Prediction of Resource Availability in Fine-Grained Cycle Sharing Systems and Empirical Evaluation [*]

Xiaojuan Ren, Seyong Lee, Rudolf Eigenmann, Saurabh Bagchi
School of Electrical and Computer Engineering, Purdue University

Abstract.
Fine-Grained Cycle Sharing (FGCS) systems aim at utilizing the large amount of computational resources available on the Internet. In FGCS, host computers allow *guest jobs* to utilize the CPU cycles if the jobs do not significantly impact the local users. Such resources are generally provided voluntarily and their availability fluctuates highly. Guest jobs may fail unexpectedly, as resources become unavailable. To improve this situation, we consider methods to predict resource availability. This paper presents empirical studies on resource availability in FGCS systems and a prediction method. From studies on resource contention among guest jobs and local users, we derive a multi-state availability model. The model enables us to detect resource unavailability in a non-intrusive way. We analyzed the traces collected from a production FGCS system for three months. The results suggest the feasibility of predicting resource availability, and motivate our method of applying semi-Markov Process models for the prediction. We describe the prediction framework and its implementation in a production FGCS system, named *iShare*. Through the experiments on an iShare testbed, we demonstrate that the prediction achieves an accuracy of 86% on average and outperforms linear time series models, while the computational cost is negligible. Our experimental results also show that the prediction is robust in the presence of irregular resource availability. We tested the effectiveness of the prediction in a *proactive scheduler*. Initial results show that applying availability prediction to job scheduling reduces the number of jobs failed due to resource unavailability.

Keywords: Cycle-sharing, Resource management, Resource availability, Prediction algorithm

1. Introduction

Distributed cycle-sharing systems have shown success through popular projects such as SETI@home [2, 14], which have attracted a large number of participants, contributing their home PCs to a scientific effort [3]. These PC owners voluntarily share the CPU cycles only if they incur no significant inconvenience from letting a foreign job (*guest process*) run on their machines. To exploit available idle cycles under this restriction, fine-grained cycle sharing (*FGCS*) systems [26, 31] al-

[*] This work was supported, in part, by the National Science Foundation under Grants No. 9974976-EIA, 0103582-EIA, and 0429535-CCF. We thank Ruben Torres for his help with the reference prediction algorithms used in our experiments.

low a guest process to run concurrently with local jobs (*host processes*) whenever the guest process does not impact the performance of the latter noticeably. For guest users, the free compute resources come at the cost of highly fluctuating availability with the incurred failures leading to undesirable completion times. The primary victims of such failures are large compute-bound guest applications, most of which are batch programs. Typically, they are either sequential or composed of multiple related jobs that are submitted as a group and must all complete before the results can be used (e.g., simulations containing several computation steps [4]). Therefore, response time rather than throughput is the primary performance metric for such compute-bound jobs. The use of this metric distinguishes our work from the use of idle CPU cycles by others, which had focused on high throughput in an environment of fluctuating resources.

In FGCS systems, resource unavailability has multiple causes and occurs frequently. First, as in a normal multi-process environment, guest and host processes run concurrently and compete for compute resources on the same machine. Host processes may be decelerated significantly by a guest process. Decreasing the priority of the guest process can only alleviate the deceleration in few situations [26]. To completely remove the impact on host processes, the guest process must be killed or migrated off the machine, which represents a failure. In this paper, we refer to such resource unavailability as *UEC* (**U**navailability due to **E**xcessive resource **C**ontention). Another type of resource unavailability in FGCS is the sudden leave of a machine — *URR*, (**U**navailability due to **R**esource **R**evocation). URR happens when a machine owner suspends resource contribution without notice, or when arbitrary hardware-software failures occur.

To achieve fault tolerance with efficiency for remote program execution, proactive approaches have been proposed in the environment of large-scale clusters [22]. These approaches explore availability prediction in job scheduling or runtime management. They achieve improved job response time compared to the methods which are oblivious to future unavailability [35]. While proactive approaches can also be applied to FGCS systems, they require successful mechanisms for availability prediction, which in turn rely on the understanding of characteristics of resource availability. However, there has been little work on predicting resource availability in large-scale distributed systems, especially in FGCS systems. While several previous contributions have analyzed the machine availability in networked environment [6, 23, 18], or the temporal structure of CPU availability in Grids [32, 21, 17], no work targets predicting availability with regard to both resource contention and resource revocation in FGCS systems.

The main contributions of this paper are the design and evaluation of an approach for predicting resource availability in FGCS systems. To understand the behavior of resource availability, we have conducted a set of studies in a production FGCS system, *iShare* [24]. We develop methods to observe and predict when a resource will become unavailable. To this end, we develop a multi-state availability model, which integrates the two classes of resource unavailability, UEC and URR. To study the predictability, we traced resource availability in an iShare testbed over a period of three months. A key observation made in analyzing these traces is that the daily patterns of resource availability are comparable to those in the most recent days. Previous work has made a similar observation [21]. It motivates our approach of applying a semi-Markov Process (SMP) to predict the *temporal reliability*, TR, which is the probability that a resource will be available throughout a given future time window. The prediction does not require any model fitting, as is commonly needed in linear regression techniques. To compute TR on a given time window, the parameters of the SMP are calculated from the host resource usages during the same time window on previous days. To alleviate the effect of deviations from the regular patterns of resource availability, we use statistical method to calculate the SMP parameters.

We show how the prediction can be realized and utilized in the iShare system that supports FGCS. We evaluate our prediction techniques in terms of accuracy, efficiency, robustness to noise (irregular occurrences of resource unavailability), and effectiveness when applying to a *proactive scheduler*. To obtain these metrics, we monitored host resource usages on a collection of machines from a computer lab at Purdue University over a period of 3 months. Users of these machines generated highly diverse workloads, which are suitable for evaluating the accuracy of our prediction method. The experimental results show that the prediction achieves accuracy of 86.5% on average and 73.3% in the worst case; it outperforms the accuracy of linear time series models [11], which are widely used in prediction techniques. The SMP-based prediction is efficient in that it increases the completion time of a guest job by less than 0.006%. It is also robust in that the high variability of host workloads disturbs the prediction results by less than 6%. Initial results of the proactive job scheduling show that, by applying our prediction method, a higher number of guest jobs can be completed successfully with improved response time, than non-predictive scheduling.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 describes our studies of resource availability. The derived multi-state availability model is shown in Section 4. Section 5 presents the studies on predictability, including trace collection and analysis. The background and application of semi-Markov Process models are described in Section 6. Section 7 discusses implementation issues of availability prediction in iShare. Experimental approaches and results of evaluating the prediction are described in Section 8.

2. Related Work

The concept of fine-grained cycle sharing was introduced in [26], where a strict priority scheduling system was developed and added to the OS kernel to ensure that host processes always receive priority in accessing local resources. Deploying such a system involves an OS upgrade, which can be unacceptable for resource providers. In our FGCS system, available OS facilities (e.g., *renice*) are utilized to limit the priority of guest processes. Resource unavailability happens if these facilities fail to prevent guest processes from impacting host processes significantly. In [26], the focus is on maintaining priority of host processes. By contrast, our work develops resource availability prediction methods, so that guest jobs can be managed proactively with improved response times.

Related contributions include work in estimating resource exhaustion in software systems [30] and critical event prediction [28, 27] in large-scale dedicated computing communities (clusters). To anticipate when a system is in danger of crashing due to *software aging*, the authors of [30] proposed a semi-Markov reward model based on system workload and resource usage. However, the data they collected deviate excessively from the supposed linear trends of resource exhaustion rate, resulting in prohibitively wide confidence intervals. The work in [28, 27] predicted general error events within a specified time window in the future. The presented analysis and prediction techniques require close observation of precedent events happened right before an error, and thus is infeasible for FGCS systems that do not have access to all the event logs on a host system.

Emerging platforms that support Grids [12] and global networked computing [9] motivated the work to provide accurate forecasts of dynamically changing performance characteristics [11] of distributed compute resources. Our work will complement the existing performance monitoring and prediction schemes with new algorithms to predict resource availability in the environment of fine-grained cycle sharing. In this paper, we compare the commonly used linear time series algorithms, which are related to our SMP-based algorithm; we show that our algorithm achieves higher prediction accuracy, especially for long-term prediction.

Other efforts have analyzed machine availability in enterprise systems [23, 6], or large Peer-to-Peer networks [5], where machine availability is defined as the machine being reachable for P2P services. While these results were meaningful for the considered application domain, they do not show how to relate machine uptimes to actual available resources that could be effectively exploited by a guest program in cycle-sharing systems. By contrast, our approach integrates machine availability into a multi-state model, representing different levels of availability of compute resources.

A few other studies have been conducted on percentages of CPU cycles available for large collections of machines in Grid systems [21, 33, 17]. In [21], the author predicted the amount of time-varying capacity available in a cluster of privately owned workstations by simply averaging the amount of available capacity over a long period. The work in [33] applied one-step-ahead forecasting to predict available CPU performance on Unix time-shared systems. This approach is applicable to short-term predictions within the order of several minutes. By contrast, our SMP-based technique predicts for future time windows with arbitrary lengths. The authors of [17] studied both machine and CPU availability in a desktop Grid environment. However, they focused solely on measuring and characterizing CPU availability during periods of machine uptimes. Instead, we predict the availability of CPU and memory resources, while taking machine downtimes into account.

3. Detecting Resource Unavailability

This section presents the studies that form the basis of our availability model, shown in Section 4. The goal is to find a practical and non-intrusive method to detect resource unavailability, especially the unavailability due to excessive resource contention. Such a detection method is critical for preventing significant slowdown experienced by host jobs. The detection would be trivial if we could measure the slowdown of host jobs directly. However, direct measurement requires pre-knowledge of contention-free performance of host jobs, which is not feasible. Therefore, we need to use observable parameters as indicators for the slowdown. By observable parameters, we mean parameters that can be obtained without special privileges on the host machine. Our overall detection method is to determine thresholds for observed CPU and memory utilization of host jobs. The thresholds constitute *noticeable slowdown* of host processes. The intuition is that resource contention is aggravated when the resource use of host jobs increases; when the resource use exceeds a threshold, contention becomes excessive and, thus, the resource becomes unavailable for guest jobs. We use offline experiments to determine the values of these thresholds on specific systems.

In the rest of this section, we first discuss the observability of both types of unavailability, UEC (unavailability due to excessive resource contention) and URR (unavailability due to resource revocation). Then we present our offline experiments to determine the thresholds.

3.1. Observability of Resource Unavailability

URR happens when machines are removed from the FGCS system by their owners, or fail due to hardware-software faults without externally visible prior symptoms. System-internal symptoms, such as memory leakage and disk block fragmentation [30], have been considered to detect failures. However, in FGCS systems, such information is often inaccessible to external uses. Therefore, in the view of guest applications, machines may suddenly become offline and the resulting URR can only be detected in that FGCS services, such as the service for job submission, are terminated. This fact supports a two-state model for URR: a machine is either available or unavailable; there are no other observable states in-between.

UEC happens when host processes incur noticeable slowdown due to resource contention from guest processes. Detecting UEC requires the quantification of *noticeable slowdown* of host processes. Our FGCS system uses the observed CPU and memory utilization of host jobs for the quantification. If the host resource utilization reaches certain thresholds, the system claims that UEC happens. The exact thresholds for what constitutes UEC may vary on OSes with different mechanisms of resource management. We use offline experiments to obtain these thresholds on specific systems. The reason to use empirical studies instead of analytical models is that developing such models is very difficult, if not impossible, considering the complexities in OS resource management. The experimental approaches and results are discussed in the next section.

3.2. Studies on Resource Contention

In our experiments, we ran guest and host jobs together. The CPU and memory usages of each job, when it is running alone, are known beforehand. We measured the reduction rate of *host CPU usage* (total CPU usage of all the host processes running on a machine) due to the contention from a guest job running concurrently. The "noticeable slowdown" of host jobs is represented by the reduction rate going above an application-specific threshold (we chose a threshold of 5%). We are interested in finding out the exact values of host resource usage when the reduction rate exceeds 5%, that is, when UEC happens.

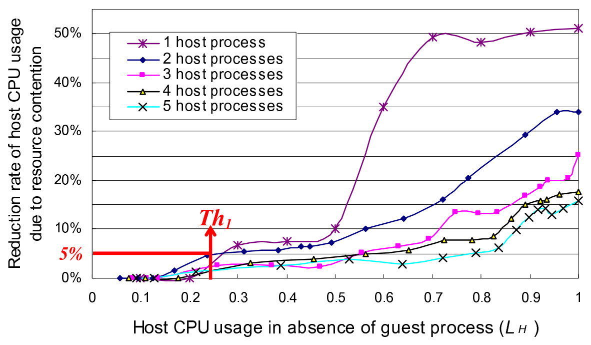
To make sure that the experimental results are not biased by particular workloads, we use representative guest applications and a broad range of host applications. In FGCS systems, guest applications are normally CPU-bound batch programs, which are sequential or composed of multiple tasks with little or no inter-task communication. Such applications arise in many scientific and engineering domains. Common examples include Monte-Carlo simulations and seismic analysis tools [4]. Because these applications use files solely for input and output, file I/O operations usually happen at the start and the end of a guest job; file transfers can be scheduled accordingly to avoid peak I/O activities on host systems. Some of the guest applications also have large memory footprints. Therefore, CPU and memory are the major resources contended by guest and host processes. Host applications, on the other hand, can be computational tasks, OS command-line utilities, etc. In our experiments, they are represented by processes with various CPU and memory usages.

We conducted a set of experiments by running host processes with various resource usages together as an aggregated *host group*. To avoid any adverse contention among multiple guest processes, no more than one guest process is allowed to run concurrently on the same machine. The priority of a running guest process is minimized (using *renice*) whenever it slows down the the host processes noticeably. If this does not alleviate the resource contention, the reniced guest process is suspended. The guest process resumes if the contention diminishes after a certain duration (1 minute in our experiments), otherwise it is terminated.

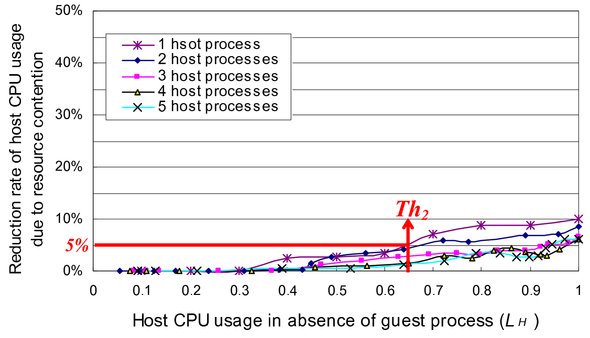
3.2.1. *Experiments on CPU Contention*

To study the contention on CPU cycles, we created a set of synthetic programs. To isolate the impact of memory contention, all the programs have very small resident sets. The host programs have *isolated CPU usage* (CPU usage of a program when it runs alone) ranging from 10% to 100%. The wall clock time (*gettimeofday*) and CPU time (*getrusage*) measurements were inserted in the synthetic programs to calculate their CPU usages and to adjust the sleep time to achieve the given isolated CPU usages. The guest process is a completely CPU-bound program. In the experiments, we ran these programs on a 1.7 GHz Redhat Linux machine.

Figure 1 presents the reduction rate of host CPU usage (the total CPU usage of all the host processes in a host group), when a guest process (G) is running together with a host group (H). Figure 1 (b)



(a) All processes have the same priority



(b) Guest process takes the lowest priority

Figure 1. Host CPU utilization under CPU contention. The x-axis (L_H) is the CPU usage of a group of host processes when the group is running alone. The y-axis shows the reduction rate of the host group's CPU usage (compared to L_H) when a guest process is running together.

shows the results when G's priority is set to 19 (lowest) while H's priority is 0. L_H is the CPU usage of a host group without interference of guest processes. To create a host group with a given L_H that consists of M ($M > 1$) processes, we randomly chose M host programs with different isolated CPU usages and ran them together without the guest process. If the total CPU usage of the M processes was equal to L_H, we chose them as a combination to generate the host group. For each tested host group, we used multiple combinations of host processes to measure the reduction rate of host CPU usage. The average of the measurements is plotted in Figure 1. This approach considers the fact that the same host workload may result from various individual host processes.

We tested host groups with L_H ranging from 10% to 100%, when M was set to 1 to 5, respectively. There are two reasons why we chose M to be no larger than 5. First, the total number of active processes started by a typical host user is usually in the range of tens. Second, as shown in Figure 1, the curves for different M converge. That is, for the same L_H, the reduction rate of host CPU usage decreases as M increases. Intuitively, in a time-sharing system, the chances that a guest process can steal CPU cycles decrease when there are more host processes running. When the size is beyond 5, the reduction saturates and, thus, there is no need to experiment with arbitrary sizes of the host group.

The results in Figure 1 show the existence of two thresholds, Th_1 and Th_2, for L_H, that can be used as indicators of noticeable slowdown of host processes. Th_1 and Th_2 are picked according to the lowest values of L_H among the different host group sizes, where the guest process needs to be set to a low priority or terminated, respectively, to keep the slowdown below 5%.

3.2.2. *Experiments on CPU Contention Using Different Methods to Control Guest Priority*

To verify that the existence of the two thresholds is not the simple result of our method of controlling guest priorities, we tested resource contention using different ways to adjust guest priorities, as used in practical FGCS systems. The two alternatives are, gradually decreasing the guest priority from 0 to 19 under heavy host workload ($L_H > Th_1$), or setting the guest priority to its lowest value whenever the guest process starts [9]. (The extreme case of terminating a guest application whenever a host application starts makes it a coarse-grained cycle sharing system [14].) In the first alternative, fine-grained values between Th_1 and Th_2 are needed to indicate different guest priorities. Relating to the second alternative, only Th_2 is needed. We conducted a set of experiments to test if these two alternatives deliver a better model of CPU availability than using the two thresholds. In these experiments, we ran the same set of synthetic programs on the 1.7 GHz Linux machine.

In the experiment for testing the first alternative, we ran a host process concurrently with a guest process of different priorities. Figure 2 presents the degradation of host CPU usage due to resource contention. When the isolated host CPU usage (L_H) is between 20% and 50%, impact of different guest priorities is trivial. This indicates that the guest process does not consume significantly more CPU by taking higher priorities than 19. When L_H is larger than 50%, the guest priority must be set to 19 (lowest) to ensure acceptable degradation of host CPU usage. Therefore, gradually decreasing guest priority does not achieve additional benefit in terms of CPU availability for guest

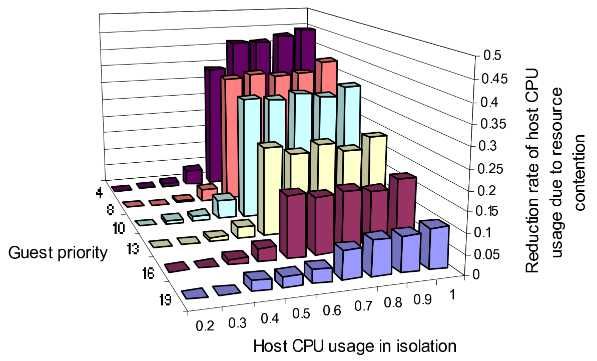


Figure 2. Reduction rate of host CPU usage due to the contention from a guest process with different priorities. This figure implies that gradually decreasing guest priority does not make the guest process consume more CPU cycles.

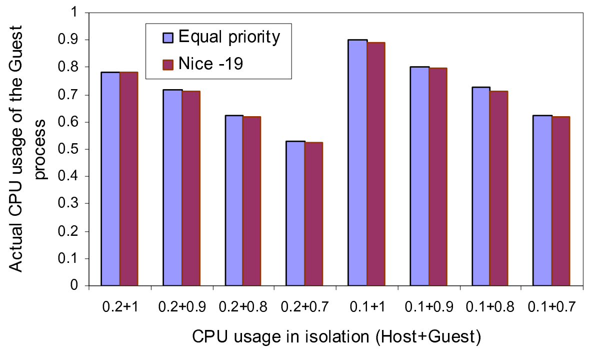


Figure 3. CPU usage of the guest process with equal and lowest priority. The x-axis is the isolated CPU usage for the coexisting host and guest processes. For example, "0.2+1" means that the isolated host and guest CPU usage is 0.2 and 1.0, respectively. The figure shows that always taking the lowest guest priority does not achieve maximum guest CPU usage.

processes. Instead, it causes higher overhead to managing guest jobs at runtime.

The experiment for the second alternative was conducted via running a set of CPU-intensive guest processes (isolated CPU usage $\geq 70\%$) with priority 0 and 19 under light host workload ($L_H \leq 20\%$) respectively. We measured the CPU usage of the guest processes and plotted the results in Figure 3. The differences between the two sets of bars in this figure show that, the guest CPU usage with priority 0 is about 2% higher on average than that with priority 19. In FGCS systems, the 2% more CPU usage can make a significant difference in job completion times if the guest job takes hours to finish. Therefore, the approach of always enforcing the lowest guest process priority is too conservative.

Table I. Resource usage of tested applications.

Workload	**CPU usage**	**Resident size**	**Virtual size**
apsi	98%	193 MB	205 MB
galgel	99%	29 MB	155 MB
bzip2	97%	180 MB	182 MB
mcf	99%	96 MB	96 MB
H_1	8.6%	71 MB	122 MB
H_2	9.2%	213 MB	247 MB
H_3	17.2%	53 MB	151 MB
H_4	21.9%	68 MB	122 MB
H_5	57.0%	210 MB	236 MB
H_6	66.2%	84 MB	113 MB

In all the above experiments, we used randomly-generated host groups without relying on any specifics in OS scheduling. Therefore, we view the existence of the two thresholds as a general, practical property of Linux systems. This also holds for Unix systems, as confirmed by our experiments on both CPU and memory contention on a Unix machine. The next section presents these experiments.

3.2.3. *Experiments on CPU and Memory Contention*

So far, we have considered CPU contention, only. To test the more complicated contention on both CPU and memory, we experimented with a set of larger applications. For guest processes, we chose four applications from the SPEC CPU2000 benchmark suite [15]: apsi, galgel, bzip2 and mcf, which are all CPU-bound. Their working set sizes range from 29 MB to 193 MB. To simulate the behaviors of actual host users on text-based terminals, we used the Musbus interactive Unix benchmark suite [20] to create various host workloads. The created workloads contain host processes for simulating interactive editing, Unix command-line utilities, and compiler invocations. We varied the size of the file being edited and compiled by the "host users" to create host processes with different usages of memory and CPU. Table I lists the resource usages of the four guest applications and the six host workloads (H_1 to H_6) created by Musbus.

We ran a guest process concurrently with each host workload on a 300 MHz Solaris Unix machine with 384 MB physical memory. For each set of processes, we measured the reduction of the host CPU usage

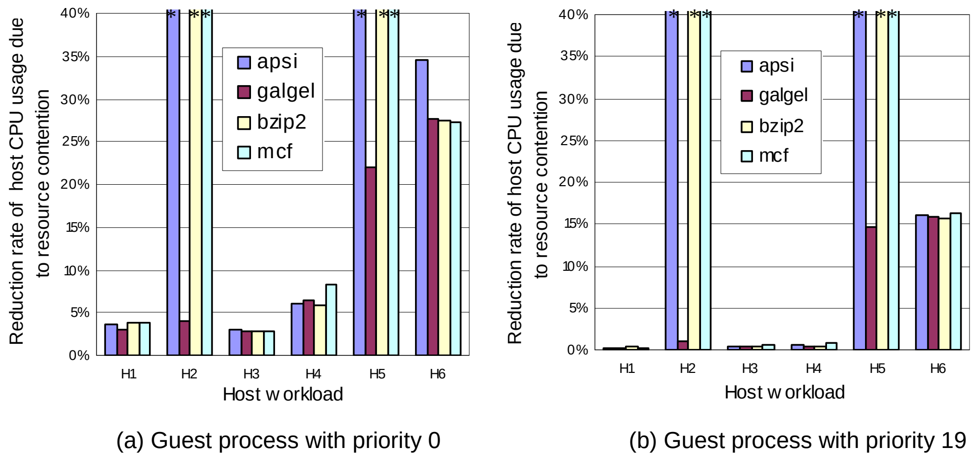


Figure 4. Slowdown of host processes under resource contention. Bars with * at the top are for the host processes dragged down due to memory thrashing.

caused by the guest process, when the guest process's priority was set to 0 and 19 respectively. The results are shown in Figure 4.

In Figure 4, memory thrashing happens when running H_2 or H_5 together with apsi, bzip2, or mcf under different priorities. In all these cases, the total working set size of the guest and host processes (including kernel memory usage of about 100 MB) exceeds the physical memory size of the machine. Changing CPU priority does little to prevent thrashing when the processes desire more memory than the system provides. Therefore, the host processes make little progress regardless of the guest priorities. The fact that memory thrashing happens for both H_2 and H_5 indicates that the occurrences of UEC with memory contention are orthogonal to host CPU usage. On the other hand, when there is sufficient memory in the system, the occurrences of CPU unavailability solely depend on the host CPU usage. For example, in Figure 4, slowdown of the host processes can be ignored for H_1 and H_3, while the guest process has to be reniced under H_4 and terminated under H_6. In these cases, the two thresholds, Th_1 and Th_2, can still be used to evaluate CPU contention. From the results in Figure 4, Th_1 is around 20% and Th_2 is between 22% (CPU usage of H_4) and 57% (CPU usage of H_5) for Solaris Unix systems.

In conclusion, memory contention and CPU contention can be isolated in detecting UEC. We do not need to consider the case of both resources under contention, since the additional effect due to one resource, when contention for another is already underway, is negligible.

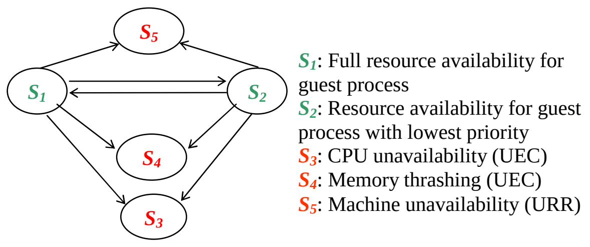

S5
S1
S2
S4
S3
S1: Full resource availability for guest process
S2: Resource availability for guest process with lowest priority
S3: CPU unavailability (UEC)
S4: Memory thrashing (UEC)
S5: Machine unavailability (URR)

Figure 5. Multi-state system for resource availability in FGCS.

4. Multi-State Availability Model

The presented results for resource contention in Section 3.2 show the feasibility of two thresholds, Th_1 and Th_2, for the measured host CPU load (L_H), that can be used to quantify the noticeable slowdown of host processes, thus the occurrences of UEC. In our FGCS testbed, consisting of Linux systems, Th_1 and Th_2 are 20% and 60% respectively. Based on the two thresholds, a 3-state model for CPU contention can be created, where the guest process is running at default priority (S_1), is running at lowest priority (S_2), or is terminated (S_3). Due to the isolation between CPU contention and memory contention, the 3-state model can be extended by adding a new unavailability state (S_4) for memory thrashing. These states are combined with URR (S_5) to give a five-state model, as presented in Figure 5. Note that the three states, S_3, S_4, and S_5, represent unrecoverable failures for guest processes. Even if the CPU or memory usage of host processes drops significantly or the host becomes available again, the guest process has already been killed or migrated off.

The formal definition of the five states is as follows:

- S_1: When the host CPU load is light ($L_H < Th_1$), the resource contention caused by a guest process can be ignored. S_1 also contains the cases when L_H transiently rises above Th_2 and the guest process is suspended;

- S_2: When the host CPU load is heavy ($Th_1 \leq L_H \leq Th_2$), the guest process's priority must be minimized to keep the impact on host processes small (slowdown $\leq$ 5%). S_2 also contains the cases when L_H transiently rises above Th_2 and the guest process is suspended;

- S_3: When the host CPU load is higher than Th_2 for a period (1 minute in our system), any guest process (with default or lowest priority) must be terminated to relieve resource contention;

- S_4: When there is no enough free memory to fit the working set of a guest process, the guest process must be immediately terminated to avoid memory thrashing;
- S_5: When the machine is revoked by its owner or incurs a system failure, URR happens, whereby resources immediately become offline.

In the above definition, S_1 and S_2 also represent the scenarios that L_H gets higher than Th_2 transiently (less than 1 minute in our experiments) and the guest process is suspended. We do not introduce a new state for a temporarily suspended guest process, because we find it very common that the host CPU load, after exceeding Th_2, will drop down shortly in a few seconds. The transiently high CPU load may be caused by a host user starting a remote X application or by some system processes.

5. Predictability Study: Trace Collection and Analysis

Based on the multi-state model presented in Section 4, we developed a module for unavailability detection and traced resource availability in an Internet-sharing systems, *iShare* [24], which supports FGCS. The goal is to find out if the availability is predictable and what factors constitute a good prediction method.

On each host machine, there is a resource monitor measuring CPU and memory usage of host processes periodically. To achieve non-intrusiveness to the host system, the monitor applies lightweight system utilities, such as *vmstat* and *prstat*. Implementation details for iShare's resource monitor are discussed in Section 7.2. The monitor is started automatically when the resource provider turns on the iShare software and its termination indicates resource revocation.

We installed and started a resource monitor on each machine in an iShare testbed, which contains 20 1.7 GHz Redhat Linux machines in a general purpose computer laboratory for student use at Purdue University. The local users on these machines are students from different disciplines. They used the machines for various tasks, such as checking emails, editing files, and compiling and testing class projects, which created highly diverse host workloads. On a tested machine, processes launched via iShare are guest processes, and all the other processes are viewed as host processes. When a resource becomes unavailable, the running guest process is terminated. Resource revocation happens when the user with access to a machine's console does not wish to share the machine with remote users, and simply reboots the machine. The

Table II. Statistics of host resource utilization.

State	S_1	S_2	S_3	S_4	S_5
Holding time	55.8%	6.6%	25.9%	9.2%	2.3%
Average available CPU					61.8%
Standard deviation (among different machines)					8.4%
Average available memory (MB)					297.4
Standard deviation (among different machines)					78.9

Table III. Resource unavailability due to different causes.

Categories	**Total number**	**UEC**		**URR**
		CPU contention	Memory contention	
#Occurrences	405–453	283–356	83–121	3–12
Percentage	100%	69–79%	19–30%	0–3%

resource behavior on these machines is consistent with the availability model in Figure 5.

We traced the availability of each tested machine for 3 months, from August to November 2005, resulting in roughly 1800 machine-days of traces. The data contains the start and end time of each occurrence of resource unavailability, the corresponding failure state (S_3, S_4, or S_5), and the available CPU and memory for guest jobs. In the following, we present our results of trace analysis.

5.1. Statistics of Resource Availability

Table II shows percentage of time that a machine stays at each state. The five states are the same as shown in Figure 5. These statistics were collected using the whole set of 1800 machine-days of traces. According to the results, a machine stays at S_1 (where resources are fully available) and at S_2 (where resources are available under minimum guest priority) for 55.6% and 6.6% of the time, respectively. This leads to the total amount of 61.8% of CPU cycles that can be utilized by guest applications. This number is lower than those reported in related work [21, 17]. The reason is that we consider host workloads in a university student environment. The workloads present more causes of resource unavailability, namely, memory thrashing and resource revocation, which are ignored in previous papers.

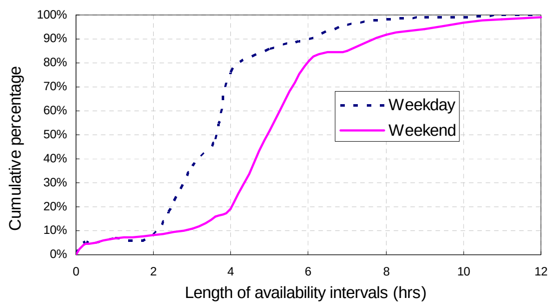
Table III lists the statistics on resource unavailability due to different causes. Number of occurrences refers to how many times a particular kind of unavailability happened during the 3 months on an individual machine, and percentage shows its relative proportion with respect to the total number of all kinds of unavailability. The two parameters were measured on each machine in the testbed, and the ranges on all the tested machines are given in Table III.

Table III shows that high host CPU load is the main cause of resource unavailability in our FGCS testbed. Because the physical memory size is larger than 1 GB on all the tested machines, memory thrashing happens less frequently. In general, UEC happens much more often than URR in FGCS systems. As discussed earlier, URR has two sources: resource providers' intentional leave and software-hardware failures. In our testbed, the first source corresponds to machine reboots, which appear in our traces as URR with intervals shorter than one minute. Software-hardware failures are represented by URR lasting longer than one minute. By examining the interval lengths for all the recorded URR, we found that around 90% of URR originated from machine reboots. This is not surprising because, on our tested machines, a local user may experience slowdown due to processes submitted by non-local users. A common user behavior in that case is rebooting the machine, thereby contributing to URR incidents.

In conclusion, UEC constitutes the major part of resource unavailability in our studied FGCS system. Regarding our goal of studying the predictability, this means that the predictability is tightly correlated with the pattern of host workloads, especially host CPU load. While previous studies have observed the possibility to coarsely estimate the aggregated CPU availability of desktop machines [17, 9], it is difficult to relate the information directly to the predictability of resource availability. In particular, the understanding of temporal characteristics of availability intervals (that is the statistical lengths of time intervals during which a resource will be available) and the frequency of unavailability occurrences is key to obtaining direct measures of the predictability. We develop such characterizations in the next two sections.

5.2. Distribution of Lengths of Availability and Unavailability Intervals

Resource availability intervals are periods during which a guest application can utilize host resources. Unavailability intervals are periods



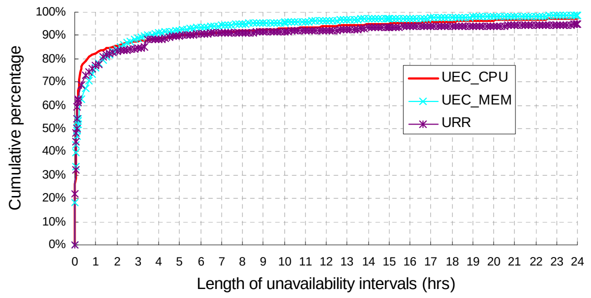


Figure 6. Cumulative distribution of lengths of availability and unavailability intervals. A date point, (x, y), means that y% of the corresponding intervals are shorter than x hours.

when the application fails or gets suspended. Facilities to predict such interval lengths provide the knowledge of how much computation power an FGCS system can deliver without interruption and when resources will return from excessive contention or revocation. Figure 6 plots the cumulative distribution of the duration of resource availability and unavailability intervals. These results were calculated from the traces of all the 20 machines during the 3 months.

From Figure 6, we see that availability intervals are shorter during weekdays, with an average of close to 3 hours, versus above 5 hours during weekends. Further, about 60% of availability intervals are between 2 and 4 hours on weekdays, and between 4 and 6 hours on weekends. The distributions of unavailability intervals with different causes present similar patterns. All the three curves rise sharply for intervals less than 5 minutes, constituting about 60% among all measured intervals. We found that they are mainly CPU peaks resulting from activities of system processes. This implies that the system could suspend a guest job for about 5 minutes upon resource unavailability. For most cases, resources will return shortly after the suspension.

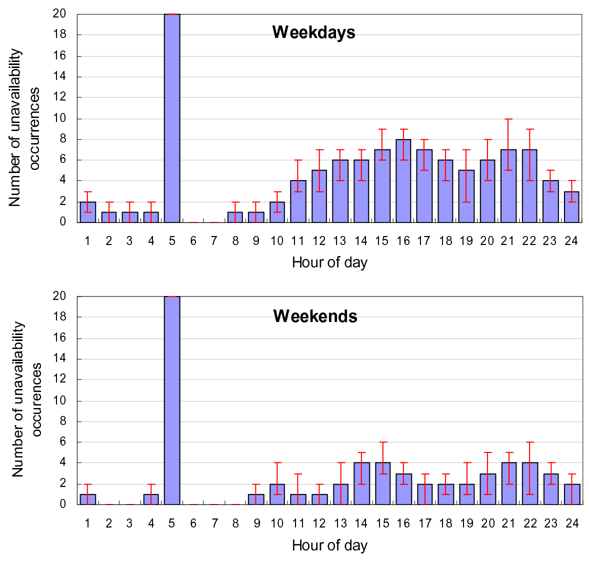


Figure 7. Occurrences of unavailability during each hour in a day. The value at hour i means the amount of unavailability occurred between $(i-1,i)$.

5.3. Daily Pattern of Failure Occurrences

To understand the more fine-grained behavior of resource availability, we counted the number of unavailability occurrences during each hour of a day on all the machines in the testbed. Figure 7 plots the distribution of unavailability occurrences during a weekday and a weekend, respectively. The value for hour i means the amount of unavailability occurred in the time interval between hour $i-1$ and i. The unavailability spanning multiple hours was counted for each of the one-hour intervals. Both the average values and the ranges over all the weekdays and weekends in the period of 3 months are depicted.

The results in Figure 7 show that the frequency of unavailability occurrences per hour is tightly correlated with the host workloads during the corresponding hour. This confirms our observation in Section 5.1. For example, unavailability happens more frequently during the day time after 10 AM with more students using the machines, and for the same time window, the amount of unavailability is larger on a weekday than on a weekend. One exception is the extremely high number (20 on both weekdays and weekends) of unavailability occurrences between 4 and 5 AM, when very few students are using the machines. We found that this is caused by the high CPU load of a system process *updatedb* (also viewed as host processes), which updates file name databases used by GNU *locate* to search for files in a system. The process is started at 4 AM every day and lasts for about 30 minutes. Therefore, the amount of unavailability happened between 4 and 5 AM is equal to the total number of machines in the testbed (20). This “exception” also shows the correlation between unavailability occurrences and host workloads.

The most important observation obtained from Figure 7 is that the deviations of unavailability frequency over the same time window across different weekdays (weekends) are small. This is evidenced by the relatively small range bars for each hour of the day. This means that the daily patterns of resource availability are comparable to those in the recent history. Previous work has made a similar observation [21]. Therefore, it is feasible to predict resource availability over an arbitrary future time window from history data for the corresponding time windows of previous weekdays or weekends. In FGCS systems, the time window can be derived from the estimated execution time of a guest job. An aggressive prediction algorithm would accommodate the small deviations of resource availability among related time windows. Our approach will use statistics on history trace to alleviate the effects of “irregular” data. More specifically, we propose to apply semi-Markov Process models for the prediction. The reasons are discussed in the next section.

5.4. Discussions on Prediction Algorithms

A number of time-series and belief-network algorithms [28] appear in the literature for predicting continuous CPU load and discrete events. Our goal is to design a prediction algorithm that achieves both high accuracy and efficiency appropriate for online uses. Several algorithms pursue *one* of these goals. For example, time-series algorithms are fast by sacrificing accuracy, especially for long-term predictions; learning algorithms, on the other hand, often require tedious processes of offline learning and massive data sets. Another prediction method used Bayesian Network models [28], which operate on acyclic transition paths and are thus inapplicable to the 5-state availability model in Figure 5, where the states S1 and S2 form a cycle.

We base our prediction algorithm on a semi-Markov Process (SMP) model, as it naturally fits the multi-state model without modification. This algorithm does not require any model fitting, as is commonly needed in linear regression techniques, and is thus efficient. To achieve high accuracy, we apply a statistical method to calculate the SMP parameters. The next section presents details of the algorithm.

6. Semi-Markov Process Models

In the multi-state availability model presented above, transitions between the states fit a semi-Markov Process (*SMP*) model, where the next transition only depends on the current state and how long the system has stayed at this state. In essence, the SMP model quantifies the dynamic structure of the multi-state model. More importantly, for our objective, it enables the efficient prediction of temporal reliability. This section presents background on SMP and shows how it can be applied for our prediction based on the availability model in Figure 5.

6.1. Background on Semi-Markov Process Models

Markov Process models are probabilistic models useful in analyzing dynamic systems [1]. A semi-Markov Process (*SMP*) extends Markov process models to time-dependent stochastic behaviors [19]. An SMP is similar to a Markov process except that its transition probabilities depend on the amount of time elapsed since the last state transition. More formally, an SMP can be defined by a tuple, (S, Q, H), where S is a finite set of states, Q is the state transition matrix, and H is the holding time mass function matrix. The most important statistics of the SMP are the interval transition probabilities, P.

$$\begin{aligned}
Q_i(j) &= Pr\{the\ process\ that\ has\ entered\ S_i\ will\ enter\ S_j \\
&\qquad in\ its\ next\ transition\}; \\
H_{i,j}(m) &= Pr\{the\ process\ that\ has\ entered\ S_i\ remains\ at\ S_i \\
&\qquad for\ m\ time\ units\ before\ the\ next\ transition\ to\ S_j\} \\
P_{i,j}(t_1, t_2) &= Pr\{the\ process\ enters\ S_j\ at\ time\ t_2, \\
&\qquad given\ that\ it\ stays\ at\ S_i\ at\ time\ t_1\} \\
&= Pr\{S(t_2) = j \mid S(t_1) = i\} \qquad (1)
\end{aligned}$$

To calculate the interval transition probabilities for a continuous-time SMP, a set of backward Kolmogorov integral equations [19] were developed, as shown in Equation 2. H' is the holding time density function matrix, the derivative of H.

$$P_{i,j}(t_1, t_2) = \sum_{k \in S} \int_{t_1}^{t_2} Q_i(k) * H'_{i,k}(u) * P_{k,j}(t_2 - u)\, du \qquad (2)$$

Basic approaches to solve these equations include numerical methods and phase approximation. While these solutions are able to achieve accurate results in certain situations, they perform poorly in many situations, such as, when the rate of transitions in the SMP is as high as exponential with time. In real applications [1], a discrete-time SMP model is often utilized to achieve simplification and general applicability under dynamic system behaviors. This simplification delivers high computational efficiency at the cost of potentially low accuracy. We argue that the loss of accuracy can be compensated by tuning the time unit of discrete time intervals to adapt to the system dynamism. In this paper, we develop a discrete-time SMP model, as described in the next section.

6.2. Semi-Markov Process Model for Resource Availability

This section discusses how a discrete-time SMP model can be applied to the availability model presented in Figure 5. The goal of the SMP model is to compute a machine's temporal reliability, TR, which is the probability of not transferring to S_3, S_4, or S_5 within an arbitrary time window, W, given the initial system state, S_{init}. The time window W is specified by a start time, W_{init}, and a length, T. Equation 3 presents how to compute TR by solving the equations in terms of Q and H. The derivation of the equation can be found in [1]. In Equation 3, $P_{i,j}(m)$ is equal to $P_{i,j}(W_{init}, W_{init}+m)$, $P^1_{i,k}(l)$ is the interval transition probabilities for a one-step transition, and d is the time unit of a discretization interval. δ_{ij} is 1 when $i = j$ and 0 otherwise.

$$\begin{aligned}
TR(W) &= 1 - \sum_{j=3}^{5} P_{init,j}(T/d) \\
P_{i,j}(m) &= \sum_{l=0}^{m} \sum_{k \in S} P^1_{i,k}(l) * P_{k,j}(m-l) \\
&= \sum_{l=1}^{m-1} \sum_{k \in S} H_{i,k}(l) * Q_i(k) * P_{k,j}(m-l) \\
P_{i,j}(0) &= \delta_{ij} \qquad\qquad j = 3,4,5 \\
& \qquad\qquad\qquad i = 1,2,3,4,5
\end{aligned} \tag{3}$$

The matrices Q and H are essential for solving Equation 3. In our design, these two parameters are calculated via the statistics on history logs collected by monitoring the host resource usages on a machine. The details on resource monitoring are explained in Section 7. To compute Q and H within an arbitrary time window on a weekday (a weekend), we derive the statistics from the data within the corresponding time windows of the most recent N weekdays (weekends). The rationale behind this is the observation that the load patterns in a given time window (e.g., from 9 to 11 am) are comparable on different weekdays (weekends) [21].

7. System Design and Implementation

We implemented the described prediction methods within the iShare [24] Internet-sharing system. iShare is an open environment for sharing both HPC resources, such as the TeraGrid facility [8], and idle compute cycles available from any Internet-connected host. This section introduces the fine-grained cycle sharing capabilities of iShare and shows how the availability prediction is implemented and utilized.

7.1. Fine-Grained Cycle Sharing in iShare

The iShare system supports the publication and discovery of compute systems and their applications [25], and it enables the remote execution of these applications on most suitable systems. Cycle-sharing happens when users submit guest jobs to the published machines, while these machines also run local jobs. A scheduler is responsible for matching guest jobs and host systems. To this end, existing techniques can be utilized to estimate the execution time [16] and the memory usage [13] of a guest job. A *proactive scheduler* would use these two quantities and pass them to the temporal reliability prediction. The predicted result is then used by the scheduler to select resources with relatively high availability or to manage the job adaptively during its execution.

Figure 8 shows the iShare framework with resource availability prediction. The *Host Node* and the *Client* show examples of a resource provider and a user, respectively. The prediction function is invoked on the host node upon a request of job submission from the client. There are three prediction-related daemons on the host node. The *iShare Gateway* communicates with remote clients and controls local guest processes. The *Resource Monitor* measures CPU and memory usage of host processes periodically. The *State Manager* stores history logs and predicts resource availability. These daemons are started automatically when resource providers turn on the iShare software and their termination indicates resource revocation. The guest process is launched for a job submission from the client.

Upon the request of a job submission on a client, the client's *Job Scheduler* queries the gateways on the available machines for their temporal reliability during the expected window of job execution, and decides on which machine(s) the job would be executed. If a machine

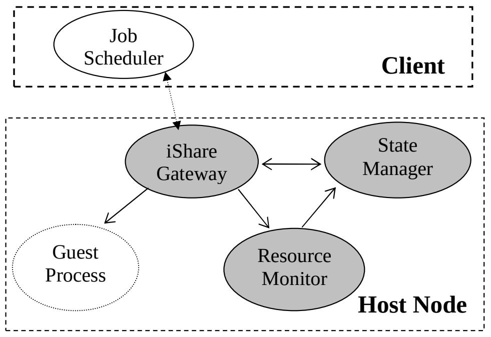


Figure 8. Processes related to resource availability prediction in iShare. Arrows indicate inter-process communication.

is selected, a guest process is launched on the machine and the corresponding resource monitor is notified of the new process id. During the job execution, the monitor detects any state transition and signals the gateway of a new transition. The gateway then reduces the priority, or kills the guest process, as needed. Checkpointing can also be used to migrate the guest process off the machine if resources become unavailable.

There are two main design challenges to implement the framework shown in Figure 8. First, the resource monitor needs to be non-intrusive to the host machine where the monitoring takes place periodically. Because resource availability prediction happens on the critical path upon the request of a job submission, the computational cost of the prediction must be negligible. Our solutions to the two challenges are described in the next two sections.

7.2. Non-intrusive Resource Monitoring

As discussed in Section 4, state transitions among S_1, S_2 and S_3 can be detected by monitoring the total CPU load of all the host processes on a machine; transitions to S_4 can be detected by monitoring the free memory size and amount of memory swap to disk on the machine. The resource monitor shown in Figure 8 uses system utilities such as *vmstat* and *prstat* on Unix and *top* on Linux, which are light-weight operations in most OS implementations, including Redhat Linux used in our experiments.

To monitor the occurrences of resource revocation (transitions to S_5), the timestamp of the most recent load measurement, $t_{monitor}$, is recorded in a special log file on the host machine. This timestamp is updated when the periodic resource monitoring occurs. To detect if a machine has become unavailable, the monitor compares the current

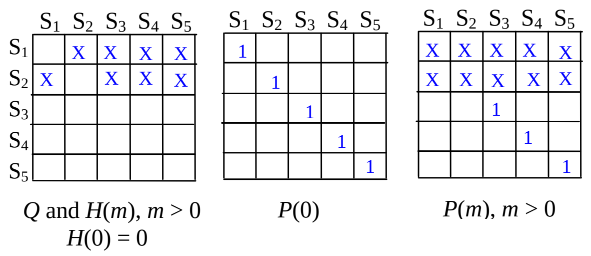


Figure 9. The sparsity of Q, H and P. The blank cells are for elements whose values are zero. Non-zero elements are labeled with a X (arbitrary values) or 1 (the value is 1).

timestamp with the saved $t_{monitor}$ at each periodic monitoring. If the gap between the two timestamps exceeds a threshold, it indicates that the resource monitor, and by implication the iShare system, had been turned off on the monitored machine (due to either system crash or machine owner's intentional leave). This is a simple solution to the important problem of avoiding the need for administrator privileges in accessing system logs for machine reboots. It is also more efficient and scalable compared to other techniques [5] for tracing machine uptimes, where a centralized unit is needed to probe all the nodes in a networked system.

7.3. Minimum Computation in Solving SMP

In our design, matrix sparsity in the SMP model is exploited to minimize the computational cost of availability prediction. Figure 9 describes the sparsity of the matrices Q, H and P in Equation 3. In this figure, all the blank cells are for zero values. The sparsity relies on two facts — it takes a finite amount of time to transition from one state to another, and states S_3, S_4 and S_5 are unrecoverable failure states.

With the sparsity shown in Figure 9, Q and $H(m)$ can be stored as an 8-element vector. As shown in Equation 3, the value of TR is decided by the summation of $P_{init,3}(T/d)$, $P_{init,4}(T/d)$ and $P_{init,5}(T/d)$, where the value of *init* is either 1 or 2. Equation 4 shows the minimum computation needed to solve the three probabilities by exploring the sparsity of Q and H. This equation shows that only six elements in $P(m)$ are required: $P_{1,3}$, $P_{1,4}$, $P_{1,5}$, $P_{2,3}$, $P_{2,4}$, and $P_{2,5}$. The total number of recursive steps is $T/d - 1$, decided by both the length of the time window, T, and the discretization interval, d. In this work, we choose the discretization interval the same as the period of resource usage monitoring. The results on computational overhead presented in Section 8 demonstrate the effectiveness of the optimization in solving SMP.

$$
\begin{aligned}
P_{1,j}(T/d) &= \sum_{l=0}^{T/d} \sum_{k \in S} H_{1,k}(l) * Q_1(k) * P_{k,j}(T/d - l) \\
&= \sum_{l=1}^{T/d-1} [H_{1,2}(l) * Q_1(2) * P_{2,j}(T/d - l) \\
&\qquad + H_{1,j}(l) * Q_1(j)] + H_{1,j}(T/d) * Q_1(j) \\
P_{2,j}(T/d) &= \sum_{l=0}^{T/d} \sum_{k \in S} H_{2,k}(l) * Q_2(k) * P_{k,j}(T/d - l) \\
&= \sum_{l=1}^{T/d-1} [H_{2,1}(l) * Q_2(1) * P_{1,j}(T/d - l) \\
&\qquad + H_{2,j}(l) * Q_2(j)] + H_{2,j}(T/d) * Q_2(j) \\
&\qquad\qquad j = 3, 4, 5 \qquad (4)
\end{aligned}
$$

8. Evaluation

We have developed a prototype of the system as described in Section 7. This section presents the experiments for evaluating our prediction techniques in terms of accuracy, efficiency, robustness to irregular patterns of resource availability, and effectiveness when applying to a proactive scheduler.

All of our experiments were conducted on the same FGCS testbed as described in Section 5. We used the same set of traces collected from August to November 2005, which present highly diverse host workloads. Because accuracy of the SMP-based prediction is mainly affected by the variety of host workloads, the testbed proved appropriate to test our prediction algorithm comprehensively.

We considered four sets of experiments. First, we measured the overhead of the resource monitoring and the prediction algorithm. Second, we tested the accuracy of our prediction algorithm by dividing the trace data for each machine into a training and a test data set. The prediction was run on the training set and the results were compared with the observed values from the test set. We also compared the prediction accuracy with that of a suite of linear time series models. Third, to test the robustness of our prediction algorithm, we inserted noise randomly into a training set and measured the difference between the prediction results by using the infected training set and those by using the original training set. Finally, we applied our prediction algorithm in a proactive

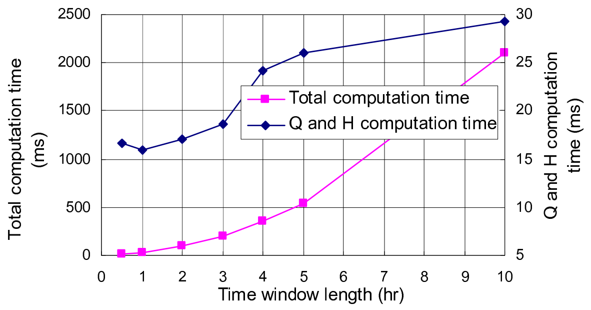


Figure 10. Computation time of the probability that a resource will be available during a given time window. This overhead adds to the execution time of a guest application.

job scheduler and tested its effectiveness in improving the execution of guest jobs. The results are presented and analyzed in the rest of this section.

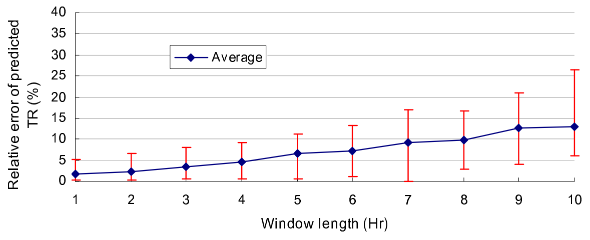
8.1. Efficiency of Availability Prediction

The overhead of the proposed prediction method includes the computational cost caused by both the resource monitoring and the SMP computation. With a sampling periodicity of 6 seconds, resource monitoring consumed less than 1% CPU and 1% memory on each tested machine in our testbed. Therefore, our resource monitoring is non-intrusive to the tested host system. To measure the computational overhead of the prediction, we measured the wall clock time of the prediction for time windows with different lengths. In Figure 10, the computation time of calculating Q and H and the whole prediction algorithm (including the computation for Q, H and TR) are plotted as a function of time window length. Recall that the goal is to predict the probability that a resource will be available during a given time window for guest job execution. As expected, the prediction over a larger time window takes longer because of the higher number of recursive steps needed. The total computation time follows a superlinear function (with exponent of 1.85) of the number of recursive steps, with the relative overhead increasing with job execution time. For the time window of 10 hours (the last point on the x-axis), the computation time for Q and H is 29.35 milliseconds and the total computation time is about 2.1 seconds. This gives the stated overhead of 0.006% for the average guest process execution time of 10 hours. Because most guest jobs in our FGCS system have completion time less than 10 hours, we can conclude that our prediction algorithm is efficient and causes negligible overhead on the completion time of typical guest jobs in FGCS systems.

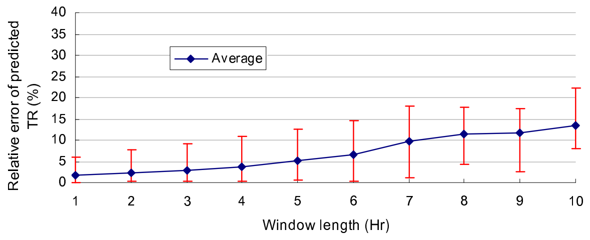
8.2. Accuracy of Availability Prediction

To test the accuracy of our prediction algorithm, we created a training and a test data set for each machine by dividing its trace data into two equal parts and choosing the first half as the training set. The parameters of the SMP model were calculated by statistics of the training data set and were then used to predict the *TR* for different time windows in the test data set. We used the actual observations from the test data set to calculate the *empirical TR*. We computed the relative error as $abs(TR_{predicted} - TR_{empirical})/TR_{empirical}$. Figure 11 plots the relative error of our prediction algorithm. The curve shows the average error of predictions on time windows with different lengths, and the bars show the minimum and maximum errors. To collect the average errors for predictions over time windows of the same length, we experimented with different start time ranging from 0:00 to 23:00 on different machines, in steps of 1 hour. As shown in Figure 11, the relative prediction error increases with the time window length. The reason is that *TR* gets close to 0 for large time windows leading to possibly large relative errors. Prediction on small time windows performs slightly worse on weekends than on weekdays, which can be explained by the smaller training size used for prediction on weekends. The prediction achieves accuracy higher than 73.38% in the worst case (maximum prediction error for time windows with length of 10 hours on weekdays). The average prediction accuracy is higher than 86.5% (average prediction accuracy for time windows with length of 10 hours on weekends) for all the studied time windows in Figure 11.

We also conducted a set of experiments to analyze the sensitivity of the prediction accuracy to the size of the training set. Intuitively, the prediction with larger training sets should perform better than that using smaller training sets. However, a large training set includes older data, which may bias the most recent pattern of host resource usages on the studied machine. We are interested in finding out if there exists a best choice of training size. Toward these goals, we divided all the trace data for weekdays into training and test sets with different size ratios. On each setting of the data, we ran the prediction over the same 240 time windows used for the experiment in Figure 11 and measured the relative prediction errors which are plotted in Figure 12. "Max-average error" is measured by first averaging over prediction errors for the time windows of the same length and then taking the maximum of all the average values. The results in Figure 12 show that there exists a sweet spot (6:4 in our experiment) for the ratio of training and test sizes. While the observation of this sweet spot may be specific to our dataset and is not intrinsic for the SMP-based prediction, its existence



(a) Prediction on weekdays



(b) Prediction on weekends

Figure 11. Relative errors of predicted *TR* (temporal reliability). Each point plots the average error of predictions over 24 time windows with different start time ranging from 0:00 to 23:00, in steps of 1 hour. The bars show minimum and maximum prediction errors.

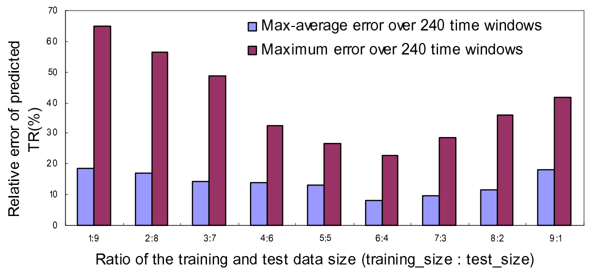


Figure 12. Relative prediction errors with different ratios of training and test data sizes for weekdays.

is important. It suggests a practical way to achieve best prediction accuracy by tuning the size of history data for arbitrary systems.

8.2.1. *Comparison with Linear Time Series Models*

Among a number of algorithms [28] for predicting continuous CPU load or discrete events, we chose linear time series models [11] as reference points for our SMP-based prediction algorithm. Linear time series models have been used for predicting CPU load in Grids [11]. The algorithms use linear regression equations to obtain future observations from a sequence of previous measurements. Compared to the SMP model, time series models consider different load levels and fit them into a liner model by ignoring the dynamic structure of load variations.

Our comparison on the two classes of models will quantify the benefits of considering the dynamic structure in resource availability prediction.

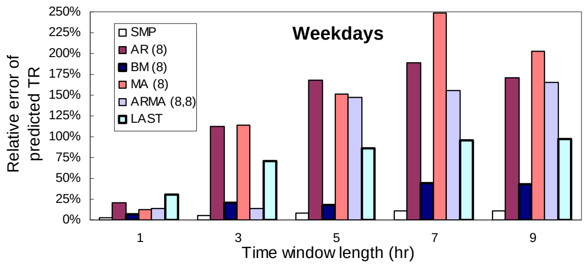
We used a set of linear time series models implemented in the RPS toolkit [10]. The models are described in Table IV. We took the same parameters for these models as used in RPS. In our experiments, we focused on the prediction accuracy of the time series models compared to our SMP-based prediction. We applied time series models to predict the state transitions in a future time window based on the samples from the previous time window of the same length. Thus, to predict transitions for 10 am-11 am on a weekday, we use historical data from 10 am-11 am from previous weekdays. The prediction accuracy is determined by the difference of the observed temporal reliability on the predicted and the measured state transitions.

In this experiment, we used the training and the test sets of equal size. We ran the prediction on each time window (starting at different time and of different lengths) on all the tested machines. For each given start time and window length, we calculate the error in the temporal reliability prediction at each machine. Then, the maximum prediction error over different machines is used as the metric of comparison, which forms the basis for a worst case comparison. Figure 13 shows the comparisons. As a representative case, we present the relative errors of predictions over time windows starting at 8:00 am on weekdays/weekends. Predictions for other time windows achieve similar results in terms of the relative differences among these algorithms.

Table IV. Linear Time Series Models

Model	Description
$AR(p)$	Autoregressive models with p coefficients
$BM(p)$	Mean over the previous N values ($N \leq p$)
$MA(p)$	Moving average models with p coefficients
$ARMA$	Autoregressive moving average models with
(p,q)	$p+q$ coefficients
LAST	Last measured value

From the results in Figure 13, we made the following observations. (1) Based on the relative prediction errors for the time windows studied, our SMP-based algorithm performs better than all 5 time series models. The advantage is more pronounced for predictions over large time windows. (2) Linear time series models are not adept at long-term predictions. This is because these models use multiple-step-ahead for



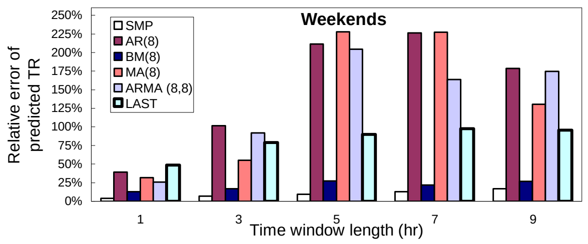


Figure 13. Maximum prediction errors of different algorithms over time windows starting at 8:00 am on weekdays and weekends.

predicting on large time windows and the prediction error increases with the number of steps lookahead.

8.3. Robustness of Availability Prediction

As we discussed earlier, the SMP-based prediction is able to accommodate deviations from the load patterns that are comparable in recent days. This ability is confirmed by the high prediction accuracy presented in Section 8.2. To further test the ability, we study its robustness to noise (irregular occurrences of unavailability) in the training data.

We injected different amounts of noise into the training data set and measured its impact on the prediction results. To inject one instance of noise, we manually inserted one occurrence of unavailability around 8:00am (when unavailability is very rare due to low resource utilization) to a training log of a weekday in the trace data collected on a machine in the testbed. The holding time of the added failure state was chosen randomly between 60 and 1800 seconds. With varying number of noise injections, we measured the *prediction discrepancy* by comparing the prediction results against the original predicted values without noise injection. Experimental results are presented in Figure 14. The prediction discrepancy bars for large time windows ($T = 5, 10$ hrs) are often negligible compared to the values for small time windows. Hence some of the bars for large time windows do not show in the figure.

Figure 14 shows that predictions on smaller time windows are more sensitive to noise. As shown by the bars for "$T = 1\ hr$", 4 instances of

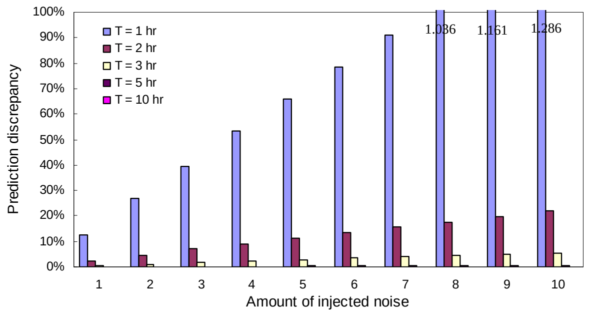


Figure 14. Prediction discrepancy with different amounts of noise injected to a training log for weekdays. T is the length of the future time window. Prediction discrepancy is the relative difference between the prediction results using the training data with and without noise injection respectively.

noise lead to a prediction discrepancy of more than 50%. On the other hand, for the time windows larger than 2 hrs, 10 instances of noise cause less than 5.56% (the bar for "$T = 3\ hr$") prediction discrepancy. The reason behind this observation is that the negative impact of noise on large time windows is alleviated by taking more history data in the prediction. Recall that our prediction utilizes history data within the corresponding time window (with the same start time and length) for predicting on a future time window.

In a practical FGCS system, such as iShare, most guest jobs are either small test programs taking less than half an hour, or large computational jobs taking several hours. For small test programs, resource unavailability happens rarely (according to Figure 6, about 90% of the availability intervals are longer than 2 hours); a job scheduler can perform well without the availability prediction. Therefore, small jobs will not suffer from the prediction inaccuracy resulted from irregular availability patterns. For large jobs taking more than 2 hours, intensive noise (10 noise events within 1 hour) causes less than 6% disturbance in our prediction algorithm. Therefore, we can conclude that our prediction algorithm is robust enough for application in practical FGCS systems.

8.4. Applying Prediction in a Proactive Scheduler

Proactive scheduling algorithms apply availability prediction to select machine resources for a given guest application. We are developing such algorithms using the SMP-based prediction. In this section, we present our proactive algorithm and the initial results of scheduling a single computational task onto FGCS resources. To evaluate the benefits of utilizing availability prediction in resource selection, we chose three algorithms for comparison: (1) a Condor-like algorithm schedules jobs to **presently** available resources by *matchmaking* [29]; (2) our proactive algorithm predicts resource availability to improve scheduling decisions; and (3) an omniscient algorithm with full knowledge of host resource utilization in the future. All these algorithms are static, i.e., a guest job is scheduled at the beginning without further migration or rescheduling.

In more detail, the Condor-like algorithm matches guest job requests with known host machine resources. These guest jobs and hosts are specified in *classified advertisements* using a semi-structured data model [29]. A given guest job can specify the ranking criteria for resource selection. In our implementation, we chose the clock rate of a host as the criterion. That is, if multiple resources match the requests, the one with the highest CPU speed will be chosen. In the omniscient algorithm, we use future traces of host workload and the associated resource availability to pre-execute the guest job, and then choose the one that finishes earliest. Therefore, it obtains optimal results among all the scheduling algorithms. It is also the fastest without causing any computational overhead to make scheduling decisions.

In our proactive algorithm, we first calculate the estimated job completion time (JCT) without considering potential failures, $JCT = TL/[\ CR * (\ 1 - L_{t_0}(TL)\ )\ ]$. In this equation, t_0 is the job submission time, CR is the clock rate of the host, $L_{t_0}(t)$ is the estimated CPU load[1] averaged over the time interval between t_0 and $t_0 + t$, and TL is the task length (the job completion time on a dedicated host machine with the average CPU speed as in our FGCS testbed). To factor future availability in resource ranking, two more parameters, *mean time to failure* ($MTTF$) and *effective task length* (ETL: the length of task expected to finish before being interrupted), are derived from the predicted temporal reliability, TR, of a host machine. Equations 5 and 6 present the computation of the two parameters.

$$\begin{aligned} MTTF &= \int_0^\infty t * Pr\{job\ fails\ at\ t\}\ dt = \int_0^\infty t\ dPr\{job\ fails\ before\ t\} \\ &= \int_0^\infty t\ d(1 - TR(t)) = -[t *\ TR(t)]|_0^\infty + \int_0^\infty TR(t)\ dt \\ &= \int_0^\infty TR(t)\ dt \qquad (5) \end{aligned}$$

$$ETL = MTTF * CR * (1 - L_{t_0}(MTTF)) \qquad (6)$$

[1] we applied the aggregated one-step-ahead algorithm [34] to obtain the average CPU load for a future time window

In Equation 5, to simplify the results of *integration by parts*, we apply the fact that TR decreases superlinearly with t in our FGCS system. To make resource selection decisions, the proactive algorithm first compares $MTTF$ with JCT (the estimated job completion time without considering failures). If, for each resource, the latter is larger, the algorithm selects the one with the largest ETL. Otherwise, it selects the one with the minimum job completion time, considering failures ($JCTF$), which can be calculated from Equation 7.

$$TL \;=\; CR * (1 - L_{t_0}(JCTF)) * \int_0^{JCTF} TR(t)\; d\,t \qquad (7)$$

We implemented the above three algorithms in the GridSim [7] simulator. There are two reasons for using simulation rather than the testbed mentioned in Section 5: we need to repeat experiments for guest jobs with different start time and lengths for all the three scheduling algorithms; and we plan to simulate an FGCS system whose scale is beyond the testbed. On the other hand, we can still use the host workload trace collected from our testbed because the host (not including guest) workload on an individual machine is orthogonal to the scale or other settings of an FGCS system.

The GridSim simulator supports the modeling and simulation of a wide range of heterogeneous resources in Grids. In our experiments, it was used for simulating job execution on a host with certain occurrences of resource unavailability, which we injected using the same traces described in Section 5. A submitted job is specified by a job id, the job (task) length, the memory usage, and the submission time. To schedule a job, the simulator first discovers a list of resource candidates. If no resource is available at the time point, it waits for 5 minutes and tries again. The job will return as *unscheduled* if no resource becomes available after the 5 minutes. A scheduled job will return either as *finished* or, if resources become unavailable during the job execution, as *failed*. A failed job will be restarted from the beginning repeatedly until it finishes successfully.

We conducted three experiments, for each of the three scheduling algorithms. In each experiment, a total number of 420 jobs were submitted. Their submission times were distributed uniformly between 6 am and 10 pm. For each submission time, we tested jobs with 7 different lengths, ranging from 0.5 to 6 hours. We measured two metrics, *job failure rate* and *average makespan*. Job failure rate shows how many jobs fail due to resource unavailability. Makespan is the time interval between a job submission and the time it completes. We measured the makespans of all the scheduled jobs (including those that failed and

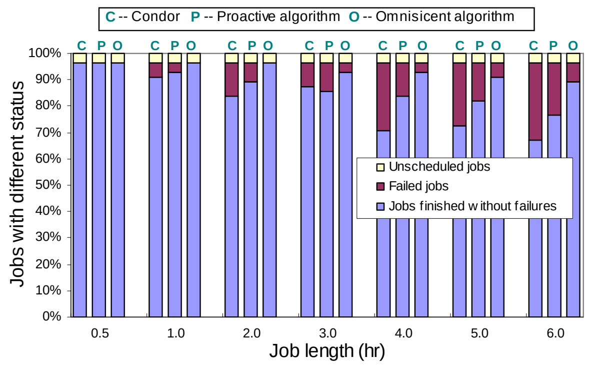


Figure 15. Percentage of jobs with different status. Unscheduled jobs can not get executed because no resource is available at the time of job submission.

then restarted) and collected the average values for each scheduling algorithm. These two metrics indicate how effective a scheduling algorithm is in selecting resources with high availability and high computing capability.

Figure 15 shows the percentage of jobs returned with different status. For each job length, about 3% of jobs were unscheduled. This value is the same for all scheduling algorithms, because a submitted job was tested identically in each algorithm. For all the jobs, our proactive algorithm achieves a better or similar failure rate compared to the Condor-like algorithm. The difference between the proactive and the omniscient algorithm is due to the availability prediction inaccuracy, which generally increases with job length. For jobs with lengths of three hours, the Condor-like algorithm obtains slightly lower failure rate than the proactive algorithm. The reason is that about 60% of availability intervals are longer than 3 hours, as shown in Figure 6, and thus the Condor-like algorithm does not suffer much from unexpected job failures. Meanwhile, the proactive algorithm is affected by the errors in availability prediction (the error is about 10% in the worst case according to Figure 11). For large jobs that experience possibly frequent failures, the proactive algorithm performs much better than the Condor-like algorithm.

Figure 16 presents the relative slowdown of the scheduling algorithms by comparing them to the omniscient algorithm. To measure the relative slowdown of an algorithm, we first collected the average makespan of all the jobs scheduled and finished (including those failed and then restarted) using this algorithm, and then compared the value to that of the omniscient algorithm. There are two sources for the slowdown: the ineffectiveness in selecting the best resource and the computational overhead of the scheduling algorithm. The omniscient

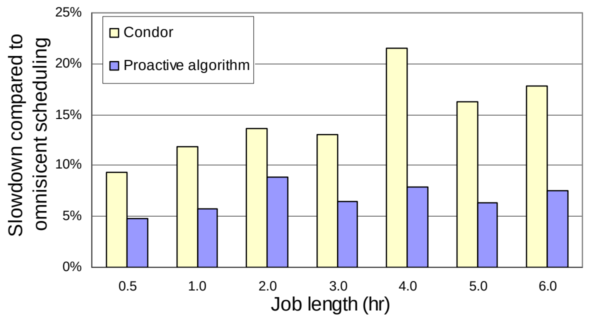


Figure 16. Slowdown of jobs under different scheduling algorithms. The baseline is the omniscient algorithm, which has full knowledge of host resource utilization in the future. The slowdown only counts jobs that finished successfully.

algorithm can make the perfect selection knowing future traces and its overhead is set to zero to serve as the baseline. The Condor-like algorithm is computationally fast, but its resource selection does not consider the available computing capability in the future. Therefore, it causes slowdown as high as 22% (for jobs of 4 hours). The proactive algorithm outperforms the Condor-like algorithm in all the cases. It improves the average makespan by 4% to 14%.

9. Conclusion and Future Work

We presented new techniques for predicting the availability of resources (compute cycles) in fine-grained cycle sharing systems. Exploiting daily patterns in the resource use history, the techniques compute the probability that a resource will be available during a given, future time window. We use this capability to find the most suitable computer system on which to execute a computational application in the iShare Internet sharing system.

A semi-Markov Process (SMP) model underlies our prediction method. Experimental results show an average prediction accuracy of 86.5% and an added overhead of 0.006% to an application. We have also found that our techniques are resilient to irregularity in history data. When applying the prediction to a proactive job scheduler, it was able to improve job failure rate and job makespan, compared to non-predictive schemes.

We have developed the prediction method in the context of a system that exhibits computer workloads in a university student environment. These workloads were highly diverse; we expect the results to hold for different environments. Several parameters, such as the threshold for tolerable impact of a guest job on a host computer, may be adjusted. In ongoing work we are evaluating our techniques in new environments. We are also exploring new applications of the prediction techniques, such as the use in job schedulers that support migration and rescheduling.

References

1. Y. Altinok and D. Kolcak. An application of the semi-markov model for earthquake occurrences in north anatolia, turkey. *Journal of the Balkan Geophysical Society*, 2(4):90–99, 1999.
2. D. P. Anderson. Boinc: A system for public-resource computing and storage. In *5th IEEE/ACM International Workshop on Grid Computing*, November 2004.
3. D. P. Anderson and G. Fedak. The computation and storage potential of volunteer computing. In *Proc. of CCGrid '06*, pages 73–80, 2006.
4. B. Armstrong and R. Eigenmann. A methodology for scientific benchmarking with large-scale application. *Performance Evaluation and Benchmarking with Realistic Applications*, pages 109–127, 2001.
5. W. Bolosky, J. Douceur, D. Ely, and M. Theimer. Feasibility of a serverless distributed file system deployed on an existing set of desktop pcs. In *ACM SIGMETRICS Performance Evaluation Review*, pages 34–43, June 2000.
6. J. Brevik, D. Nurmi, and R. Wolski. Automatic methods for predicting machine availability in desktop grid and peer-to-peer systems. In *Proc. of CCGrid'04*, pages 190–199, 2004.
7. R. Buyya and M. Murshed. GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience*, 14, 2002.
8. C. Catlett. The philosophy of TeraGrid: Building an open, extensible, distributed terascale facility. In *Proc. of CCGrid'02*, 2002.
9. A. Chien, B. Calder, S. Elbert, and K. Bhatia. Entropia: Architecture and performance of an enterprise desktop grid system. *Journal of Parallel and Distributed Computing*, 63(5):597–610, 2003.
10. P. Dinda and D. O'Hallaron. An extensible toolkit for resource prediction in distributed systems. Technical Report CMU-CS-99-138, School of Computer Science, Carnegie Mellon University, July 1999.
11. P. A. Dinda and D. R. O'Halaron. An evaluation of linear models for host load prediction. In *Proc. of HPDC'99*, page 10, August 1999.
12. I. Foster and C. Kesselman. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11:115–128, 1997.
13. M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. of the ACM POPL'03*, pages 185–197, 2003.
14. http://setiathome.ssl.berkeley.edu/. SETI@home: Search for extraterrestrial intelligence at home.
15. http://www.spec.org/osg/cpu2000. "spec cpu2000 benchmark".
16. N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley. Predictive application-performance modeling in a computational grid environment. In *Proc. of HPDC'99*, pages 47–54, 1999.

17. D. Kondo, M. Taufer, C. L. Brooks, H. Casanova, and A. A. Chien. Characterizing and evaluating desktop grids: An empirical study. In *Proc. of IPDPS'04*, April 2004.
18. D. Long, A. Muri, and R. Golding. A longitudinal survey of internet host reliability. In *14th Symposium on Reliable Distributed Systems*, pages 2–9, September 1995.
19. M. Malhotra and A. Reibman. Selecting and implementing phase approximations for semi-markov models. *Commun. Statist. -Stochastic Models*, 9(4):473–506, 1993.
20. K. McDonell. Taking performance evaluation out of the 'stone age'. In *Proc. of the Summer USENIX Conference*, pages 8–12, 1987.
21. M. W. Mutka. Estimating capacity for sharing in a privately owned workstation environment. *IEEE Trans. On Software Engineering*, 18(4):319–328, 1992.
22. A. J. Oliner, R. Sahoo, J. Moreira, M. Gupta, and A. Sivasubramaniam. Fault-aware job scheduling for bluegene/l systems. In *Proc. of IPDPS'04*, pages 64–73, April 2004.
23. J. Plank and W. Elwasif. Experimental assessment of workstation failures and their impact on checkpointing systems. In *28th International Symposium on Fault-Tolerant Computing*, pages 48–57, June 1998.
24. X. Ren and R. Eigenmann. ishare - open internet sharing built on p2p and web. In *Proc. of EGC'05*, pages 1117–1127, February 2005.
25. X. Ren, Z. Pan, R. Eigenmann, and Y. C. Hu. Decentralized and hierarchical discovery of software applications in the ishare internet sharing system. In *Proc. of PDCS'04*, pages 124–130, 2004.
26. K. D. Ryu and J. Hollingsworth. Resource policing to support fine-grain cycle stealing in networks of workstations. *IEEE Trans. on Parallel and Distributed Systems*, 15(9):878–891, 2004.
27. R. Sahoo, M. Bae, R. Vilalta, J. Moreira, S. Ma, et al. Providing persistent and consistent resources through event log analysis and predictions for large-scale computing systems. In *Workshop on Self-Healing, Adaptive, and Self-Managed Systems*, June 2002.
28. R. Sahoo, A. J. Oliner, I. Rish, M. Gupta, et al. Critical event prediction for proactive management in large-scale computing clusters. In *Proc. of the ACM SIGKDD*, pages 426–435, August 2003.
29. D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2004.
30. K. Trivedi and K. Vaidyanathan. A measurement-based model for estimation of resource exhaustion in operational software systems. In *Proc. of ISSRE'99*, pages 84–93, November 1999.
31. D. Vyas and J. Subhlok. Volunteer computing on clusters. In *12th Workshop on Job Scheduling Strategies for Parallel Processing*, 2006.
32. R. Wolski. Experiences with predicting resource performance on-line in computational grid settings. *ACM SIGMETRICS Performance Evaluation Review*, 30(4):41–49, 2003.
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. L. Yang, J. M. Schopf, and I. Foster. Conservative scheduling: Using predicted variance to improve scheduling decisions in dynamic environments. In *Proc. of the ACM/IEEE conference on Supercomputing*, page 31, 2003.

35. Y. Y. Zhang, M. Squillante, A. Sivasubramaniam, and R. K. Sahoo. Performance implications of failures in large-scale cluster scheduling. In *10th Workshop on Job Scheduling Strategies for Parallel Processing*, June 2004.