

Reliable and Efficient Checkpoint/Recovery in Shared Grid Environments

Tanzima Zerine Islam
School of ECE, Purdue University
tislam@purdue.edu
<http://web.ics.purdue.edu/~tislam>
Advisors: Saurabh Bagchi and Rudolf Eigenmann
School of ECE, Purdue University
{sbgchi, eigenmann}@purdue.edu

I. INTRODUCTION & MOTIVATION

In a Fine-Grained Cycle Sharing (FGCS) system [1], machine owners voluntarily share their unused CPU cycles with guest jobs, as long as their performance degradation is tolerable. However, for guest users, these free computation resources come at the cost of fluctuating availability due to resource contention or resource revocation by the owners. As the size and the boundary of a grid expand, so grows the frequency of failures. The completion times of the long running compute-bound jobs fluctuate widely due to this effect.

To achieve high performance in the presence of resource volatility, *checkpointing* and *rollback* [2] has been widely applied. The research questions that we investigate in this work are:

- Where do we store the checkpoints and how does this affect the overall performance of the guest applications?
- How does a reliable storage repository for these checkpoints improve the performance of the guest jobs?
- How to harness the power of multiple cores available on grid resources to improve the performance of the checkpoint-recovery process?

Obviously, one of the quick answers to the first question is to employ dedicated powerful checkpoint servers with large amounts of available storage. But, this state-of-the-art solution gives rise to a number of performance and feasibility issues that have motivated us to try and answer these questions from a different point of view.

- First, a computational grid generally does not have any dedicated network to handle the load of transferring potentially gigabytes of checkpoints between a compute host and a remote storage server. So, transferring large sized checkpoints through a shared network may leave the network congested. This, in turn, increases the overall checkpointing time and affect other applications.
- Second, a physically close storage server, chosen based on round trip time (RTT), may have low available network bandwidth between the compute host and itself. Hence, it may result in higher checkpoint transfer overhead, 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 the guest processes.
- Fourth, a random choice of grid resources to store the checkpoints does not work either because once that storage host becomes unavailable, the checkpoints get lost.

Our work is motivated by the issues reported by scientists using DiaGrid (also known as BoilerGrid) [3] for running their compute-intensive jobs with checkpoint-recovery. For this, we have developed a framework, FALCON, for reliable execution of applications in a *shared grid environment* [4] and answer the first two questions. To answer the third one, we have identified the checkpoint storing phase of FALCON to be the bottleneck and are leveraging the inherent data-parallelism present in the checkpoints to harness the computational power of multiple cores present in the grid resources. A reliable and efficient checkpoint recovery scheme, such as FALCON, will enable the idea of a nation wide grid without any organization to actually invest in purchasing expensive powerful machines and setup a cluster to be able to take full advantage of a computational grid.

II. BACKGROUND & RELATED WORK

In our previous work [5] and [6], we developed a failure model for computation hosts and used it for predicting the availability of both the computation and the storage hosts. But it fell short in addressing the fact that a storage host in a shared grid environment utilizes different types of resource. Also, we did not address either the issue of load-balancing for storage hosts or that of flash crowd of large checkpoints overwhelming a storage host. Additionally, none of the related contributions [2], [7], [8] have looked into utilizing data-parallelism of the checkpoints to take advantage of multi-core technology available on the grid machines.

Production grid systems such as Condor [8], take checkpoints of applications periodically and store them in dedicated centralized servers. In contrast, we take a distributed storage approach and leverage the idle storage resources in a grid environment.

Recent research [9] 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.

We use a well known technique called erasure encoding for storing data in a distributed manner to tolerate failure [10], [11], [12].

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. There have been a lot of studies that use failure modeling of compute hosts for scheduling jobs on a grid [14]. In contrast, our work looks at availability prediction of storage hosts and reduces the overheads of the checkpoint-recovery phases to ultimately improve the performance of the guest applications.

III. APPROACH & UNIQUENESS

Our approach for storing the application checkpoints is a distributed one. We list the steps as follows and discuss their uniqueness compared to the related literature.

A. Novel Multi-state Failure Model for Storage Hosts

First, we have designed a *multi-state failure model* for predicting the reliability and availability of storage hosts in a shared grid environment. Related works such as [6], [15] have developed failure models for compute hosts, which fall short for resources with dual roles in a shared grid environment.

Figure 1 presents our new five state failure model for storage hosts. From high-level, this failure model depicts the states a storage host may reside in depending on its availability (available or not) and %I/O utilization. From a compute host's point of view, it prefers a storage host that has high availability and is lightly loaded. From a storage host's point of view, if the %I/O utilization is too high, it does not accept any more request from compute hosts. It also does not accept any new request from other compute hosts if the number of compute hosts already sending data reaches certain threshold. These two states together ensure load-balancing among storage hosts and prevent the problem of flash crowd from occurring. Details of the failure model can be found in [4].

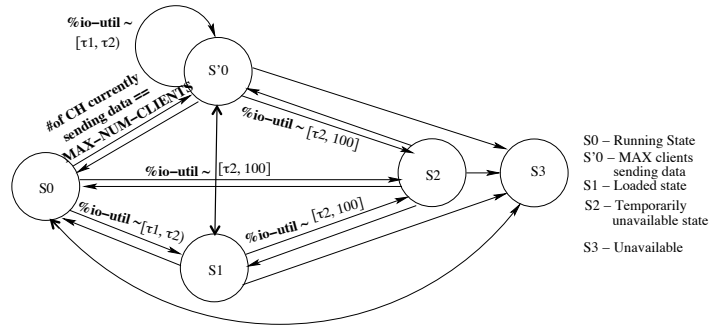


Fig. 1. Novel multi-stage storage host failure model

B. Failure-aware Storage Selection

Second, we have designed and developed a *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 [11], [6]. In this regard, we take the following steps:

- First, we predict the correlated reliability of the storage hosts compared to a compute host. We define *correlated temporal reliability* as the probability that a storage host is available when a compute host is not. The rationale, behind using the correlated temporal reliability instead of the absolute reliability of a storage host, is that, if the failures of a storage host is correlated with that of a compute host, the checkpoints stored on that storage host will not be available when needed. This might often be the case if both the resources show similar maintenance schedule. We calculate a score, *Correlated Temporal Reliability Score (CTRS)* based on the equation shown in Figure 2(a). The basic idea expressed in this equation (Figure 2(a)) is that, the storage hosts that have higher probability of being in lightly loaded states will result in a higher value of CTRS. Storage hosts that have reliability above certain threshold, are going to be ranked according to their %I/O load.
- Second, we design an objective function that calculates the difference between the cost and the benefit of checkpointing. The cost is in terms of the overhead of storing a checkpoint and the benefit of restarting from this checkpoint after a failure. Our goal is to pick the $m + k$ storage hosts that minimize the objective function described in Equation (b) shown in Figure 2. A detailed description of the objective function can be found in [4].

$$CTRS(SH_k, CH_l) = \begin{cases} S_1 + S_2, & \text{if } S_1 \geq \gamma; \\ S_1, & \text{otherwise.} \end{cases}$$

where,

$$Pr_{CH_l}(i) = Pr\{CH_l \text{ in state } i\}$$

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

$$\zeta = \begin{cases} Pr_{SH_k, CH_l}(0|0) \\ + Pr_{SH_k, CH_l}(1|0) \\ + Pr_{SH_k, CH_l}(2|0), & \text{if } Pr_{CH_l}(0) > 0; \\ \gamma, & \text{otherwise.} \end{cases}$$

$$S_1 = \min[\zeta, \gamma]$$

$$S_2 = \begin{cases} \alpha \times Pr_{SH_k, CH_l}(0|1) \\ + (1 - \alpha) \times Pr_{SH_k, CH_l}(1|1), & \text{if } Pr_{CH_l}(1) > 0; \\ 0, & \text{otherwise.} \end{cases}$$

(a)

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

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

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

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

$$CTRS'(SH_i, CH_j) = \max[1 - C_i, CTRS(SH_i, CH_j)]$$

(b)

Fig. 2. (a) Calculation of Correlated Temporal Reliability Score. The higher the value of CTRS, the better. From high-level, storage hosts with their reliability above certain threshold (γ) are ranked according to their %I/O load. (b) Objective function that ranks storage hosts according to CTRS and available bandwidth between a compute host and a storage host.

- Figure 3 shows the block diagram of the *an efficient checkpointing method* that provides fault-tolerance to the process of checkpointing data. Our method uses parallelism offered by multiple fragments being stored in multiple storage hosts to reduce checkpoint and recovery overheads. Different steps of the algorithm are,
 - Compression by b number of parallel threads from index $i \times b$ by thread i . Here, b is a configurable parameter that represents the number of parallel threads to spawn. This step reduces the bottleneck of compressing a potentially gigabytes sized checkpoint data. We have found (Figure 4(b)) that the compression ratio of these checkpoints are very high (e.g., a benchmark application MCF [16] has that of 85.63%). So, this step reduces the amount of network I/O as well as improves the recovery overhead because less amount of data needs to be transferred.
 - Use erasure coding on the compressed checkpoint blocks, one block per thread, to break it into $m + k$ fragments. After this step, there will be $b \times (m + k)$ fragments.
 - Send the k^{th} checkpoint fragment of all the blocks to the k^{th} of the chosen storage hosts.
 - During recovery, first fetch the checkpoint fragments, then decode them using erasure decoding algorithm and then decompress each block in parallel to get the original checkpoint data.

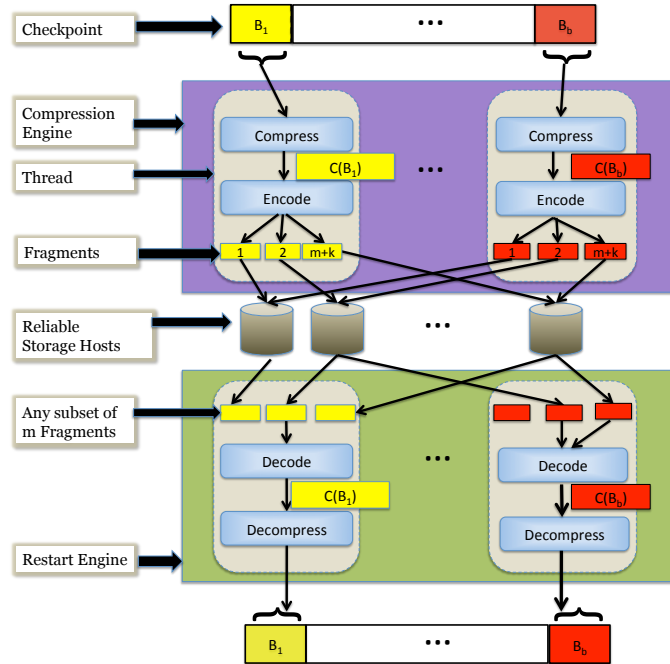


Fig. 3. This figure shows the block diagram of FALCON. FALCON exploits the data-level parallelism of the checkpoints to efficiently handle large sized data. Multiple cores available in the computational resources aid FALCON in making the checkpointing system scalable and efficient.

By employing b threads to compress, encode and transfer b blocks in parallel, we have potentially reduced the amount of data input to the compression algorithm and also, reduced network I/O. Also, by sending different fragments of data to different storage hosts by different threads in parallel, we are able achieve high aggregated network I/O bandwidth.

IV. RESULTS & CONCLUSION

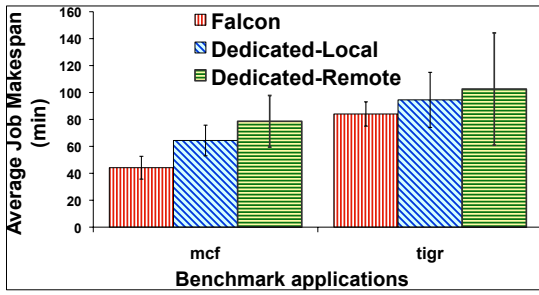
A. Overall Average Job Execution Time

The ultimate goal of any fault tolerant system is to improve the application execution time. For evaluating the overall impact of our system, we integrated three checkpoint storing schemes with two benchmark applications. FALCON uses shared resources available to DiaGrid, Dedicated-local with a local checkpoint server (lab machine connected to the campus-wide LAN at Purdue) and Dedicated-remote with a remote checkpoint server (machine at University of Notre Dame connected to Internet). We submitted jobs in DiaGrid. These applications took checkpoint once every 5 minutes. Here, the Dedicated-remote scheme represents the situation when jobs submitted from one university go to run in another university.

From Figure 4, we can observe that FALCON outperforms Dedicated-local and Dedicated-remote by 11% and 44% respectively for the application MCF with checkpoint size of 1.6GB.

B. Efficiency in Handling Simultaneous Clients

The objective of this experiment is to show how the performance of different schemes scale with the load imposed by concurrent writers. The observation that can be made from Figure 5(a) is that increase in the number of concurrent clients makes the dedicated scheme suffer more. In other words, the performance of FALCON scales well as the number of clients in a grid increases.

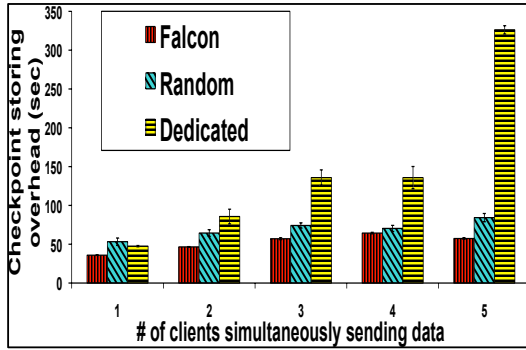


Applications	mcf	TIGR-I	TIGR-II	TIGR-III
Original Checkpoint Size (MB)	1677	946	500	170
Compressed Checkpoint Size (MB)	241	201	153	129
Compression Ratio	85.63%	78.75%	69.4%	24.12%

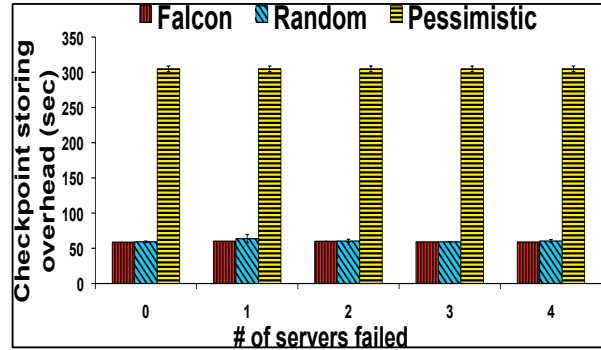
(a)

(b)

Fig. 4. (a) 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. (b) Sizes of different application checkpoints and their compression ratio.



(a)



(b)

Fig. 5. (a) Average execution time of the algorithms vs number of clients concurrently sending data to the servers. Checkpoint storing overhead includes the time to write the data to disk at the storage host end. The performance of FALCON scales well. (b) As a number of servers fail, FALCON re-chooses new storage hosts fast enough not to significantly deteriorate the performance of the checkpointing phase.

C. 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 smartly 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 made in Figure 5(b) is that the overhead of re-choosing storage hosts using history and available bandwidth is no worse than choosing them randomly. However, Figure 5(a) shows why FALCON is better than Random. Pessimistic however incurs large overhead due to measuring bandwidth between compute and storage hosts every time. This shows that FALCON’s design choice of measuring history and available bandwidth out of the critical path yields robustness at no extra cost.

D. Checkpoint Storing & Recovery Overheads

Figure 6 shows the decomposition of both the checkpointing and the recovery overheads of FALCON. Here, the Dedicated scheme uses a remote checkpoint server. The observations that can be made from Figure 6(a) are:

- compression and erasure encoding of a large checkpoint enable FALCON to take advantage of network level parallelism. The gain is 37.5% during the checkpointing phase and 75% during the recovery phase for a checkpoint of size 1.6GB. As checkpoint sizes grow, this disparity will grow even more and will hinder applications taking large checkpoints, to take advantage of fault tolerance in a grid.
- since recovery overhead is incurred each time an application gets evicted, lower recovery overhead directly improves the performance of an application.
- time to compress seems to dominate the checkpointing overhead. As checkpoint sizes grow, compression overhead grows more than that of encoding and transfer. An increase in checkpointing overhead will translate into higher execution time for applications that take synchronous checkpoints.

E. Parallel vs Sequential Checkpoint Storage

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 Figure 7(a) is that by breaking a large checkpoint up into multiple blocks and then working on each block in parallel reduces the overhead significantly. The multi-core architecture of FALCON improves the compression overhead

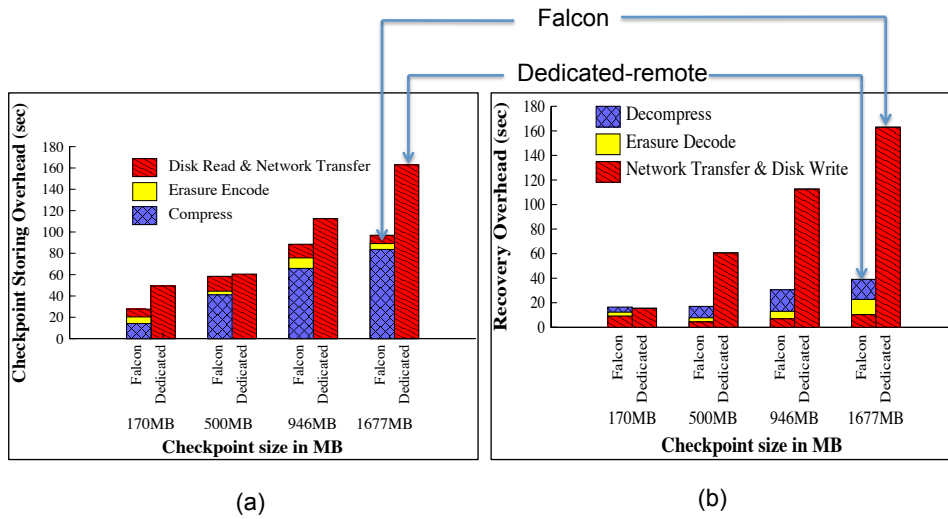


Fig. 6. (a) Breakdown of the overheads of the different steps taken by FALCON during the checkpointing phase. (b) Breakdown of the overheads of the different steps taken by FALCON during the recovery of a checkpoint.

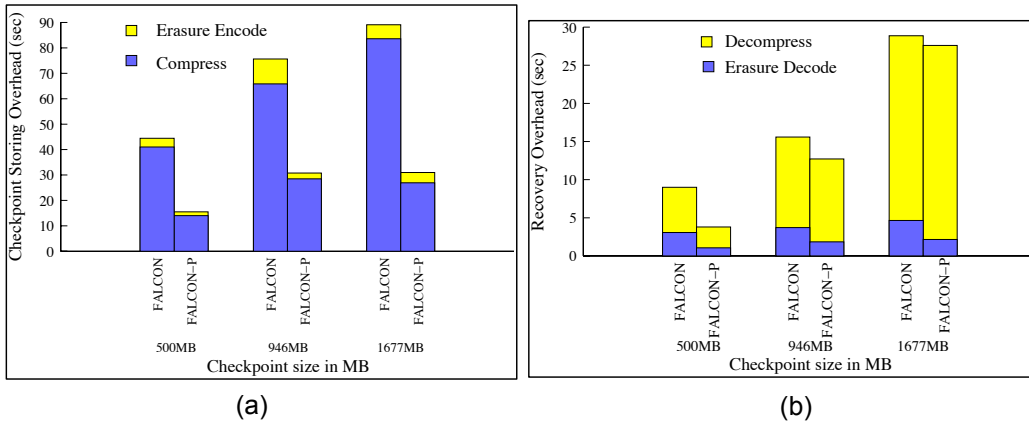


Fig. 7. (a) The overhead of compressing and encoding checkpoints of different sizes with and without taking advantage of thread-level parallelism. (b) The overheads of erasure decoding and decompression — with and without taking advantage of thread-level parallelism. The decompression phase also includes the time to write the reconstructed checkpoint to disk.

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%.

F. Parallel vs Sequential Checkpoints Retrieval

This experiment compares the recovery overheads incurred by the two architectures. The observations that can be made from Figure 7(b) 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.

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 FALCON (i.e., FALCON-P).

G. 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 scalable and efficient checkpoint storing and retrieval technique that leverages the multiple cores of the computation machines to handle large-sized checkpoint data, of the order of gigabytes. Finally, we deployed FALCON in DiaGrid, a multi-university production Condor system at Purdue University and ran experiments with benchmark applications in this system. 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. We believe that FALCON is the ultimate solution to the issue of reliable and efficient execution of applications in grid environments.

REFERENCES

- [1] K. Ryu and J. Hollingsworth, "Resource Policing to Support Fine-Grain Cycle Stealing in Networks of Workstations," *IEEE Transactions on Parallel And Distributed Systems*, pp. 878–892, 2004.
- [2] 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.*, vol. 34, no. 3, pp. 375–408, 2002.
- [3] "<http://www.rcac.purdue.edu/boilergrid/>."
- [4] T. Z. Islam, S. Bagchi, and R. Eigenmann, "Falcon: A system for reliable checkpoint recovery in shared grid environments," in *SC '09: Proceedings of the Conference on High Performance Computing Networking, Storage and Analysis*. New York, NY, USA: ACM, 2009, pp. 1–12.
- [5] 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)*, 2006, pp. 19–23.
- [6] 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*, 2007, pp. 33–42.
- [7] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, "Collective operations in application-level fault-tolerant mpi," in *ICS '03*, 2003, pp. 234–243.
- [8] D. Thain, T. Tannenbaum, and M. Livny, "Distributed computing in practice: the Condor experience." *Concurrency - Practice and Experience*, vol. 17, no. 2-4, pp. 323–356, 2005.
- [9] J. Walters and V. Chaudhary, "A Comprehensive User-level Checkpointing Strategy for MPI Applications," TR 2007-1, The State University of New York, Buffalo, NY, Tech. Rep., 2007.
- [10] 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*, 2005, pp. 336–345.
- [11] de Camargo, R. Y., Cerqueira, Renato, and K. Fabio, "Strategies for storage of checkpointing data using non-dedicated repositories on grid systems," in *MGC*, 2005, pp. 1–6.
- [12] 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.
- [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] B. Rood and M. Lewis, "Scheduling on the Grid via multi-state resource availability prediction," in *Grid '08*, 2008, pp. 126–135.
- [15] B. Rood and M. J. Lewis, "Multi-state grid resource availability characterization," in *GRID '07*, 2007, pp. 42–49.
- [16] "<http://www.spec.org/cpu2006/>."