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 host users. 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. Checkpointing and migration techniques help overcome such failures. However, these techniques, if oblivious to future failures, may cause significant overhead and thus undesirable job response times. This paper presents a method to predict resource failures in FGCS systems. The prediction method enables proactive management with greatly improved job response times. 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.**

I. Introduction

The opportunity of harvesting cycles on idle PCs over the Internet has long been recognized, since the majority of compute cycles go unused [23]. Distributed cycle-sharing systems have shown success through popular projects such as SETI@home [16], 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 processes*) run on their own machines. To exploit available idle cycles under this restriction, fine-grained cycle sharing (*FGCS*) systems [29] 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 jobs. Most of these jobs are batch programs. They are either sequential, or composed of multiple related jobs that are submitted as a unit 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.

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, such as CPU, memory, and I/O bandwidth, 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 [29]. 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 in remote program execution, prevalent systems [13], [32] deploy checkpointing and migration. However, such mechanisms cause nontrivial overhead which offset their benefits in improving the response time of guest jobs [37]. With the assumption of knowing future resource failures, proactive approaches, such as scheduling guest jobs to the machines least likely to fail and turning on checkpointing adaptively, are able to improve job response time effectively [24]. Successful failure prediction is key to these proactive approaches. However, there have been few studies on resource 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], [20], [25], or the temporal structure of CPU availability in Grids [19], [23], [34], no work targets predicting resource 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. The failure model integrates the two classes of failures, FRC and FCC, in a multi-state space which is derived from the observed values of *host resource usage*, that is the resource usage of all the host processes on a machine, upon the occurrences of failures. 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 usage 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 [23]. Deviations from these regular patterns are accommodated in our approach by the statistic method that calculates the SMP.

We will show how the prediction can be implemented and utilized in a system, iShare [27], that supports FGCS. Our implementation targets at the low computational overhead as well as the effectiveness of the failure prediction. To evaluate our prediction method, we monitored the host resource usage 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, which outperforms the prediction accuracy of linear time series models [11]. 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 II reviews related work. Section III 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 IV. In Section V, implementation issues of the failure prediction in iShare are discussed. Experimental approaches and results are described in Section VI and Section VII respectively.

II. Related Work

The concept of fine-grained cycle sharing was introduced in [29], 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. However, deploying such a system involves an OS upgrade, which can be unacceptable for resource providers. In FGCS systems, available OS facilities (e.g., *renice*) are utilized to limit guest processes' priority. Resource failures happen if these facilities fail to prevent guest processes from impacting host processes significantly. Instead of the focus on maintaining priority of host processes in [29], our work targets at resource failure prediction, so that guest jobs can be scheduled proactively with improved response times.

Related contributions include work in estimation of resource exhaustion in software systems [33] and critical event prediction [30], [31] 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 [33] proposed a semi-Markov reward model based on system workload and resource usage to estimate the time of failure of a system. However, the data they collected tend to fluctuate a great deal from the supposed linear trends, resulting in prohibitively wide confidence intervals. The work in [30], [31] predicted the occurrences of general error events within a specified time window in the future. The analysis and prediction techniques presented in these work are not well suited for failures occurring in FGCS, where resources are non-dedicated and their availability change dynamically.

Emerging platforms that support Grids [12] and global networked computing [8] motivated the work to provide accurate forecasts of dynamically changing performance characteristics [11], [35] of distributed compute resources. Our work will complement the existing performance monitoring and prediction schemes with new algorithms to predict failures caused by resource contention and resource revocation 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.

There have been some research efforts in measuring and analyzing machine availability in enterprise systems [4], [25], or large Peer-to-Peer networks [3], [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 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], [23], [36]. In [23], 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 [36] 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 aims at predicting over future time windows with arbitrary lengths. The authors of [19] 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.

III. Multi-state 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 the observability can be related to a rigorous mathematical model.

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 gateway 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 or suspend them, with the expectation that the impact on host processes will disappear. These actions need to be modeled and the modeling requires the ability to detect "noticeable slowdown" of host processes. However, because we do not know the original performance of host processes isolated from guest processes, it is practically difficult to measure the slowdown. To avoid this problem, we use observable parameters, specifically **measured** host resource usage, as indicators for "noticeable slowdown".

To study the connections between host resource usage and occurrences of FRC, we conducted a set of experiments to simulate resource contention among general guest and host processes in FGCS. We measured the values of host resource usage upon the occurrences of FRC. The experimental results indicate that, based on the measured host resource usage w/ or w/o guest processes running concurrently, it is able to detect the occurrences of FRC. The results also identify the existence of a set of thresholds for host resource usage, based on which a multi-state failure model for FRC can be developed.

In this section, our empirical studies on resource contention and the derived failure model are presented. To explain the generality of our experiments, the section first introduces the resource usage patterns of typical guest applications in FGCS.

A. Resource Usages of Guest Applications in FGCS

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 parameter-space searches. 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. Therefore, CPU and memory are the major resources contended by guest and host processes. For example, memory thrashing happens if a guest process's resident size exceeds the amount of free memory on a machine. In this case, resource failure happens and the guest process has to be terminated.

To avoid any adverse contention among multiple guest processes, in our FGCS system, no more than one guest process are allowed to run concurrently 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, otherwise it is terminated. The "noticeable slowdown" in our system is quantified by the slowdown of host processes going above an application specific threshold (we chose a threshold of $> 5\%$).

B. Experiments on Resource Contention

We conducted a set of experiments by running host processes with various resource usages together as an aggregated *host-group*. We measured the slowdown of the host-group as the reduction of its CPU utilization when a compute-intensive guest process is running concurrently. We experimented on CPU contention using a set of synthetic programs. Real benchmark applications were applied to study the contention on both CPU and memory.

Because the empirical studies are not the focus of this paper, we only present the experimental approaches and the observations drawn from the experiments here. We concentrate on how our resource failure model can be derived from the observations. Detailed experimental results are presented in a separate paper [26].

Experiments on CPU Contention

To study the contention on CPU cycles, we created a set of synthetic programs. 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 guest process is a 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*, that is the 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 usage randomly distributed between 10% and 100%. The same host-group settings were used when G's priority was set to 19 (lowest) and 0 respectively while H's priority was 0. The measured reduction rates were plotted as a funtion of isolated host CPU usage, L_H. There is no need to experiment on host-groups with exhaustive number of processes. This is because that, in a time-sharing system, the chances that a guest process can steal CPU cycles decrease when there are more host processes running. Therefore, for host-groups with the same L_H, the reduction rate of host CPU usage decreases as the number of processes increases.

The experimental results 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 the lowest values of L_H where the guest process needs to be reniced and suspended respectively to keep the slowdown below 5%. Because we experimented on randomly-generated host-groups without relying on any specifics in OS scheduling, we can conclude that the existence of the two thresholds are ubiquitous. Exact values of the two thresholds may change on OS systems with different CPU scheduling policies.

Based on the two thresholds, a 3-state failure model for CPU contention can be derived, where the guest process is running at default priority (S_1), is running at lowest priority (S_2), or is terminated (S_3). More specifically, the three states are:

- S_1: when the host CPU usage is low ($L_H < Th_1$), the resource contention due to a guest process can be ignored (slowdown of host processes is below 5%);
- S_2: when the host CPU usage is heavy ($Th_1 \leq L_H \leq Th_2$), the guest process's priority has to be minimized to make the slowdown of host processes unnoticeable ($\leq 5\%$);
- S_3: when the host workload is higher than Th_2, any running guest process (with default or lowest priority) has to be paused or terminated to relieve the resource contention.

Note that, under the first two states, L_H is approximately the same as the **measured** host CPU usage when a guest process running concurrently. Therefore, it is able to decide when a guest job is to be reniced or terminated by simply monitoring if the host CPU usage exceeds Th_1 or Th_2.

In practical FGCS systems, resource contention can be controlled in different ways. The two alternatives are, gradually decreasing the guest priority from 0 to 19 under heavy host workload, or minimizing the guest priority whenever it starts [8]. In the first alternative, S_2 is divided into fine-grained states indicating different guest priorities. Relating to the second alternative, S_1 and S_2 would be combined into one state. We have done a set of experiments to test if these two alternatives deliver a better model of CPU availability than the 3-state model discussed above. The details of the experiment settings and results are presented in [26]. From these results, we arrived at the conclusion that, gradually decreasing the guest priority introduces redundant states, while always taking the lowest guest priority slows down the guest process unnecessarily under light host workload ($L_H < Th_1$). The fine-grained states introduced by the first alternative are redundant, because they are basically the same as S_2 in terms of the CPU availability for guest processes. These experiments further prove that the choice for the three states is not arbitrary. The 3-state model reflects the levels of CPU availability accurately without adding redundant resource states or conservative restriction on guest processes.

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 four applications from the SPEC CPU2000 benchmark suite [17]. All of the four 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 [22] 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 six host workloads with different usages of memory and CPU.

We ran a guest process concurrently with each host workload on a 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 exceeds the free physical memory of the machine. The reason for the thrashing is that one process has page faults that require another process pages to be flushed to disk. Changing CPU priority does little to prevent thrashing when two processes desire more memory than the system has. Therefore the host processes make little progress no matter what priority the guest process takes. Second, the occurrences of the failures due to memory contention are orthogonal to the host CPU usage. On the other hand, when there is sufficient memory in the system, the occurrences of resource failures caused by CPU contention solely depend on the host CPU usage. Therefore, the impact of host memory usage can be ignored whenever there is enough free memory to hold a guest process. In this scenario, the two thresholds, Th_1 and Th_2, can still be used to evaluate CPU contention.

In conclusion, the memory contention and CPU contention can be isolated in detecting FRC. The 3-state model for CPU contention can be extended by adding a new failure state, S_4, for memory thrashing. The 4-state model represents the different levels of resource availability due to resource contention in FGCS.

C. Multi-State Failure Model

The resource states relating to FRR and FRC are combined to give a five state system presented in Figure 1. In Figure 1, S_3 indicates the situation where the host CPU usage has exceeded Th_2 for a duration (1 minute in our experiments) and the guest process has to be migrated off. S_4 presents resource failures caused by memory threshing, where the guest process needs to be terminated immediately. S_5 is for all the failures caused by resource revocation where resource immediately become offline. 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.

The transitions among S_1, S_2, and S_3 are decided by the measured host CPU usage. The corresponding thresholds (Th_1 and Th_2) are different on arbitrary OS systems. In our FGCS testbed, consisting of Linux systems, Th_1 and Th_2 are 20% and 60% respectively. Transitions to S_4 happen when the free memory size is less than the working set size of a guest

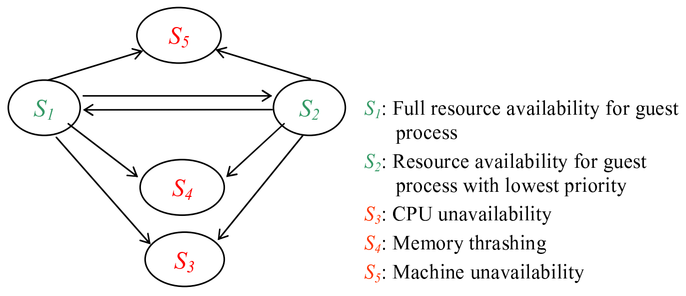

S_5
S_1
S_2
S_4
S_3
S_1: Full resource availability for guest process
S_2: Resource availability for guest process with lowest priority
S_3: CPU unavailability
S_4: Memory thrashing
S_5: Machine unavailability

Fig. 1. Multi state system for resource failures in FGCS. The arrows present state transitions. S_3 and S_4 are for failures caused by resource contention. S_5 is for failures caused by resource revocation.

process. Note that, states S_3, S_4, and S_5 are all unrecoverable failure states for guest processes. Even if the CPU usage or the 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.

IV. 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 states for resource availability. More importantly, for our objective, it enables the efficient prediction of temporal reliability. This section presents the background on SMP and how a SMP is applied for the resource failure prediction based on the failure model in Figure 1.

A. Background on Semi-Markov Process Models

Semi-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 [21]. An SMP is similar to a Markov process except that its transition probabilities depend on the amount of time elapsed since the last change in the state. 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.

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

The most important statistics of the SMP are the interval transition probabilities, P.

$$P_{i,j}(t_1, t_2) = Pr\{S(t_2) = j | S(t_1) = i\} \tag{2}$$

To calculate the interval transition probabilities for a continuous-time SMP, a set of backward Kolmogorov integral equations [21] are developed. Basic approaches to solve these equation include numerical methods [9] and phase approximation [21]. Numerical methods solve the Kolmogorov integration equations with particular mathematical techniques, such as Laplace-transform inversion. Phase approximation fits the holding time distribution to commonly used distribution functions, such as a Weibull or a Log-normal distribution. While these solutions are able to achieve accurate results in certain situations, they are not applicable to general SMP models and the worst performance may be affected adversely if an SMP can go through exponentially many transitions for a specific time interval [9]. 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 Equation 3. In our design, the time unit of discretization can be adjusted adaptively based on the temporal characteristics of resource state variation. Details for the adaptive approach are discussed in Section V.

B. Semi-Markov Process Model for Resource Availability

This section discusses how the 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 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.

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

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 usage on a machine. The details on resource monitoring are explained in Section V. 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 daily load patterns are comparable over the corresponding time windows over a weekday (a weekend) [23].

V. System Design and Implementation

The proposed prediction approach is implemented within an Internet-sharing system called *iShare* [15], [27]. iShare is an open environment for sharing both HPC resources from the Grid community, such as the TeraGrid facility [6], 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 in the system.

A. Fine-Grained Cycle Sharing in iShare

In iShare, a Peer-to-Peer (P2P) network is employed for resource dissemination among providers and consumers of resources [28]. The cycle-sharing happens when resource consumers submit guest jobs to the machines published and owned by the resource providers. Existing techniques can be utilized to estimate the execution time [18] and the memory usage [14] of the job. A job scheduler would use *a priori* knowledge of the estimated characteristics of the guest job and its arrival time to fit to our 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.

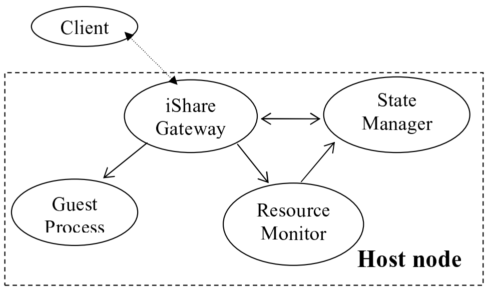


Fig. 2. The system framework of resource failure prediction in iShare. The four circles on host node depict processes created on the host. The arrows among them are for inter-process communication with Unix domain sockets.

Figure 2 shows the framework of resource failure prediction in iShare. The *Host Node* and the *Client* show examples of a provider and a consumer respectively. The prediction is invoked on the host node upon a request of job submission from the client. There are four entities on the host node, an *iShare Gateway* for communicating with remote clients and controlling local guest processes; a *Resource Monitor* for monitoring host resource usage, i.e., the total CPU usage and memory usage of host processes; a *State Manager* for storing history logs and predicting resource failures; and a guest process launched for a job submitted by the client. The first three entities exist on each host machine as daemons. They are started automatically when the providers turn on the iShare software to enable resource contribution. Therefore, the termination of the daemons indicates resource revocation and can be utilized for detecting FRR.

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 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, when transition to state S_2, S_3 or S_4 is detected. 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.

B. Non-intrusive Resource Monitoring

As discussed in Section III, 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 that we used for our experiments. The monitoring period can be set dynamically based on the frequency of the change of resource usage level by the host processes. The dynamic setting is done using an exponential increase in the period. Starting from a period (pd) of 1 sec, the period is doubled if for a threshold amount of time, no state change happens within the period pd. Once the monitoring period has stabilized, the reasoning is that no state change happens within a monitoring interval. However, random sampling is also done outside of the regular monitoring period to determine if the period has to be decreased.

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. 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 procedure forms an easy-to-use and accurate means of detecting the event indicating machine unavailability. It is a simple solution to the important problem of avoiding the need for administrator privileges in accessing system logs that indicate shutdown and restart of the machine. It is also more efficient and scalable compared to other techniques [3], [5], where a centralized unit is needed to probe all the nodes in a networked system.

C. Minimum Computation in Solving SMP

In our design, the matrix sparsity in the SMP model is exploited to minimize the computational cost of the resource failure prediction. Figure 3 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.

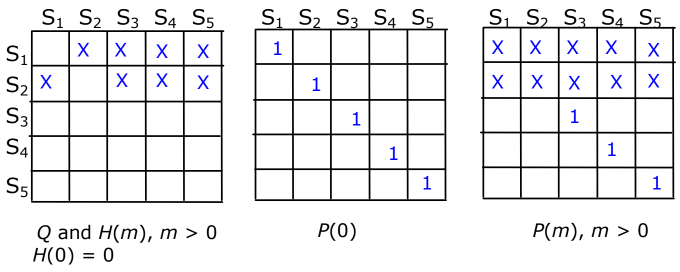


Fig. 3. The sparse pattern of Q, H and P. The blank cells are for elements whose values are zero. Non-zero elements are labeled with a X (arbitrary values) or 1 (the value is 1).

With the sparsity shown in Figure 3, Q and $H(m)$ can be stored as an 8-element vector rather than a 5 by 5 matrix. 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 $P_{2,5}$. In this way, the computational cost in each recursive step is decreased 3.125 (25 divided by 6) times compared to the original SMP model. 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 computational overhead of the optimized prediction algorithm is presented in Section VII, which proves the effectiveness of the minimum computation in solving SMP.

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

VI. Experimental Approach

We have developed a prototype of the system as described in Section V. This section presents the experimental approach for measuring the system.

A. Experimental Testbed

All of our experiments were conducted on a testbed for FGCS. The testbed contains a collect of 1.7 GHz Redhat Linux machines in a computer lab at Purdue University. Because our approach predicts the resource failures happened on an individual machine by exploring host resource usage from the recent history, the variety of host workloads rather than the scale of the testbed will affect the experimental results. In our testbed, the host users 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 host workloads with totally different resource usage patterns. Therefore, our testbed provides highly variable host workloads, which are 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. The resource contention between these two types of processes leads to the FRC as described in Section I. 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 sharing between iShare users and other host users on the testbed reflects the resource failure model presented in Section III. We installed our system and started a resource monitor on each machine in the testbed. The resource monitoring was performed every 6 seconds and instantaneous resource usage (usage since the last monitoring) were measured. The host resource usage on these machines were traced for 3 months.

We did 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 to evaluate the accuracy of the prediction. 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. All the experiment results are presented and analyzed in Section VII.

B. Comparative Algorithm: Linear Time Series Models

A number of time-series and belief-network algorithms [31] appear in the literature for prediction in continuous CPU load or discrete events. After studying various algorithms, we chose linear time series models to compare with 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 Rule-based Classification algorithm [31], which only provides the conditional probability of an event occurring, given the observation of other events. Meanwhile, time series models have been successfully applied in various areas, including host load prediction [11] and prediction of throughput in wireless data networks [7].

Linear time series models have been used for predicting CPU load in Grids [11]. The algorithms can predict for future observations from a sequence of previous measurements. Both the measured values and the predicted values have to be changing with roughly a fixed periodicity. 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 *TR* on the predicted and the measured state transitions.

We used a set of linear time series models implemented in the RPS toolkit [10]. The models are described in Table I. 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 and compared with that of our SMP-based prediction.

VII. Experimental Results

This section presents the experimental results on evaluating the efficiency, accuracy and robustness of our prediction method. The "resource failures" mentioned in this section refer to both classes of failures, FRC and FRR.

A. Efficiency of Resource 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 period of 6 seconds, the 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*) is plotted as a function of time window length. Recall that the prediction is to predict the probability that no failures will happen during a given time window. As expected, the prediction over a larger time window takes longer because of the more recursive steps needed. The total computation time follows a superlinear function (with exponent of 1.85) of the number of recursive steps. For the time window of 10 hours (the last point on the x-axis), the computation time for Q and H is 29.35 millisecond 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. We can conclude that our prediction algorithm is efficient and causes negligible overhead on the completion time of typical guest jobs in FGCS systems. Note that these jobs are typically large programs with completion time of the order of at least tens of minutes.

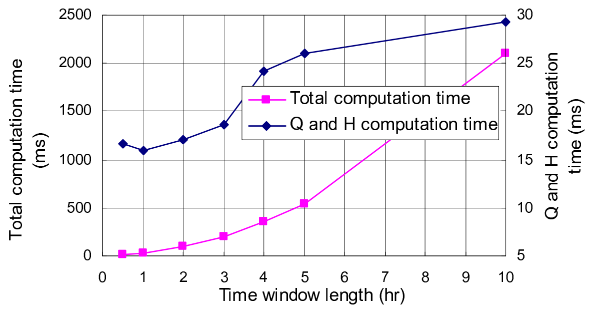
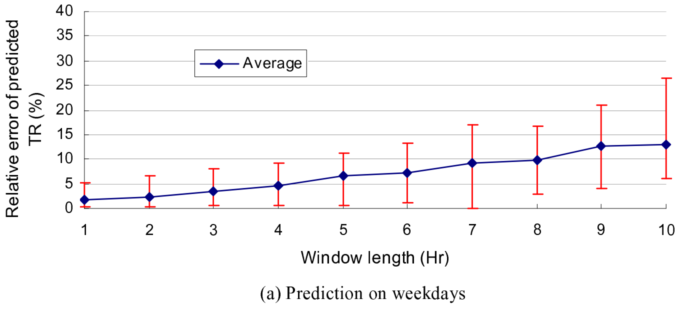


Fig. 4. Computation time of resource failure prediction for time windows with different lengths. The prediction is to predict the probability that no failures will happen during a given time window.

B. Accuracy of Resource 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. The actual observations from the test data set were used to calculate the $empirical\ TR$. The $predicted\ TR$ and the $empirical\ TR$ were used to compute the relative error as $abs(TR_{predicted} - TR_{empirical})/TR_{empirical}$. Figure 5 plots the relative errors of our prediction algorithm. The curve shows the average errors of predictions on time windows with different lengths, and the bars at each point show the related 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.



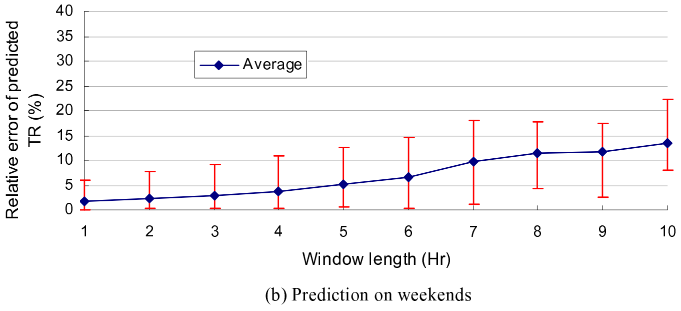


Fig. 5. Relative errors of predicted *TR*. Each point plots the average error of predictions over 24 time windows with different start time ranging from 0:00 to 23:00, in steps of 1 hour. The bars on each point show the related minimum and maximum prediction errors.

We also conducted a set of experiments to analyze the sensitivity of the prediction accuracy to the size of training sets. Intuitively, the prediction with larger training sets should perform better than that using smaller training sets. However, a large training set may include "older" data, which may bias the most recent pattern of host resource usage on the studied machine. We are interested in finding out if there exists a best choice of training size and what factors constitute such a choice.

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 prediction achieves the best accuracy (max-average error $\leq 7.96\%$ and maximum error $\leq 22.71\%$), when the ratio of training to test data sizes is 6:4. This observation can be used to decide the size of the history data to be used for the prediction given the test set. We argue that the observation is useful for the prediction on machines with highly diverse host workloads, which are similar to those in our testbed. For machines with relatively static host workloads, predictions using training data sets of different sizes tend to achieve close results.

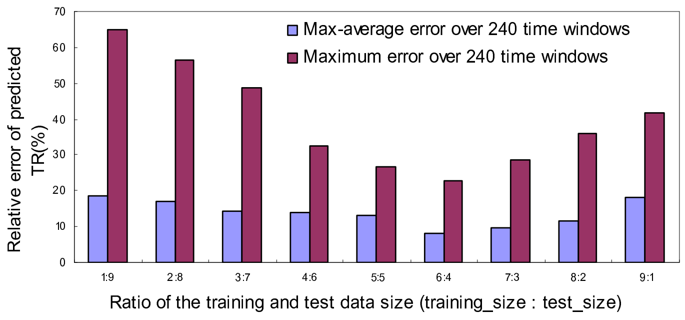


Fig. 6. Relative prediction errors with different ratios of training and test data sizes for weekdays

Comparison with Linear Time Series Models

To compare with our prediction algorithm, we applied linear time series algorithms to predict temporal reliability and measured their prediction accuracy. The tested time series models are shown in Table I. They can be obtained from the RPS toolkit implementation [10]. Figure 7 shows comparisons of various time series models and our prediction algorithm (SMP) in terms of prediction accuracy. As a representative case, we present the relative errors of the predictions over time windows starting at 8:00 am on weekdays. In this experiment, we used the training and test set with equal size.

TABLE I
LINEAR TIME SERIES MODELS

Model	Description
$AR(p)$	Purely 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

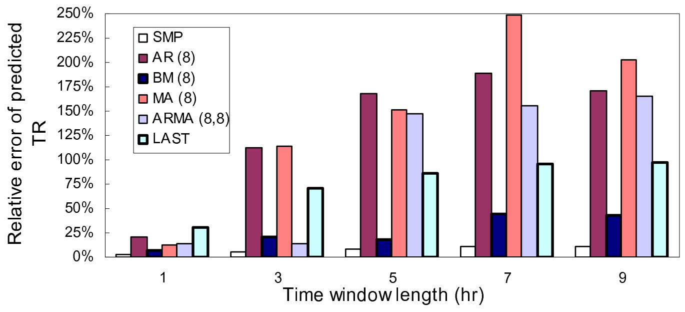


Fig. 7. Maximum prediction errors of different algorithms over time windows starting at 8:00 am on weekdays.

From the results in Figure 7, we made the following observations. (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. (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. In conclusion, our resource failure prediction algorithm achieves higher accuracy than all the 5 linear time series models, especially for predictions on large time windows.

C. *Robustness of Resource Failure Prediction*

To study the robustness of our prediction algorithm, 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 resource 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. The choice of log into which to inject the noise does not affect the prediction results; we randomly picked one log file for all the noise studies. With an arbitrary amount of noise, we measured the *prediction discrepancy* by comparing the prediction results with 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 do not show up in the figure.

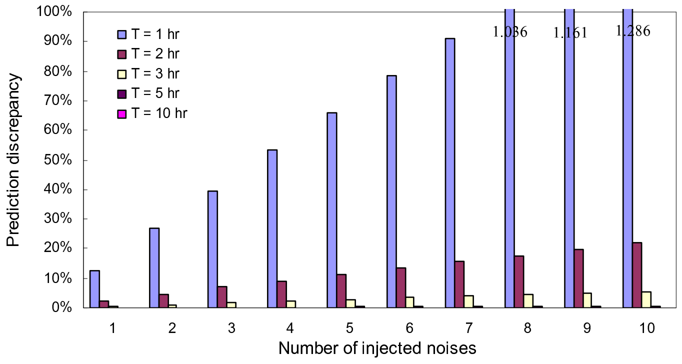


Fig. 8. Prediction discrepancy with different amounts of noise injected to a training log for weekdays. T is the length of the future time window for the prediction. Prediction discrepancy is the relative difference between the prediction results by using the training data with noise injection and those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

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 time. For large jobs taking more than 2 hours, intensive noise (10 amounts of noise 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 fine-grained cycle sharing systems.

VIII. 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 adds less than 0.006% overhead to a guest job and the prediction accuracy is higher than 86.5% on average. The effectiveness of the prediction in accomodating the deviations of host workloads was also tested, and the results show that the impact of the deviations on our prediction is negligible. These results in total verify that our resource failure prediction is efficient, accurate and robust.

In future work, we study factors in the data set that further improve the prediction accuracy. A candidate is information of the specific day of the weekday or weekend. This would involve a longer period of monitoring and data collection. An immediate task is to integrate our prediction framework with a proactive job scheduler in the iShare Internet sharing system.

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 *CCGrid'04*, pages 190–199, April 2004.

[5] F. E. Bustamante and Y. Qiao. Friendships that last: Peer lifespan and its role in p2p protocols. In *International Workshop on Web Content Caching and Distribution '03*, September 2003.

[6] Charlie Catlett. The philosophy of TeraGrid: Building an open, extensible, distributed terascale facility. In *Proc. CCGRID*, 2002.

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

[8] 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.

[9] G. Ciardo, R. Marie, B. Sericola, and K. S. Trivedi. Performability analysis using semi-markov reward processes. *IEEE Trans. Comput.*, C-39(10):1251–1264, 1990.

[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 *HPDC'99*, page 10, August 1999.

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

[13] W. Gentzsh. Sun Grid Engine: towards creating a compute power grid. In *Int. Symposium on Cluster Computing and the Grid*, 2001.

[14] M. Hofmann and S. Jost. Static prediction of heap space usage for first-order functional programs. In *Proc. of the 30th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 185–197, 2003.

[15] http://peak.ecn.purdue.edu/ParaMount/iShare/. The ishare project.

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

[17] http://www.spec.org/osg/cpu2000. Spec cpu2000 benchmark.

[18] N. H. Kapadia, J. A. B. Fortes, and C. E. Brodley. Predictive application-performance modeling in a computational grid environment. In *Proc. HPDC*, pages 47–54, 1999.

[19] D. Kondo, M. Taufer, C. L. Brooks, H. Casanova, and A. A. Chien. Characterizing and evaluating desktop grids: An empirical study. In *IPDPS'04*, April 2004.

[20] 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.

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

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

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

[24] A. J. Oliner, R.K. Sahoo, J.E. Moreira, M. Gupta, and A. Sivasubramaniam. Fault-aware job scheduling for bluegene/l systems. In *IPDPS '04*, pages 64–73, April 2004.

[25] 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.

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

[27] X. Ren and R. Eigenmann. ishare - open internet sharing built on p2p and web. In *European Grid Conference*, pages 1117–1127, February 2005.

[28] X. Ren, Z. Pan, R. Eigenmann, and Y. Charlie Hu. Decentralized and hierarchical discovery of software applications in the ishare internet sharing system. In *Proc. PDCS*, 2004.

[29] K. D. Ryu and J. Hollingsworth. Resource policing to support fine-grain cycle stealing in networks of workstations. *IEEE Transactions on Parallel and Distributed Systems*, 15(9):878–891, 2004.

[30] R.K. 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.

[31] R.K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, et al. Critical event prediction for proactive management in large-scale computing clusters. In *Proceedings of the ACM SIGKDD*, pages 426–435, August 2003.

[32] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The condor experience. *Concurrency - Practice and Experience*, 17(2-4), 2004.

[33] K. Trivedi and K. Vaidyanathan. A measurement-based model for estimation of resource exhaustion in operational software systems. In *Proceedings of the 10th International Symposium on Software Reliability Engineering*, pages 84–93, November 1999.

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

[35] R. Wolski, N. Spring, and J. Hayes. The network weather service: A distributed resource performance forecasting service for metacomputing. *Journal of Future Generation Computing Systems*, 15(5-6):757–768, 1999.

[36] 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.

[37] Y. Y. Zhang, M.S. 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.