# Empirical Studies on the Behavior of Resource Availability in Fine-Grained Cycle Sharing Systems

Xiaojuan Ren and Rudolf Eigenmann
School of ECE, Purdue University
West Lafayette, IN, 47907
Email: {xren,eigenman}@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 local host users. Such resources are generally provided voluntarily and their availability fluctuates highly. Guest jobs may fail unexpectedly, as resource becomes unavailable. We present empirical studies on the detection and predictability of resource availability in FGCS systems. A multi-state availability model is derived from a study of resource behavior. The model combines generic hardware-software failures with domain-specific resource behavior in FGCS. To understand the predictability, we traced resource availability in a production FGCS system for three months. We found that the daily patterns of resource availability are comparable to those in recent history. This observation suggests the feasibility of predicting future resource availability, which can be applied for proactive management of guest jobs.*

## 1 Introduction

Distributed cycle-sharing systems have shown success through popular projects such as SETI@home [6], which have attracted a large number of participants, contributing their home PCs to a scientific effort. These PC owners voluntarily share the CPU cycles only if they incur no significant inconvenience from letting a foreign job (*guest process*) run on their machines. To exploit available idle cycles under this restriction, fine-grained cycle sharing (*FGCS*) systems [14] 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 failures leading to undesirable completion times of guest jobs. The primary victims of such failures are large compute-bound guest applications, most of which are batch programs. Typically, they are either sequential or composed of multiple related jobs that are submitted as a group and must all complete before the results can be used (e.g., simulations containing several computation steps [1]). 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 unavailability has multiple causes and has 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 may be decelerated significantly by a guest process. Decreasing the priority of the guest process can only alleviate the deceleration in few situations [14]. To completely remove the impact on host processes, the guest process must be killed or migrated off the machine, which represents a failure. In this paper, we refer to such resource unavailability as *UEC* (**U**navailability due to **E**xcessive resource **C**ontention). Another type of resource unavailability in FGCS is the sudden leave of a machine — *URR*, (**U**navailability due to **R**esource **R**evocation). URR happens when a machine owner suspends resource contribution without notice, or when arbitrary hardware-software failures occur.

To achieve fault tolerance with efficiency for remote program execution, proactive approaches have been proposed in the environment of large-scale clusters [10]. These approaches explore availability prediction in job scheduling or runtime management. They achieve significantly improved job response time compared to the methods which are oblivious to future unavailability [18]. While proactive approaches can also be applied to FGCS systems, they require successful mechanisms for availability prediction, which in turn rely on the understanding of the characteris-

tics of resource availability. Unfortunately, little is known about resource availability in FGCS systems. Although several previous contributions have measured the distribution of general machine availability in networked environment [2, 11], or the temporal structure of CPU availability in Grids [8], no work has studied availability with regard to both resource contention and resource revocation in FGCS systems.

To understand the behavior of resource availability, we have conducted a set of empirical studies in a production FGCS system, iShare [12]. Our studies focus on the observability and predictability of when a resource will become unavailable. The studies on observability show that UEC can be detected by observing *host resource usage* (that is the resource usage of all the host processes on a machine) and URR is accompanied by the termination of FGCS services. Based on these studies, we develop a multi-state availability model and apply this model to detect resource unavailability in our FGCS system. To evaluate the predictability, we traced resource unavailability in an iShare testbed over a period of three months. A key observation made in this trace analysis is that the daily patterns of resource availability are comparable to those in the history. This suggests the feasibility to predict resource availability over an arbitrary time window. The prediction will use the history data for the corresponding time windows of recent days.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 presents the discussion on detecting resource unavailability and the empirical studies on resource contention. The multi-state availability model is described in Section 4. In Section 5, the studies on predictability, including trace collection and analysis, are presented.

## 2 Related Work

The concept of fine-grained cycle sharing was introduced in [14], where a strict priority scheduling system was developed and added to the OS kernel to ensure that host processes always receive priority in utilizing resources. However, 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 guest priorities. Resource unavailability happens if these facilities fail to prevent guest processes from impacting host processes significantly. In [14], the focus was on maintaining priority of host processes. By contrast, our work studies resource availability behavior with the final goal of improving completion times of guest jobs via proactive management.

Emerging platforms that support Grids [5] and global networked computing [4] motivated the work to provide accurate forecasts of performance characteristics [16] of distributed compute resources. Our work will complement the existing performance monitoring and prediction schemes with the empirical studies on resource availability. These studies take us a step further towards proactive job management in systems that support fine-grained cycle sharing.

There have been several research efforts in measuring and analyzing machine availability in enterprise systems [11, 2], 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 exploited by a guest program. By contrast, our approach integrates machine availability into a multi-state model, representing different levels of availability of compute resources.

A few other studies have been conducted on percentages of CPU cycles available for large collections of machines in Grid systems [17, 8]. The monitored CPU availability in [17] was obtained on time-shared systems assuming equal priorities of concurrent jobs. The authors of [8] studied both machine and CPU availability in a desktop Grid environment similar to FGCS. However, they conducted intrusive measurements by starting real applications on tested nodes. By contrast, we develop a model based on which resource unavailability, including CPU unavailability, can be detected in a non-intrusive way. Furthermore, we study the availability of general compute resources in a host system, while the studies in [8] solely focused on CPU cycles.

## 3 Detecting Resource Unavailability

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

In the rest of this section, we first discuss the observability of both types of unavailability, UEC (unavailability due to excessive resource contention) and URR (unavailability due to resource revocation). Then we present the details of our offline experiments on resource contention. The goal of these experiments is to determine the thresholds of host resource utilization that constitute UEC.

## 3.1 Observability of Resource Unavailability

URR happens when machines are removed from the FGCS system by their owners, or fail due to hardware-software faults without externally visible prior symptoms. System-internal symptoms, such as memory leakage and disk block fragmentation [15], have been considered to detect failures. However, in FGCS systems, such information is often inaccessible to external uses. Therefore, in the view of guest applications, machines may suddenly become offline and the resulting URR can only be detected in that FGCS services, such as the service for job submission, are terminated. This fact supports a two-state model for URR: a machine is either available or unavailable; there are no other observable states in-between.

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

## 3.2 Studies on Resource Contention

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

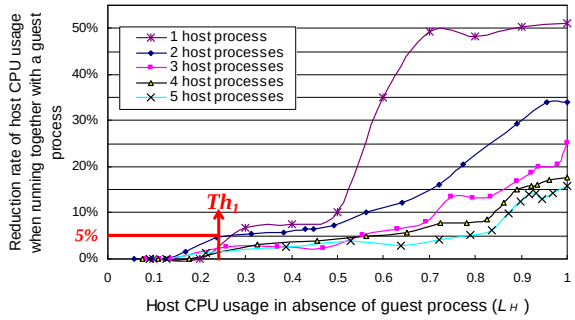
To make sure that the experimental results are not biased by arbitrary workloads, we use representative guest applications and a broad range of host applications. In FGCS systems, guest applications are normally CPU-bound batch programs, which are sequential or composed of multiple tasks with little or no inter-task communication. Such applications arise in many scientific and engineering domains. Common examples include Monte-Carlo simulations and seismic analysis tools [1]. Because these applications use files solely for input and output, file I/O operations usually happen at the start and the end of a guest job; file transfers can be scheduled accordingly to avoid peak I/O activities on host systems. Some of the guest applications also have large memory footprints. Therefore, CPU and memory are the major resources contended by guest and host processes. Host applications, on the other hand, can be computational tasks, OS command-line utilities, etc. In our experiments, they are represented by processes with various CPU and memory usages.

We conducted a set of experiments by running host processes with various resource usages together as an aggregated *host group*. To avoid any adverse contention among multiple guest processes, no more than one guest process is allowed to run concurrently on the same machine. The priority of a running guest process is minimized (using *renice*) whenever it 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 contention diminishes after a certain duration (1 minute in our experiments), otherwise it is terminated.

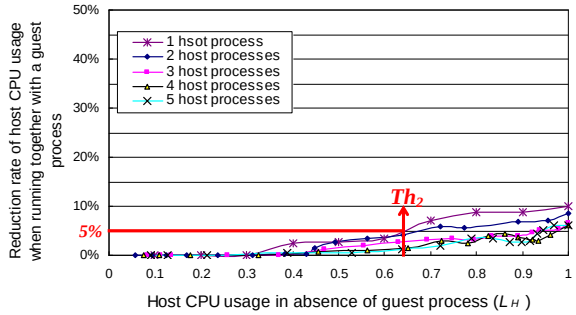
### 3.2.1 Experiments on CPU Contention

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

Figure 1 presents the reduction rate of host CPU usage (the total CPU usage of all the host processes in a host group), when a guest process ($G$) is running together with a host group ($H$). Figure 1 (b) shows the results when $G$'s priority is set to 19 (lowest) while $H$'s priority is 0. $L_H$ is the CPU usage of a host group without interference of guest processes. To create a host group with a given $L_H$ that consists of $M$ ($M > 1$) processes, we randomly chose $M$ host




(a) All processes have the same priority. $Th_1$ indicates the lowest value of $L_H$, above which host jobs can be slowed down by larger than 5%.




(b) Guest process takes the lowest priority. $Th_2$ indicates the lowest value of $L_H$, above which host jobs can be slowed down by larger than 5% even with minimum guest priority.

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

programs with different isolated CPU usages and ran them together without the guest process. If the total CPU usage of the $M$ processes was equal to $L_H$, they were chosen as a combination to generate the host group. For each tested host group, multiple combinations of host processes were used to measure the reduction rate of host CPU usage. The average of the measurements is plotted in Figure 1. This approach considers the fact that the same host workload can come from various individual host processes.

We tested host groups with $L_H$ ranging from $10\%$ to $100\%$, when $M$ was set to 1 to 5, respectively. There are two reasons why we chose $M$ to be no larger than 5. First, the total number of active processes started by a typical host user is usually in the range of tens. Second, as shown in Figure 1, the curves for different $M$ converge. That is, for the same $L_H$, the reduction rate of host CPU usage decreases as $M$ increases. Intuitively, in a time-sharing system, the chances that a guest process can steal CPU cycles decrease when there are more host processes running. When the size is beyond 5, the reduction saturates and therefore experiments do not need to be conducted for arbitrary sizes of the host group.

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

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

To verify that the existence of the two thresholds is not the simple result of our method of controlling guest priorities, we tested resource contention using different ways to adjust guest priorities, as used in practical FGCS systems. The two alternatives are, gradually decreasing the guest priority from 0 to 19 under heavy host workload ($L_H > Th_1$), or set the guest priority to its lowest value whenever the guest process starts [4]. (The extreme case of terminating a guest application whenever a host application starts makes it a coarse-grained cycle sharing system [6].) In the first alternative, fine-grained values between $Th_1$ and $Th_2$ are needed to indicate different guest priorities. Relating to the second alternative, only $Th_2$ is needed. We conducted a set of experiments to test if these two alternatives deliver a better model of CPU availability than using the two thresholds. In these experiments, we ran the same set of synthetic programs on the 1.7 GHz Linux machine.

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

The experiment for the second alternative was conducted via running a set of CPU-intensive guest processes (isolated CPU usage $\geq 70\%$) with priority 0 and 19 under light host workload ($L_H \leq 20\%$) respectively. The CPU usage of the guest process was measured and plotted in Figure 3. The differences between the two sets of bars in this figure show that, the guest CPU usage with priority 0 is about $2\%$ higher

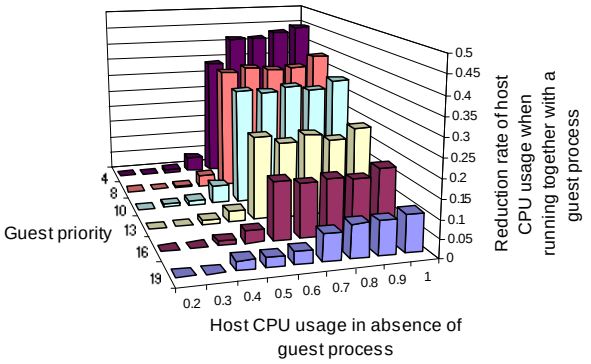



**Figure 2. Reduction rate of host CPU usage due to the contention from a guest process with different priorities. This figure implies that gradually decreasing guest priority does not make the guest process consume more CPU cycles.**

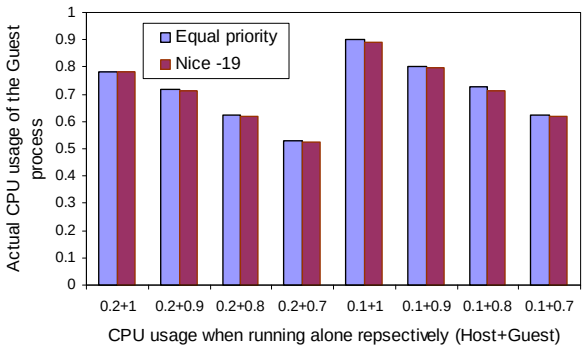



**Figure 3. CPU usage of the guest process with equal and lowest priority. The $x$-axis is the isolated CPU usage for the coexisting host and guest processes. For example, "0.2+1" means that the isolated host and guest CPU usage is $0.2$ and $1.0$, respectively. The figure shows that always taking the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

on average than that with priority 19. In FGCS systems, the 2% more CPU usage can make a significant difference in job completion time because some guest jobs take hours to finish. Therefore, the approach of always enforcing the lowest guest process priority is too conservative.

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.

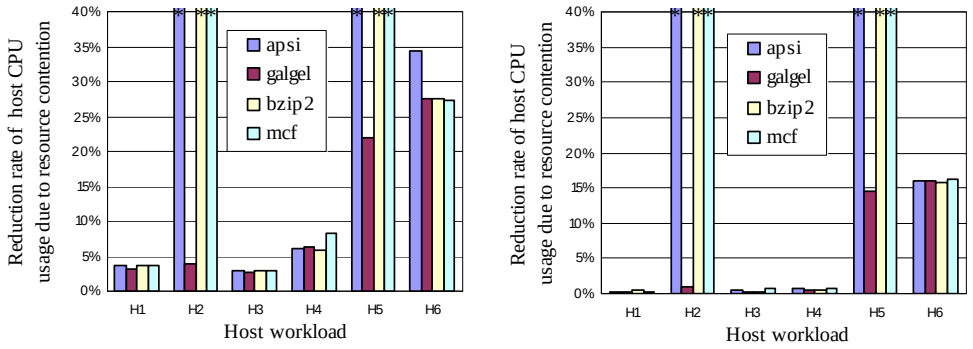
### 3.2.3 Experiments on CPU and Memory Contention

So far, we have considered CPU contention, only. To test the more complicated contention on both CPU and memory, we experimented with a set of larger applications. For guest processes, we chose four applications from the SPEC CPU2000 benchmark suite [7]: apsi, galgel, bzip2 and mcf, which are all CPU-bound. Their working set sizes range from 29 MB to 193 MB. To simulate the behaviors of actual interactive host users on text-based terminals, we used the Musbus interactive Unix benchmark suite [9] to create various host workloads. The created workloads contain host processes for simulating interactive editing, Unix command-line utilities, and compiler invocations. We varied the size of the file being edited and compiled by the "host users" to create host processes with different usages of memory and CPU. Table 1 lists the resource usages of the four guest applications and the six host workloads ($H_1$ to $H_6$) created by Musbus.

We ran a guest process concurrently with each host workload on a 300 MHz Solaris Unix machine with $384$ MB physical memory. For each set of processes, we measured the reduction of the host CPU usage caused by the guest process, when the guest process's priority was set to 0 and $19$ respectively. The results are shown in Figure 4.

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

In conclusion, memory contention and CPU contention can be isolated in detecting UEC. We do not need to consider the case of both resources under contention, since the




(a) Guest process with priority 0

(b) Guest process with priority 19

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

**Table 1. Resource usage of tested applications.**

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

additional effect due to one resource, when contention for another is already underway, is negligible.

## 4 Multi-State Availability Model

The presented results for resource contention in Section 3.2 show the feasibility of two thresholds, $Th_1$ and $Th_2$, for the measured host CPU load ($L_H$), that can be used to quantify the noticeable slowdown of host processes, thus the occurrences of UEC. In our FGCS testbed, consisting of Linux systems, $Th_1$ and $Th_2$ are 20% and 60% respectively. Based on the two thresholds, a 3-state model for CPU contention can be created, where the guest process is running at default priority ($S_1$), is running at lowest priority ($S_2$), or is terminated ($S_3$). Due to the isolation between CPU contention and memory contention, the 3-state model can be extended by adding a new unavailability state ($S_4$) for memory thrashing. These states are combined with

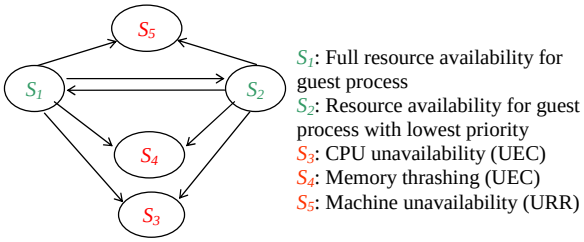



**Figure 5. Multi-state system for resource availability in FGCS.**

URR ($S_5$) to give a five-state model, as presented in Figure 5. Note that, the three states, $S_3$, $S_4$, and $S_5$, are all unrecoverable failure states 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.

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

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

# 5 Predictability Study: Trace Collection and Analysis

Based on the multi-state model presented in Section 4, we developed an unavailability detection module and traced resource availability in an Internet-sharing system, *iShare* [12], which supports FGCS. The goal is to derive the predictability of resource availability.

In iShare, resource publication and discovery [13] are enabled by a Peer-to-Peer network. Cycle sharing happens when resource consumers submit guest jobs to published machines. On each published machine, there is a resource monitor measuring CPU and memory usage of host processes periodically. To achieve non-intrusiveness to the host system, the monitor applies lightweight system utilities, such as *vmstat* and *prstat*. The monitor is started automatically when the resource provider turns on the iShare software and its termination indicates resource revocation.

We installed and started a resource monitor on each machine in an iShare testbed, which contains 20 1.7 GHz Redhat Linux machines in a general purpose computer laboratory for student use at Purdue University. The local users on these machines are students from different disciplines. They used the machines for various tasks, e.g., checking emails, editing files, and compiling and testing class projects, which created highly diverse host workloads. On a tested machine, processes launched via iShare are guest processes, and all the other processes are viewed as host processes. An occurrence of resource unavailability leads to the termination of the running guest process. 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, resource behavior on these machines is consistent to the availability model in Figure 5.

The availability of each tested machine was traced 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 occurrence of resource unavailability, the corresponding failure state ($S_3$, $S_4$, or $S_5$), and the available CPU and memory for guest jobs. In the following, we present our results of trace analysis.

**Table 2. Resource unavailability due to different causes.**

| Categories | Total amount | UEC | | URR |
|---|---|---|---|---|
| | | CPU contention | Memory contention | |
| Frequency | 405–453 | 283–356 | 83–121 | 3–12 |
| Percentage | 100% | 69–79% | 19–30% | 0–3% |

## 5.1 Resource Unavailability with Different Causes

Table 2 lists the statistics on resource unavailability due to different causes. In Table 2, frequency is the total amount of unavailability during the 3 months on an individual machine, and percentage shows the relative values. The two parameters were measured on each machine in the testbed, and the ranges on all the tested machines are given in Table 2.

Table 2 shows that high host CPU load is the main cause of resource unavailability in our FGCS testbed. Because the physical memory size is larger than 1 GB on all the tested machines, memory thrashing happens less frequently. In general, UEC happens much more often than URR in FGCS systems. As discussed earlier, URR has two sources: resource providers' intentional leave and software-hardware failures. In our testbed, the first source corresponds to machine reboots, which appear in our traces as URR with intervals shorter than one minute. Software-hardware failures are represented by URR lasting longer than one minute. By examining the interval lengths for all the recorded URR, we found that around $90\%$ of URR originated from machine reboots. This is not surprising because, on our tested machines, a local user may experience slowdown due to remote users other than iShare applications and then reboot the machine. Such machine reboots would be very rare on hosts used by only one local user, such as home PCs.

In conclusion, UEC constitutes the major part of resource unavailability in an FGCS system. Regarding our goal of studying the predictability, this means that the predictability is tightly correlated with the pattern of host workloads, especially host CPU load. While previous studies have observed the possibility to coarsely estimate the aggregated CPU availability of desktop machines [8, 4], it is difficult to relate the information directly to the predictability of resource availability. In particular, the understand-

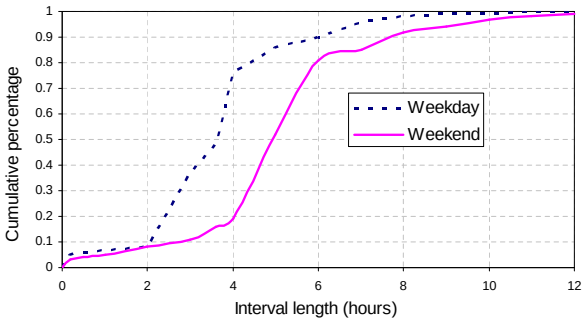



**Figure 6. Cumulative distribution of lengths of resource availability intervals.**

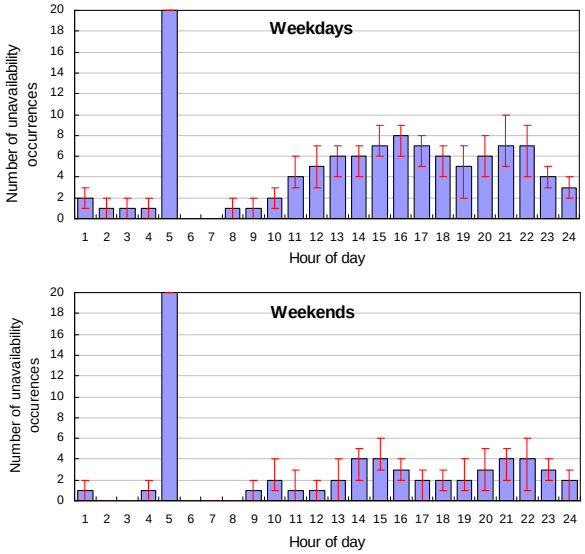



**Figure 7. Occurrences of unavailability during each hour in a day. The value at hour $i$ means the amount of unavailability occurred between ($i-1$,$i$).**

ing of temporal characteristics of availability intervals (that is the statistical lengths of time intervals throughout which a resource will be available) and the frequency of unavailability occurrences is key to obtaining direct measures of the predictability. We develop such characterizations in the next two sections.

### 5.2 Distribution of Lengths of Resource Availability Intervals

Resource availability intervals are periods during which a guest application may utilize host resources or get suspended, but does not fail. Facilities to predict such interval lengths provide the knowledge of how much computation power an FGCS system can deliver without interruption. Figure 6 plots the cumulative distribution of the duration of resource availability intervals, both for weekdays and weekends. These results were calculated from the traces of all the 20 machines during the 3 months.

From Figure 6, we see that intervals are shorter during weekdays, with an average of close to 3 hours, versus above 5 hours during weekends. Further, about $60\%$ of intervals are between 2 and 4 hours on weekdays, and between 4 and 6 hours on weekends. Such characteristics make it possible to coarsely estimate the available computational power that a guest application can expect from our FGCS system. We also note that both curves are relatively flat for intervals between 5 minutes and 2 hours, denoting that host resources rarely exhibit availability in that range. The small intervals that are shorter than 5 minutes constitute about $5\%$ among all measured intervals. We found that they are mainly small gaps resulting from variations of high host workloads. This implies that the system should wait for about 5 minutes before harvesting a machine recently released from heavy host workloads.

### 5.3 Daily Pattern of Failure Occurrences

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

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

day and lasts for about 30 minutes. Therefore, the amount of unavailability happened between 4 and 5 AM is equal to the total number of machines in the testbed (20). This "exception" also shows the correlation between unavailability occurrences and host workloads.

The most important observation obtained from Figure 7 is that, the deviations of unavailability frequency over the same time window across different weekdays (weekends) are small. This means that the daily patterns of resource availability are comparable to those in the recent history. Therefore, it is feasible to predict resource availability over an arbitrary future time window, if the prediction uses history data for the corresponding time windows from previous weekdays or weekends. In FGCS systems, the time window can be derived from the estimated execution time of a guest job. An aggressive prediction algorithm would accommodate the small deviations of resource availability among related time windows. One approach is to use statistics on history trace to alleviate the effects of "irregular" data.

## 6 Conclusion and Future Work

In this paper, we studied the detection and predictability of resource availability in fine-grained cycle sharing (FGCS) systems. The ultimate goal of this work is to develop availability prediction algorithms used for proactive job management. Based on the experiments of resource contention among guest and host jobs, a multi-state availability model is derived. This model enables the detection of unavailability by observing resource usage of host processes and liveness of FGCS services in a non-intrusive way. To test the predictability, we traced resource availability in a production FGCS testbed for three months. Analysis on the trace data shows that, while lengths of resource availability intervals can be coarsely estimated, the frequencies of unavailability occurrences are comparable for the same time windows across different days.

In the future work, we plan to collect trace on testbeds with different patterns of host workloads, for example a testbed containing enterprise desktop resources. We expect that data collected on the proposed testbeds will present similar predictability, because the testbed used in this work already provides highly diverse workloads. Another future effort is to develop availability prediction algorithms and evaluate them on various testbeds.

## References


[1] 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.

[2] J. Brevik, D. Nurmi, and R. Wolski. Automatic methods for predicting machine availability in desktop grid and peer-to-peer systems. In *Proc. CCGrid*, pages 190–199, April 2004.

[3] 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.

[4] 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.

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

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

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

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

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

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

[11] 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.

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

[13] 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. PDCS*, pages 124–130, 2004.

[14] 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.

[15] K. Trivedi and K. Vaidyanathan. A measurement-based model for estimation of resource exhaustion in operational software systems. In *Proc. ISSRE*, pages 84–93, November 1999.

[16] 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.

[17] 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.

[18] 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.