulative average for the first w timesteps. Afterwards, the cumulative approach is subject to recurrent fluctuations in the presence of periodic activities besides timestep computation while the sliding window approach provides stability.

The above monitoring and adaptive prediction may not be able to capture all fluctuations. For example, if these fluctuations occur after the partial execution is terminated, the partial execution is not long enough to detect their recurrent pattern, or the fluctuations are not big enough. However, our overall observation is that with a reasonable length of partial execution, our enhanced prediction scheme will capture fluctuations that are "important". If fluctuations are sparse or of low cost, they will safely be disregarded by our approach, as they would not have a large impact on the overall prediction accuracy in the first place.

3.4 Summary of Prediction Methodology

Because this prediction is observation-based, as long as application A is executed in the same way across platforms, prediction accuracy will not be affected by differing system characteristics, such as:

- processor families, generations and clock frequencies,
- bus interconnects (for shared-memory systems),
- communication interconnects (for networked clusters),
- memory and cache configurations,
- connections from compute nodes to shared disks, and
- system software, such as operating systems and I/O libraries.

Our experiments in Section 4 demonstrate the above.

A major objective of our performance prediction scheme is always to minimize the overhead for partial executions. The question is, *e.g., can we estimate the performance on the target platform using 64 processors with the relative performance observed in partial runs using 8 processors only*? Section 4 also presents our empirical study along this direction. The cheaper and more reusable the partial executions are, the higher we can expect the acceptance of our approach to be by end users.

One issue related to prediction cost is the number of partial executions. This number can be automatically configured, especially by considering the actual performance variance observed from previous partial executions on a given system. In our experiments detailed in the next section, we observed very small variances in partial executions with negligible impact on cross-platform prediction accuracy. However, larger variances may occur, depending on the machine configuration, job execution environment, and the nature of each application. The number of partial executions should be increased if high variance is observed. In fact, one unique side benefit of our observation-based prediction is that partial executions can be used to take a glimpse at performance variances caused by typical workloads on a given system. The prediction models presented in this paper use the average timing from multiple partial executions. This can be easily extended to utilize statistic distributions for giving a safer maximum wall time estimate for job schedulers using cross-platform performance prediction.

4. PERFORMANCE RESULTS

In this section, we present prediction accuracy (as defined in Section 3.3) and other results with our approach using partial execution and the notion of relative performance. Some of the experiments were conducted using the partial execution APIs described in Section 3.2 while others were obtained from scientists who benchmarked per-timestep execution time for their applications. In the second case, we took the first n timesteps to simulate partial executions. The applications form a diverse group of representative large-scale simulations, with a variety of computation models. More specifically, the codes we chose cover four out of seven key simulation areas identified by DOE Office of Science [11]: climate, high-energy physics, combustion, and fusion.

4.1 Experiment Platforms

Our application performance data are collected from subsets of ten different parallel computers with eight distinct types of parallel architectures. Table 1 summarizes their technical configurations.

Table 1: Parallel platforms used in the evaluation

Name	Location	Architecture	CPU	No. nodes	Procs/node	Mem/node	OS	Shared FS
Datastar-690	SDSC	IBM SP4	1.7GHz Power4	8	32	128GB	AIX	GPFS
Datastar-655	SDSC	IBM SP4	1.5GHz Power4	176	8	16GB	AIX	GPFS
Henry2	NCSU	IBM Blade Center	2.8/3.0GHz Xeon	100	2	4GB	Linux	NFS
Ram	ORNL	SGI Altix	1.5GHz Itanium2	256	1	8GB	Linux	XFS
Turing	UIUC	Apple Xserver	2GHz G5	640	2	4GB	Mac OS	NFS
Frost	LLNL	IBM SP3	375MHz Power3	64	16	16GB	AIX	GPFS
Cheetah	ORNL	IBM SP4	1.3GHz Power4	27	32	32/64/128GB	AIX	GPFS
Phoenix	ORNL	Cray X1	vector	512	1	2TB global	UNICOS/mp	StorNext
Seaborg	NERSC	IBM SP3	375MHz Power3	380	16	16/32/64GB	AIX	GPFS
TeraGrid	NCSA	Cluster	1.3/1.5GHz Itanium2	887	2	4/12GB	Linux	GPFS/NFS

4.2 Base Model

We evaluated our partial execution method with two benchmarks from the DOE ASCI Purple suite [35], a set of large-scale parallel codes with inputs resulting in hours of execution. This suite also comprises a mixture of scientific domains, types of meshes, and computation/communication models.

The two applications we successfully ported and tested on four platforms (Datastar-690, Datastar-655, Henry2, and Ram) are Sphot, a 2-D Monte Carlo photon transport code, and sPPM, a 3-D gas dynamics code. For both of them, we chose a problem size that results in hours of execution on the above platforms using 8 processors. These systems yield very small performance variances, and our prediction results are based on 1-2 full executions and 2-5 partial executions. More specifically, the largest standard deviation on the above platforms in partial execution overhead is 2.3% of the mean value, which is not significant enough to affect the prediction accuracy. Furthermore, the cost of inserting partial execution API calls is small compared to cross-platform porting overhead. It took a graduate student without prior knowledge of the codes mostly minutes (less than an hour) to identify the major timestep loop in these applications.[2]

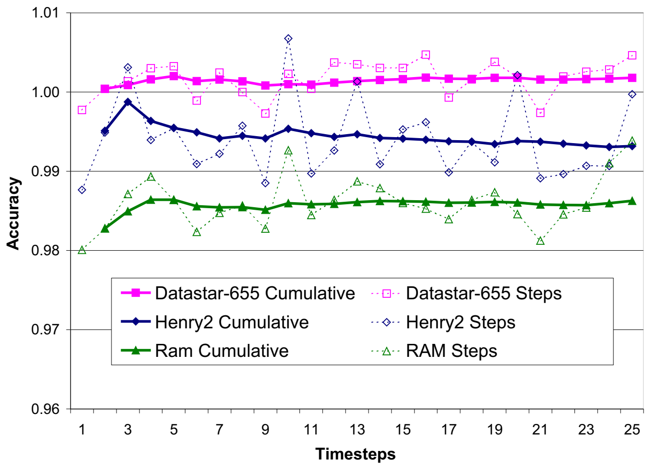

Figure 1: Sphot prediction accuracy using Datastar-690 as the reference platform

Figure 1 shows the prediction results for Sphot, using Datastar-690 as the reference platform and the other three systems as target platforms. For each platform, we portray two prediction methods: *Steps*, where the relative performance used in the ith prediction point is based on the pair of execution time values from timestep i on the reference and target platforms, and *cumulative*, where the relative performance is based on the cumulative average of execution times from timestep 1 to i on both platforms (the base prediction model).

In general, Figure 1 demonstrates that our prediction using partial execution yields very accurate results: for all three target systems, the prediction error is within 1.5%. In addition, it demonstrates the following: **(1)** High accuracy can be reached at a very early stage of execution. Even with the first timestep, when initialization and warm-up effects should perturb results on all three target platforms, the prediction accuracy is higher than 98%. Within 5 timesteps, the accuracy on all systems stabilizes at even higher levels. **(2)** Partial executions can deliver accurate predictions at a very low cost. In this full execution, Sphot executes thousands of timesteps. On the most time-consuming platform (in this case Ram), the full execution took more than 11 hours while our partial execution of 25 timesteps only took 6 minutes. Moreover, as mentioned earlier our prediction model is accurate even with fewer timesteps as input. Therefore, a partial execution's cost can be bounded by a small maximum wall time for a job. Even when this partial execution itself is terminated prematurely, our model still generates reasonable observation-based predictions. **(3)** The "cumulative" method works better than the "steps" method by smoothing out small irregularities in per-timestep execution times. **(4)** With such uniformly high accuracy, it appears that the selection of a reference platform is not important, at least for this code.

4.3 Filter Model and Initialization Overhead

For sPPM, our filter model is required to obtain accurate predictions. The simple, cumulative model did not suffice for one particular platform, namely RAM. Figure 2 depicts the per-timestep time of the simulation, which increases significantly during the first few timesteps and then drops back before stabilizing after 20 timesteps. This kind of behavior is not observed on Datastar-690, as shown in the same figure, or on any other platform. We suspect that on this NUMA machine, the operating system has been enhanced to perform page migration based on initial memory access patterns. As such, pages are moved to the node of most frequently accesses to exploit locality, while nodes issuing less frequent accesses may experience longer latencies.

Our filter model can detect and compensate for this one-time overhead, as explained in Section 3.2. Figure 2 shows the difference in prediction accuracy with and without this "ramp filter". With the non-discriminative cumulative average method, the

[2]However, porting these codes to diverse platforms and executing batch jobs takes significantly more time and effort.

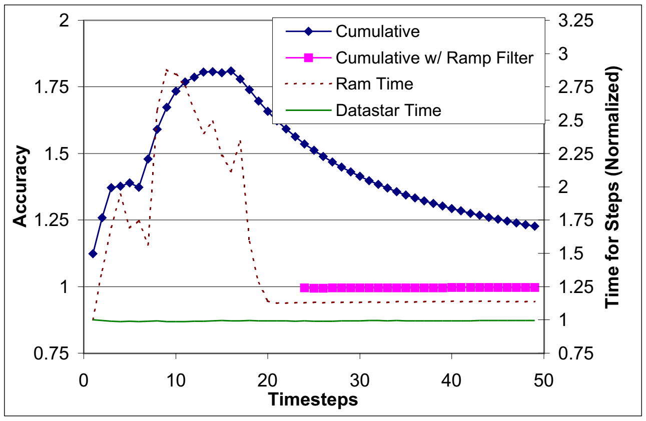

Figure 2: Normalized per-timestep execution time (right axis) and prediction accuracy (left axis) for sPPM

prediction error can reach 80%, and the effect of misleading relative performance lingers for many timesteps after the anomaly disappears. In contrast, with the improved prediction, the first 23 timesteps will not produce prediction results as they are classified as unstable. Right after that, the prediction instantly yields a consistently high accuracy of over 99%.

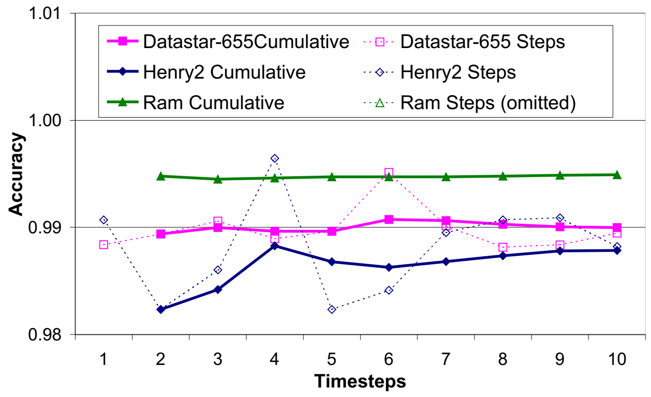

Figure 3: sPPM prediction accuracy using Datastar-690 as the reference platform

Figure 3 depicts the prediction accuracy for sPPM on all three target timesteps. Ram data points represent the first timesteps *after* the relative performance stabilizes using the filter model. sPPM has far more expensive timesteps than Sphot, with each timestep taking around 3 minutes and full runs taking almost 10 hours on Datastar-690 and 655. Therefore, we run only 10 timesteps in our partial executions. Again, the accuracy is remarkably high, at above 98% just after the first timestep.

Comparing the relative performance for Sphot and sPPM also reveals interesting facts. For Sphot, Ram is by far the *worst* platform, where each timestep takes more than 3.5 times as long as on Datastar-690. For sPPM, however, it is by far the *best* platform, where each timestep takes slightly more than 1/6 of the time on Datastar-690. This dramatic contrast is likely due to the different communication and computation patterns of the two codes. For example, sPPM uses frequent large messages, which may benefit from Altix' distributed shared memory architecture. Such phenomena suggest that relative performance across platforms can vary dramatically from application to application (in this case, a 20+ times difference). This also shows that system parameters, such as similar CPU frequencies on Ram and Datastar-690, do not offer significant information.

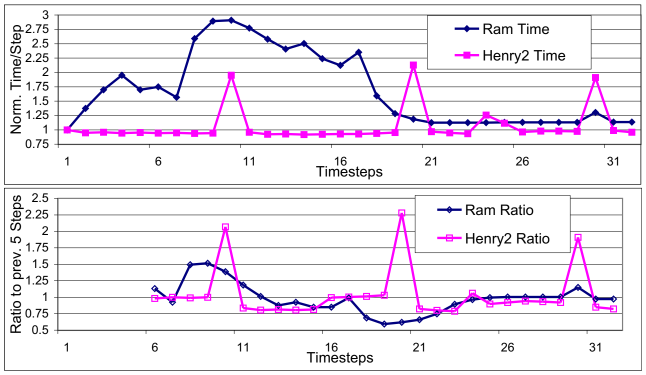

Figure 4: sPPM with high I/O frequency on Ram and Henry2. Top: normalized timestep overhead; bottom: ratio between current timestep overhead and average of last 5 timesteps

4.4 Sliding Window and Periodic I/O

Next, we consider predictions for executions with periodic I/O activities writing snapshot or checkpoint data. To save I/O time, most applications choose to periodically generate output every k computational timesteps. Between Sphot and sPPM, the latter provides an easier interface to adjust this I/O frequency. The runs shown above used a default low I/O frequency. Figure 4 depicts sPPM results from runs with a much higher I/O frequency ($k = 10$).

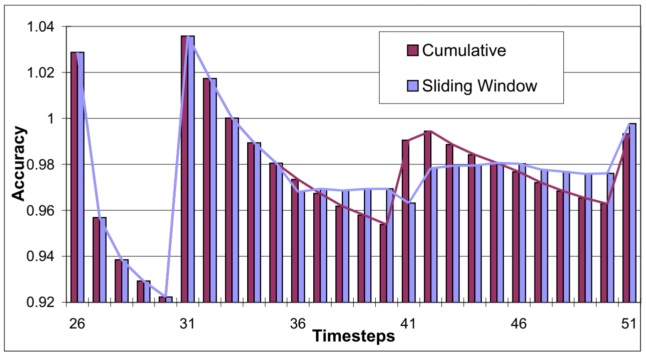

Figure 5: sPPM (high I/O frequency) predictions from Ram to Henry2, cumulative and sliding window models

Figure 4 and Figure 5 depict the I/O effect when the performance of sPPM is predicted from Ram to Henry2. This experiment serves a second purpose, namely to show that our ramp filter is valid on a reference platform as well. From the per-timestep timing curves shown in Figure 4, I/O "spikes" can be clearly identified after the execution stabilizes. On Henry2, where computation is faster than on Ram, I/O is significantly slower. Using our filter model that calculates the ratio of each current timestep versus the average of the previous 5 steps, we can successfully identify these I/O spikes as recurring behavior. We can also capture k, the aforementioned periodic I/O frequency in terms of number of timesteps.

We address recurring I/O activities with the *sliding window model*, as discussed in Section 3.3. Figure 5 shows the prediction results (after the initial noise is filtered out on the reference

655 Conf.	8×1	4×2	2×4	1×8
Accuracy	1.002	0.991	1.012	1.004

Table 2: Prediction accuracy for Datastar-655 varying (*#-processors-per-node* × *#-nodes*), total # procs = 8.

655 Conf.	2×1	4×1	8×1	8×2	8×4
Accuracy	0.988	0.993	1.002	0.978	0.981

Table 3: Prediction accuracy for Datastar-655 varying (*#-processors-per-node* × *#-nodes*), total # procs = 2..32

system). For the first 10 timesteps, the sliding window is growing, so the two models perfectly overlap. After that, however, the cumulative algorithm shows a periodic fluctuation in accuracy while the sliding window algorithm is more stable. We believe that the sliding window model will show more significant advantage if I/O is more frequent and of larger costs.

Finally, we study different processor/node configurations on the reference and target platforms. As mentioned earlier, our partial execution uses the same number of processors and the same problem size as in the full execution. However, with today's large SMP nodes, a particular configuration may not easily be reproduced across platforms. We subsequently assess the prediction accuracy with the cumulative average method from Datastar-690 (with 32-processor nodes) to Datastar-655 (with 8-processor nodes).

On Datastar-690, we ran all experiments on one node. On Datastar-655, in the first group of tests we fixed the number of processors at 8 and varied the number of nodes (1, 2, 4, and 8). In the second group of tests we increased the total number of processors from 2 to 32, where we always tried to minimize the number of nodes to use on Datastar-655. As demonstrated by Table 2 and Table 3, the prediction accuracy remains high in both cases, with no significant variance caused by the different processor/node configurations.

4.5 Limitations of the Models

For the applications studied so far, our models resulted in high accuracy. The simplicity of the models is appealing, but it also limits its application range where such high accuracy is delivered. In the following, we study the effects of our low-cost model on 1) applications with variable overhead per timestep, and 2) reusing partial execution results for different input problem sizes and different numbers of nodes. For each of these aspects, our method shows less accurate predictions with errors ranging from 5-37%. We also identify future directions of research toward equally tight predictions as previously reported. Finally, we discuss methods to detect arising inaccuracies and an "early-warning system" for users in the presence of anomalies not covered by our prediction model.

Variable Timestep Overhead:

GENx is a multi-component rocket simulation code developed at the University of Illinois [10]. The performance data was obtained from model-validation runs simulating lab-scale rockets. We chose this particular simulation since it has an interesting property: the number of particles in its fluid dynamics code increases as time goes on. Therefore, unlike any other codes demonstrated in this paper, this per-timestep execution overhead grows gradually up to a factor of 1.8 at the final point at 1550 timesteps. We use this code to assess the limitations of our model in the presence of variable timestep overhead.

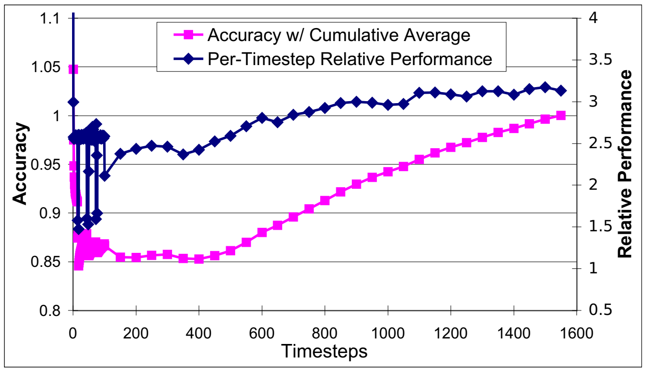

Figure 6: GENx relative performance (right axis) and prediction accuracy w/ cumulative average (left axis), one data point per first 100 steps, then every 50 steps

Figure 6 depicts timing and prediction results from two 60-processor runs, which took more than 5 hours on the reference platform (Turing) and 15 hours on the target platform (Frost). It shows that as the simulation progresses, the relative performance between the target and reference platforms gradually increases as well, *i.e.*, the costs per timestep grows *faster* on Frost than on Turing. However, this increase in relative performance (around 24%) is fairly small compared to the increase in per-timestep overhead (around 45% on Turing and 80% on Frost). A partial execution of 10 timesteps (taking 1.5 minutes on Turing and 4 minutes on Frost) would have yielded a prediction accuracy of 91.6%. Overall, the worst accuracy produced by a partial execution of the first n timesteps ($n = 1, 2, ...1550$) is around 85%. We can extend our filter model to detect when the timestep overhead fails to converge within a certain error margin to then alert users about the potential loss of prediction accuracy. Depending on the intended use of predictions, a 85% accuracy may or may not be considered sufficient, and it is upon the user to make such a decision after being alerted by our "early warning system".

We are currently investigating different approaches to tighten predictions for variable timestep overheads. In one approach, micro-benchmarks are utilized to determine cross-platform differences at various levels. When combined with the growth rate of the problem observed on the reference platform, a new growth rate can be predicted on the target platform that takes architectural differences into account. Examples of such aspects include differences in latencies within the memory hierarchy (from caches over memory down to disk I/O) or the interconnect. Our observation-based approach is to be enhanced to collect metrics from hardware performance counters that provide the means to attribute application characteristics to micro-benchmark behavior.

Reusing Partial Execution Results for Varying Number of Nodes and Input:

GYRO [14] is a code for the numerical simulation of tokamak micro-turbulence solving time-dependent, nonlinear gyrokinetic-Maxwell equations. We obtained a large set of Gyro benchmarking results from ORNL and NCSU researchers who conducted Gyro runs with 3 problem inputs (B1-std, B2-cy, and B3-gtc) on 5 platforms (Cheetah, Ram, Phoenix, Seaborg, and Teragrid) using a variety of processor numbers. Unlike the GENx simulation discussed above, Gyro has extremely stable per-timestep overheads.

Our prediction models achieve high accuracy (exceeding 98%) for cross-platform predictions for the same input data and number of nodes. Furthermore, the choice of the reference system among the five platforms does not affect the prediction accuracy. Instead of reporting these base accuracy test results, we leverage the abundance of experimental configurations in this case to examine if a pair of partial executions can be *reused* to make prediction across different input problem sizes or different numbers of processors.

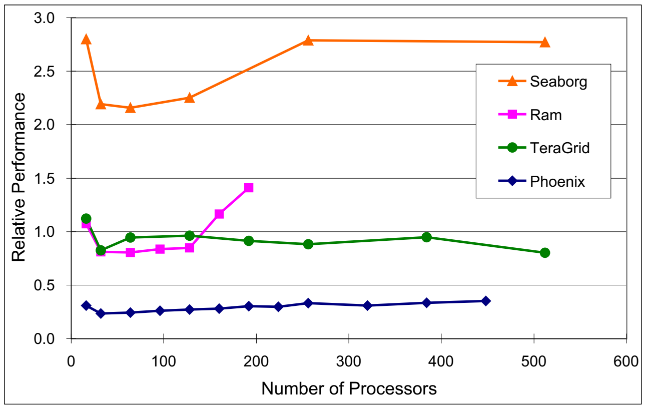

Figure 7: Gyro B1-std relative performance to Cheetah, using different number of processors

	Phoenix	Ram	Seaborg	Teragrid
# Pred.	11	6	5	7
Avg. Error	12.1%	25.5%	16.7%	25.8%

Table 4: Average errors of predictions for different numbers of processors relative to 16-processor runs

Figure 7 depicts the relative performance of four target platforms against Cheetah for different numbers of processors (a multiple of 16, limited by hardware availability/configuration on each platform) to compute B1-std. We see that the level of consistency across different numbers of processors varies from platform to platform. Since using a small number of processors is likely to be cheaper (faster to get a job scheduled), we applied the relative performance observed in the run using the fewest processors (16) for each platform when predicting the overall execution time for the other process numbers. Table 4 shows the average prediction error, which varies between 12% and 26%.

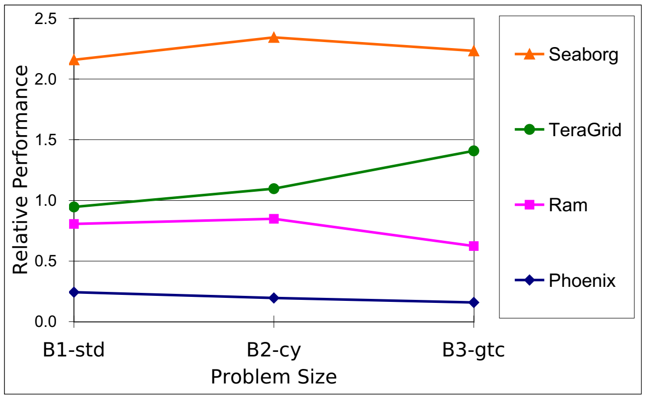

Figure 8: Gyro performance relative to Cheetah for 64 processors with different input problem sizes

	Phoenix	Ram	Seaborg	Teragrid
Avg. Error	37.9%	17.0%	5.6%	23.2%

Table 5: Average errors of predictions from B1-std to B2-cy and B3-gtc

Similarly, Figure 8 plots the relative performance across different problems, and Table 5 shows the average prediction error when we use the relative performance from computing the smallest input problem to predict for the other two problems. This would also reduce the costs of partial executions. *E.g.*, B3-gtc takes up to 3 times longer than B1-std. Here, the average error varies between 5% and 38%. Note that this group of Gyro results is not completely fair to our prediction method, as these several input problems not only bring different amount of computation but also different computation components to a certain degree.

The above results clearly indicate the trade-off between partial execution overhead and prediction accuracy. Predictions for processor (node) scalability are subject to different computation/communication ratios, which are application dependent [35]. We are currently assessing benefits from using micro-benchmark results in conjunction with a small number of partial runs on selected numbers of nodes, to enhance prediction accuracy in the presence of changing computation/communication ratios while reducing total partial execution costs. Problem (input) scaling, on the other hand, provides a more subtle challenge as growth rates in computation or communication often differ and may follow non-linear relations [35]. While user-provided data on input data sizes might help, we prefer a black-box approach due to the wider applicability of our approach. Instead of user interaction, input sizes can be inferred from memory requirements of partial executions along with observed computation and communication overheads. Such data can be automatically obtained using system calls and hardware performance counters behind our instrumentation API.

5. CONCLUSION

In this paper, we demonstrated the benefit of a black-box style, observation-based performance prediction approach built on the notion of relative performance and utilizing affordable, short partial application executions. We presented a base prediction algorithm utilizing relative performance derived from partial execution results and enhanced it to handle one-time and periodic relative performance fluctuations caused by system initialization or periodic activities such as I/O. We believe the merits of our approach lie in its *simplicity*, *portability*, and *cost-effectiveness* that make it ideal for performance prediction as a general service to HPC users and grid schedulers.

In addition, a major contribution of our work is given by the thorough evaluation of our prediction models based on relative performance from multiple production-scale real-world codes on a total of ten large parallel computers. We obtained highly accurate prediction at extremely low cost for a wide variety of platforms and multiple applications. We further identified limitations of our models and we discussed an “early warning system” to alert users to potentially less accurate predictions. We also outlined a number of approaches to tackle these scenarios ranging from variable timestep overhead to problem and processor scaling. We are currently assessing these approaches to design a tool for fully-automated performance predictions based on partial execution of applications with and without obvious timestep characteristics.

6. ACKNOWLEDGEMENTS

This work was supported in part by NSF grants CNS-0406305, CCF-0429653, CAREER CCR-0237570, and Xiaosong Ma's joint appointment between NCSU and ORNL. SDSC, ORNL, and NCSU HPC Center provided computational resources. The authors of the ASCI Purple benchmarks helped during the installation and benchmarking of these applications. Patrick Worley at ORNL, Hongzhang Shan at LBNL, as well as Kumar Mahinthakumar and Sarat Sreepathi at NCSU provided benchmarking data for Gyro. Robert Fiedler and Xiangmin Jiao at UIUC provided the GENx benchmarking data.

7. REFERENCES

[1] V. Adve and R. Sakellariou. Application representations for multiparadigm performance modeling of large-scale parallel scientific codes. *The International Journal of High Performance Computing Applications*, 14(4), 2000.

[2] B. Armstrong and R. Eigenmann. Performance forecasting: Towards a methodology for characterizing large computational applications. In *Proceedings of the International Conference on Parallel Processing*, August 1998.

[3] R. Bagrodia, E. Deeljman, S. Docy, and T. Phan. Performance prediction of large parallel applications using parallel simulations. In *Principles Practice of Parallel Programming*, 1999.

[4] D.H. Bailey and A. Snavely. Performance modeling: Understanding the present and predicting the future. In *Euro-Par Conference*, August 2005.

[5] R Bell and L. John. Improved automatic testcase synthesis for performance model validation. In *International Conference on Supercomputing*, pages 111–120, June 2005.

[6] J. Bourgeois and F. Spies. Performance prediction of an nas benchmark program with chronosmix environment. In *Proceedings of the 6th International Euro-Par Conference*, 2000.

[7] S. Browne, J. Dongarra, N. Garner, K. London, and P. Mucci. A scalable cross-platform infrastructure for application performance tuning using hardware counters. In *Proceedings of Supercomputing*, 2000.

[8] M. Clement and M. Quinn. Automated performance prediction for scalable parallel computing. *Parallel Computing*, 23(10), 1997.

[9] David E. Culler, Richard M. Karp, David A. Patterson, Abhijit Sahay, Klaus E. Schauser, Eunice Santos, Ramesh Subramonian, and Thorsten von Eicken. LogP: Towards a realistic model of parallel computation. In *Proceedings of the 4th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 1993.

[10] W. Dick and M. Heath. Whole system simulation of solid propellant rockets. In *Proceedings of the 38th AIAA/ASME/SAE/ASEE Joint Propulsion Conference and Exhibit*, Indianapolis, IN, July 2002.

[11] DOE Office of Science. *Database and File Systems Working Group Summary, SLAC Data Management Workshop*, 2003.

[12] E. Duesterwald, C. Cascaval, and S. Dwarkadas. Characterizing and predicting program behavior and its variability. In *Proceedings of the 12th International Conference on Parallel Architectures and Compilation Techniques*, 2003.

[13] M. Faerman, A. Su, R. Wolski, and F. Berman. Adaptive performance prediction for distributed data-intensive applications. In *Proceedings of Supercomputing*, 1999.

[14] M. Fahey and J. Candy. GYRO: A 5-d gyrokinetic-maxwell solver. In *Proceedings of Supercomputing*, 2004.

[15] T. Fahringer and H. Zima. A static parameter based performance prediction tool for parallel programs. In *Proceedings of the International Conference on Supercomputing*, 1993.

[16] S. Jarvis, D. Spooner, H. Keung, G. Nudd, J. Cao, and S. Saini. Performance prediction and its use in parallel and distributed computing systems. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[17] W. Jones, L. Pang, D. Stanzione, and W. Ligon. Bandwidth-aware co-allocating meta-schedulers for mini-grid architectures. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2004.

[18] D. Kerbyson, H. Alme, A. Hoisie, F. Petrini, H. Wasserman, and M. Gittings. Predictive performance and scalability modeling of a large-scale application. In *Proceedings of Supercomputing*, 2001.

[19] D. Kerbyson, A. Hoisie, and H. Wasserman. A comparison between the earth simulator and alphaserver systems using predictive application performance models. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[20] W. Ligon. Research directions in parallel I/O for clusters. In *Proceedings of the IEEE International Conference on Cluster Computing*, 2002.

[21] C. Lim, Y. Low, B. Gan, and W. Cai. Implementation lessons of performance prediction tool for parallel conservative simulation. In *Proceedings of the 6th International Euro-Par Conference*, 2000.

[22] W. Liu and M. Huang. EXPERT: expedited simulation exploiting program behavior repetition. In *Proceedings of the 18th Annual International Conference on Supercomputing*, 2004.

[23] X. Ma, M. Winslett, J. Lee, and S. Yu. Improving MPI-IO output performance with active buffering plus threads. In *Proceedings of the International Parallel and Distributed Processing Symposium*, 2003.

[24] G. Marin and J. Mellor-Crummey. Cross-architecture performance predictions for scientific applications using parameterized models. In *SIGMETRICS*, 2004.

[25] A. Mu'alem and D. Feitelson. Utilization, predictability, workloads, and user runtime estimates in scheduling the ibm sp2 with backfilling. *IEEE Transactions on Parallel and Distributed Systems*, 12(6), 2001.

[26] R. Reussner and G. Hunzelmann. Achieving performance portability with SKaMPI for high-performance MPI programs. In *Proceedings of the International Conference on Computational Science*, 2001.

[27] H. Shan, L. Oliker, and R. Biswas. Job superscheduler architecture and performance in computational grid environments. In *Proceedings of Supercomputing '03*, 2003.

[28] J. Simon and J. Wierum. Accurate performance prediction for massively parallel systems and its applications. In *Proceedings of the 2nd International Euro-Par Conference*, 1996.

[29] C. Smith. Open source metascheduling for virtual organizations with the community scheduler framework (CSF). Technical whitepaper,

http://www.platform.com/resources/whitepapers/.

[30] W. Smith, I. Foster, and V. Taylor. Predicting application run times using historical information. In *Proceedings of IPPS Workshop on Job Scheduling Strategies for Parallel Processing*, 1998.

[31] 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.

[32] Supercluster.org. SILVER design specification. http://www.supercluster.org/silver/specoverview.shtml.

[33] V. Taylor, X. Wu, and R. Stevens. Prophesy: an infrastructure for performance analysis and modeling of parallel and grid applications. *ACM SIGMETRICS Performance Evaluation Review*, 30(4), 2003.

[34] A. van Gemund. Symbolic performance modeling of parallel systems. *IEEE Transactions on Parallel and Distributed Systems*, 14(2), 2003.

[35] J. Vetter and F. Mueller. Communication characteristics of large-scale scientific applications for contemporary cluster architectures. *Journal of Parallel Distributed Computing*, 63(9):853–865, September 2003.

[36] A. Wagner, H. Sreekantaswamy, and S. Chanson. Performance models for the processor farm paradigm. *IEEE Transactions on Parallel and Distributed Systems*, 8(5), 1997.

[37] X. Zhang and Z. Xu. Multiprocessor scalability predictions through detailed program execution analysis. In *Proceedings of the 9th ACM International Conference on Supercomputing*, 1995.