Resource Failure Prediction in Fine-Grained Cycle Sharing Systems

Xiaojuan Ren Seyong Lee Rudolf Eigenmann Saurabh Bagchi
School of ECE, Purdue University
West Lafayette, IN, 47907
Email: {xren,lee222,eigenman,sbagchi}@purdue.edu

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 of a host. A characteristic of such resources is that they are generally provided voluntarily and their availability fluctuates highly. Guest jobs may incur* resource failures *because of unexpected resource unavailability. To provide fault tolerance to guest jobs without adding significant computational overhead, failure prediction is required. This paper presents a method to predict resource failures in FGCS systems. It applies a semi-Markov Process and is based on a novel failure model, combining generic hardware-software failures with domain-specific failures in FGCS. We describe the failure 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 accuracy above* 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 failures.*

1 Introduction

The opportunity of harvesting cycles on idle PCs over the Internet has long been recognized [19]. Distributed cycle-sharing systems have shown success through popular projects such as SETI@home [12], which have attracted a large number of participants donating time on their home PCs to a scientific effort. The PC owners voluntarily share the CPU cycles only if they incur no significant inconvenience from letting a foreign job (*guest process*) run on their own machines. To exploit available idle cycles under this restriction, fine-grained cycle sharing (*FGCS*) systems [25] allow 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 *resource failures* leading to undesirable completion time of guest jobs. The primary victims of such resource 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 being used (e.g., simulations containing several computation steps [2]). Therefore, response time rather than throughput is the primary performance metric for such compute-bound jobs. The use of this metric represents an extension to the traditional use of idle CPU cycles, which had focused on high throughput in an environment of fluctuating resources.

In FGCS systems, resource failures have multiple causes and have to be expected frequently. First, as in a normal multi-process environment, guest and host processes are running concurrently and competing for compute resources on the same machine. Host processes can be decelerated significantly by a guest process. Decreasing the priority of the guest process can only alleviate the deceleration in few situations [25]. To completely remove the impact on host processes, the guest process must be killed or migrated off the machine, which represents a resource failure. In this paper, we refer to such resource failures as *FRC* (**F**ailures caused by **R**esource **C**ontention). Another type of resource failures in FGCS is the sudden unavailability of a machine — *FRR*, (**F**ailures caused by **R**esource **R**evocation). FRR happens when a machine owner suspends resource contribution without notice, or when arbitrary hardware-software failures occur.

To achieve fault tolerance for remote program execution, proactive job management, such as turning on checkpointing adaptively based on the results of failure prediction, has been proposed in the environment of large-scale clusters [20]. Proactive approaches achieve significantly improved job response time [31] compared to the methods which are oblivious to future failures. While these approaches can also be applied to FGCS systems, they require

successful failure prediction mechanisms. However, there have been few studies on failure prediction in large-scale distributed systems, especially in FGCS systems. Although several previous contributions have measured the distribution of general machine availability in networked environment [4, 21, 16], or the temporal structure of CPU availability in Grids [29, 19, 15], no work targets predicting failures caused by 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 failures in FGCS systems. We develop a multi-state failure model and apply a semi-Markov Process (SMP) to predict the *temporal reliability*, which is the probability that no resource failure will occur on a machine in a future time window of given length. The failure model integrates the two classes of failures, FRC and FRR, in a multi-state space which is derived from the observed values of *host resource usages* (the resource usages of all the host processes on a machine). The prediction does not require any training phase or model fitting, as is commonly needed in linear regression techniques. To compute the temporal reliability 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. A key observation leading to our approach is that the daily patterns of host users' workloads are comparable to those in the most recent days [19]. Deviations from these regular patterns are accommodated in our approach by the statistical method that calculates the SMP parameters.

We show how the prediction can be realized and utilized in a system, iShare [23], that supports FGCS. We evaluate our prediction techniques in terms of accuracy, efficiency, and robustness to noise (irregular occurrences of failures). 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. Host users on these machines generated highly diverse workloads, which are suitable for evaluating the accuracy of our prediction approach. The experimental results show that the prediction achieves the accuracy above 86.5% on average and above 73.3% in the worst case, and outperforms the prediction accuracy of linear time series models [9], which are widely used prediction techniques. The SMP-based prediction is also efficient and robust in that, it increases the completion time of a guest job of less than 0.006% and the intensive noise in host workloads disturbs the prediction results by less than 6%.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 presents the multi-state failure model and its derivation from empirical studies. The background and application of the semi-Markov Process are described in Section 4. In Section 5, implementation issues of the failure prediction in iShare are discussed. Experimental approaches and results are described in Section 6 and Section 7, respectively.

2 Related Work

The concept of fine-grained cycle sharing was introduced in [25], 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 failures happen if these facilities fail to prevent guest processes from impacting host processes significantly. In [25], the focus is on maintaining priority of host processes. By contrast, our work develops resource failure 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 [28] and critical event prediction [27, 26] in large-scale dedicated computing community (clusters). In order to anticipate when a system is in danger of crashing due to software aging, the authors of [28] proposed a semi-Markov reward model based on system workload and resource usage. However, the data they collected tend to fluctuate a great deal from the supposed linear trends of resource exhaustion rate, resulting in prohibitively wide confidence intervals. The work in [27, 26] predicted the occurrences of general error events within a specified time window in the future. The presented analysis and prediction techniques are not well suited for failures occurring in FGCS, where resources are non-dedicated and their availability changes dynamically.

Emerging platforms that support Grids [10] and global networked computing [7] motivated the work to provide accurate forecasts of dynamically changing performance characteristics [9] of distributed compute resources. Our work will complement the existing performance monitoring and prediction schemes with new algorithms to predict failures in the environment of fine-grained cycle sharing. In this paper, we compare the commonly used linear time series algorithms which are related work to our SMP-based algorithm, and show that our algorithm achieves higher prediction accuracy, especially for long-term prediction.

Research efforts have analyzed machine availability in enterprise systems [21, 4], or large Peer-to-Peer networks [3] (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 up times to actual available resources that could be effectively exploited by a guest program in cycle-sharing systems. On

the other hand, our approach integrates machine availability into a multi-state failure 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 [19, 30, 15]. In [19], 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 [30] applied the 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 [15] 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 availability. Instead, we target at predicting the availability of both machines and their compute resources in FGCS systems.

3 Resource Failure Model

A failure model that represents the two types of resource failures, FRC and FRR, is the basis for detecting and predicting these failures. To define such a model, we study the level of observability to detect resource failures and how a failure model can be derived from the observability.

3.1 Observability of Resource Failures

FRR happen when machines are removed from the FGCS system by the owners or fail due to hardware-software faults. FRR can be detected by the termination of FGCS services, such as the service for job submission. This detection method indicates a two-state failure model for FRR: a machine is either available or unavailable; there are no other observable states in-between. For FRC, the failures happen when host processes incur noticeable slowdown due to resource contention from guest processes. Before terminating the guest processes, a FGCS system will first decrease their priority, with the expectation that the impact on host processes will disappear. These actions need to be modeled and the modeling requires the quantification of *noticeable slowdown* of host processes. The system uses observable parameters of host resource usage as indicators for the slowdown. By observable parameters, we mean parameters, such as CPU and memory utilization, that can be obtained without special privileges on the host machine. The reason of using these indicators is that, at runtime, it is not possible to measure the slowdown of the host processes directly because the performance without contention is not known. The overall technique we use is to determine the threshold for what constitutes *noticeable slowdown* of the host process and thus implies the occurrence of an FRC. Then, we use offline empirical studies to determine the values of the observable parameters of host resource usage when such slowdown occurs.

We use empirical studies instead of an analytical model, because developing such a model is very difficult, if not impossible, considering the complexities in OS resource management. To make sure that the empirical studies are not biased by arbitrary workloads, we use representative guest applications and a broad range of host applications. The experimental approaches and observations are discussed in the next section. Because the empirical studies are not the focus of this paper, we concentrate on deriving the resource failure model from the studies. Details for the experimental results can be found in a separate paper [22].

3.2 Empirical Studies on Resource Contention

In FGCS systems, guest applications are typically 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 seismic applications and Monte-Carlo simulations. 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.

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, only one guest process is allowed to run at one time on the same machine. The priority of a running guest process is minimized (using *renice*) whenever it causes noticeable slowdown on the host processes. If this does not alleviate the resource contention, the reniced guest process is suspended. The guest process resumes if the resource contention diminishes after a certain duration (1 minute in our experiments), otherwise it is terminated. In the experiments, the "noticeable slowdown" is quantified by the reduction rate of *host CPU usage* (total CPU usage of all the host processes running on a machine) going above an application-specific threshold (we chose a threshold of $> 5\%$). The reduction rate can be simply obtained by running the host processes in isolation and then together with a guest process.

3.2.1 Experiments on CPU Contention

To study the contention on CPU cycles, we created a set of synthetic programs. The main component in each program is a loop with some computation and process sleeping. To isolate the impacts 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, these programs were ran on a 1.7 GHz Redhat Linux machine.

We measured the reduction rate of host CPU usage (total CPU usage of all the processes in a host group), when resource contention happens between a guest process (G) and the host group (H). We tested on host groups containing different numbers of host processes with isolated CPU usages of each process randomly distributed between 10% and 100%. G's priority was set successively to 19 (lowest) and 0 while H's priority was 0. The measured reduction rates were plotted as a function of isolated host CPU usage, L_H. Intuitively, in a time-sharing system, the chances that a guest process can steal CPU cycles decrease when there are more host processes running. This trend of decreasing CPU utilization for the guest process with increasing size of the host group is indeed experimentally seen for host group sizes from 1 to 5. When the size is beyond 5, the reduction saturates and therefore experiments do not need to be conducted for arbitrary sizes of the host group.

The experimental results on CPU contention indicate 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 reniced or terminated, respectively, to keep the slowdown below 5%. According to the trend described earlier, these thresholds would typically be for the host group of size 1. The reasoning above indicates that for any larger sized host group, the slowdown would be *less than* 5% at the thresholds Th_1 and Th_2.

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 set the guest priority to its lowest value whenever the guest process starts [7]. (The extreme case of terminating a guest application whenever a host application starts makes it a coarse grained cycle sharing system [12].) 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 have done a set of experiments to test if these two alternatives deliver a better model of CPU availability than using the two thresholds mentioned above. The details of the experiments are presented in [22]. From the results, we arrived at the conclusion that, gradually decreasing the guest priority introduces redundancy, while always taking the lowest guest priority slows down the guest process unnecessarily under light host workload ($L_H < Th_1$). The fine-grained values introduced by the first alternative are redundant, because they are observed to have the same effect as Th_2 in terms of the CPU availability for host processes. These experiments show that the choice for the two thresholds is not arbitrary. They reflect the levels of CPU availability accurately without introducing redundancy or imposing an overly-conservative restriction on guest processes.

In all the above experiments, we used randomly-generated host groups without relying on any specifics in OS scheduling. The existence of the two thresholds is therefore viewed 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.

3.2.2 Experiments on CPU and Memory Contention

To test the more complicated resource contention on both CPU and memory, we experimented with a set of real applications. For guest processes, we chose applications from the SPEC CPU2000 benchmark suite [13]. All of the applications are CPU-bound. Their working set sizes rang from 29 MB to 193 MB, which represent the range of memory usages of typical scientific and engineering applications. To simulate the behaviors of actual interactive host users on text-based terminals, we used the Musbus interactive Unix benchmark suite [18] 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" and created host workloads with different usages of memory and CPU. The CPU usages of these workloads range from 8% to 67%, and their memory usages range from 53 MB to 213 MB.

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 caused by the guest process, when the guest process's priority was set to 0 and 19, respectively. Two observations can be derived from the experimental results. First, memory thrashing happens when the total working set size of the guest and host processes (including kernel memory usage) exceeds the phys-

ical memory size of the machine. Changing CPU priority does little to prevent thrashing when the processes desire more memory than the system has. Second, when there is sufficient memory in the system, the occurrences of FRC caused by CPU contention, solely depend on the host CPU usage. And, in this scenario, the two thresholds, Th_1 and Th_2, can still be used to evaluate CPU contention. Therefore, the impact of host memory usage can be ignored whenever there is enough free memory to hold a guest process.

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

3.3 Multi-State Failure Model

The above experimental results on CPU contention 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 FRC. 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), respectively. Due to the isolation between CPU contention and memory contention, the 3-state model can be extended by adding a new failure state (S_4) for memory thrashing. These resource states are combined with FRR (S_5) to give a five-state model, as presented in Figure 1.

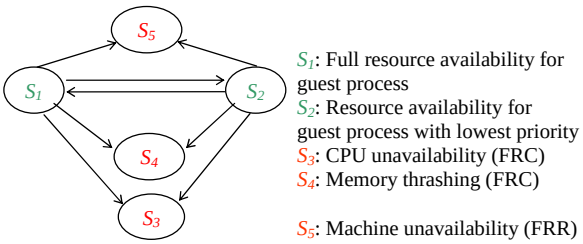

Figure 1. Multi state system for resource failures in FGCS.

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 due to 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 steadily higher than Th_2, any guest process (with default or lowest priority) must be terminated to relieve the resource contention;
- S_4: When there is not enough free memory to fit the working set of a guest process, any guest process must be terminated to avoid memory thrashing;
- S_5: When the machine is revoked by its owner or incurs a system failure, FRR occurs 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 (last 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 which exceeds Th_2 will drop down shortly after several seconds. The transiently high CPU load may be caused by a user starting remote X applications or by some system processes.

The proposed prediction algorithm is to predict the probability that a machine will never transfer to S_3, S_4, or S_5 within a future time window. Note that, these three states represent unrecoverable failures for guest processes. Even if the CPU or memory usage of host processes drops significantly or the host is reintegrated into the system, the guest process is already killed or migrated off and no state is left on the host.

4 Semi-Markov Process Model

In the multi-state failure 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 failure model in Figure 1.

4.1 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 [17]. 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.

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

To calculate the interval transition probabilities for a continuous-time SMP, a set of backward Kolmogorov integral equations [17] were developed. 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.

4.2 Semi-Markov Process Model for Resource Availability

This section discusses how a discrete-time SMP model can be applied to the failure model presented in Figure 1. The goal of the SMP model is to compute a machine's temporal reliability, *TR*, which is the probability of never 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 2 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 2, $P_{i,j}(m)$ 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.

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

The matrices Q and H are essential for solving Equation 2. 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 5. 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) [19].

5 System Design and Implementation

The proposed prediction approach is implemented within an Internet-sharing system called *iShare* [23]. iShare is an open environment for sharing both HPC resources (from the Grid community), such as the TeraGrid facility [5], and idle compute cycles available from any Internet-connected host. This section introduces the fine-grained cycle sharing in iShare and shows how the resource failure prediction is implemented and utilized.

5.1 Fine-Grained Cycle Sharing in iShare

In iShare, a Peer-to-Peer (P2P) network is applied for resource publication and discovery [24]. The cycle-sharing happens when resource consumers submit guest jobs to the published machines. Existing techniques can be utilized to estimate the execution time [14] and the memory usage [11] of a guest job. A job scheduler would use these two quantities and pass them to the temporal reliability prediction. The predicted result can be used by the scheduler to select the machines with relatively high availability or to manage the job adaptively during its execution.

Figure 2 shows the iShare framework with resource failure prediction. The *Host Node* and the *Client* show examples of a provider and a consumer, 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 failures. 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.

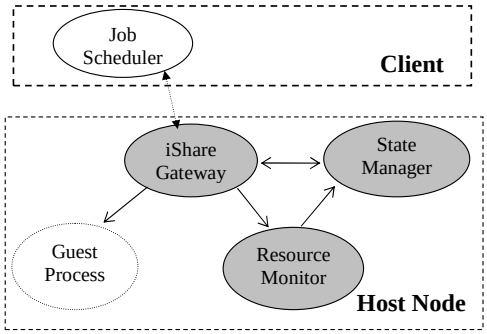

Figure 2. The software modules related to resource failure prediction in iShare. The four circles on host node depict processes created on the host. The arrows among them are for inter-process communication.

Upon the request of a job submission on a client, the client's *Job Scheduler* queries the gateways on the available machines for their temporary reliability within the future time window of job execution, and decides on which machine(s) the job would be executed. If a machine 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 renices, or kills the guest process accordingly. Checkpointing can also be used to migrate the guest process off the machine if resource failure happens.

There are two main design challenges to implement the framework shown in Figure 2. First, the resource monitor needs to be non-intrusive to the host machine where the monitoring takes place periodically. Second, because resource failure prediction happens in 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.

5.2 Non-intrusive Resource Monitoring

As discussed in Section 3, 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 on the machine. The resource monitor shown in Figure 2 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 machine unavailability has happened, the monitor compares the current 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 [3] for tracing machine up times, where a centralized unit is needed to probe all the nodes in a networked system.

5.3 Minimum Computation in Solving SMP

In our design, matrix sparsity in the SMP model is exploited to minimize the computational cost of failure prediction. Figure 3 describes the sparsity of the matrices Q, H and P in Equation 2. 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.

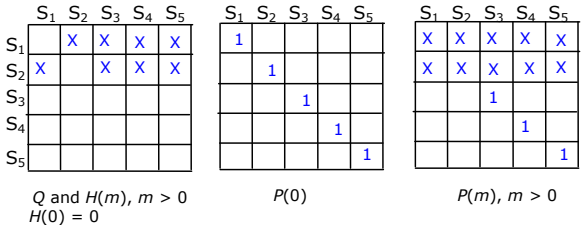

Figure 3. The sparsity of Q, H and P. The blank cells are for elements whose values are zero. Non-zero elements are labeled with a X (arbitrary values) or 1 (the value is 1).

With the sparsity shown in Figure 3, Q and $H(m)$ can be stored as an 8-element vector. As shown in Equation 2, 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 3 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 7 prove the effectiveness of the optimization in solving SMP.

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

6 Experimental Approach

We have developed a prototype of the system as described in Section 5. This section presents the experimental approach for evaluating the performance.

6.1 Experimental Testbed

All of our experiments were conducted on an FGCS testbed. The testbed contains a collection of 1.7 GHz Redhat Linux machines in a general purpose computer laboratory for student use at Purdue University. The local users on the tested machines are students from different disciplines. They used the machines for various tasks, e.g., checking emails, editing files, and compiling and testing class projects, which created highly diverse host workloads. Because the effectiveness of the SMP-based prediction is mainly affected by the variety of host workloads, the testbed proved appropriate to test our prediction algorithm comprehensively.

On each tested machine, processes launched via the iShare gateway are guest processes, and all the other processes are viewed as host processes. 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. Therefore, the resource behavior on these machines reflects the failure model in Figure 1. We installed and started a resource monitor on each machine in the testbed, which measured host resource usage every 6 seconds. We recorded these data for 3 months, from August to November in 2005, resulting in roughly 1800 machine-days of traces. The data contains the start and end time of each failure occurrence, the corresponding failure state (S_3, S_4, or S_5), and the available CPU and memory for guest jobs. Statistical results show that the number of failures happened on an individual machine during the 3 months ranges from 405 to 453 (for different machines). The frequency of failure occurrences is substantial, and this motivates the development of prediction techniques. Furthermore, our failure trace presents comparable patterns of host workload as observed by previous work on different testbeds [19].

We considered three 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. The prediction accuracy was also compared with that of a suite of linear time series models discussed in the next section. Finally, 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. The results are presented and analyzed in Section 7.

6.2 Reference Algorithm: Linear Time Series Models

A number of time-series and belief-network algorithms [27] appear in the literature for prediction of continuous CPU load or discrete events. After studying various algorithms, we chose linear time series models as reference points for our SMP-based prediction algorithm. Other existing algorithms are not well suited for use in the prediction of resource failures in FGCS. One example is the Bayesian Network model [27], which can be reduced to a state space without acyclic transition paths, inapplicable for the 5-state failure model in Figure 1. Time series models have been successfully applied in diverse areas, including host load prediction [9] and prediction of throughput in wireless data networks [6].

Linear time series models have been used for predicting CPU load in Grids [9]. The algorithms use linear regression equations to obtain future observations from a sequence of previous measurements. Compared to the SMP model, time series models only consider different load lev-

els 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 failure prediction. In our experiments, we used 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. The prediction accuracy is determined by the difference of the observed temporal reliability on the predicted and the measured state transitions.

We used a set of linear time series models implemented in the RPS toolkit [8]. The models are described in Table 1. 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.

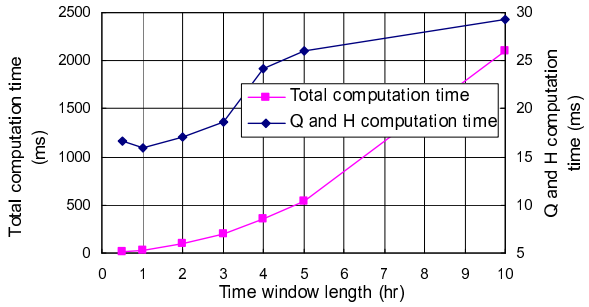

Figure 4. Computation time of resource failure prediction for time windows with different lengths. The prediction is to predict the probability that no failures will happen during a given time window.

7 Experimental Results

7.1 Efficiency of Failure Prediction

The overhead of the proposed resource failure prediction 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 resource failure prediction for time windows with different lengths. In Figure 4, 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 no failures will happen 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.

7.2 Accuracy of Failure 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 5 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 5, 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 5.

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.

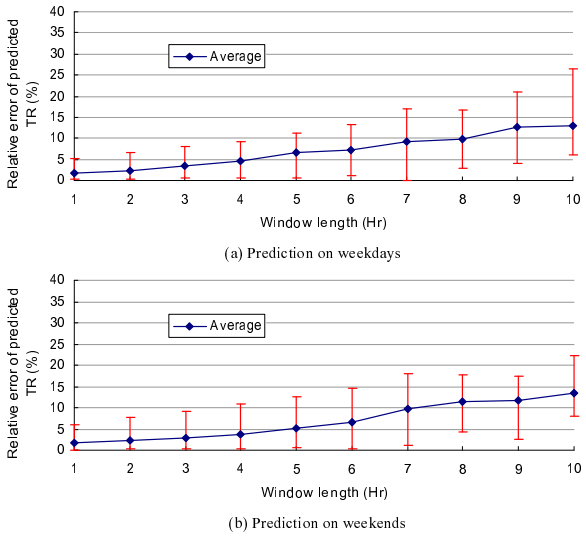

Figure 5. Relative errors of predicted TR. Each point plots the average error of predictions over 24 time windows with different start time ranging from 0:00 to 23:00, in steps of 1 hour. The bars show minimum and maximum prediction errors.

However, a large training set include 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 5 and measured the relative prediction errors which are plotted in Figure 6. "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 6 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 is important. It suggests a practical way to achieve best prediction accuracy by tunning the size of history data for arbitrary systems.

7.2.1 Comparison with Linear Time Series Models

To compare with our prediction algorithm, we applied linear time series algorithms from the RPS toolkit [8] to predict temporal reliability and measured their prediction accuracy. The tested time series models are shown in Table 1. In this experiment, we used the training and the test sets of equal

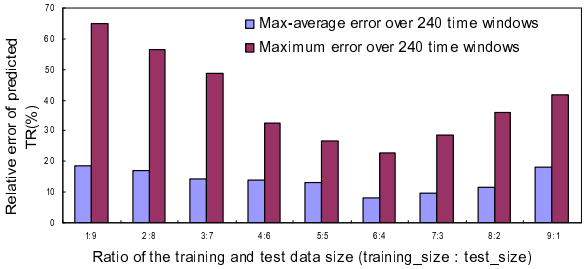

Figure 6. Relative prediction errors with different ratios of training and test data sizes for weekdays.

size. We ran the prediction on each time window (starting at different time and of different lengths) on all the tested machines. The maximum prediction error over different machines is used as the metric. It indicates the worst case of prediction accuracy.

Figure 7 shows the comparisons. As a representative case, we present the relative errors of predictions over time windows starting at 8:00 am on weekdays. Predictions for other time windows (including those on weekends) achieve similar results in terms of the relative differences among these algorithms. Due to space limit, we do not include all the results in this paper.

Table 1. 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
$ARMA$ (p,q)	Autoregressive moving average models with $p+q$ coefficients
LAST	Last measured value

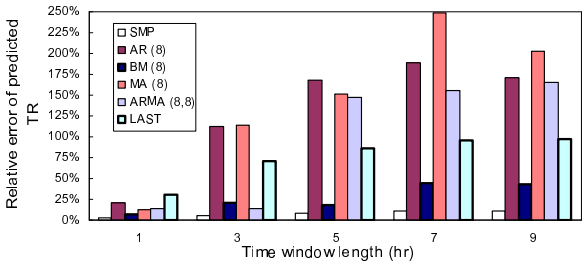

Figure 7. Maximum prediction errors of different algorithms over time windows starting at 8:00 am on weekdays.

From the results in Figure 7, we made the following ob-

servations. (1) Based on the relative prediction errors for the time windows studied, our SMP-based algorithm performs better than all of the 5 time series models. The advantage is more pronounced for predictions over large time windows. (2) Linear time series models are more adept at short-term prediction. This is because these models use multiple-step-ahead for predicting on large time windows and the prediction error increases with the number of steps lookahead.

7.3 Robustness of Failure Prediction

As we discussed earlier, the SMP-based prediction is able to accommodate the deviations from the load patterns that are comparable in recent days. This ability is confirmed by the high prediction accuracy presented in the previous section. To further test this ability, we study its robustness to noise (irregular occurrences of failures) 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 failure around 8:00am (when failure 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 8. The prediction discrepancy bars for large time windows ($T = 5, 10$ hrs) are often negligible compared to the values associated with small time windows. Hence some of the bars for large time windows do not show up in the figure.

Figure 8 shows that predictions on smaller time windows are more sensitive to noise. As shown by the bars for "$T = 1\ hr$", 4 instances of 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, they can be restarted upon the occurrences of resource failures without causing significant delay in job response times. For large jobs taking more than 2 hours, intensive noise (10 amounts of noise within 1 hour)

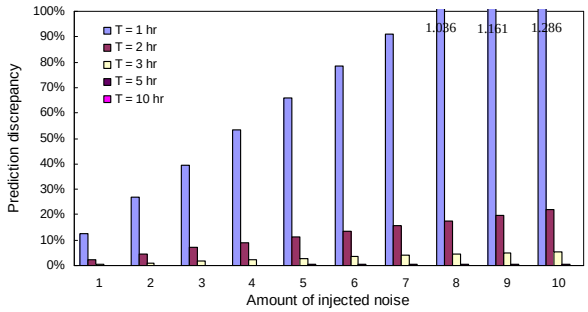

Figure 8. Prediction discrepancy with different amounts of noise injected to a training log for weekdays. T is the length of the future time window for the prediction. Prediction discrepancy is the relative difference between the prediction results using the training data with noise injection and those using the original training set.

causes less than 6% disturbance in our prediction algorithm. Therefore we can conclude that our prediction algorithm is robust enough for application in practical fine-grained cycle sharing systems.

8 Conclusion and Future Work

In this paper, we developed a multi-state model to represent the characteristics of resource failures in FGCS systems. We applied a semi-Markov Process (SMP) to predict the probability that no resource failure will happen in a future time window, based on the host resource usage history. The SMP-based prediction was implemented and tested in the iShare Internet sharing system. Experimental results show that the prediction algorithm achieves accuracy higher than 86.5% on average and adds less than 0.006% overhead to a guest job. The effectiveness of the prediction in accommodating the deviations in host workloads was also tested, and the results show that our prediction is resilient to noise in history data. In summary, the resource failure prediction is accurate, efficient, and robust.

In future work, we plan to test our prediction mechanisms on testbeds with different workload patterns, such as a testbed containing enterprise desktop resources. We expect that our prediction will perform well on the proposed testbeds, because, in this work, the prediction was already tested in an environment with highly diverse workloads. A next task is also to integrate our prediction framework with a proactive job scheduler in the iShare Internet sharing system.

Acknowledgment

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 algorithms used in our experiments.

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] 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.

[3] 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.

[4] 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.

[5] C. Catlett. The philosophy of TeraGrid: Building an open, extensible, distributed terascale facility. In *Proc. of CCGrid'02*, 2002.

[6] L. Cheng and I. Marsic. Modeling and prediction of session throughput of constant bit rate streams in wireless data networks. In *Proc. of WCNC'03*, March 2003.

[7] 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.

[8] 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.

[9] 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.

[10] I. Foster and C. Lesselmann. Globus: A metacomputing infrastructure toolkit. *The International Journal of Supercomputer Applications and High Performance Computing*, 11:115–128, 1997.

[11] 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.

[12] http://setiathome.ssl.berkeley.edu/. SETI@home: Search for extraterrestrial intelligence at home.

[13] http://www.spec.org/osg/cpu2000. "spec cpu2000 benchmark".

[14] 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.

[15] 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.

[16] 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.

[17] M. Malhotra and A. Reibman. Selecting and implementing phase approximations for semi-markov models. *Commun. Statist. -Stochastic Models*, 9(4):473–506, 1993.

[18] K. McDonell. Taking performance evaluation out of the 'stone age'. In *Proc. of the Summer USENIX Conference*, pages 8–12, 1987.

[19] M. W. Mutka. Estimating capacity for sharing in a privately owned workstation environment. *IEEE Trans. On Software Engineering*, 18(4):319–328, 1992.

[20] 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.

[21] 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.

[22] X. Ren and R. Eigenmann. Empirical studies of resource failure behavior in fine-grained cycle sharing systems. Technical Report ECE-HPCLab-05202, High-Performance Computing Lab, ECE, Purdue University, December 2005.

[23] X. Ren and R. Eigenmann. ishare - open internet sharing built on p2p and web. In *Proc. of EGC'05*, pages 1117–1127, February 2005.

[24] 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.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] R. Wolski. Experiences with predicting resource performance on-line in computational grid settings. *ACM SIGMETRICS Performance Evaluation Review*, 30(4):41–49, 2003.

[30] 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.

[31] 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.