Failure-Aware Checkpointing in Fine-Grained Cycle Sharing Systems

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

Abstract

Fine-Grained Cycle Sharing (FGCS) systems aim at utilizing the large amount of idle computational resources available on the Internet. Such systems allow guest jobs *to run on a host if they do not significantly impact the local users of the host. Since the hosts are typically provided voluntarily, their availability fluctuates greatly. To provide fault tolerance to guest jobs without adding significant computational overhead, we propose failure-aware checkpointing techniques that apply the knowledge of resource availability to select checkpoint repositories and to determine* checkpoint intervals. *We present the schemes of selecting reliable and efficient repositories from the non-dedicated hosts that contribute their disk storage. These schemes are formulated as 0/1 programming problems to optimize the network overhead of transferring checkpoints and the work lost due to unavailability of a storage host when needed to recover a guest job. We determine the checkpoint interval by comparing the cost of checkpointing immediately and the cost of delaying that to a later time, which is a function of the resource availability. We evaluate the failure-aware techniques on an FGCS system called* iShare, *using trace-based simulation. The results show that our techniques achieve better application performance than the prevalent methods which use checkpointing with a fixed periodicity on dedicated checkpoint servers.*

1 Introduction

The opportunity of harvesting cycles on idle PCs over the Internet has long been recognized [11]. Distributed cycle-sharing systems have shown success through popular projects such as SETI@home [9], which have attracted a large number of participants donating time on their home PCs to a scientific effort. The PC owners voluntarily share the CPU cycles only if they incur no significant inconvenience from letting a foreign job (*guest process*) run on their own machines. To exploit available idle cycles under this restriction, fine-grained cycle sharing (*FGCS*) systems [23] 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 fluctuating availability due to either failures of the host machine, its removal from the FGCS system, or the eviction of a guest process causing resource contention. The primary victims of such resource volatility are large compute-bound guest programs that favor fast response times rather than high throughput. Most of these programs are either sequential or composed of several tasks with little communication in between. The numbers of tasks are relatively small, and the tasks are submitted as a group and must all complete before the results being used (e.g., simulations containing several computation steps [2]). Therefore, response time is the primary performance metric for such compute-bound jobs.

To achieve high performance in the presence of resource volatility, *checkpointing* and *rollback* have been widely applied [15]. These techniques enable an application to periodically save a checkpoint—a copy of the application's state— onto a stable storage that is connected to the computation node(s) through a network. The application recovers from a failure by rolling back to the latest checkpoint and continues execution on other computation node(s). While these techniques can also be applied to FGCS systems, a number of performance and feasibility issues must be considered.

First of all, general FGCS systems do not provide dedicated storage facilities [18, 23, 22]. While some hosts are willing to contribute their disk storage as well as CPU cycles, the storage also presents fluctuating availability. This leads to the possible loss of checkpoints, thereby causing the application to roll further back either to the last available *valid* checkpoints or the beginning of execution. Moreover, FGCS systems do not include a dedicated network that can efficiently handle the load of transferring checkpoints between the *computation hosts* (the host on which the guest process is executing) and the *storage hosts* (the hosts that have contributed storage for checkpointing). The high utilization of a shared network will negatively impact the performance of the overall system. A third issue is that generating a checkpoint and rolling back to the recent checkpoint both hurt application performance. While the penalty of rolling back decreases as an application checkpoints more frequently, the application suffers from the overhead of generating more checkpoints. To find a good tradeoff, one must determine a suitable *checkpoint interval*, which is the duration between the creation of two consecutive checkpoints. This issue is broader than for FGCS systems; however it assumes additional significance here due to the concern for availability of a set of checkpoints.

The main contributions of this paper are novel techniques that apply the knowledge of resource availability to address the above issues. (1) For storing checkpoints, we select *checkpoint repositories* among a set of storage hosts by considering both their future availability and their network performance. The selected repositories are used to save a set of redundant, small fragments generated by encoding the original checkpoint with erasure codes [1]. We formulate the problem of selecting checkpoint repositories as 0/1 programming models and apply a fast greedy algorithm to solve these models. This approach of dynamically selecting non-dedicated checkpoint storage distinguishes our work from the prevalent solution of using pre-installed checkpoint servers. (2) To determine the checkpoint interval, we design a *one-step look ahead* heuristic. It compares the cost of checkpointing immediately with the cost of delaying that to a later time. The costs are functions of the resource availability of the computation host and the storage hosts. The heuristic is different from the analytical methods proposed in related work, which computes the ideal checkpoint interval by assuming an arbitrary distribution function of failure arrivals [10, 12, 26]. We will show that our heuristic is more efficient in terms of the impacts of checkpointing on application performance.

We show how the failure-aware checkpointing techniques can be built upon our previous work on resource availability prediction in FGCS systems [22]. While these techniques can be applied to general compute-bound programs, we focus on *sequential* scientific programs as a representative case in this paper. We choose the overall execution time, or *makespan*, of these programs as the metric for examining the efficiency of our techniques. To obtain such metric, we collected traces of resource availability and host workloads from a production FGCS testbed over a period of 3 months, and tested the failure-aware techniques using trace-based simulation. We also compared with the methods that store checkpoints on pre-installed servers and perform checkpointing at a user-specified periodicity. These methods are widely applied in production systems such as Condor [25]. Our results show that the failure-aware checkpointing techniques achieve better application performance, while they do not rely on dedicated hardware resources, which are infeasible to be installed in FGCS systems.

The rest of the paper is organized as follows. Section 2 reviews related work. Section 3 presents a summary of our previous work on resource availability prediction. Our techniques of failure-aware checkpointing are described in Section 4 and Section 5. Section 6 discusses the issues of realizing these techniques in an FGCS system, called iShare [18]. Experimental approaches and results are described in Section 7.

2 Related Work

Checkpointing is a widely studied technique for ameliorating the effects of resource volatility in high-performance parallel computing and distributed systems [25, 14, 16, 6]. Related contributions include the checkpointing facilities provided in production systems and the research efforts on using resource availability to improve checkpointing performance. In the following, we compare our work with these two contributions.

Production systems that support Grids and global networked computing, such as Condor [25] and Sun Grid Engine [8], checkpoint applications upon failures or at user-defined intervals and store checkpoints onto a set of dedicated servers. We have employed their mechanisms of checkpoint generation and checkpoint transfers [3, 24] in our system design. However, installing dedicated checkpoint servers is not feasible for FGCS systems, because relying on such dedicated hardwares contradicts the objective of harvesting idle CPU cycles from existing systems. Moreover, recent research has shown that the overheads of checkpointing can offset its benefits if the checkpoint interval is not chosen carefully [28]. These facts motivate our work of applying resource availability prediction to select relatively reliable, non-dedicated checkpoint storage and to determine suitable checkpoint intervals, which will complement the existing facilities of checkpointing.

Some recent efforts in checkpointing research have used erasure coding for storage in distributed systems [6, 1, 5]. The relevant property of erasure codes that we use is that they are resilient to losses of some fragments of the checkpoint. Th previous work analyzed the performance of erasure coding in the systems with fast network and dedicated storage servers. By contrast, we target loosely coupled environments consisting of non-dedicated resources. We solve the problem of selecting checkpoint repositories among unstable storage resources.

Research has been done in solving the problem of optimal checkpoint interval selection [26, 10, 12]. Work in this area is typically predicated on simplifying assumptions. Most authors have assumed that the distribution of availability times conforms to an arbitrary function such as Poisson [26], allowing closed-form calculation of the optimal checkpoint intervals. In order to produce expressions that are analytically tractable, another common assumption is that failures do not occur during storing a checkpoint or during recovery [10]. These assumptions restrict generality, especially in FGCS systems, where the availability of resources fluctuates highly.

A few other studies have been conducted on using temporal or spatial information of resource availability in checkpointing [28, 29]. While these studies also considered the awareness of failures, none of them solve the problem of selecting reliable and efficient checkpoint repositories. The work in [28] considered turning on checkpointing only when failures were expected. Different from our work, they determined which jobs to checkpoint. To achieve fast turn around time in desktop Grid systems, the authors of [29] used checkpointing to migrate jobs to hosts located in night-time zones around the globe. The proposed heuristics solely focused on the issue of when and where to migrate the job.

3 Background of Resource Availability Prediction

Failure-aware checkpointing requires successful mechanisms for availability prediction. We have developed prediction methods that achieve high accuracy as well as efficiency appropriate for online uses. In our prediction techniques [22, 21], we applied a semi-Markov Process (SMP) to predict the *temporal reliability*, TR, of a computation host. This is the probability that the host will be available (i.e., not fail) throughout a given future time window. Solving the SMP gives us a value of $TR(x)$, the probability that there will be no failure between now and time x in the future. This algorithm does not require any model fitting, as is commonly needed in linear regression techniques. To compute the temporal reliability on a given time window, the parameters of the SMP are calculated from the host resource usages during the same time window on previous days. A key observation leading to our approach is that the daily patterns of host workloads are comparable to those in the most recent days [19, 11]. Deviations from these regular patterns are handled in our approach by the statistical method that calculates the SMP parameters.

The SMP-based prediction has been implemented in a system, iShare [18], that supports FGCS. In previous work, we focused on the specific resource of CPU cycles and evaluated our prediction techniques in terms of accuracy, efficiency, and robustness to noise (irregular occurrences of resource unavailability). The experimental results show that the prediction achieves an accuracy above 86.5% on average and above 73.3% in the worst case, and outperforms the prediction accuracy of linear time series models [7], which are widely used prediction techniques. The SMP-based prediction is also efficient and robust in that it increases the completion time of a guest job of less than 0.006% and the intensive noise in host workloads disturbs the prediction results by less than 6%.

In the following sections, we will show how the prediction techniques can be applied to optimize checkpointing efficiency.

4 Selecting Checkpoint Repositories

In our FGCS system, a guest application is checkpointed during its execution on a computation host. A checkpoint contains the entire memory state of the application and is used when the application fails and needs to continue on a new host. Application checkpoints cannot be generated right before job preemption or be stored locally on the computation host, because the host can be reclaimed unexpectedly without leaving enough time to migrate the checkpoint off. Therefore, we need to checkpoint applications during their execution and keep the checkpoints on networked storage.

Due to the lack of dedicated storage servers in FGCS systems, we store checkpoints on non-dedicated storage hosts – host machines that provide disk storage for guest users. Given that the number of storage hosts is usually larger than the number of checkpoint repositories required, we need to determine the selection of repositories. The goal is to minimize the network overhead (N) of transferring checkpoints, as well as the re-execution cost (R) caused by losing a checkpoint that is needed to recover a guest job. The two parameters present different performance concerns in FGCS systems. While the network overhead measures how a guest application would impact the hosts that share their network channels with the guest, the re-execution cost affects the guest application performance itself.

In our FGCS system, we split and encode a checkpoint of size n into $m+k$ fragments of size n/m. The checkpoint fragments are then stored on $m+k$ different storage hosts. We use erasure coding for this purpose, which guarantees that the checkpoint can be recovered even if up to k storage hosts fail. We will discuss the techniques to generate and encode checkpoints in Section 6. In the following, we show how to select the $m+k$ repositories from a set of, say, V storage hosts, given that V is much larger than $m+k$.

4.1 Network Overhead of Transferring Checkpoints

The network overhead of transferring a checkpoint fragment to or from a storage host can be quantified by the latency, that is the fragment size, n/m, divided by the effective end-to-end bandwidth (between the computation host and the storage host), bw. This latency represents how long a fragment uses the network. It is relevant in an FGCS system since a checkpoint is not to be considered permanent till it has been transferred over the network and saved to the remote storage.

A fine-grained modeling of the network overhead would consider how loaded a shared network channel is: the queuing at network routers/cards and the number of active network sessions. However, it is infeasible to collect such information in a large network consisting of non-dedicated resources. The effective end-to-end bandwidth, on the other hand, can be monitored and forecast by using facilities of the Network Weather Service (NWS) [27] that are publicly available. Therefore, it is a reasonable metric for evaluating the impacts of transferring checkpoints over shared network channels. Based on this metric, the total network overhead for transferring a checkpoint is the summation of the latencies incurred by each fragment, that is, $N = \sum_{i=1}^{V} C_i * \frac{n}{m}/bw_i$. This computation of N is with respect to the current computation host on which the guest job is executing. In the formula, bw_i is the end-to-end bandwidth of the storage host i; C_i is 1 if storage i is selected as a repository, otherwise it is 0.

4.2 Availability of Checkpoint Repositories

Similar to other resources in FGCS systems, checkpoint repositories are volatile. For example, to preserve the non-intrusiveness to host users, guest data are stored and retrieved only when there is no *intensive* I/O activities on a host. Therefore, an FGCS system needs to monitor the host I/Os and preempt the checkpoint read/write if necessary. In our FGCS system, if more than k repositories are unavailable for retrieving the checkpoint fragments, the application to be recovered will lose all the computation and restart. The incurred re-execution cost, thus, can be formulated as the total CPU time spent on the previous computation nodes.

We have applied the same semi-Markov Process (SMP) method for predicting the availability of both CPU cycles and disk storage, although the underlying availability models for the two types of resources are different. The difference comes from the diverse behaviors of resource contention. While the contention on CPU cycles is tightly related to the actual CPU usages of host applications [19], the contention on disk I/O bandwidth is related to both the I/O loads (i.e., the amount of data to read or write) and the locality of data accesses issued by the contended processes, indicating the needs of monitoring both parameters in the host system. However, it is not feasible for an FGCS system to measure the data locality of disk I/Os. To provide a practical way for detecting and predicting the availability of disk storage, we modeled disk I/Os as on/off events. More specifically, the operation of a checkpoint read/write needs to be paused if the rate of host disk I/Os is larger than a small threshold (e.g., 100 blocks per second) regardless of the data access locality. This threshold accounts for the I/Os of system daemons such as *syslogd*, which can be handled concurrently with checkpoint I/Os. In this way we can monitor the availability of a storage host using system facilities of *iostat* and *vmstat*.

The validity of such availability model has been verified through experiments [20]. Using this model, the SMP-based prediction achieves an accuracy higher than 93.7% on average and above 78.3% in the worst case. More details about the modeling and prediction methods can be found in [20]. In the rest of the paper, we refer to the predicted temporal reliability of storage as TRS. Recall that temporal reliability is the probability that the resource will never fail during a given time window.

4.3 The 0/1 Programming Models

We develop two schemes to select the checkpoint repositories: an optimistic scheme and a pessimistic scheme. The optimistic scheme assumes that most storage hosts present high availability. A set of checkpoint repositories is selected before a job starts on a computation host and is used during the entire execution on this host. A new set of repositories will be selected when the job continues on another computation host. A job may execute partially on a host and then be migrated for several reasons, including load balancing and data nearness. The pessimistic scheme, on the other hand, expects the storage hosts to fail frequently and updates the selection of repositories at each checkpoint interval. While the optimistic scheme is more efficient when the availability of storage hosts does not fluctuate greatly during the execution time of an application, it may lead to undesirable re-execution cost in highly volatile environments. Thus we need to choose from the two schemes for different levels of resource availability. In the following, we describe how these two schemes can be mathematically formulated into different 0/1 programming models.

In the optimistic scheme, the mean time to failure of the computation host, $MTTF_{cmp}$, is used as the expected time when the guest job fails and needs to be recovered from a recent checkpoint. If no more than k repositories fail within $MTTF_{cmp}$, the guest job can be continued. Otherwise, all the previous execution before starting on the current computation host, which is referred to as T_{curr} (the time units spent on performing useful computation for the guest job so far), and the execution on this host will be lost. We include T_{curr} to the re-execution cost because all the old checkpoints are discarded once the job moves to the current computation host, making the previous checkpoint repositories available for other guest applications. Thus, the expected re-execution cost can be estimated as, $R = (T_{curr} + MTTF_{cmp})$*Pr{ *more than* k *storage hosts fail within* $MTTF_{cmp}$}. The probability of more than k storage hosts failing is the summation of a series of values, each of which is the probability of an arbitrary number (larger than k) of storage hosts failing. To obtain a closed-form expression, we consider the probability that at least one host fails within the time of $MTTF_{cmp}$, that is $1 - \prod_{i=1}^{V} TRS'_i(MTTF_{cmp})$, where TRS'_i is 1 if host i is not selected, otherwise it is equal to TRS_i. This simplification leads to a larger expected value of R, but it does not conflict with our goal of selecting the most reliable storage hosts.

Given these terms, the optimistic scheme for selecting storage hosts can be presented by a 0/1 integer programming model shown in Equation 1.

$$\begin{aligned} &min\{ \tfrac{MTTF_{cmp}}{CI} * \textstyle\sum_{i=1}^{V}(C_i * \tfrac{n}{m}/bw_i) - \\ &\qquad (T_{curr} + MTTF_{cmp}) * \textstyle\prod_{i=1}^{V} TRS'_i(MTTF_{cmp}) \ \} \\ &\textstyle\sum_{i=1}^{V} C_i \ = \ m + k \\ &TRS'_i(t) \ = \ max[1 - C_i, \ TRS_i(t)] \end{aligned} \tag{1}$$

In the above equation, the value of $\frac{MTTF_{cmp}}{CI}$ (CI is the length of the next checkpoint interval) is an approximation of how many checkpoints are generated within $MTTF_{cmp}$.

Different from the optimistic model, the pessimistic scheme selects the repositories at each checkpoint interval right before the checkpoint is committed. The loss of more than k checkpoint fragments only matters if the guest job fails within the next interval. Thus the expected R can be calculated as, $R = T_{curr} * [1 - \prod_{i=1}^{V} TRS'(CI)] * \frac{1-TR(t+CI)}{TR(t)}$. In this formula, T_{curr}, TRS' and CI have the same definitions as in Equation 1; *t* is the time when the decision of repository selection is made. The expression of $\frac{1-TR(t+CI)}{TR(t)}$ represents the probability of the guest job not failing on the computation host before $t + CI$ given that it has survived till time t. The whole optimization model for the pessimistic scheme is described in Equation 2.

$$\begin{aligned} &min\{ \ \textstyle\sum_{i=1}^{V}(C_i * \tfrac{n}{m}/bw_i) - \\ &\qquad T_{curr} * \textstyle\prod_{i=1}^{V} TRS'(CI) * \tfrac{1-TR(t+CI)}{TR(t)} \ \} \\ &\textstyle\sum_{i=1}^{V} C_i \ = \ m + k \\ &TRS'_i(t) \ = \ max[1 - C_i, TRS_i(t)] \end{aligned} \tag{2}$$

In Equation 2, the weight on selecting reliable storage hosts increases as T_{curr}. The intuition is that, as more job gets done, it is more critical to keep the checkpoint on reliable storage. The weight also increases as the computation host is expected to fail with a high probability, that is $TR(t + CI)$ is close to 0. When $TR(t + CI)$ is large, we can actually bypass the computation of R in Equation 2. This way, we ignore the penalty of losing checkpoint fragments since the guest job is expected to be alive at the next checkpoint interval.

4.4 Greedy Algorithm to Solve the Models

As with general 0/1 integer programming models, Equation 1 and Equation 2 can both be solved by the methods of branch and bound [13]. However, these methods usually introduce nontrivial computation overhead, which impacts the application performance. To solve this problem, we propose an efficient greedy algorithm that iteratively includes the storage host causing minimum increment to the objective function in Equation 1 or Equation 2. The greedy algorithm has three steps:

- Initialization: For each storage host, compute TRS_i and N_i (calculated as $\frac{n}{m}/bw_i$). The objective function is initialized to 0.

- Bootstrap: Rank the storage hosts by N_i (in increasing order) and TRS_i (in decreasing order), respectively. This results in two sorted arrays, A_N and A_{TRS}. If a storage host appears in the first $m + k$ elements of both A_N and A_{TRS}, it is selected. Based on the currently selected storage hosts, the objective function is updated according to Equation 1 or Equation 2.

- Iteration: If the number of selected storage hosts is less than $m + k$, calculate the objective function by including one unselected host at a time. The host causing the minimum increase to the objective function is selected. If the number of the selected hosts is $m + k$, terminate the iteration.

Note that the relative ordering of the different hosts may change from one iteration to the next and therefore the objective function has to be evaluated for *all* the unselected hosts at each iteration. This observation arises from the nature of the objective function — the summation for the first term and the product for the second term. Thus, practically, if a number of hosts have been selected and the combined probability of all of them staying up is low, then in the following selection, network performance will be more emphasized. In the above algorithm, each iteration traverses all the unselected hosts and the number of iterations is $m + k$ in the worst case, resulting in the complexity of $O(V(m + k) + VlogV)$. Given that V is usually much larger than $m + k$, the greedy algorithm is more efficient than the branch and bound methods whose complexity is $O(\frac{V!}{(m+k)!(V-m-k)!})$.

5 Checkpoint Intervals

The optimal interval between checkpoints in an application execution balances the cost of generating checkpoints with the cost of recovering from a failure by rolling back to the latest checkpoint. While many analytical solutions to this problem have been proposed, we develop a heuristic that is more effective since it does not rely on the failures following a certain distribution. By simply looking at the future availability, the heuristic decides if checkpointing should be performed at each "step" during an application execution. The heuristic causes trivial computational overhead by reusing the availability predicted upon a job submission or recovery. We next discuss the details of this heuristic.

We refer to our method of deciding checkpoint intervals as a *one-step look ahead heuristic*: we divide a job execution into multiple steps of a fixed length which we will explain below. At the beginning of each step, we decide if the job needs to be checkpointed by comparing the cost of checkpointing at that moment and the cost of delaying it to the next step. The former cost is the *checkpoint overhead* (the delay required to generate a checkpoint) plus the rollback overhead resulted from the job failing before the next step. The latter cost is solely a rollback overhead. Note that rollback overhead is generally the cost of re-execution from the recently checkpointed job state to the state right before the job failed. Thus it is approximately the length of time from the recent checkpointing to the failure occurrence. While the checkpoint overhead can be measured offline and can be assumed, typically, as a constant which we refer to as C, the rollback overhead is a random variable given that the job may fail at any moment between the current step at t and the next step at $t + \delta t$. Therefore, we present the rollback overhead as an expected value derived from the temporal reliability of the computation host, i.e., TR. In short, the one-step look ahead heuristic can be formulated as a yes-or-no decision: the job is checkpointed at t if the condition in Equation 3 is true; it repeats checking the condition at the next step of $t + \delta t$.

$$\begin{array}{rcl} C + \int_t^{t+\delta t}(s-t)\frac{f(s)}{TR(t)}ds & \leq & \int_t^{t+\delta t}(s-t_0)\frac{f(s)}{TR(t)}ds \\ C & \leq & (t-t_0)\frac{TR(t)-TR(t+\delta t)}{TR(t)} \end{array} \tag{3}$$

In the above equation, $f(s)$ is the failure pdf for the computation host; $\frac{f(s)}{TR(t)}$ is the conditional probability of failing at s ($s \geq t$) given that the job has survived till time t. t_0 is the time of the previous checkpointing, and δt is the step length. Ideally, we would choose δt as small as possible so that the rollback overhead is minimized. However, there are a number of factors that prevent us from doing so. First, for a typical computation host, its temporal reliability rarely changes between very short time intervals. Thus, for most steps, the condition in Equation 3 remains false until $t - t_0$ gets large enough or $TR(t)$ gets close to 0. Another issue is that the checkpoint interval should be no shorter than the checkpoint latency: a new checkpoint can be generated only when the previous checkpointing has completed. Note that the checkpoint latency is composed of the checkpoint overhead and the latency of transferring checkpoint fragments over the network. It is typically several minutes in our FGCS system. In this work, we choose the step length as the checkpoint latency, which is the minimum we can use.

6 System Design and Implementation

In this section, we describe how the techniques of failure-aware checkpointing are implemented in our FGCS system, called iShare [18]. The process of checkpointing is composed of two parts: (1) generating a copy of the state of a guest job; (2) encoding the checkpointed state into fragments and storing them on a set of storage hosts. When a failure occurs, the guest job recovers in the following manner. First, a new computation host is chosen to take place of the failed host. Next, the new host collects data from the checkpoint repositories and decodes the data to regenerate the recent checkpoint. Finally, the guest job continues from the checkpoint. To implement these facilities of checkpointing and recovery, there are several design choices that need to be made. We discuss these design choices in the following.

6.1 Creating the Checkpoints

A program can be checkpointed either by the OS or by the program itself. We adopted user-level checkpointing because it does not require any modification to the OS or the guest program. This is essential for FGCS systems, where resource sharing happens among unknown, and possibly untrusted entities. In our FGCS system, we apply the user-level checkpointing library from Condor [25]. Guest programs only need to be re-linked to include this library. Checkpointing is invoked by signaling a guest process, and at recovery time, it appears to the process that it has just returned from the signal handler. The created checkpoint is first saved to the local disk of the computation host, and then is encoded using Erasure coding, as described in the next section, and written to the sockets that transfer the encoded fragments to the storage hosts. The original guest process is continued once the encoding is done. Its computation is overlapped with the network I/O of transferring the checkpoints. Thus the checkpoint overhead contains the time of creating checkpoint, writing to the local disk and encoding. The checkpoint latency includes both this overhead and the latency of transferring checkpoints. We will discuss the measure of checkpoint overheads in Section 7.

6.2 Encoding and Storing the Checkpoints

As discussed in Section 4.2, storage hosts are volatile, implying the need of redundancy in storing the checkpoints. To this end, we break a checkpoint into several fragments, adding some redundancy to enable recovery from a subset of the fragments. Two common techniques for splitting data into redundant fragments are the use of *erasure coding*, such as *information dispersal algorithms* [6], and the addition of parity information [16]. We apply Michael Rabin's classic information dispersal algorithm (IDA) [17] in our FGCS system, because it provides different fault tolerance levels by merely tuning certain parameters. More specifically, it allows coding a vector of size n into $m + k$ vectors of size n/m, such that regenerating the vector is possible using only m encoded vectors [6]. Thus, it is possible to tolerate k failures with a storage overhead of only kn/m elements. The encoding and decoding both happen on the computation hosts, adding to the overheads of the guest application. Our measurements show that this overhead is less than 20 seconds for the checkpoint size of 200 MB, which is acceptable for typical guest applications.

We implemented the above IDA scheme with the values of parameters m and k provided by guest users. These parameters affect both the data redundancy and the computation overhead of IDA. We leave the decision of choosing the tradeoff between reliability and performance to the users. We also implemented the APIs of transferring checkpoint fragments to/from networked storage. The network transfers to the different storage hosts are performed in parallel. In Section 7.1, we will show how the parallelism helps to reduce the network transfer latency.

6.3 Recovering from Failures

When a guest job can not continue on the computation host, the scheduler on the job submission client will select a new host. The scheduler collects the set of appropriate machines along with their load and temporal reliability. Then it invokes the resource selection algorithm. This algorithm is slightly different from the one used for job submission, as presented in our previous work [21]. First, the task size is updated to reflect the portion of the job finished so far. Second, the latency of collecting checkpoint fragments is added to the job makespan. This makes it more favorable to select the machine close to the previous checkpoint repositories. After the resource selection, the client updates the checkpoint repositories for the new computation host. Finally, the computation host regenerates the state from the checkpoint fragments and continues the execution.

As with all checkpointing systems, the checkpointing in FGCS must be made fault-tolerant itself. For example, the computation host or a checkpoint repository may become unavailable, while checkpointing or recovery is underway. To this end, our system makes sure that each checkpoint fragment remains valid until the next checkpoint has been completed. The computation host controls this process. After all the fragments of a checkpoint are transfered and stored successfully, the host notifies the repositories that they may discard the previous checkpoint fragments. This protocol ensures that there is always a valid copy of checkpoint on the repositories.

7 Evaluation

We have developed a prototype of the system described in Section 6. This section presents the experiments for evaluating the failure-aware checkpointing techniques in terms of their effectiveness in improving job makespans. We choose makespan out of the other metrics, such as work loss or resource utilization, because it directly measures the job response times observed by guest users.

We considered four sets of experiments. First, we measured the efficiency of our checkpointing and recovery implementations. The experiments were conducted by checkpointing a set of scientific programs and transferring the checkpoints over a campus-wide WAN. Second, we tested the effectiveness of the failure-aware schemes of checkpoint repository selection and compared with the alternatives that ignore resource availability in the selection. Third, we compared our one-step look ahead heuristic of checkpoint interval decision with the method of checkpointing at a fixed periodicity. Finally, we applied the two failure-aware techniques together. We compared with Condor's checkpointing mechanism of using pre-installed checkpoint servers and checkpointing periodically.

In the last three sets of experiments, we used trace-based simulation because the experiments need to be repeated on diverse guest jobs for the tested techniques. For each technique, we measured the average makespans of a set of synthetic guest jobs with diverse submission times, *normalized job lengths* (i.e., job completion time when running on a dedicated computation node without host workloads), and checkpoint sizes. To ensure that a submitted job will be executed on the same host for different experimental settings, we used the same job scheduling algorithm in all the experiments. The details of the scheduling algorithm can be found in a previous paper [21].

In the simulation, we set the checkpointing parameters, e.g., checkpoint overhead and the settings of m and k in

IDA, based on the experimental results of checkpointing real scientific programs. To obtain the traces of host workloads and resource availability, we monitored the host resource usages in an iShare testbed, which contains 20 1.5 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, creating highly diverse host workloads. We collected the traces for 3 months, from September 2006 to November 2006, resulting in roughly 1800 machine-days of traces.

We used the GridSim simulator [4] in our experiments. GridSim supports the modeling and simulation of a wide range of heterogeneous resources in Grids. In our experiments, it was applied to simulate the job execution on a host with specific workloads and availability characteristics. Using the traces collected from the iShare testbed, we simulated a campus-wide WAN consisting of 40 hosts. Half of the hosts were computation nodes with CPU speed of 2.0 GHz. The other 20 1.5 GHz machines with 18 GB disk space were used as storage hosts in our experiments. They were connected to the computation nodes via 10 Mbps network links. We added the failure-aware checkpointing modules to the GridSim simulator. The process of checkpointing and recovery was simulated by simply pausing a guest job for the time interval that reflects the corresponding overhead. GridSim simulated the network transfer of checkpoints by increasing the system clock according to the latency computed from the data size and the effective bandwidth [4].

7.1 Efficiency of Checkpointing and Recovery

The metrics of checkpoint overhead and recovery overhead present the inherent cost of checkpointing, which are important factors that affect application performance. We need to know their quantitative characteristics to set up our simulation. To obtain these metrics, we tested the efficiency of checkpointing and recovery using a set of programs in the SPEC CPU2000 benchmark suite. We chose these programs because they present the typical checkpoint sizes of scientific applications. These programs were linked with Condor's checkpointing library and the encoding was done following the IDA scheme.

To measure the checkpoint overhead, we ran the programs on a Pentium IV 2.0 GHz machine with 1 GB of memory. We collected the wall clock time of creating and encoding a checkpoint, respectively. Figure 1 shows the results. Due to the space limitation, we listed the results for the five programs with the largest checkpoint sizes among the benchmark suite. Their checkpoint sizes are listed in Table 1.

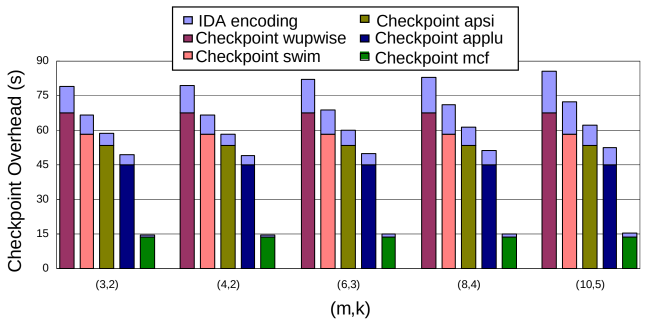


Figure 1. Checkpoint overhead for five programs in the SPEC CPU2000 benchmark suite. The x-axis shows the parameters for the IDA encoding: m is the number of fragments needed for regenerating the data and k is the number of redundant fragments.

Table 1. Checkpoint Size

Program	wupwise	swim	apsi	applu	mcf
Checkpoint size (MB)	175.4	154.1	120.6	105.5	22.5

In Figure 1, the top of each bar shows the time of IDA encoding and the lower part is the time of generating a checkpoint. Both times increase with the checkpoint size. The encoding time is also affected by the parameters of IDA: m (the number of basic fragments) and k (the number of redundant fragments). For all five programs, checkpoint creation constitutes more than 80% of the total overhead. Therefore, the use of IDA introduces marginal overhead while it enables to save checkpoints onto non-dedicated storage efficiently and reliably.

As discussed before, recovery overhead contains the time of retrieving checkpoints from the repositories and the time of IDA decoding. To measure the actual latency of network transfer, we used machines on a campus-wide WAN. The computation node was a Pentium IV 2.0 GHz machine. It was connected to the repositories, a set of Pentium IV 1.5 GHz machines, via 10 Mbps network links. Figure 2 lists the measurements of recovery overhead.

Two observations can be drawn from Figure 2. First, using IDA leads to lower recovery overhead than sending the whole checkpoint to a storage directly (the point corresponding to m = 1, k = 0). The reason is that, while IDA generates k redundant fragments, the size of data needed for recovery is always the original checkpoint size; reading the data concurrently from m storage nodes is more efficient than reading from a single storage. Another observation is that there exists a sweet spot (m = 6, k = 3) for the recovery overhead. This can be explained by the lower granularity of

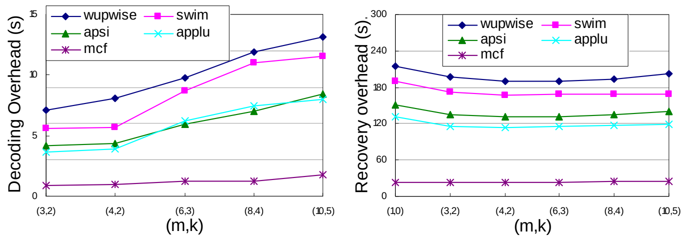


Figure 2. Recovery overhead for five programs in the SPEC CPU2000 benchmark suite.

parallelism in data transfer for smaller (m, k) and, on the other hand, the higher decoding overhead caused by larger (m, k). Because the latency of data transfer is much larger than the decoding overhead, the latency effect dominates in determining the sweet spot.

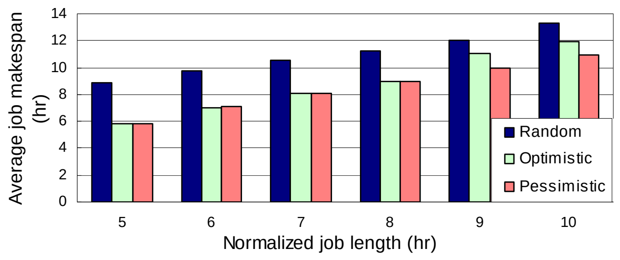
Based on the above results, we choose the parameter settings (m, k) as (6, 3) in the remaining experiments. We set the checkpoint size, n, to be between 100 MB and 500 MB. The corresponding overheads can be calculated by the linear models derived from the results in Figure 1 and Figure 2. Table 2 summarizes the parameters. These parameters reflect the system-specific performance of our checkpointing implementations. They are appropriate to use in our simulation, which has similar settings to the tested system. Note that the actual latencies of checkpoint transfer depend on the host workload traces used in the simulation. Thus we do not include the parameters of checkpoint latency and recovery overhead in the following table.

Table 2. Checkpointing Parameters

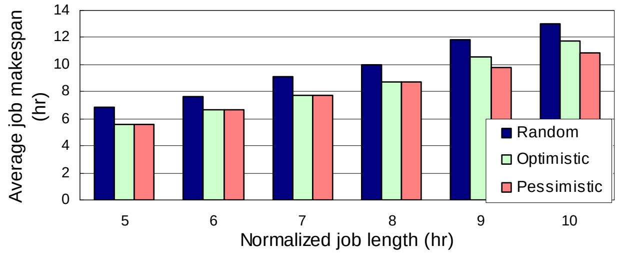
Checkpoint size, n	$(100, 500)$ MB
IDA	$m = 6$, $k = 3$
Checkpoint overhead (s)	$0.43 * n + 5.63$
Decoding overhead (s)	$0.06 * n - 0.29$

7.2 Evaluation on Checkpoint Repository Selection

This section presents the evaluation on the two schemes of checkpoint repository selection described in Section 4. These schemes aim to choose reliable repositories via considering storage availability. By analyzing the traces collected from the iShare testbed, we found that the storage hosts are available for about 70% of the time on average and failures happen more frequently during 11 AM and 10 PM, when the host workloads are relatively high. To specifically examine the benefits of applying the knowledge of resource availability in selecting the repositories, we compared with the method of random selection. This results in three strategies of repository selection: random, optimistic and pessimistic. For each strategy, we measured the average makespans over a total of 720 jobs, whose submission times distributed uniformly between 12 AM and 10 PM. For each submission time, we tested jobs with normalized lengths ranging from 5 to 10 hours. We excluded the jobs taking less than 5 hours because, by using the job scheduling algorithm that considers resource availability [21], they usually incur few failures and thus do not need to be checkpointed.



(a) Average makespans of the jobs running under low resource availability



(b) Average makespans of all the tested jobs

Figure 3. Average job makespans for different strategies of checkpoint repository selection. Optimisitic and pessimistic are our proposed strategies while random provides the baseline. Lower bars mean better performance.

To remove the impacts of checkpoint intervals on the repository selection, we set the jobs to be checkpointed hourly. Figure 3 show the results for the three selection strategies. Figure 3-(a) lists the average makespans over the jobs executed between 11 AM and 10 PM, during which resource availability is relatively low [19]. Figure 3-(b) lists the average makespans over all the 720 jobs.

Figure 3 shows that the failure-aware strategies achieve better job makespans than the strategies oblivious to storage availability. The former improves job makespans by 34.87% in the best case (for the jobs with normalized length of 5 hours in Figure 3-(a)). On average, by considering resource availability, the optimistic strategy and the pessimistic strategy achieve an improvement of 13.04% and

15.29%, respectively. We also see that, for jobs taking less than 9 hours, the optimistic strategy is slightly better than the pessimistic one (the difference is trivial and thus does not show up in the figure); while it is worse for longer jobs. The former observation comes from the computational overhead of more selections needed by the pessimistic strategy. This overhead will be pronounced for large V and $m+k$ (we set the two values to be 20 and 9, respectively, in our simulation). The latter is the case because the reliability of storage gets close to zero for very large $MTTF_{cmp}$, indicating that less than m storage hosts can survive through the execution on a computation host. It is worthwhile to point out that the poor performance of the optimistic scheme on long jobs can be ameliorated by choosing a larger k if there are enough storage hosts in the system. We will evaluate how the values of m and k affect the performance of the two failure-aware strategies in future work. According to the above results, when running relatively small jobs (e.g., shorter than 9 hours in our testbed) or using high level of redundancy in the IDA, it is better to use the optimistic strategy. Otherwise, the pessimistic strategy is more safe to use.

7.3 Evaluation on Checkpoint Interval Selection

We have conducted experiments to compare our one-step look ahead heuristic in checkpoint interval selection with the approach of checkpointing at a fixed periodicity. We do not compare with the analytical method of calculating checkpoint intervals [26] because it requires expensive computation to fit failures to arbitrary distribution models and to solve the optimal intervals. For example, while hyper-exponential distributions present accurate models for resource availability, it is numerically difficult to find the maximum likelihood estimator parameters [12]. Related work has shown that the incurred computational overhead leads to small positive effects on application performance even when using more accurate distribution functions [12].

In these experiments, to remove the effects of using different checkpoint repositories, we simulated a set of checkpoint repositories that were always available and were used by all the guest jobs. We used the same set of 720 jobs as described in Section 7.2 and measured the improvement of average job makespans when using the one-step look ahead heuristic over periodic checkpointing. We first chose the checkpoint period as 1 hour and measured the improvement over different checkpoint sizes. Then we set the checkpoint sizes of all the jobs to be 300 MB and examined the improvement by comparing with checkpointing at different periodicities. The results are shown in Figure 4 and Figure 5, respectively.

Figure 4 shows that the one-step look ahead heuristic improves the average job makespans for different checkpoint

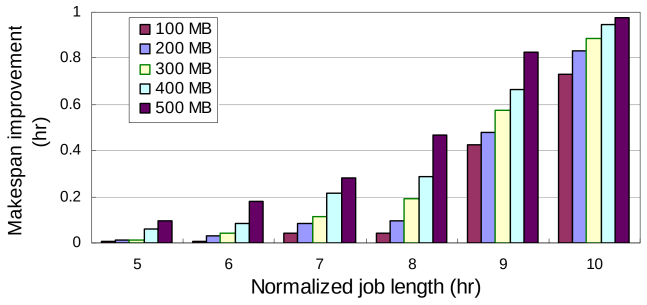


Figure 4. Improvement of average job makespans when using the one-step look ahead heuristic instead of checkpointing hourly. Each bar shows the results for a specific checkpoint size.

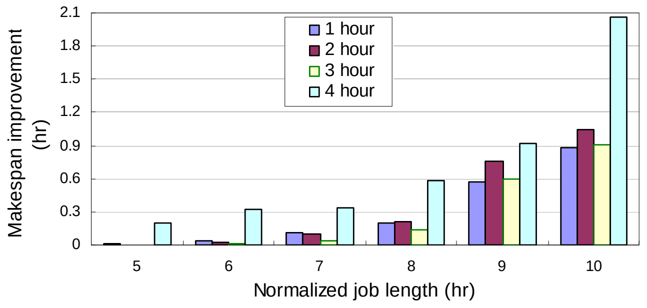


Figure 5. Improvement of average job makespans when using the one-step look ahead heuristic instead of checkpointing periodically. Each bar shows the results for a specific periodicity of checkpointing. All the jobs have the checkpoint size of 300 MB.

sizes and job lengths. The improvement is achieved by the adaptiveness of the heuristic to resource availability. The heuristic determines if checkpointing is needed at the periodicity of several minutes, which is the length of the checkpoint latency in our experiments. Based on Equation 3, checkpointing is performed only when the resource availability is low enough, making the cost of work loss higher than that of checkpoint overhead. The resulting improvement increases with checkpoint size, as large size indicates high checkpoint overhead. Furthermore, it also increases with normalized job length because longer jobs tend to incur more failures, meaning that the periodic checkpointing leads to higher work loss than checkpointing adaptively.

There exists a sweet spot for the checkpoint periodicities in Figure 5: checkpointing every 3 hours. This can be explained by the fact that most availability intervals of the computation hosts are within 2 and 4 hours in the traced testbed [19]. Therefore, for small jobs that are subject to fewer failures, checkpointing every 3 hours obtains better performance than checkpointing at a higher frequency. Meanwhile, for long jobs that suffer from relatively frequent failures, the results are similar to those of checkpointing hourly. Checkpointing at a lower frequency, e.g., every 4 hours, leads to poor performance because of the large penalty caused by rolling back to the recent checkpointed state.

From the results in Figure 4 and Figure 5, we can conclude that the one-step look ahead heuristic is able to adapt to different levels of resource availability and thus achieves better application performance than checkpointing at static periodicities.

7.4 Overall Evaluation

In this section, we show the performance of applying the two failure-aware techniques together. We collected the average job makespans over the total 720 jobs and compared them among three strategies: "*optimistic storage + adaptive interval*", "*pessimistic storage + adaptive interval*", and "*dedicated storage + periodic interval*". The first two strategies present the different combinations of our failure-aware techniques. The third one uses a dedicated checkpoint server and checkpoints guest jobs at a pre-defined periodicity, which are mechanisms applied in production systems such as Condor [25].

In all the experiments, the checkpoint sizes of the tested jobs were randomly chosen between 100 MB and 500 MB. We set the checkpoint periodicity to be 3 hours, which achieved the best performance among all periodic checkpointing schemes shown in Figure 4. In production systems, this value is provided by guest users that usually do not have in-depth knowledge about the behaviors of computation hosts. Thus we are comparing to the best case that periodic checkpointing can get. To simulate a dedicated checkpoint server, we added a machine with unlimited disk space and connected with all the computation hosts via 10 Mbps network links. The checkpoint server was available 98% of the time, representing a higher level of availability than the non-dedicated resources.

Figure 6 shows the results for the three checkpointing strategies. In addition to average makespans, we also plot the 95% confidence intervals for the average values. All the confidence intervals are smaller than 1 hour, meaning that the size of the samples used in our experiments is large enough to obtain statistically meaningful results.

In Figure 6, the scheme of using the optimistic repository selection and the adaptive checkpoint interval achieves the best performance for jobs shorter than 9 hours. However, the difference between the two failure-aware schemes (optimistic and pessimistic) is trivial, which is also shown

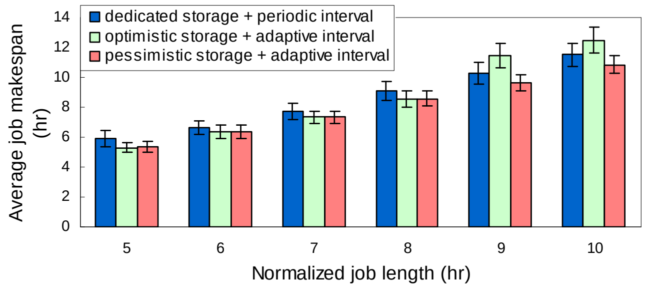


Figure 6. Overall evaluation of the failure-aware checkpointing techniques. Each bar shows the average makespan of the jobs submitted at different times. The interval on top of a bar is the 95% confidence interval for the average value.

in Figure 3. The periodic checkpointing leads to higher frequency of checkpointing and thus worse makespans. For jobs longer than 9 hours, the failure-aware scheme of using pessimistic repository selection performs best since failures become more frequent. The assumption of the optimistic scheme that the failure characteristic at the beginning of the execution is representative all through the execution becomes less true toward the end of the long guest jobs. Again, the performance of the optimistic scheme can be improved by encoding checkpoints to more redundant fragments.

In conclusion, our failure-aware checkpointing techniques outperform or come close to the methods that rely on dedicated hardware resources and checkpoint with the best fixed periodicity. It is worthwhile to mention that in practical FGCS systems, it is unlikely to have a dedicated storage server with availability as high as high as 98%. In that case, our proposed scheme will have a higher performance improvement.

8 Conclusion

We presented new failure-aware checkpointing techniques that improve the performance of guest applications in fine-grained cycle sharing (FGCS) systems. These techniques apply the predictive knowledge of resource availability to select reliable checkpoint repositories out of non-dedicated failure prone storage hosts and to determine suitable checkpoint intervals. We have evaluated our techniques on an FGCS system via trace-based simulation. Experimental results show that the schemes of selecting reliable checkpoint repositories improve job makespans up to 15.3% when compared to the method of random selection. We have also found that our heuristic of determining checkpoint intervals adapts to the dynamism of resource availability and thus outperforms checkpointing with a fixed periodicity. Moreover, by comparing to Condor's mechanisms that use dedicated checkpoint servers and pre-defined checkpoint periodicity, our techniques improve the application performance by up to 9.4%.

In future work, we plan to examine our checkpointing techniques in two more aspects. First, we will measure the effectiveness of the repository selection schemes by encoding checkpoints into different numbers of basic and redundant fragments. Second, we will use a production testbed to measure the impacts of transferring checkpoints over a shared network. In such a network, multiple transfers will be dependent and will affect the network latency incurred by each. We can use runtime network bandwidth measurements between pairs of hosts to adapt our scheme to this reality.

References

[1] M. K. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *Proc. of DSN'05*, pages 336–345, 2005.

[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] J. Basney and M. Livny. Managing network resources in condor. In *Proc. of HPDC'00*, pages 298–299, 2000.

[4] R. Buyya and M. Murshed. GridSim: A toolkit for the modeling and simulation of distributed resource management and scheduling for grid computing. *Concurrency and Computation: Practice and Experience*, 14:1175–1220, 2002.

[5] C. Cachin and S. Tessaro. Optimal resilience for erasure-coded byzantine distributed storage. In *International Conference on Dependable Systems and Networks (DSN)*, pages 115–124, 2006.

[6] R. de Camargo, R. Cerqueira, and F. Kon. Strategies for storage of checkpointing data using non-dedicated repositories on grid systems. In *Proc. 3rd Int'l Workshop on Middleware for Grid Computing*, pages 1–6, 2005.

[7] P. A. Dinda and D. R. O'Halaron. An evaluation of linear models for host load prediction. In *Proc. of HPDC'99*, page 10, 1999.

[8] W. Gentzsh. Sun Grid Engine: towards creating a compute power grid. In *Int. Symposium on Cluster Computing and the Grid*, pages 35–39, 2001.

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

[10] Y. Ling, J. Mi, and X. Lin. A variational calculus approach to optimal checkpoint placement. *IEEE. Trans. on Computers*, 50(7):699–708, 2001.

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

[12] D. Nurmi, J. Brevik, and R. Wolski. Minimizing the network overhead of checkpointing in cycle-harvesting cluster environments. In *Proc. of Cluster'05*, 2006.

[13] C. H. Papadimitriou and K. Steiglitz. *Combinational Optimization: Algorithms and Complexity*. Dover Publications, 1998.

[14] 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, 1998.

[15] J. S. Plank. An overview of checkpointing in uniprocessor and distributed systems. Technical Report UT-CS-97-372, Department of Computer Science, University of Tennessee, 1997.

[16] J. S. Plank, K. Li, and M. A. Puening. Diskless checkpointing. *IEEE Trans. on Parallel and Distributed Systems*, 9(10):972–986, 1998.

[17] M. O. Rabin. Efficient dispersal of information for security, load balancing, and fault tolerance. *Journal of the ACM*, 36(2):335–348, 1989.

[18] X. Ren and R. Eigenmann. iShare - Open internet sharing built on P2P and web. In *Proc. of EGC'05*, pages 1117–1127, 2005.

[19] X. Ren and R. Eigenmann. Empirical studies on the behavior of resource availability in fine-grained cycle sharing systems. In *International Conference on Parallel Processing*, pages 3–11, 2006.

[20] X. Ren, R. Eigenmann, and S. Bagchi. Availability prediction for non-dedicated storages in fine-grained cycle sharing systems. Technical Report ECE-HPCLab-06201, HPC Lab, School of ECE, Purdue University, 2006.

[21] X. Ren, S. Lee, R. Eigenmann, and S. Bagchi. Prediction of resource availability in fine-grained cycle sharing systems and empirical evaluation. *In submission to the Journal of Grid Computing (Submission date: September 2006)*.

[22] X. Ren, S. Lee, R. Eigenmann, and S. Bagchi. Resource availability prediction in fine-grained cycle sharing systems. In *Proc. of HPDC'06*, pages 93–104, 2006.

[23] 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.

[24] D. Thain, J. Basney, S. Son, and M. Livny. The kangaroo approach to data movement on the grid. In *Proc. of HPDC'01*, pages 325–333, 2001.

[25] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: The condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2004.

[26] N. H. Vaidya. Impact of checkpoint latency on overhead ratio of a checkpointing scheme. *IEEE. Trans. on Computers*, 46(8):942–927, 1997.

[27] 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.

[28] 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*, 2004.

[29] D. Zhou and V. Lo. Wave scheduler: Scheduling for faster turnaround time in peer-based desktop grid systems. In *Proc. of the 11th Workshop on Job Scheduling Strategies for Parallel Processing*, 2005.