J Grid Computing (2014) 12:593–613
DOI 10.1007/s10723-014-9297-4

Reliable and Efficient Distributed Checkpointing System for Grid Environments

Tanzima Zerin Islam · Saurabh Bagchi · Rudolf Eigenmann

Received: 8 April 2013 / Accepted: 13 March 2014 / Published online: 20 May 2014

Abstract In Fine-Grained Cycle Sharing (FGCS) systems, machine owners voluntarily share their unused CPU cycles with guest jobs, as long as their performance degradation is tolerable. However, unpredictable evictions of guest jobs lead to fluctuating completion times. Checkpoint-recovery is an attractive mechanism for recovering from such "failures". Today's FGCS systems often use expensive, high-performance dedicated checkpoint servers. However, in geographically distributed clusters, this may incur high checkpoint transfer latencies. In this paper we present a distributed checkpointing system called FALCON that uses available disk resources of the FGCS machines as shared checkpoint repositories. However, an unavailable storage host may lead to loss of checkpoint data. Therefore, we model the failures of a storage host and develop a prediction algorithm for choosing reliable checkpoint repositories. We experiment with FALCON in the university-wide Condor testbed at Purdue and show improved and consistent performance for guest jobs in the presence of irregular resource availability.

Keywords Checkpoint · Checkpointing · Recovery · Reliability · Cycle sharing system · FGCS · Condor · Efficient · Data parallel checkpointing · Erasure encoding · Checkpoint/Restart

T. Z. Islam (✉)
Lawrence Livermore National Laboratory, Box 808, L-560, Livermore, CA, 94551-0808, USA
e-mail: islam3@llnl.gov

S. Bagchi · R. Eigenmann
Center for Applied Scientific Computing (CASC), Lawrence Livermore National Laboratory, Livermore, CA, USA

S. Bagchi
e-mail: sbagchi@purdue.edu

R. Eigenmann
e-mail: eigenman@purdue.edu

1 Introduction

A Fine-Grained Cycle Sharing (FGCS) system [1] aims at utilizing the large amount of idle computational resources available on the Internet. In such a cycle sharing system, PC owners voluntarily make their CPU cycles available as part of a shared computing environment, but 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, an FGCS system allows a guest process to run *concurrently* with the jobs belonging to the machine owner (*host processes*). However, for guest users, these free computation resources come at the cost of fluctuating availability due to "failures". Here we define failures to be due either to the eviction of a guest process from a machine due to resource contention, or due to conventional hardware and software failures of a machine, which we call resource revocation. The primary victims of such resource volatility are large compute-bound guest programs whose completion times fluctuate widely due to this effect. Most of these programs are either sequential or composed of several coarse-grained tasks with little communication in between.

To achieve high performance in the presence of resource volatility, *checkpointing* and *rollback* have been widely applied [2]. These techniques enable an application to periodically save a checkpoint - a snapshot of the application's state - onto a stable storage that is connected to the computation host(s) through a network. A job may get evicted from its execution machine any time and can recover from this failure by rolling back to the latest checkpoint. Evictions may occur due to software or hardware failure, host work load increasing beyond a threshold or simply owner of the machine has returned.

Most production FGCS systems, such as Condor [3], store checkpoints to dedicated storage servers. These are few in number, are well-provisioned, and maintained such that 24×7 availability is achieved. This solution works well when a cluster only belongs to a small administrative domain or there are a large number of storage servers. However, it does not scale well with the growing sizes of grids having as participants thousands of home users, and geographically separated university campuses and research labs. For example, the production Condor pool at Purdue University (PU), called "DiaGrid" [4], is one of the largest such pools in the country with 20,000 processors in it. It has machines "flocking" from Indiana University (IU), University of Notre Dame (ND), and Purdue University Calumet (PUC). While DiaGrid has been using the dedicated storage server solution, a number of performance and feasibility issues have been encountered.

First, FGCS systems do not include a dedicated network that can efficiently handle the load of transferring potentially gigabytes of checkpoints between a *compute host (CH)* (the host on which the guest process is executing) and the *storage hosts (SH)* (the hosts that have contributed storage). Moreover, for multi-university grids, the current mechanism for storing checkpoints in dedicated servers may cause large network latencies since the compute and storage hosts may be located at large network distances. We have collected traces that show 12 % of jobs submitted to DiaGrid between March 5th, 2009 and March 12th, 2009 from PU actually ran on ND's machines. The high-level contribution of our work is to reduce amount of data by compressing them and break a large one to one communication of transferring checkpoint to one to many small transfers. This mechanism yields higher overall network bandwidth while transferring checkpoints.

Second, even if multiple storage servers could be provisioned and made available throughout the grid, a mechanism based on round trip time (RTT) to choose the closest storage host for saving checkpoint data, an option available in Condor, may not perform well. This is because a physically close host may observe huge network traffic during checkpoint transfer, making it less preferable than a distant one. Our technique takes the available bandwidth between two hosts into account while selecting a storage host. This ensures that the overall latency of transferring checkpoints between these hosts will be low. This technique could potentially select a physically far away host for storage if available bandwidth to near by hosts were lower due to congestion.

Third, a dedicated storage server will become loaded as the number of guest processes concurrently sending data increases—which will ultimately cause degradation in the performance of these guest processes. Today's storage server approach can be improved by making a pool of servers available, and then performing load-balancing. However, the same situation will still occur as the size of a grid increases (new computation hosts join). This decreases the ratio between the number of compute hosts to that of the storage servers. A more scalable solution is to utilize the unused storage capacity of grid resources to store these checkpoints.

Our work is motivated by these issues reported by scientists using DiaGrid for running their compute-intensive jobs with checkpoint-recovery. Our work therefore is to develop a framework for reliable execution of applications in a *shared storage environment*—an environment where a host can serve as both an execution host for a guest job as well as a storage host for saving checkpoints of others—as opposed to dedicated checkpoint servers. For this, we first propose a novel multi-state failure model for the shared storage hosts. Then, we propose a failure prediction scheme and apply this to the multi-state failure model to choose reliable and less loaded storage hosts to serve as checkpoint repositories. Finally, we propose an algorithm for efficient checkpoint storage and recovery. This algorithm uses erasure encoding [5] to break checkpoint data into multiple fragments, such that the checkpoint data can be reconstructed from a subset of the fragments. We realize our algorithms in a practical system called FALCON.

The major contributions of our work are —

- *A novel multi-state failure model* for storage hosts in shared storage environment for ensuring load-balancing across different storage hosts. Previous work has only considered a failure model for compute hosts [6].
- *Failure-aware storage selection technique* that selects a set of reliable and lightly loaded storage hosts for a compute host, based on their availability and available bandwidth between the compute host and the storage hosts. Previous work has not considered the multiplicity of factors related to storage hosts that affect the performance of checkpointing and recovery [6, 7].
- *An efficient method* that provides fault-tolerance to the process of checkpointing data as well as uses parallelism offered by multiple fragments being stored in multiple storage hosts to reduce checkpoint and recovery overheads. This approach leverages prior work in erasure coding for fault-tolerance [5] while using it in a different context (shared grid environments) and using the parallelism afforded by it.
- We have implemented and evaluated FALCON on the production Condor testbed of Purdue University—DiaGrid—with multiple sequential benchmark applications. The experiments ran on DiaGrid show that performance of an application with FALCON improves between 11 % and 44 %, depending on the size of checkpoints and whether the storage server for Condor's solution was located close to the compute host. Also, we show that the performance of FALCON scales as the checkpoint sizes of different scientific applications increase.

The rest of the paper is organized as follows. Section 2 presents a comprehensive summary of our previous work on failure-aware checkpointing and resource availability prediction. Our major contributions are described in detail in Section 3. Section 4 discusses data parallelism and the parallel architecture of our system. Section 5 presents implementation details of FALCON. Then, experimental approaches and results are discussed in Section 6. Section 7 reviews some related works. Finally, we conclude the paper in Section 8.

2 Background on Failure-Aware Checkpointing

Failure-aware checkpointing builds on mechanisms that predict the availability of the involved compute and storage hosts. In our previously developed prediction techniques [8], we applied a *multi-state failure model* to predict the *Temporal Reliability*, TR, of compute hosts. TR is the probability that a host will be available throughout a given future time window. Quantitatively, $TR(x)$ is the probability that there will be no failure between now and time x in the future. To compute TR, we applied a Semi Markov Process (SMP) model, where the probability of transitioning to a state in the future depends on the current state and the time spent in this state. The parameters of this model are calculated from the host resource usages during the same time window on previous days, since in many environments, the daily pattern of host workloads are comparable to those in the most recent days [9].

In other work [6], we proposed two algorithms for selecting reliable storage hosts in an FGCS system, where non-dedicated host machines provide disk storage for saving checkpoint data. A checkpoint is taken periodically and contains the entire memory state of an application. A checkpoint is used by a job for recovery when it gets rescheduled to another machine after the current host fails. For reliable storage in an FGCS system, we considered two criteria: the network overhead due to saving and recovery of checkpoints, and the availability of the storage hosts.

This work [6] applied the knowledge of network connectivity and of resource availability to predict reliability of a checkpoint repository from a set of storage hosts. This work also proposed a *one-step look ahead* heuristic to determine the optimal checkpoint interval. It compares the cost of checkpointing immediately with the cost of delaying that to a later time and uses that to adjusted checkpoint intervals.

Our prior work left some questions unanswered. First, it applied the failure-model for compute hosts to predict the availability of storage hosts. These two kinds of hosts offer resources with different characteristics; hence they will have different failure models. Second, there was no notion of load-balancing for storage hosts. Thus, a storage host that is predicted to have high availability in the near future will see a flash crowd of large checkpoints from several concurrently executing jobs. The checkpoints are often large in size (e.g., with the mcf benchmark application that we experiment with, the size is about 1.7 GB), and this load disbalance can cause significant perturbation to the FGCS system. For example, the machine owner can see slow I/O for his own jobs during such flash crowds. Third, for predicting reliability of a storage host, we used absolute temporal reliability even though *correlated temporal reliability* is the important criterion. By correlated temporal reliability, we mean what is the likelihood of the storage host being available, *when needed*, i.e., when the compute host has a failure. It is at that time that the checkpoint is needed for recovering the guest process on a different machine. Fourth, our prior work used a static bandwidth measure, given by the network specification, to estimate the network overhead. We find that the actual bandwidth available for a large checkpoint transfer may vary significantly from the static measure. Finally, and most significantly from an implementation and deployment effort, our prior work performed a simulation of the checkpoint-based recovery scheme, using the GridSim toolkit. In this paper, we present a fully functional system executing on Purdue's BoilerGrid.

We compare the performance of FALCON with two other checkpoint repository selection schemes:

- Dedicated: This scheme uses a pre-configured checkpoint server to store checkpoints. These are generally powerful machines with very high availability. This is a supported current mode of usage for checkpointing in BoilerGrid, as in many other production Condor systems.
- Random: This scheme selects storage hosts randomly. Here, we assume that this scheme employs the same checkpoint store and retrieve methods as FALCON, except that it chooses storage hosts randomly at the beginning of each checkpoint interval.

3 Design for Robust Checkpointing

In this section, first we discuss our proposed novel multi-state failure model for storage hosts. The roles that a host assumes exhibit different characteristics - computation hosts execute guest jobs requiring CPU and memory resources whereas storage hosts handle I/O load. Therefore, failure models for these two types of resources are different. We propose a failure model specialized to storage hosts in a shared computing environment.

Second, we propose a new failure prediction technique to select reliable checkpoint repositories by considering correlation of failures between compute and storage hosts. Third, we present an algorithm that fragments checkpoints using erasure coding and concurrently saves them to multiple storage hosts. The coding introduces redundancy such that a subset of the fragments can be used for the recovery of the application. This section discusses the single-threaded design of the compression and the erasure encoding algorithms. Section 4.1 presents the multi-threaded architecture of FALCON.

3.1 Novel Multi-state Failure Model for Storage Hosts

In an FGCS system, storage hosts are often non-dedicated, shared hosts contributing their unused disk spaces. Checkpoint data saved by guest jobs in these storage hosts may get lost when the storage hosts become unavailable due to resource contention or resource revocation. This is of particular concern for long-running compute-intensive applications. The model for recovering a guest job when it is evicted from a machine is that the guest process migrates to another compute host and uses the last checkpoint fragments to recover and re-execute from the checkpoint. In this situation, it is clearly advantageous to choose a storage host that:

- is going to be available with high probability *when a compute host becomes unavailable*.
- is less likely to have high I/O load. This ensures load-balancing across storage hosts and is crucial for an FGCS system, since creating load on an already busy host will reduce the performance of host and guest jobs.

– has large available bandwidth to the computation host so that the transfer of checkpoint fragments incurs lower network latency.

To predict failures of storage hosts, we propose a novel multi-state failure model. In our previous work [6], we developed a failure model for compute hosts and applied it to storage hosts. Since the underlying availability models of the two types of resources - CPU cycles and disk storage - are different, applying the same failure model to both these resources is inadequate.

Figure 1 presents our new five-state failure model for storage hosts. The states are defined as follows:

(i) S_0: storage host is running with I/O load $< \tau_1$ and number of compute hosts sending checkpoint data concurrently is < MAX-CLIENTS, (ii) S_0': number of compute hosts sending checkpoint data concurrently is = MAX-CLIENTS, (iii) S_1: I/O load of storage host is between $[\tau_1, \tau_2)$, (iv) S_2: I/O load of storage host is between $[\tau_2, 100\,\%]$ (v) S_3: storage host is not available due to resource revocation.

Here, the states S_0' and S_2 ensure load-balancing since storage hosts in either of these states do not accept any more request for storing checkpoint data. Knowledge about states S_1 and S_2 is used during storage selection to rank storage hosts according to their likelihood of becoming loaded in the future. Note that, state S_0' has been separated from state S_1 and S_2 because this state represents a transient state of a storage host. A compute host only uses the knowledge of states running, loaded and temporarily unavailable to predict load on a storage host machine. The knowledge of S_0' is only used by the storage host to reject requests from new compute hosts for storing checkpoint and thus enforce load balancing. State S_3 is an absorbing state because we assume the failures are irrecoverable. Even if in practice the failure can be recovered, the time to recover is large enough and unpredictable enough to be useless for the current guest job.

When a compute host requests a storage host to save checkpoint data, depending on which state the storage host is in, it replies back. When the storage host is in either S_0 or S_1, it replies "ok", and the compute host continues with sending data. Otherwise, if it is in either S_0' or S_2, it does not accept any new request.

3.2 Failure-aware Storage Selection

3.2.1 Temporal Availability

Similar to other resources in FGCS systems, checkpoint repositories are volatile. To predict availability of a storage host SH_k in a given time window with respect to a compute host CH_l, we define *Correlated Reliability Load Score (CRLS)* as:

$$CRLS(SH_k, CH_l) = \begin{cases} LI(SH_k, CH_l) + \gamma, & \text{if } \mathrm{CR}(\mathrm{SH_k}, \mathrm{CH_l}) \geq \gamma; \\ CR(SH_k, CH_l), & \text{otherwise.} \end{cases} \tag{1}$$

where,

$$\begin{aligned} Pr_{CH_l}(i) &= Pr\{CH_l \text{ in state } i\} \\ Pr_{SH_k,CH_l}(i|j) &= Pr\{SH_k \text{ in state } i | CH_l \text{ in state } j\} \end{aligned}$$

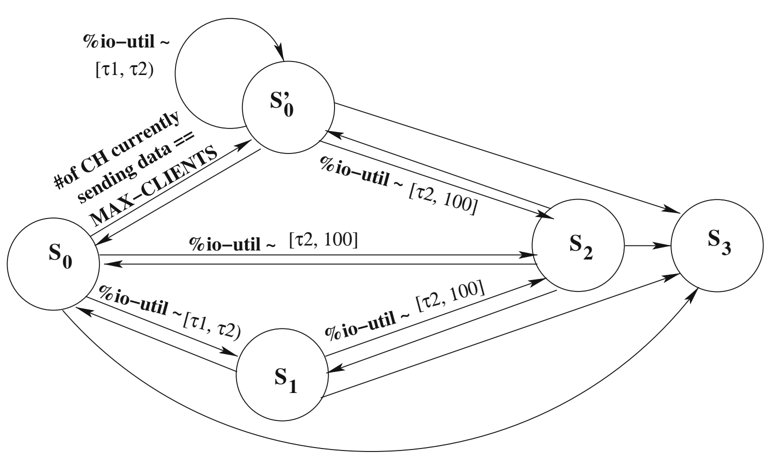


Fig. 1 New multi-state storage host failure model. Here, S_0: running state, S_0': state where the maximum number of compute hosts are sending data, S_1: loaded state, S_2: Temporarily unavailable state and S_3: unavailable state

$$CR(SH_k, CH_l) = \begin{cases} Pr_{SH_k,CH_l}(S_0|down) \\ +Pr_{SH_k,CH_l}(S_1|down)+ \\ Pr_{SH_k,CH_l}(S_2|down), & \text{if } \mathrm{Pr_{CH_l}(down)} > 0; \\ \gamma, & \text{otherwise.} \end{cases}$$

$$LI(SH_k, CH_l) = \begin{cases} \alpha \times Pr_{SH_k,CH_l}(S_0|up) \\ +(1-\alpha) \times Pr_{SH_k,CH_l}(S_1|up), & \text{if } \mathrm{Pr_{CH_l}(up)} > 0; \\ 0, & \text{otherwise.} \end{cases}$$

In (1), $Pr_{CH_l}(down)$ and $Pr_{CH_l}(up)$ denote the probability that the compute host CH_l is down (state S_0) and up (state S_1) respectively. Here, $CR(SH_k, CH_l)$ and $LI(SH_k, CH_l)$ denote the *correlated reliability* and the *load indicator* between SH_k and CH_l respectively. $CR(SH_k, CH_l)$ and $I(SH_k, CH_l)$ are probabilities while $CRLS$ is not—it is a score $\in$ $[0, 2]$. In (1), we first calculate $CR(SH_k, CH_l)$ as the total probability that the storage host SH_k remains up when the compute host CH_l is down. Note that, here we are adding up the probabilities corresponding to storage host SH_k being running, loaded, or temporarily unavailable. The intuition is that if the checkpoint data is needed (since the compute host has gone down), then the storage host will allow the read of the data, even if it is loaded. We consider storage hosts having $CR(SH_k, CH_l) \geq \gamma$ (γ is a configurable parameter, we chose $\gamma = 0.95$ for our experiments) as very reliable. The equation rounds the reliability component of $CRLS$ to γ since we consider that reliability scores greater than γ are high enough to be considered equivalent and also may be statistically indistinguishable due to the inherent noisiness of the measurements. Beyond this point, we would want to give less weight to the less loaded storage hosts and therefore add their load indicators to γ to make less loaded machines have high $CRLS$ value. For storage hosts having reliability $< \gamma$, we do not consider their load because we want to ensure that the most reliable storage hosts get chosen first. We calculate the load indicator $LI(SH_k, CH_l)$ as a weighted probability of SH_k being in the two tolerably loaded states, namely, S_0 and S_1.

The weight α should be chosen based on the load characteristics of the grid resources. This value should be higher (close to 1) if the variability in load among grid resources is low to make sure that less loaded hosts are favored more during storage host selection technique. For our experiments, we could have chosen any value between 0.6 to 1 and found that for BoilerGrid, these values do not impact the storage host selection process significantly. Hence, we arbitrarily selected the value of α to be 0.75.

3.2.2 Available Bandwidth

In addition to failure-prediction, checkpoint transfer overhead is one of the key factors in storage repository selection. Our previous work [6] used effective bandwidth between a compute and a storage host to calculate network overhead. Effective bandwidth is the maximum possible bandwidth that a link can deliver. But the actual bandwidth available between a compute host and a storage host may be far less than this quantity. So, it is more accurate to use available bandwidth between two hosts to access the overhead of transferring data between them. *Available bandwidth (ABw)* is the unused capacity of a link or end-to-end path in a network and is a time-varying metric. We define network overhead of transferring a checkpoint of size n with erasure coding parameters (m, k) from compute host CH_j to storage host SH_i in (2). The parameters (m, k) mean that a total of $m+k$ checkpoint fragments are stored and any m of them may be used to recover the entire checkpoint.

$$\text{network overhead, } N_{i,j} = \frac{n/m}{ABw_{(SH_i, CH_j)}} \tag{2}$$

In (2), $N_{i,j}$ represents the network overhead of sending a checkpoint from compute host CH_j to storage host SH_i. ABw_{SH_i,CH_j} is the available bandwidth between storage host SH_i and compute host CH_j.

3.2.3 Objective Function

We define an objective function in (3) that tries to balance the checkpoint storing overhead with the re-execution cost if that checkpoint had not been taken. An application incurs overhead during a checkpoint storing phase while benefits from the fact that it does not have to re-execute from the very beginning and can easily restart from the state saved in the latest checkpoint. The difference of these two quantities is the ultimate price that the application pays. Clearly, a lower value is desirable. FALCON selects storage hosts such that they minimize this objective function. For a particular compute host CH_j, $m + k$ storage hosts are selected for storing that many erasure coded checkpoint fragments.

$$\mathcal{F} = \frac{MTTF_{cmp}}{CI} \times \sum_{i=1}^{V} (C_i \times N_{i,j}) - (T_{curr} + MTTF_{cmp}) \times \prod_{i=1}^{V} CRLS'(SH_i, CH_j) \tag{3}$$

$$\sum_{i=1}^{V} C_i = (m + k) \tag{4}$$

$$CRLS'(SH_i, CH_j) = max[1 - C_i, CRLS(SH_i, CH_j)] \tag{5}$$

Here, $MTTF_{cmp}$ is the mean time to failure of a compute host, CI is the length of a checkpoint interval and T_{curr} is the time units spent on performing useful computation for the job so far. V is the total number of storage hosts. The variable C_i is an indicator variable, set to 1 for the storage host that is selected and 0 for the one that is not. Our goal is to pick the $m + k$ storage hosts so as to minimize the objective function $\mathcal{F}$. The first term corresponds to the overhead of storing the checkpoints. The term $\frac{MTTF_{cmp}}{CI}$ approximates the number of checkpoints generated within $MTTF_{cmp}$. The second term corresponds to the re-execution cost—a larger value means lower re-execution cost. Equation 5 forces the formulation to only consider the storage hosts that will be selected. We developed a similar objective function in our previous work [6]. However, (3) uses different measures of network overhead and reliability score.

3.2.4 Storage Selection Algorithm

To choose storage hosts that minimize the objective function in (3), we devise a greedy algorithm. Consider again that the storage selection is being done by compute host CH_j.

We first sort the storage hosts in decreasing order of $CRLS(SH_i, CH_j)$ and increasing order of $N_{i,j}$. If a storage host appears in the first $m + k$ elements of both the sorted lists, it is selected. Then, the value of $\mathcal{F}$ is calculated with $C_i = 1$ for the chosen hosts and 0 for others. This will be used as the baseline value of $\mathcal{F}$ when considering further hosts to add.

If the number of selected storage hosts is less than $m + k$, the objective function is calculated by including one unselected host at a time. The host causing the minimum increase to the objective function is selected. When the number of selected hosts is $m + k$, the algorithm terminates; otherwise the algorithm continues adding one storage host in each iteration. 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.

When a storage host becomes unavailable later during a checkpoint interval, the compute host needs to re-choose a new one to replace it. Re-choosing another host from previously unselected ones is also done based on minimizing the same objective function. Our system design is such that this storage selection process occurs in parallel to the actual application and to the algorithm that uses this decision to send checkpoint fragments to the selected repositories. Since checkpoint repository selection occurs out of the algorithm's critical path, this speeds up the checkpoint storing algorithm. But the tradeoff of this design choice is that there may be stale information being used to choose the storage hosts. This can be addressed by configuring the periodicity with which these measurements and list updates take place. The period should be smaller if the underlying grid environment is highly volatile. The statistical analysis of the eviction characteristics in BoilerGrid (as represented in Table 2) shows that on average 1.3 evictions occur per hour. For all our experiments, we have configured this periodicity of calculating the objective function to once every 10 sec.

3.2.5 Discussion

The resource availability of FGCS systems is highly volatile and the network bandwidth and latency can change any time. For any of these reasons, a less than optimal storage host may be selected by FALCON at one instance. Section 5.3 describes the component of FALCON that is responsible for re-ordering storage hosts based on the objective function described in Section 3.2.4. If any of these storage hosts become busy or unavailable, FALCON replaces the "bad" storage host on the fly. Section 6.2.2 presents that the cost of doing so is negligible.

One additional consideration that needs to be taken into account is the amount of disk space available on FGCS resources. Grid resources can be personally owned laptops, lab machines in universities, or servers with varying degree of available disk quota. For demonstrating the techniques developed in this paper, we used a list of storage hosts that have more than $1GB$ available and are willing to store checkpoints. Such a list can be populated automatically by making computational resources let their available disk space and their willingness to share their storage space known to the resource manager. The latter solution requires support from the Condor resource manager and is an engineering extension that is beyond the scope of this work.

The value of m and k depends on how much redundancy is sought for the system. The more volatile a system is, the higher the value of k should be. However, higher value of k means more redundancy, hence increase in data by $\frac{k}{m} \times 100\ \%$. Since FALCON uses compression to reduce the overall size of checkpoints considerably, even with redundancy, the total amount of data transferred is reduced. Section 6.2.1 shows an example of the amount of data FALCON transfers after compression and erasure encoding.

FALCON focuses on sequential and large running jobs taking system-level checkpoints. System-level checkpoints are well compressible even by a general purpose compression algorithm such as Gzip since they include the current state of memory which may include large regions of "0"s in them. Islam *et. al* in [10] presents an interesting technique called "data-aware compression" that is very effective for hard to compress checkpoints such as application-level checkpoints.

3.3 Single-Threaded Method for Checkpoint Recovery

In our previous work [6] we proposed two algorithms - *Optimistic* and *Pessimistic* for selection of storage repositories. The Optimistic scheme selects a set of storage hosts at the very beginning of a job's execution and uses this set to save checkpoint data during each checkpoint interval. This set is updated only when a job migrates to a different execution machine. On the other hand, the Pessimistic scheme selects a new set of storage hosts at the beginning of each checkpoint interval. While Optimistic ignores inherent dynamism that is present in resource availability, Pessimistic results in unnecessary overhead [6]. Here, we develop a new algorithm that chooses storage hosts on an as-needed basis, always keeping $m + k$ fragments. It releases the resource availability assumptions from our prior work and updates the selection of storage hosts on an as-needed basis. It takes into account the changing load and fluctuating resource availability. Our algorithm for storing checkpoint data has the following steps:

1. Read chosen storage host list generated by algorithm described in Section 3.2
2. Read checkpoint from disk
3. Compress the checkpoint
4. Erasure encode the checkpoint into $m + k$ fragments (erasure coding with parameters (m, k))
5. Send fragments concurrently to storage hosts
6. If any of the chosen storage hosts is in state S_2 or S_3, re-choose another host from the list of unselected ones. The same greedy algorithm, described in Section 3.2, is used to reselect. Send the remaining fragments concurrently.
7. Repeat step 6 until all the fragments are sent or a new checkpoint is generated. If a new checkpoint is generated then start from step 1 and abandon the remaining checkpoint fragments.

4 Data Parallelism and Parallel Architecture of FALCON

Since checkpoint sizes of the biology applications can potentially range from couple of megabytes to the order of gigabytes, processing them in their entirety may take a long time. But, by dividing a checkpoint data into a number of blocks and then processing each block in parallel may speed up the processing of the checkpoints in a computation host. Most of the commodity machines that are part of a grid, are multi-core machines. So, by taking advantage of thread level parallelism (TLP) to process the large sized checkpoint data, the processing overhead can be reduced significantly. In this paper, we have redesigned the checkpoint storing and retrieving process of FALCON [11] to do exactly that. In the evaluation section, we will refer to this architecture as FALCON-P, differentiated from the FALCON architecture presented so far in the paper.

4.1 Design of Parallel Checkpoint-Recovery Scheme

4.1.1 Parallel Checkpointing

To take advantage of TLP, we divide a checkpoint into b blocks where b is a configurable parameter. The value of b should not be more than the number of cores available on that particular computation host. Then we spawn b number of threads where the i^{th} of these reads from index $i \times \lceil n/b \rceil$ to $(i+1) \times \lceil n/b \rceil - 1$. Since multiple threads can read a file concurrently, the overhead of reading a file decreases. In addition to that, now each thread has to deal with a smaller sized data set. After reading a mutually exclusive block, each thread then goes on to compress the data, erasure encode each block into $m + k$ fragments and transfer them to the chosen storage hosts. We send the i^{th} fragment of each block to the i^{th} storage host chosen by our storage selection technique Section 3.2. The rationale behind storing one fragment from each block on a storage host is that an available storage host implies the availability of 1 fragment per block for each of the blocks. In this way, our analysis for selecting a set of reliable and efficient storage hosts still holds true.

4.1.2 Parallel Recovery

During the recovery phase, a number of threads equal to the number of blocks a checkpoint was divided into during the checkpoint storing phase, work in parallel to reconstruct the blocks. The i^{th} thread does the following:

- fetch the $m + k$ fragments of the i^{th} block in parallel
- erasure decode these fragments to build the compressed i^{th} block
- decompress the i^{th} block and write to a file

While, the network transfer, decoding and decompression phases take the advantage of TLP, writing the blocks to the disk is a serial operation.

5 FALCON Structure

Figure 2 presents the system level block diagram of FALCON. In this figure, each large box represents one component and each small box represents a module. We have designed our system such that some modules run off the critical path of our checkpoint-recovery schemes.

FALCON consists of three major components:

- *Compute host component (CHC)* takes care of failure-aware checkpointing and is submitted along with the guest process to the compute host.
- *Storage host component (SHC)* is a user application that runs in storage hosts. This component implements the multi-state failure model explained in Section 3.1.
- *History server component (HSC)* runs on any machine and periodically collects states of

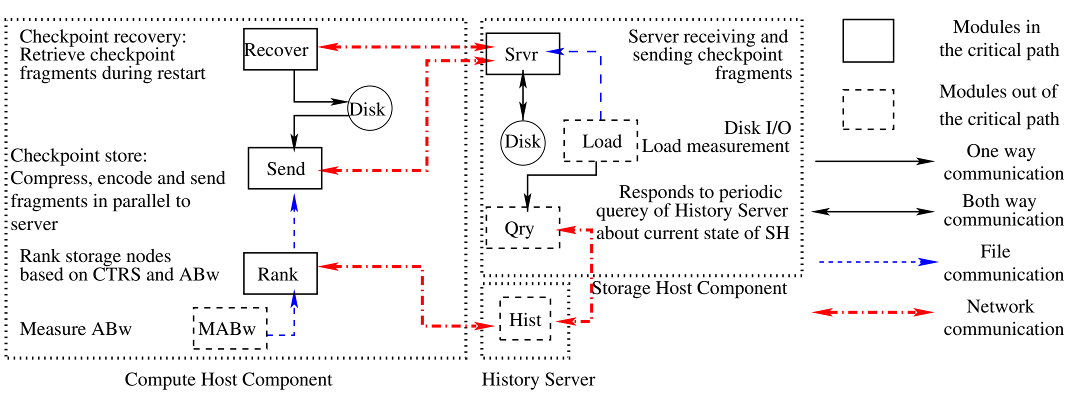


Fig. 2 System level block diagram of our system. Here, the Compute Host Component represents FALCON modules that run in a compute host, the Storage Host Component represents FALCON modules that run in a storage host and History Server represents a module that keeps history logs of the availability states of the compute and storage hosts

compute and storage hosts. Modules in the CHC and the SHC communicate with this component to calculate $CRLS$.

FALCON is integrated with Condor and some of the design decisions were driven by the design of Condor. For example, all the modules are implemented as user-level processes. Detailed description of each module in each component is given in the following subsections.

5.1 Compute Host Component (CHC)

Compute host component is the part of FALCON that runs on a computation host. The CHC consists of four modules:

- Module **`MABw`** is a process that periodically measures available bandwidth between this computation host to all the storage hosts and appends them to a file. For available bandwidth measurement, we have used Spruce [12], a light-weight available bandwidth measurement tool. Spruce provides a server module that runs as a part of the CHC and a client module that runs as a part of the SHC. For our experiments, we have used a period of 10 seconds for measuring the available bandwidth.
- Module **`Rank`** is a process that implements the greedy algorithm described in Section 3.2.4. It iteratively ranks the storage hosts and produces an ordered list of storage hosts.
- Module **`Send`** implements an algorithm that is responsible for storing and retrieving the checkpoints. In the single threaded architecture of FALCON [11], this module reads the checkpoint data from disk, compresses it and then uses erasure coding to break the compressed checkpoint data into $m + k$ fragments where m and k are parameters of erasure coding. For erasure coding, we modified the `zfec` implementation [13] to convert it to use only C so that we can run on all the machines of BoilerGrid. These checkpoint fragments are sent in parallel to storage host module `Srvr`. In the multi-threaded architecture, the checkpoint reading, compression and encoding schemes are conducted in blocks by parallel threads.
- Module **`Recover`** is responsible for retrieving checkpoint fragments during a rollback phase from storage repositories, then decoding the fragments to one compressed data and then uncompressing it to produce original checkpoint data - that a guest process uses to restart. Checkpoint fragments are fetched from storage hosts in parallel.

The modules are light-weight, requiring little CPU and memory resources.

5.2 Storage Host Component (SHC)

The SHC consists of three modules:

- Module **`Load`** measures disk I/O load periodically. This process runs in parallel to the actual server module `Srvr` and generates required load information that module `Srvr` uses to determine which state the storage server is in according to Fig. 1.
- Module **`Srvr`** implements the server logic for receiving and sending checkpoint data to variable number of compute hosts. It updates the variable current-state at the beginning of each request received from Module `Send` of the CHC.
- Module **`Qry`** is a parallel process that responds to the history server's query about which state the storage host is currently in. It receives values of the state variables from module `Load` (I/O load) and module `Srvr` (the number of compute hosts currently being served).

5.3 History Server Component (HSC)

This can be run as a user process on any machine. This component pings each compute host to note if that machine is up or down and communicates with storage hosts to receive their current status. Note that, the HSC only takes 4 states of the storage hosts into account based on load - namely, S_0, S_1, S_2 and S_3. This information then is stored in log files as a {current time stamp, current state} tuple. The HSC computes $CRLS$ using (1). Our current implementation uses a central server approach. This design can be extended to a distributed implementation.

6 Evaluation

We have developed a complete system, as described in Section 5. We ran experiments on a production

Condor testbed—BoilerGrid by integrating our work with standard benchmark applications. These applications were chosen from SPEC CPU 2006 and BioBench [14] benchmark suites. SPEC CPU 2006 is widely used for benchmarking CPU-intensive programs while BioBench consists of well known biomedical applications. Table 1 shows the sizes of checkpoints that were generated by these applications and their compressibility. This section presents the experiments for evaluating the system in terms of its checkpoint recovery overheads and its effectiveness in improving job makespan. In Sections 6.1 and 6.2, we present the evaluation of FALCON (the single-core version) and in Section 6.3, we present the evaluation of FALCON-P (the multi-core version).

We have organized our experiments to measure both fine-grain (*micro benchmark experiments*) and coarse-grain (*macro benchmark experiments*) metrics. While the micro benchmark experiments compare overheads of different checkpoint-recovery schemes under controlled experimental conditions, the macro benchmark experiments evaluate the effectiveness in improving job makespan running on Purdue's condor environment, the BoilerGrid. Schemes that we compare with are:

– Dedicated: Condor's scheme where a dedicated storage server is used for saving checkpoint data.
– Random: A scheme where the storage hosts for saving the erasure encoded checkpoints are randomly chosen from among all available storage hosts.
– Pessimistic: A scheme presented in [6] that assumes that resource fluctuation is very common and re-chooses all the storage hosts at the beginning of each checkpoint interval.

Note that, the default scheme of Condor is to send checkpoint back to the submitter machine (the machine from which the job was submitted). This scheme is similar to that of using a dedicated server and hence performs no better than our reference Dedicated algorithm (in fact, it can be significantly worse, if the submitter sits behind a low-bandwidth connection). We do not compare FALCON with the Optimistic scheme because the assumption of not having any fluctuation in the grid environment by the Optimistic scheme does not hold in practice due to the volatility of the environment.

Checkpoint storing overhead includes time:

– for FALCON : (i) Read chosen storage host list from disk (ii) Read checkpoint from disk (iii) Compress (iv) Erasure encode and write fragments to disk (v) Read fragments from disk (vi) Send checkpoint fragments in parallel to storage hosts
– for Dedicated : (i) Read checkpoint from disk (ii) Send checkpoint to storage server
– for Random : (i) Choose storage hosts randomly (ii) Follow steps (ii) - (vi) of FALCON

Recovery overhead includes time:

– for FALCON : (i) Fetch the minimum required fragments from storage hosts. This time includes reading checkpoint at the storage host end, network transfer and writing to disk at the compute host end (ii) Erasure decode and write compressed checkpoint data to disk (iii) Decompress and write to disk
– for Dedicated : (i) Fetch checkpoint data from storage server. This time includes reading checkpoint at the storage host end, network transfer and writing to disk at the compute host end
– For Random : (i) Follow steps (i) - (iii) of FALCON

For all our macro and micro benchmark experiments, we set erasure coding parameters to $(3, 2)$ - meaning 3 fragments are required and 2 are redundant.

Table 1 Checkpoint sizes of different applications

Applications	mcf	TIGR-I	TIGR-II	TIGR-III
Original Checkpoint Size (MB)	1677	946	500	170
Compressed Checkpoint Size (MB)	241	201	153	129
Compression Ratio	85.63 %	78.75 %	69.4 %	24.12 %

MCF and TIGR are benchmark applications part of SPEC CPU 2006 and BioBench respectively. TIGR-I, TIGR-II and TIGR-III are runs of TIGR with different input sizes

6.1 Macro Benchmark Experiments

This section presents results of our macro benchmark experiments - experiments that we ran by submitting scientific applications to BoilerGrid and measuring their average makespan — the time difference between submission and completion of the job minus the time it spent in the idle or the suspended states. A job submitted to Condor remains idle until it gets scheduled to a suitable machine. Condor jobs can specify their requirements for disk space, memory, machine architecture, operating system etc. in a submission script and a scheduler matches these requirements with machines that are available for running Condor jobs. Since this idle time is in no way related to checkpoint-recovery scheme, we exclude it from calculating makespan.

Checkpointing in Condor is non-blocking for the applications - the only blocking part is till the checkpoint is locally stored. Condor then transfers this checkpoint to appropriate storage repository as configured. This non-blocking technique efficiently hides checkpoint transfer overhead from makespan of applications. It is during restart when applications need to fetch checkpoints to the execution machine and restart. This recovery overhead directly adds up to an application's makespan.

The recovery overhead is incurred as many times as there are evictions of the applications from the compute hosts. We empirically measured this in BoilerGrid and present the failure characteristics in Table 2. We used 1.3 evictions per hour per job as the rate of eviction for our experiments in Section 6.1.1.

In Section 6.1.1 we compare average job makespan of applications using different checkpoint repository techniques. The decompositions of checkpoint storing and recovery overheads are shown in Section 6.1.2 and Section 6.1.3 respectively. For all the macro benchmark experiments we have used the sequential checkpoint storing and recovery schemes of FALCON.

Table 2 Statistical analysis of the eviction characteristics in BoilerGrid

N	μ	σ	*Range*
116	1.3130	0.2172	[1.0298,2.3931]

The table shows number of jobs for which we collected data (N), average number of evictions per hour (μ), standard deviation (σ) and range

6.1.1 Overall Evaluation

For overall evaluation of different schemes, we collected average job makespan of two benchmark applications. We integrated three checkpoint schemes: FALCON, Dedicated with a local checkpoint server (lab machine connected to the campus-wide LAN at Purdue) and Dedicated with a remote checkpoint server (machine at University of Notre Dame connected to Internet) with these applications and submitted jobs in BoilerGrid. Note that, University of Notre Dame is a part of this multi-university grid. For all cases, these applications took checkpoint once every 5 minutes. Here, using the Dedicated-Remote scheme represents the situation when jobs submitted from one university go to run at another university but the checkpoint server is at the first university. This is exactly the situation in Boiler-Grid for applications that run on other university machines.

From Fig. 3, we see that FALCON outperforms Dedicated-Local and Dedicated-Remote in actual application runs. In actual runs on BoilerGrid, applications on an average will see performance that lies between that of Dedicated-local and Dedicated-Remote since applications do go to run on machines at other campuses. The reasons for performance improvement of FALCON are many-fold — (i) FALCON chooses storage repositories that are more efficient to access (ii) checkpoint fragments saved by FALCON are much smaller in size due to compression and encoding, compared to the checkpoint size that Dedicated schemes store. Section 6.1.3 explains that smaller checkpoint size results in lower recovery overhead and hence improved job makespan and (iii) the fragments are retrieved in parallel from the chosen storage hosts. More about the contribution of each of the techniques (compression, load balancing and parallel network transfer) in improving the recovery overhead is discussed in Section 6.2.5.

6.1.2 Checkpoint Storing Overhead

Figure 4 shows decomposition of overhead of FALCON and Dedicated for a single store operation. Dedicated scheme uses a remote checkpoint server.

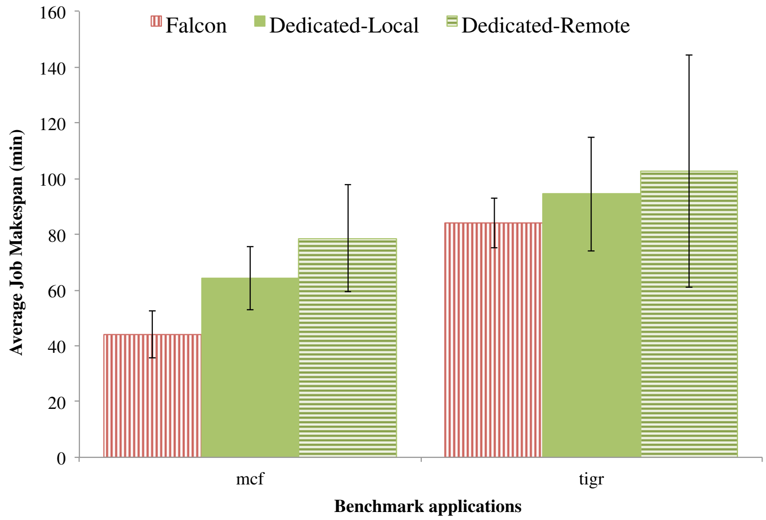


Fig. 3 Average job makespan of different applications. Here, Dedicated-Local represents the dedicated scheme using a local checkpoint server and Dedicated-Remote represents the dedicated scheme using a remote checkpoint server

One observation that can be drawn from Fig. 4 is that as checkpoint size increases, increase in checkpoint storing overhead of Dedicated becomes much higher than FALCON. The overhead is dominated by the disk read and network transfer time, which increases with increasing checkpoint sizes. However, FALCON's design of compressing the checkpoints and transferring the smaller checkpoint fragments in parallel speeds this up.

6.1.3 Recovery Overhead

In Fig. 5 we plot recovery overhead incurred by FALCON and Dedicated schemes for a single recovery operation.

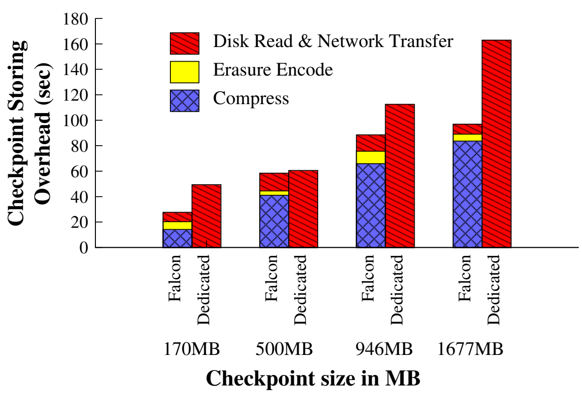


Fig. 4 Average checkpoint storing overhead vs different checkpoint sizes

One observation that can be made from Fig. 5 is that as checkpoint size increases, Dedicated scheme suffers due to large network transfer overhead. Compression by FALCON results in smaller checkpoint data and hence reduced network transfer overhead. As Table 1 shows, compression ratio increases as size of checkpoint data increases. This justifies our approach of incurring a little overhead at the compute host side for compression with the benefit of significant improvement in recovery overhead. Note that, lower recovery overheads directly translate to better performance for an application.

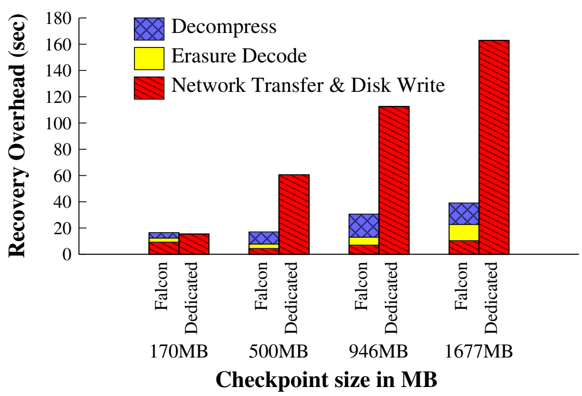


Fig. 5 Recovery overhead of four different checkpoint sizes generated by applications in Table 1. This figure shows contributions of different components in total recovery time. Erasure coding parameters are (m = 3, k = 2). Fetch and disk write time also includes the time to read checkpoint data from storage repositories

6.2 Micro Benchmark Experiments

The objective of the micro benchmark experiments is to show off specific features of FALCON under controlled experimental conditions. We conducted three sets of experiments to compare: (i) efficiency of different schemes in handling concurrent clients, (ii) efficiency in handling storage failures and (iii) performance improvement due to load balancing. For these experiments, we used checkpoint data of 500MB generated by application TIGR. As storage hosts for FALCON we used 11 - 1.86 GHz Intel Core 2 Duo machines with 80GB of hard disk connected to the campus-wide 100 Mbps LAN and 1 - 2.00 GHz laptop with 160GB of hard disk connected to a DSL modem. As dedicated storage server we used another lab machine with configuration 2.66 GHz Intel Core 2 Duo with 80GB of hard disk space and connected to the campus-wide LAN. This machine was always available.

6.2.1 Efficiency in Handling Simultaneous Clients

The objective of this experiment is to show how the performance of different schemes scale with load imposed by multiple concurrent clients. In this experiment, the checkpoint storing overheads of different schemes, in addition to the factors described in Section 6, include time to write the checkpoint data to disk at the storage host end. We vary the number of compute hosts simultaneously sending data and measure the overhead for storing checkpoints.

Two observations that can be drawn from Fig. 6 are:

1. As the number of clients simultaneously sending data increases, the checkpoint scheme with a dedicated server suffers more than FALCON. Even though with erasure coding FALCON introduces 40 % more data, due to compression the total amount actually sent by FALCON is less than that of Dedicated. Total amount of data sent by FALCON with compression and erasure coding is:

$$datasent = 5 \times \frac{153}{3} MB = 255MB < 500MB$$

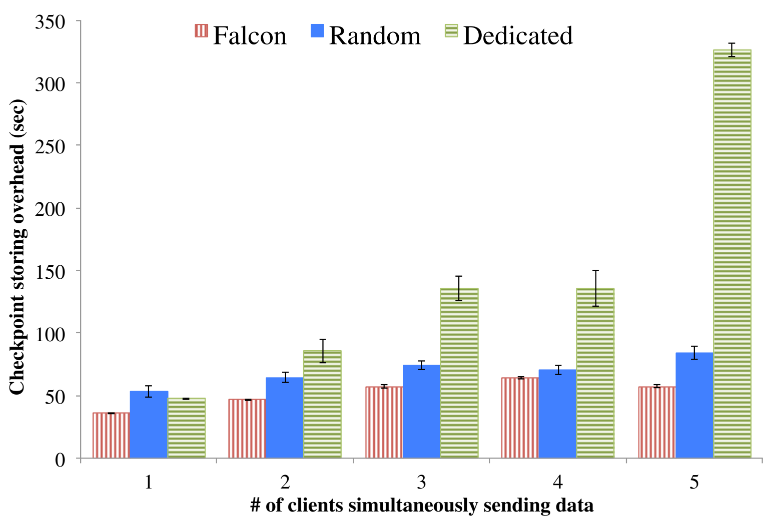
2. Checkpoint storing overhead of Random is larger than that of FALCON because Random chose the laptop behind the slow network connection 8 % of the time. Because of low available bandwidth between compute host and this laptop, FALCON never chose it.

6.2.2 Efficiency in Handling Storage Failures

Since storage hosts in FGCS are non-dedicated resources, a protocol must be able to handle unavailability of storage hosts efficiently. The objective of this experiment is to compare the added overhead of re-choosing storage hosts by FALCON with that of Random and the more conservative approach of Pessimistic [6]. For this experiment, we killed the storage daemons running in those storage hosts to make them appear unavailable.

One observation that can be drawn from Fig. 7 is that the overhead of re-choosing storage hosts using history and available bandwidth is no worse than that

Fig. 6 Average execution time of the algorithms vs number of clients concurrently sending data to the servers



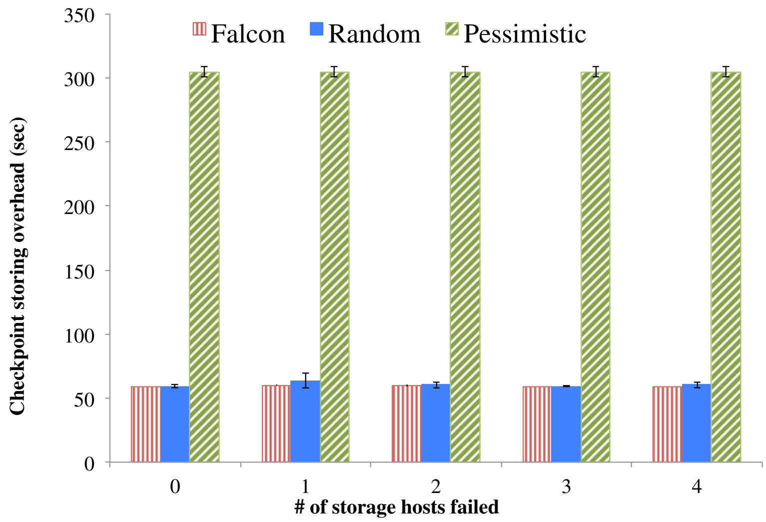


Fig. 7 Average checkpoint storing overhead of different schemes with variable number of unavailable storage hosts. This overhead includes time to re-choose storage hosts to replace unavailable ones

of choosing them randomly. This shows that FALCON's design choice of measuring history and available bandwidth out of the critical path yields robustness at no extra cost. But it is clear from Section 6.2.2 that the scheme employed by FALCON chooses storage hosts wisely. Pessimistic however incurs large overhead due to measuring bandwidth between compute and storage hosts at the beginning of every checkpoint storing instance.

6.2.3 Load Balancing vs Checkpointing Overhead

In this experiment, we compare the overheads of storing checkpoints when the workload in storage hosts varies. The objective of this experiment is to evaluate the effectiveness of FALCON's load balancing technique. For this experiment, we generated background I/O load in a single storage host out of the 5 chosen ones using a file system benchmark application Bonnie [15]. The checkpointing overheads of both the techniques, in addition to the factors described in Section 6, include the time for the storage hosts to write data to disk. Additionally, the overhead of the scheme with load balancing includes time to rechoose a storage host to replace the overloaded one.

The observation that can be made from Fig. 8 is that the difference between the load balancing and no load balancing cases comes up when load on a storage host becomes $\geq$ 80 %. As high I/O load may imply high

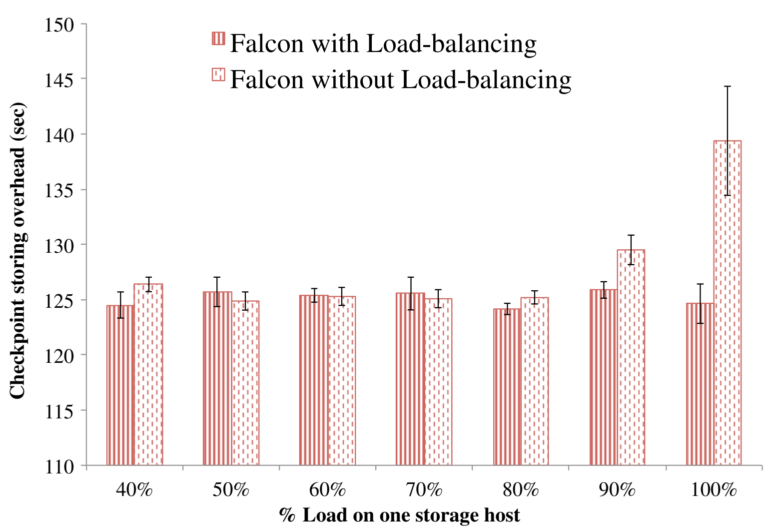


Fig. 8 Average checkpoint storing overhead of different schemes with various I/O loads on one of the storage hosts. This overhead includes the time for storage hosts to write checkpoint data and acknowledge

CPU utilization as well, the model with load balancing benefits by not sending data to this host. We set τ_2 to 80 % for FALCON with load balancing based on this observation. Ensuring balanced load among the shared storage resources is utterly important because these resources are shared by the owners voluntarily. Hence, taking advantage of these resources in a way so that the actual host's performance does not degrade beyond a threshold is as crucial a parameter as the performance benefit gained.

6.2.4 Parallel vs Sequential Retrieval of Checkpoints

In this experiment, we compare the network transfer overheads incurred by retrieving the checkpoint fragments sequentially with that of retrieving them in parallel. The size of the checkpoint fragments was 940MB each and we generated 100 % I/O load on the loaded storage hosts using Bonnie [15]. There were 5 storage hosts. The objective of this experiment is to evaluate the effectiveness of FALCON's parallel data retrieval technique when a subset of the storage hosts containing the checkpoint fragments becomes loaded. Here, the sequential scheme retrieves fragments from m storage hosts including all the loaded ones.

Figure 9 demonstrates the advantage of employing parallelism in retrieving the fragments from storage hosts. The sequential scheme performs poorly because it fetches the checkpoint fragments from the loaded hosts and can only complete after all the m fragments are fetched. So if even any one of the hosts is busy and the sequential scheme starts retrieving data from that host, it has to finish the transfer. In contrast, the parallel scheme is done as soon as any m of the fragments arrive. So, it often happens that there are m not so loaded hosts and the retrieval process finishes early.

6.2.5 Contributions of Compression, Load balancing, and Network level Parallelism

In this experiment, we compare the contributions of each of the three schemes—compression, load balancing, and parallel retrieval of checkpoint fragments—in improving the recovery overhead and in turn, in improving the performance of the applications. For this, we successively remove one of the schemes from FALCON while keeping the other two schemes. The checkpoint used to run this experiment is that of the application TIGR-I (Table 1).

In this experiment, PXfer stands for the parallel network transfer of the checkpoint fragments, SXfer stands for the sequential network transfer, D stands for decoding, and UZ stands for decompression. Each scheme consists of a combination of multiple such schemes. For example, PXfer+D+UZ implies that this scheme retrieves checkpoint fragments in parallel, decodes them, and then decompresses them. The first observation from Fig. 10 is that the largest contribution in improving recovery overhead comes from compressing the checkpoint data. The highly compressible nature of these checkpoint data can result in a compression factor as large as 86 % (for mcf) and 79 % for this application (Table 1). This in turn reduces the network transfer overhead. Also, if the checkpoint is not compressed, the decoding overhead increases. An important point to note is that even though the

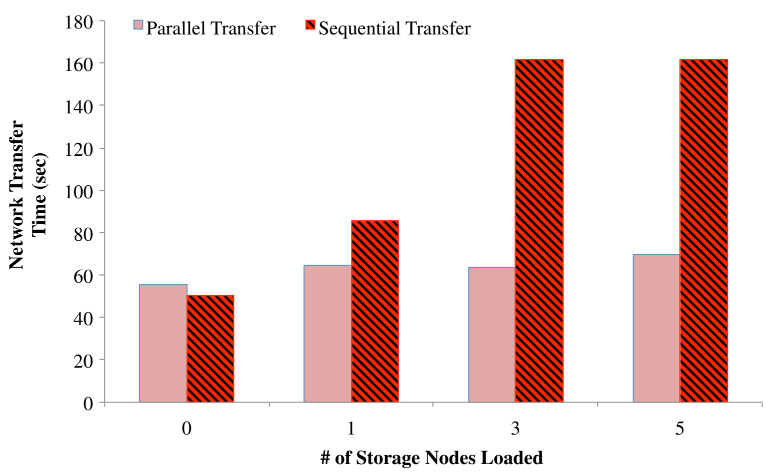


Fig. 9 Parallel vs sequential retrieval of the checkpoint fragments during a recovery phase. Note that the loaded storage hosts are included in the list of m hosts which the sequential scheme contacts for retrieving the checkpoint fragments

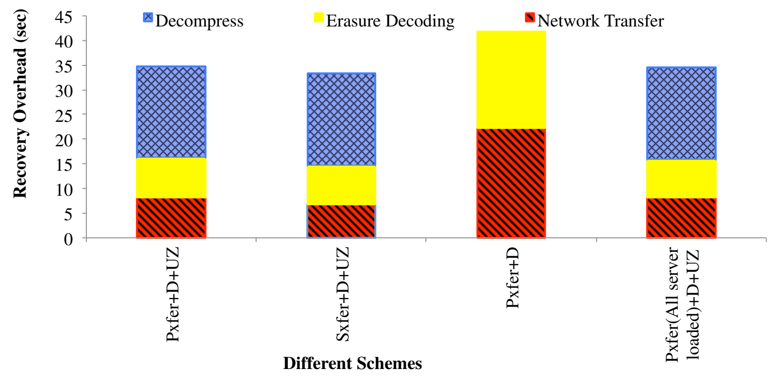


Fig. 10 The contributions of each of the 3 techniques—compression, load balancing, and parallel network transfer—in improving recovery overhead. Note that, lower recovery overhead reduces the makespan of an application and hence results in improved performance in Section 6.1.1

decompression overhead is the most dominating component in the recovery overhead, it is worthwhile to compress and decompress. Otherwise encoding very large checkpoints ($\geq$ 1GB) incurs very high memory cost and requires very long time, if at all possible. Second, due to the small size of each checkpoint fragment, network transfer overhead of both the parallel and the sequential schemes are comparable. The advantage of using the parallel scheme over the sequential one in retrieving the checkpoint fragments is discussed in Section 6.2.4. Third, due to the smaller sizes of the checkpoint fragments, the overhead of retrieving the m fragments from storage hosts with 100 % I/O load is comparable to that of retrieving them from non-loaded ones. But the point to note is that load-balancing while it does not have a very prominent contribution in lowering the recovery overhead in this experiment, it has an impact on the checkpoint transfer overhead (Section 6.2.3). Load balancing is also crucial because the storage hosts are shared resources. So, during the checkpoint storing phase, if compute hosts disregard the fact that a storage host is loaded and put more load on it by sending the bulk of checkpoint data, then the performance of the host jobs on that storage host may degrade considerably. This may cause the owner to remove his resource from the pool.

6.3 Single-threaded vs Multi-threaded Architecture of Falcon

In this section, we discuss the experiments that we ran to compare the checkpoint storing and recovery overheads of the two different architectures of Falcon. At first, we ran an experiment to find out the value of b, the number of blocks that the checkpoint should be decomposed into, that should be used to get high overall performance improvement. Then, we used this value of b to run experiments by varying the checkpoint sizes. The goal of these experiments is to compare the performance of Falcon architectures with and without the TLP option turned on. This new multi-threaded architecture works by breaking up the checkpoint data, compressing and creating $m + k$ fragments for each block in parallel. Transferring the checkpoint blocks in parallel over the network was already part of the single threaded architecture of Falcon. So, we only compare the overheads of parallel compression and encoding during the checkpoint storing phase and parallel decoding and decompression during the recovery phases of the two schemes. Lower compression and encoding overheads translate into lower overall checkpoint storing and recovery overheads.

6.3.1 Determine the Degree of Parallelism

In this experiment, for each size of checkpoint, we varied the number of threads working concurrently on the data. The machine that we used to run this experiment was an 8 core SMP (Symmetric Multi-processor) machine.

The observation that can be made from Fig. 11 is that the decrease in compression overhead levels off as the number of cores concurrently used increases.

In addition to that, the ability of a parallel program's performance to scale with the number of cores is the result of a number of interrelated factors. Some of the hardware related limiting factors are memory-cpu bus bandwidth and the amount of memory available per core on a shared memory machine. As the number of cores used increases, the amount of memory available to each of the cores decreases and hence

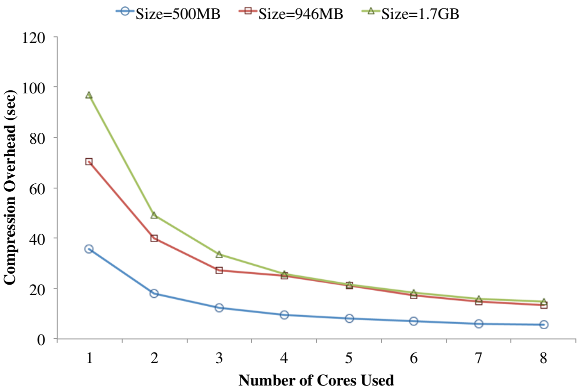


Fig. 11 The change in compression overhead for different sizes of checkpoint data with the increase in the number of cores working concurrently

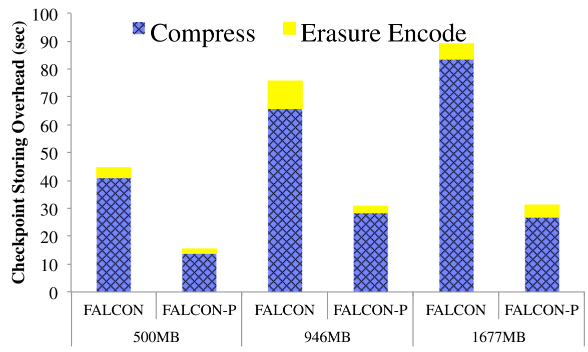


Fig. 12 The overhead of compressing and encoding checkpoints of different sizes with and without taking advantage of TLP

results in memory contention when more than 50 % of the cores start being used. Based on this experiment we set the configurable parameter b to half of the total number of cores, i.e., 4 for the rest of the experiments.

6.3.2 Comparison of Checkpointing Overheads

In this experiment, we compare the overheads of compressing a checkpoint data and then encoding it into $m+k$ fragments with that of breaking up a checkpoint data into b blocks and then compressing and encoding each block in parallel. For this experiment, we set (m, k) to $(3, 2)$ and $b = 4$. In the figure, the scheme "FALCON" represents the single threaded architecture where as "FALCON-P" represents the multi-threaded one. The checkpoints were all generated by using different inputs to the same benchmark application TIGR.

One observation that can be made from Fig. 12 is that by breaking a large checkpoint up into multiple blocks and then working on each block in parallel reduces the overhead significantly. Table 3 shows a complete break down of the amount of data that each core had to deal with. The multi-core architecture of FALCON improves the compression overhead by up to 67 % (higher gains for larger checkpoint sizes). Also, the encoding overhead reduces since now the size of the input data to each invocation of the erasure encoding algorithm is reduced by more than 50 %.

6.3.3 Comparison of Recovery Overheads

This experiment compares the recovery overheads incurred by the two architectures.

The observations that can be made from Fig. 13 are that:

- decoding overhead decreases since the fragment sizes to work with are smaller
- as checkpoint size increases, the difference between the decompression overheads decreases. The reason is that all the threads write to the same file and disk writes cannot be made in parallel. Hence with the increase in the size of a checkpoint data, there is not much improvement in decompression overhead.

Even though the improvement in the recovery overhead diminishes as checkpoint sizes increase, the reduction in the checkpoint storing overhead is significant. This justifies the use of thread level parallelism in the new architecture of FALCON (i.e., FALCON-P).

6.3.4 Discussion

The objective function developed in (3) in general ranks storage hosts with lower I/O load and are "faster" higher. In cases where a previously selected storage host becomes unavailable or slow, when the compute host component contacts the storage host component, it accesses the current load on the storage host and replies back saying that the storage host is experiencing very high load. With that reply, the compute host component reselects another storage host that has not been selected already and sends its checkpoint fragment. Since the cost of reselecting a storage host is comparable to randomly selecting a storage host, as shown in Section 6.2.2, the checkpointing phase of FALCON is not impacted by a previously

Table 3 A comparison of the input sizes to the compression and the erasure encoding algorithms in two different architectures of FALCON

App.	Input: Compression Algo.(MB)		Input: Encoding Algo.(MB)	
	FALCON	FALCON-P (per thread)	FALCON	FALCON-P
mcf	1677	419.25	241	(65,64,62,62)
TIGR-I	946	236.5	201	(83,49,36,34)
TIGR-II	500	125	153	(40,40,39,34)

Here, FALCON represents the single threaded and FALCON-P represents the multi-threaded architecture

selected storage host suddenly becoming slow. During recovery phase, FALCON starts reading all $m+k$ fragments in parallel from $m+k$ storage hosts and only waits for any m fragments to appear at the compute node. This is possible due to the fact that FALCON can recover a checkpoint from any m fragments. This ensures that the recovery time for FALCON is only dominated by the slowest of the m fastest storage hosts. As long as the number of slow storage hosts is less than or equal to k, the recovery overhead of FALCON is not impacted at all.

7 Related Work

Checkpoint-recovery is a widely used technique for providing fault-tolerance in high-performance parallel computing and distributed systems [2]. Related contributions include checkpointing facilities provided in production systems for MPI applications [16]

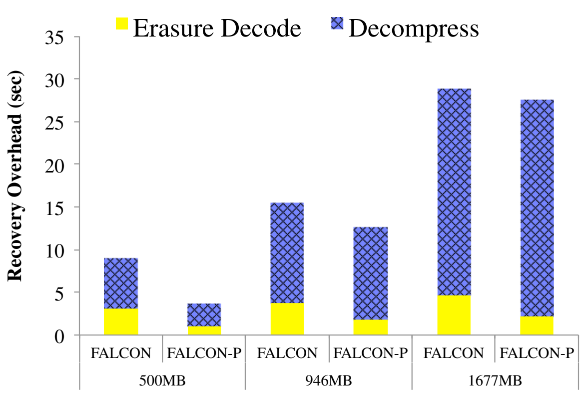


Fig. 13 The recovery overhead of two different architectures of FALCON

and improving checkpointing performance. Production grid systems such as Condor [3], take checkpoints of applications periodically and store them in dedicated servers. However, relying on such dedicated servers does not leverage the idle storage resources in grid environment. Moreover, recent research [17] has shown that using non-dedicated storage can actually result in improved performance of guest applications if a reliable set of such resources can be chosen. These results motivate our work of applying resource availability prediction to select reliable, non-dedicated checkpoint repositories.

Erasure encoding for storing data in a distributed manner to tolerate failure is a well-known technique. Related work such as [5] discusses in detail a fault-tolerant method of checkpointing and recovery using erasure coding. On the other hand, [7] compares different techniques of introducing redundancy in checkpoint data to improve fault-tolerance of applications in a shared storage environment. Erasure coding is also a popular technique for providing reliable access to data in peer-to-peer networks [18]. The OceanStore project [19] creates massive scale redundant copies of data using (among other techniques) erasure coding. The work makes contributions in efficient read operation and Byzantine fault-aware replication. The model is not that of FGCS systems and therefore the notion of guest jobs and their evictions due to resource contention is not significant.

[20] is an empirical study based on actual Condor trace. It characterizes the reasons of resource unavailability in Condor and proposes a multi-state grid resource availability characterization. A few other studies use failure modeling of compute hosts for scheduling jobs on a grid [21].

Feller et al. in [22] presents a framework that integrates different checkpointing protocols and independently checkpoint a distributed application within a heterogeneous grid environment. Ansel et al. presents DMTCP (distributed multithreaded checkpointing) in [23] which is a transparent user-level checkpointing package for distributed applications. These frameworks focus on addressing the question of how to take checkpoint or restart which is orthogonal to the research question addressed by FALCON. While the former two present checkpointing frameworks, the latter (our work) presents techniques to store checkpoints in distributed manner ensuring both efficiency and reliability. Techniques presented in this paper can be incorporated within any checkpoint-restart framework for leveraging unused disk space on grid resources for storing checkpoints.

Our work in FALCON employs some well-known techniques, to improve fault-tolerance of data (by erasure coding) and to improve performance of guest processes (by checkpointing and recovery). However, distinct from prior work, we try to address the unanswered issues of choosing reliable storage hosts in a shared grid environment, balancing load across them, and finally using them for storing and retrieving checkpoints.

8 Conclusion

We have designed, developed and evaluated FALCON, a system that provides fault-tolerant execution of applications in FGCS systems without any dedicated storage server. Since FGCS systems do not include dedicated network, any practical checkpointing system should reduce the data transferred between two points. Our system achieves so by applying compression and storing erasure encoded fragments that are of much smaller size to a set of storage hosts. We present a load-balancing multi-state failure model for these shared storage resources and apply knowledge of this model to predict reliability. We present new checkpoint storing and retrieving techniques that efficiently handle large-sized checkpoint data, of the order of gigabytes. Finally, we run experiments in BoilerGrid, a multi-university production Condor system at Purdue University. Experiments show that FALCON provides consistency in running times and improves overall performance of jobs by 11 % to 44 % over the mechanisms of using dedicated checkpoint servers or choosing storage hosts randomly. In ongoing work, we are extending FALCON to handle parallel applications, and leverage the possibility of coexistence enabled by multi-core machines.

Acknowledgment We thank Mohammed Sayeed and Preston Smith of Rosen Center for Advanced Computing (RCAC) at Purdue for their help with BoilerGrid. We thank Mohammad Sajjad Hossain for the C version of the erasure encoding software. This work was supported, in part, by the National Science Foundation under grants No. 0751153-CNS, 0707931-CNS, and 0833115-CCF and by the Purdue Research Foundation through a Special Initiative Research Grant (SIRG). This work was performed under the auspices of the U.S. Department of Energy by Lawrence Livermore National Laboratory under Contract DE-AC52-07NA27344. Document #LLNL-JRNL-649440.

References

1. Ryu, K., Hollingsworth, J.: Resource policing to support fine-grain cycle stealing in networks of workstations. IEEE Trans. Parallel Distrib. Syst., 878–892 (2004)
2. Elnozahy, E.N.M., Alvisi, L., Wang, Y., Johnson, D.B.: A survey of rollback-recovery protocols in message-passing systems. ACM Comput. Surv. (CSUR) **34**(3), 375–408 (2002)
3. Thain, D., Tannenbaum, T., Livny, M.: Distributed computing in practice: The Condor experience. Concurr. Pract. Experience **17**(2–4), 323–356 (2005)
4. Boilergrid: A large, high-throughput, distributed computing system. [Online]. Available: http://www.rcac.purdue.edu/userinfo/resources/boilergrid/
5. Aguilera, M.K., Janakiraman, R., Xu, L.: Using erasure codes efficiently for storage in a distributed system. In: International Conference on Dependable Systems and Networks (DSN), pp. 336–345 (2005)
6. Ren, X., Eigenmann, R., Bagchi, S.: Failure-aware checkpointing in fine-grained cycle sharing systems. In: Proceedings of the 16th International Symposium on High Performance Distributed Computing, pp. 33–42 (2007)
7. De Camargo, R.Y., Cerqueira, R., Kon, F.: Strategies for storage of checkpointing data using non-dedicated repositories on grid systems. In: Proceedings of the 3rd International Workshop on Middleware for Grid Computing, pp. 1–6. ACM (2005)
8. Ren, X., Lee, S., Eigenmann, R., Bagchi, S.: Resource failure prediction in fine-grained cycle sharing systems. In: IEEE International Symposium on High Performance Distributed Computing (HPDC), pp. 19–23 (2006)
9. Ren, X., Eigenmann, R.: Empirical studies on the behavior of resource availability in fine-grained cycle sharing systems. In: International Conference on Parallel Processing (ICPP), pp. 3–11 (2006)
10. Islam, T.Z., Mohror, K., Bagchi, S., Moody, A., de Supinski, B.R., Eigenmann, R.: Mcrengine: A scalable

checkpointing system using data-aware aggregation and compression. In: International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp 1–11. IEEE (2012)
11. Islam, T.Z., Bagchi, S., Eigenmann, R.: Falcon: A system for reliable checkpoint recovery in shared grid environments. In: International Conference for High Performance Computing, Networking, Storage and Analysis (SC), pp. 1–12. ACM, New York (2009)
12. Strauss, J., Katabi, D., Kaashoek, F.: A measurement study of available bandwidth estimation tools. In: ACM SIGCOMM Conference on Internet measurement, pp. 39–44. ACM (2003)
13. Wilcox-O'Hearn, Z.: Zfec Homepage, [Online]. Available: http://allmydata.org/trac/zfec (2008)
14. Albayraktaroglu, K., Jaleel, A., Wu, X., Franklin, M., Jacob, B., Tseng, C.-W., Yeung, D.: Biobench: A benchmark suite of bioinformatics applications. In: IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), pp. 2–9 (2005)
15. Bray, T.: The Bonnie home page. Located at http://www.textuality.com/bonnie (1996)
16. Bronevetsky, G., Marques, D., Pingali, K., Stodghill, P.: Collective operations in application-level fault-tolerant mpi. In: Proceedings of the 17th Annual International Conference on Supercomputing, pp. 234–243. ACM (2003)
17. Walters, J., Chaudhary, V.: A comprehensive user-level checkpointing strategy for MPI applications, Technical report, 2007-1, The State University of New York, Buffalo, NY, Tech. Rep. (2007)
18. Rodrigues, R., Liskov, B.: High Availability in DHTs: Erasure Coding vs Replication. In: Peer-to-Peer Systems IV 4th International Workshop IPTPS 2005 (2005)
19. Rhea, S., Eaton, P., Geels, D., Weatherspoon, H., Zhao, B., Kubiatowicz, J.: Pond: the OceanStore prototype. In: Proceedings of the 2nd USENIX Conference on File and Storage Technologies (FAST) (2003)
20. Rood, B., Lewis, M.J.: Multi-state grid resource availability characterization. In: IEEE/ACM International Conference on Grid Computing, pp. 42–49. IEEE Computer Society (2007)
21. Rood, B., Lewis, M.J.: Scheduling on the grid via multi-state resource availability prediction. In: IEEE/ACM International Conference on Grid Computing, pp 126–135 (2008)
22. Feller, E., Mehnert-Spahn, J., Schoettner, M., Morin, C.: Independent checkpointing in a heterogeneous grid environment. Futur. Gener. Comput. Syst. **28**(1), 163–170 (2012)
23. Ansel, J., Arya, K., Cooperman, G.: Dmtcp: Transparent checkpointing for cluster computations and the desktop. In: IEEE International Symposium on Parallel & Distributed Processing, pp. 1–12. IEEE (2009)