# FALCON - A System for Reliable Checkpoint Recovery in Shared Grid Environments

Tanzima Zerin Islam
School of ECE
Purdue University
West Lafayette, IN
tislam@purdue.edu

Saurabh Bagchi
School of ECE
Purdue University
West Lafayette, IN
sbagchi@purdue.edu

Rudolf Eigenmann
School of ECE
Purdue University
West Lafayette, IN
eigenman@purdue.edu

## ABSTRACT

In Fine-Grained Cycle Sharing (FGCS) systems, machine owners voluntarily share their unused CPU cycles with guest jobs, as long as the performance degradation is tolerable. For guest users, free resources come at the cost of unpredictable "failures", where failures are defined as disruption in the guest job's execution due to contention from the processes of the machine owner or the conventionally understood hardware and software failures. These unpredictable failures lead to unpredictable completion times. Checkpoint-recovery has long been used for providing reliability in failure-prone computing environments. Today's production FGCS systems, such as Condor, use expensive, high-performance dedicated checkpoint servers, even though they could take advantage of free disk resources offered by the clusters' commodity machines. Also, in large, geographically distributed clusters, dedicated checkpoint servers may incur high checkpoint transfer latencies. In this paper we consider using available, free disk resources as shared storage hosts for serving as checkpoint repositories. Doing so raises new challenges in providing fault-tolerance, because a failing storage host may lead to a loss of saved application states. We model failures of such shared storage hosts and develop a prediction algorithm for such failures and then choosing an appropriate set of storage hosts. We describe the development of our system called FALCON in the production university-wide Condor testbed at Purdue University, named "BoilerGrid". Through experiments on BoilerGrid, we show that FALCON provides improved and consistent performance to guest jobs by ensuring reliability and robustness in the presence of irregular resource availability.

## Categories and Subject Descriptors

H.4 [**Performance of Systems**]: Fault-tolerance

SC09 November 14-20, 2009, Portland, Oregon, USA
(c) 2009 ACM 978-1-60558-744-8/09/11âĂę $10.00.

## General Terms

Reliability, Failure Model, Performance

## Keywords

Checkpointing, Cycle-Sharing Systems, Reliability, Condor.

## 1. INTRODUCTION

A Fine-Grained Cycle Sharing (FGCS) system [17] 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 [5]. 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 node(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 [18], 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 "BoilerGrid" [1], 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 BoilerGrid 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 BoilerGrid between March 5th, 2009 and March 12th, 2009 from PU actually ran on ND's machines.

*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 node may observe huge network traffic during checkpoint transfer, making it less preferable than a distant one.

*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.

Our work is motivated by these issues reported by scientists using BoilerGrid 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 node 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 nodes to serve as checkpoint repositories. Finally, we propose an algorithm for efficient checkpoint storage and recovery. This algorithm uses erasure encoding [8] 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 [10].
- *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 [4, 10].
- *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 [8] 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—BoilerGrid—with multiple sequential benchmark applications. The experiments ran on BoilerGrid 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 presents implementation details of FALCON. Then, experimental approaches and results are discussed in Section 5. Section 6 reviews some related works. Finally, we conclude the paper in Section 7.

## 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 [11], 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 [10], 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 [10] 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.

Resource failure in Cycle Sharing Systems may be classified as - failure due to excessive resource contention (FRC) and failure due to resource revocation (FRR) [12]. FRC represents situations when the performance of local jobs degrades due to contention for resources by local and guest jobs and FRR represents situation where owner of a machine turns it off or resource becomes unavailable due to plain software or hardware failure. In this paper, we denote both FRC and FRR with *failure*.

## 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 nodes 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 nodes 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 nodes. The coding introduces redundancy such that a subset of the fragments can be used for the recovery of the application.

### 3.1 Novel Multi-state Failure Model for Storage Hosts

In an FGCS system, storage hosts are often non-dedicated, shared nodes contributing their unused disk spaces. Checkpoint data saved by guest jobs in these storage nodes may get lost when the storage nodes 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 node will reduce the performance of host and guest jobs.
- has large available bandwidth to the computation node 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 [10], 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 saving 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

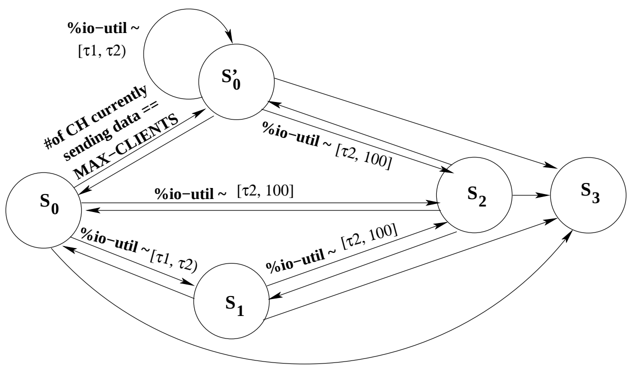

**Figure 1: New multi-state storage host failure model. Here, $S_0$: running state, $S_0'$: state where the maximum number of compute hosts are sending data, $S_1$: loaded state, $S_2$: Temporarily unavailable state and $S_3$: unavailable state.**

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 for checkpoint storage from new compute hosts 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:

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

where,

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

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

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

In Equation 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 Equation 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 storge 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 $CR(SH_k, CH_l) \geq \gamma$ ($\gamma$ is a configurable parameter, we chose $\gamma = 0.95$ for our experiments) as very reliable. The equation rounds the reliability component of $CRLS$ to $\gamma$ since we consider that reliability scores greater than $\gamma$ 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 weightage to the less loaded storage hosts and therefore add their load indicators to $\gamma$ 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 nodes 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 $\alpha$ is chosen as 0.75 in our experiments.

### 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 [10] 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 Equation 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.

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

In Equation 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 Equation 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 nodes 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.~~

$$\mathcal{F} = \frac{MTTF_{cmp}}{CI} \times \sum_{i=1}^{V} (C_i \times N_{i,j})$$

$$-(T_{curr} + MTTF_{cmp}) \times \prod_{i=1}^{V} CRLS'(SH_i, CH_j)\} \qquad (3)$$

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

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

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 checkpints. 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 [10]. However, Equation 3 uses different measures of network overhead and reliability score.

### 3.2.4 Storage Selection Algorithm

To choose storage nodes that minimize the objective function in Equation 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 nodes 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. Greater is the volatility in the underlying grid environment, smaller should be the setting of the period. 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 seconds.

## 3.3 New Method for Checkpoint Recovery

In our previous work [10] 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 [10]. 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 node from the list of unselected ones. The same greedy algorithm, described in Section 3.2, is used to rechoose. 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. 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 compute and storage hosts. Modules in CHC and 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.

### 4.1 Compute Host Component (CHC)

Compute host component is the part of Falcon that runs on an execution host machine. 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 to a file. For available bandwidth measurement, we have used Spruce [6], a light-weight available bandwidth measurement tool. Spruce provides a server module that runs as a part of CHC and a client module that is a part of SHC. For our experiments, we have used a period of measuring available bandwidth of 10 seconds.
- 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 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 [20] 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`.
- 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.

Most of the modules are light-weight, requiring little CPU and memory resources.

### 4.2 Storage Host Component (SHC)

The SHC consists of three modules:

- Module `Load` measures disk I/O load periodically. This process runs in parallel to actual server module `Srvr` and generates required load information that module `Srvr` uses to determine which state the storage server is in according to Figure 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 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` (number of compute hosts currently being served).

### 4.3 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, 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 Equation 1. Our current implementation uses a central server approach. This design can be extended to a distributed implementation.

## 5. EVALUATION

We have developed a complete system, as described in Section 4. 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 [7] benchmark suites. SPEC CPU 2006 is widely used for benchmarking CPU-intensive programs while BioBench consists of well known biomedical applications. This section presents the experiments for evaluating the system in terms of its checkpoint recovery overheads and its effectiveness in improving job makespan.

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 dedicated storage server is used for saving checkpoint data.

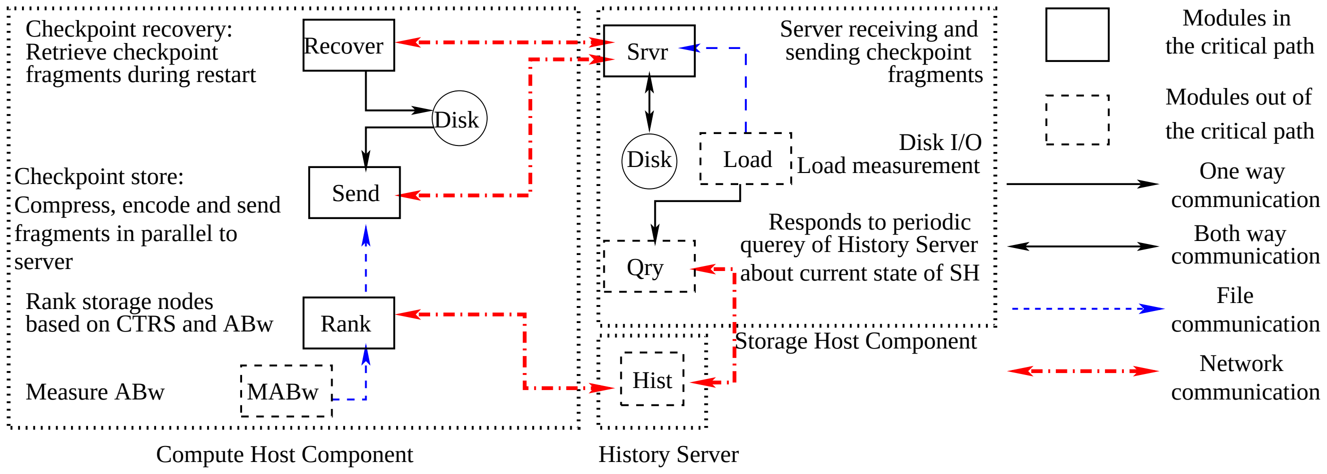
Checkpoint recovery:
Retrieve checkpoint fragments during restart
Recover
Disk
Checkpoint store:
Compress, encode and send fragments in parallel to server
Send
Rank storage nodes based on CTRS and ABw
Rank
Measure ABw
MABw
Compute Host Component
Srvr
Server receiving and sending checkpoint fragments
Disk
Load
Disk I/O
Load measurement
Qry
Responds to periodic querey of History Server about current state of SH
Storage Host Component
Hist
History Server
Modules in the critical path
Modules out of the critical path
One way communication
Both way communication
File communication
Network communication

**Figure 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.**

| 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% |

**Table 1: Checkpoint sizes of different applications. mcf and TIGR are benchmark applications part of SPEC and BioBench respectively. TIGR-I, TIGR-II and TIGR-III are runs of TIGR with different input sizes.**

- Random: A scheme where the storage hosts for saving the erasure encoded checkpoints are randomly chosen from among all available storage hosts.

~~Note that, t~~he default scheme of Condor is to send checkpoint back to submitter machine (machine from which the job is submitted). This scheme ~~is similar to that of using a dedicated server and hence performs no~~ better than ~~our reference D~~edicated algorithm ~~(~~ in fact, it can be significantly worse, if the submitter sits behind a low-bandwidth connection~~)~~.
Checkpoint storing overhead includes time:

- For Falcon ~~to~~: (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 ~~to~~: (i) Read checkpoint from disk (ii) Send checkpoint to storage server
- For Random ~~to~~: (i) Choose storage nodes randomly (ii) Follow steps (ii) - (vi) of Falcon

Recovery overhead includes time:

- For Falcon ~~to~~: (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) ~~De~~compress and write to disk
- For Dedicated ~~to~~: (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 to: (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.

## 5.1 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 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 show the failure characteristics in Table 2. We use 1.3 evictions per hour as the rate of eviction for our experiments in Section 5.1.1. In Section 5.1.1 we compare average job makespan of applications using different checkpoint repository techniques. The decompositions of checkpoint storing and recovery overheads are shown in Section 5.1.2 and Section 5.1.3 respectively.

| $N$ | $\mu$ | $\sigma$ | *Range* |
|---|---|---|---|
| 116 | 1.3130 | 0.2172 | [1.0298,2.3931] |

**Table 2: Statistical analysis of the eviction characteristics in BoilerGrid. The table shows number of jobs for which we collected data ($N$), average number of evictions per hour ($\mu$), standard deviation ($\sigma$) and range.**

### 5.1.1 *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.

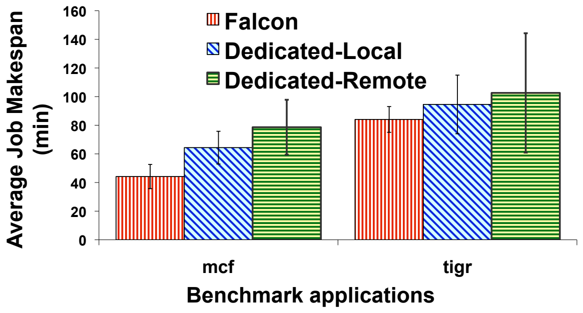

**Figure 3: Average job makespan of different applications. Here, Dedicated-Local represents the dedicated scheme using a local checkpoint server and Dedicated-Remote represents the dedicated scheme using a remote checkpoint server.**

From Figure 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 5.1.3 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 5.2.5.

### 5.1.2 *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.

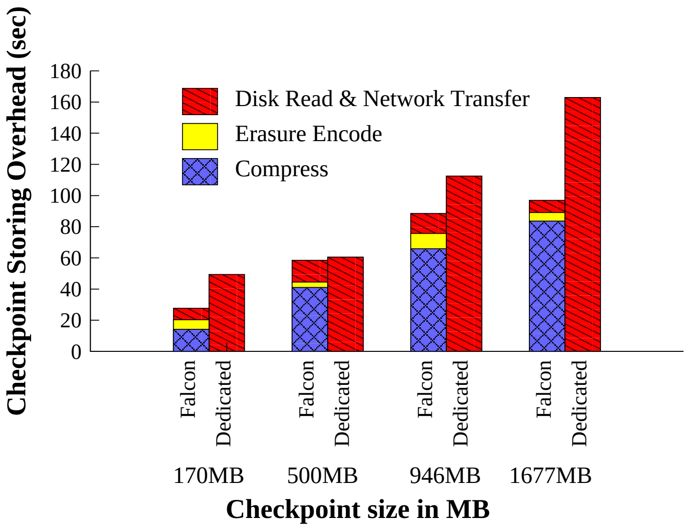

**Figure 4: Average checkpoint storing overhead vs different checkpoint sizes.**

One observation that can be drawn from Figure 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 parallelizing the transfer of smaller checkpoint fragments speeds this up.

### 5.1.3 *Recovery Overhead*

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

One observation that can be made from Figure 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

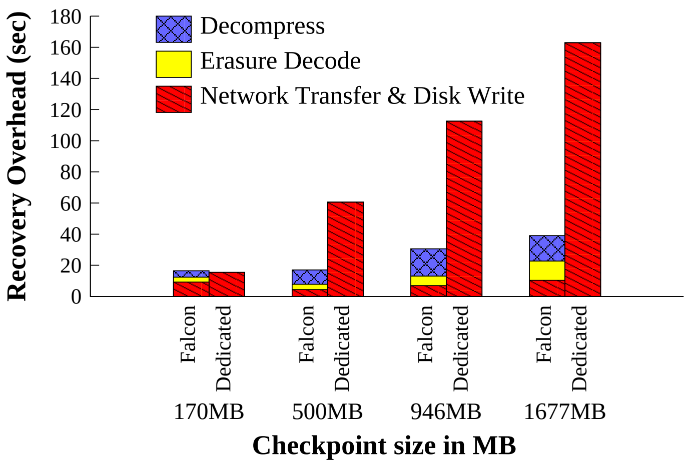

**Figure 5: Recovery overhead of four different checkpoint sizes generated by applications in Table 1. This figure shows contributions of different components in total recovery time. Erasure coding parameters are (m=3, k=2). Fetch and disk write time also includes the time to read checkpoint data from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

overheads directly translate to better performance for an application.

## 5.2 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.

### 5.2.1 *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 5, 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 Figure 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,

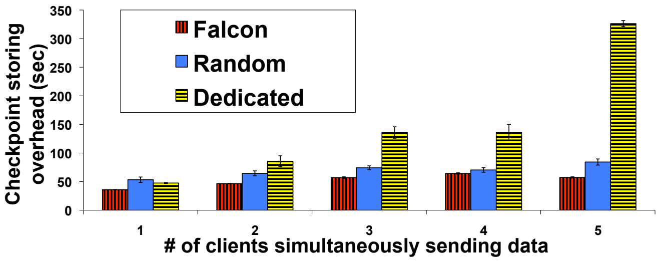

**Figure 6: Average execution time of the algorithms vs number of clients concurrently sending data to the servers.**

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.

### 5.2.2 *Efficiency in Handling Storage Failures*

Since storage hosts in FGCS are non-dedicated resources, a protocol must be able to handle unavailability of storage nodes efficiently. The objective of this experiment is to compare the added overhead of re-choosing storage nodes by FALCON with that of Random and the more conservative approach of Pessimistic [10]. For this experiment, we killed the storage daemons running in those storage hosts to make them appear unavailable.

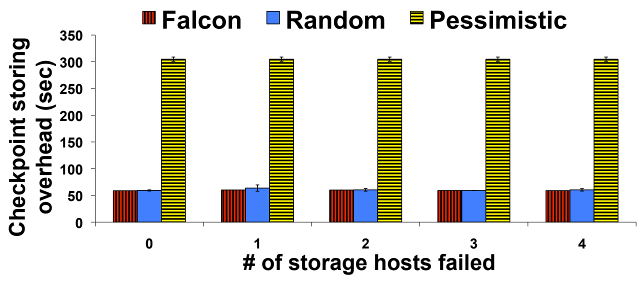

**Figure 7: Average checkpoint storing overhead of different schemes with variable number of unavailable storage hosts. This overhead includes time to re-choose storage hosts to replace unavailable ones.**

One observation that can be drawn from Figure 7 is that the overhead of re-choosing storage hosts using history and available bandwidth is no worse than that 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 5.2.2 that the scheme employed by FALCON chooses storage nodes wisely. Pessimistic however incurs large overhead due to measuring bandwidth between compute and storage hosts at the beginning of every checkpoint storing instance.

### 5.2.3 *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 [2]. The checkpointing overheads of both the techniques, in addition to the factors described in Section 5, 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.

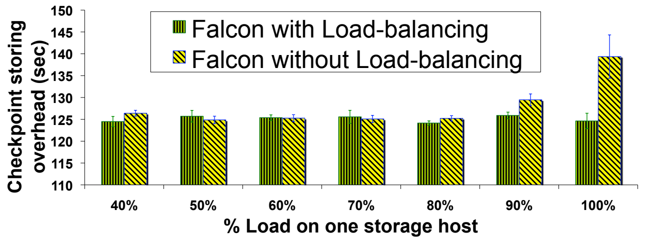

**Figure 8: Average checkpoint storing overhead of different schemes with various I/O loads on one of the storage hosts. This overhead includes the time for storage hosts to write checkpoint data and acknowledge.**

Since $\tau_2$ is set as 80%, in Figure 8, the difference between the load balancing and no load balancing cases comes up when load on a storage host becomes $\geq$ 80% and FALCON with load balancing scheme re-chooses. As high I/O load may imply high CPU utilization as well, the model with load balancing benefits by not sending data to this host. 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.

### 5.2.4 *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 nodes using Bonnie [2]. There were 5 storage nodes ~~and the erasure encoding parameters were set at~~ (3,2) ~~- meaning 3 required and 2 redundant fragments~~. The objective of this experiment is to evaluate the effectiveness of FALCON's parallel data retrieval technique when a subset of the storage nodes containing the checkpoint fragments becomes loaded. Here, the sequential scheme retrieves fragments from $m$ storage nodes including all the loaded ones.

Figure 9 demonstrates the advantage of employing parallelism in retrieving the fragments from storage nodes. The

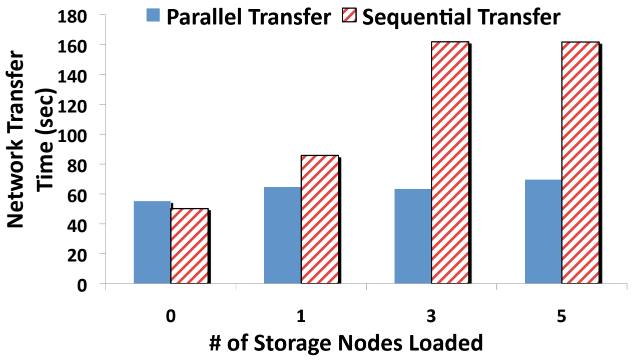

**Figure 9: Parallel vs sequential retrieval of the checkpoint fragments during a recovery phase. Note that the loaded storage nodes are included in the list of $m$ nodes which the sequential scheme contacts for retrieving the checkpoint fragments.**

sequential scheme performs poorly because it fetches the checkpoint fragments from the loaded nodes and can only complete after all the $m$ fragments are fetched. So if even any one of the nodes is busy and the sequential scheme starts retrieving data from that node, 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 nodes and the retrieval process finishes early.

### 5.2.5 *Contributions of Compression, Load balancing, and 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).

The first observation from Figure 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 uncompression overhead is the most dominating component in the recovery overhead, it is worthwhile to compress and uncompress. Otherwise encoding very large checkpoints (>= 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 5.2.4. Third, due to the smaller sizes of the checkpoint fragments, the overhead of retrieving the $m$ fragments from storage nodes 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

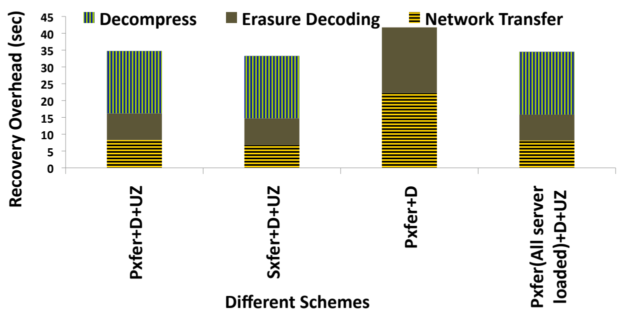

**Figure 10: This figure shows the contributions of each of the 3 techniques—compression, load balancing, and parallel network transfer—in improving recovery overhead. Note that, lower recovery overhead reduces the makespan of an application and hence results in improved performance in Section 5.1.1. Here, 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 uncompression. 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 uncompresses them.**

recovery overhead in this experiment, it has an impact on the checkpoint transfer overhead (Section 5.2.3). Load balancing is also crucial because the storage nodes are shared resources. So, during the checkpoint storing phase, if compute hosts disregard the fact that a storage node 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 node may degrade considerably. This may cause the owner to remove his resource from the pool.

## 6. RELATED WORK

Checkpoint-recovery is a widely used technique for providing fault-tolerance in high-performance parallel computing and distributed systems [5]. Related contributions include checkpointing facilities provided in production systems for MPI applications [3] and improving checkpointing performance. Production grid systems such as Condor [18], 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 [19] 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 [8] discusses in detail a fault-tolerant method of checkpointing and recovery using erasure coding. On the other hand, [4] 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 [14]. The OceanStore project [13] 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.

[16] 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 [15].

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 nodes in a shared grid environment, balancing load across them, and finally using them for storing and retrieving checkpoints.

## 7. 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. 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 a new checkpoint store and retrieve technique that efficiently handles 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.

## 8. ACKNOWLEDGMENT

We thank Mohammed Sayeed and Preston Smith of Rosen Center for Advanced Computing (RCAC) at Purdue for their help with BoilerGrid. We thank 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). We would also like to thank the anonymous reviewers for their valuable feedback.

## 9. REFERENCES

[1] http://www.rcac.purdue.edu/boilergrid/.

[2] T. Bray. The Bonnie home page. *Located at http://www.textuality.com/bonnie*, 1996.

[3] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill. Collective operations in application-level fault-tolerant mpi. In *ICS '03*, pages 234–243, 2003.

[4] de Camargo, R. Y., Cerqueira, Renato, and K. Fabio. Strategies for storage of checkpointing data using non-dedicated repositories on grid systems. In *MGC*, pages 1–6, 2005.

[5] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[6] D. K. Jacob Strauss and F. Kaashoek. A measurement study of available bandwidth estimation tools. In *IMC*, pages 39–44, 2003.

[7] A. K., J. A., X. Wu, F. M., J. B., C.-W. Tseng, and Y. D. Biobench: A benchmark suite of bioinformatics applications. In *ISPASS '05*, pages 2–9, 2005.

[8] M. K. Aguilera, R. Janakiraman, and L. Xu. Using erasure codes efficiently for storage in a distributed system. In *DSN '05: Proceedings of the 2005 International Conference on Dependable Systems and Networks*, pages 336–345, 2005.

[9] X. Ren and R. Eigenmann. Empirical studies on the behavior of resource availability in fine-grained cycle sharing systems. In *ICPP '06: Proceedings of the 2006 International Conference on Parallel Processing*, pages 3–11, 2006.

[10] X. Ren, R. Eigenmann, and S. Bagchi. Failure-aware checkpointing in fine-grained cycle sharing systems. In *Proceedings of the 16th international symposium on High performance distributed computing*, pages 33–42, 2007.

[11] X. Ren, S. Lee, R. Eigenmann, and S. Bagchi. Resource Failure Prediction in Fine-Grained Cycle Sharing Systems. In *Proc. of Fifteenth IEEE International Symposium on High Performance Distributed Computing (HPDC-15)*, pages 19–23, 2006.

[12] X. Ren, S. Lee, R. Eigenmann, and S. Bagchi. Prediction of resource availability in fine-grained cycle sharing systems empirical evaluation. *J. Grid Comput.*, 5(2):173–195, 2007.

[13] S. Rhea, P. Eaton, D. Geels, H. Weatherspoon, B. Zhao, and J. Kubiatowicz. Pond: the OceanStore prototype. In *Proc. of the 2nd USENIX Conference on File and Storage Technologies (FAST)*, 2003.

[14] R. Rodrigues and B. Liskov. High Availability in DHTs: Erasure Coding vs. Replication. In *Peer-to-Peer Systems IV 4th International Workshop IPTPS 2005*, 2005.

[15] B. Rood and M. Lewis. Scheduling on the Grid via multi-state resource availability prediction. In *Grid '08*, pages 126–135, 2008.

[16] B. Rood and M. J. Lewis. Multi-state grid resource availability characterization. In *GRID '07*, pages 42–49, 2007.

[17] K. Ryu and J. Hollingsworth. Resource Policing to Support Fine-Grain Cycle Stealing in Networks of Workstations. *IEEE Transactions on Parallel And Distributed Systems*, pages 878–892, 2004.

[18] D. Thain, T. Tannenbaum, and M. Livny. Distributed computing in practice: the Condor experience. *Concurrency - Practice and Experience*, 17(2-4):323–356, 2005.

[19] J. Walters and V. Chaudhary. A Comprehensive User-level Checkpointing Strategy for MPI Applications. Technical report, TR 2007-1, The State University of New York, Buffalo, NY, 2007.

[20] Z. Wilcox-O'Hearn. Zfec Homepage. *Located at http://allmydata.org/trac/zfec*, 2008.