Can Checkpoint/Restart Mechanisms Benefit from Hierarchical Data Staging? *

Raghunath Rajachandrasekar, Xiangyong Ouyang, Xavier Besseron,
Vilobh Meshram, and Dhabaleswar K. Panda

Network-Based Computing Laboratory
Department of Computer Science and Engineering
The Ohio State University
{rajachan,ouyangx,besseron,meshram,panda}@cse.ohio-state.edu

Abstract. Given the ever-increasing size of supercomputers, fault resilience and the ability to tolerate faults have become more of a necessity than an option. Checkpoint-Restart protocols have been widely adopted as a practical solution to provide reliability. However, traditional checkpointing mechanisms suffer from heavy I/O bottleneck while dumping process snapshots to a shared filesystem. In this context, we study the benefits of data staging, using a proposed hierarchical and modular data staging framework which reduces the burden of checkpointing on client nodes without penalizing them in terms of performance. During a checkpointing operation in this framework, the compute nodes transmit their process snapshots to a set of dedicated staging I/O servers through a high-throughput RDMA-based data pipeline. Unlike the conventional checkpointing mechanisms that block an application until the checkpoint data has been written to a shared filesystem, we allow the application to resume its execution immediately after the snapshots have been pipelined to the staging I/O servers, while data is simultaneously being moved from these servers to a backend shared filesystem. This framework eases the bottleneck caused by simultaneous writes from multiple clients to the underlying storage subsystem. The staging framework considered in this study is able to reduce the time penalty an application pays to save a checkpoint by 8.3 times.

Keywords: checkpoint-restart, data staging, aggregation, RDMA

1 Introduction

Current High-End Computing (HEC) systems operate at petaflop or multi-petaflop level. As we move towards Exaflop systems, it is becoming clear that such systems will be comprised of millions of cores and components. Although

* This research is supported in part by U.S. Department of Energy grants #DE-FC02-06ER25749 and #DE-FC02-06ER25755; National Science Foundation grants #CCF-0621484, #CCF-0833169, #CCF-0916302, #OCI-0926691 and #CCF-0937842; grant from Wright Center for Innovation #WCI04-010-OSU-0;

each component has only a very small chance of failure, the combination of all components has a much higher chance of failure. The Mean Time Between Failures (MTBF) for typical HEC installations is currently estimated to be between eight hours and fifteen days [19, 7]. In order to continue computing past the MTBF of the system, fault-tolerance has become a necessity. The most common form of fault-tolerant solution on current generation system is checkpointing. An application or library periodically generates a checkpoint that encapsulates its state and saves it to a stable storage (usually a central parallel filesystem). Upon a failure, the application can be rolled back to the last checkpoint.

Checkpoint/Restart support is provided by most of the commonly used MPI stacks [8, 12, 6]. Checkpointing mechanisms are notoriously known for their heavy I/O overhead to simultaneously dump images of many parallel processes to a shared filesystem. Many studies have been carried out to tackle this I/O bottleneck [16, 5]. For example, SCR [15] proposes a multi-level checkpoint system that stores data to the local storage on compute node, and relies on redundant data copy to tolerate node failures. It requires a local disk or RAM disk to be present at each compute node to store checkpoint data. There are many disk-less clusters, and a memory-intensive application can effectively disable RAM disk by using up most of system memory. Hence its applicability is constrained.

With the rapid advances in technology, many clusters are being built with high performance commercial components such as high-speed low-latency networks and advanced storage devices such as Solid State Drives (SSDs). These advanced technologies provide an opportunity to redesign existing solutions to tackle the I/O challenges imposed by Checkpoint/Restart. In this paper, we propose a hierarchical data staging architecture to address the I/O bottleneck caused by Checkpoint/Restart. Specifically we want to answer several questions:

1. How to design a hierarchical data staging architecture that can relieve compute nodes from the relatively slow checkpoint writing, so that applications can quickly resume execution?
2. How to leverage high speed network and new storage media such as SSD to accelerate staging I/O performance?
3. How much of a performance penalty will the application have to pay to adopt such a strategy?

We have designed a hierarchical data staging architecture that uses a dedicated set of staging server nodes to offload checkpoint writing. Experimental results show that the checkpoint time, as it appears to the application, can be 8.3 times lesser compared to the basic approach for which each application process directly writes checkpoint to a shared Lustre filesystem.

The rest of the paper is organized as follows. In section 2, we give a background about the key components involved in our design. In Section 3, we propose our hierarchical staging design. In section 4, we present our experiments and evaluation. Related work is discussed in Section 5, and in section 6, we present the conclusion and future work.

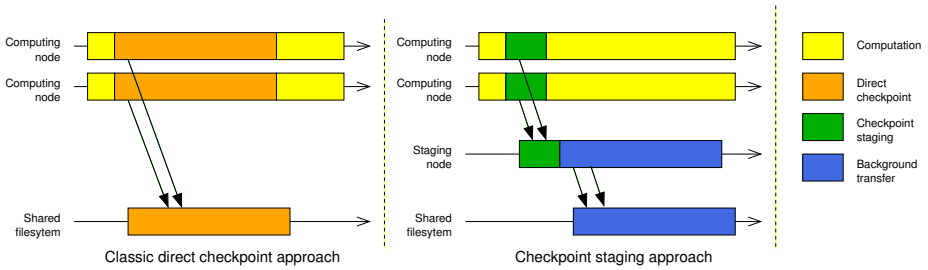


Fig. 1. Comparison between the direct checkpoint and the checkpoint staging approaches

2 Background

Filesystem in Userspace (FUSE). Filesystem in Userspace (FUSE) [1] is a software that allows the creation of a virtual filesystem in the user level. It relies on a kernel module to perform privileged operations at the kernel level, and provides a userspace library to communicate with this kernel module. FUSE is widely used to create filesystems that do not really store the data itself but relies on other resources to effectively store the data.

InfiniBand and RDMA. InfiniBand is an open standard of high speed interconnect, which provides send-receive semantics, and memory-based semantics called Remote Direct Memory Access (RDMA) [13]. RDMA operations allow a node to directly access a remote node's memory contents without using the CPU at the remote side. These operations are transparent at the remote end since they do not involve the remote CPU in the communication. InfiniBand empowers many of today's Top500 Super Computers [3].

3 Detailed Design

The central principle of our Hierarchical Data Staging Framework is to provide a fast and temporary storage area in order to absorb the I/O load burst induced by a checkpointing operation. This fast staging area is governed by, what we call, a *Staging server*. In addition to what a generic compute-node is configured with, staging servers are over-provisioned with high-throughput SSDs and high-bandwidth links. Given the fact that such hardware is expensive, this design avoids the need to install them on every compute-node.

Figure 1 shows a comparison between the classic direct-checkpointing and our checkpoint-staging approaches. On the left, with the classic approach, the checkpoint files are directly written on the shared filesystem. Due to the heavy I/O burden imposed on the shared filesystem by the checkpointing operation, the parallel writes get multiplexed, and the aggregate throughput is reduced. This increases the time for which the application blocks, waiting for the checkpointing operation to complete. On the right, with the staging approach, the

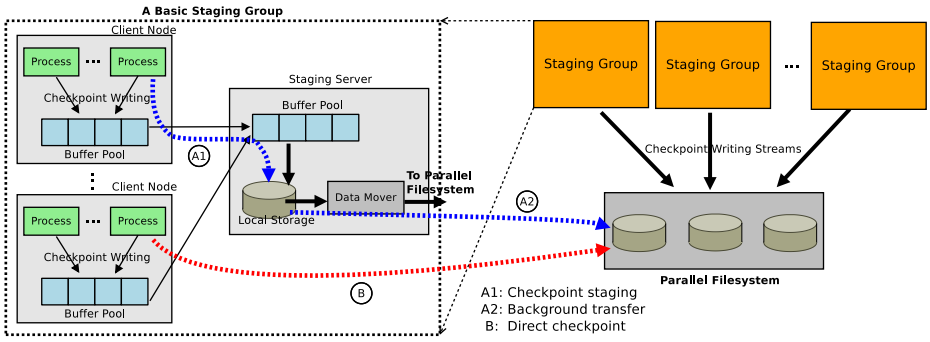


Fig. 2. Design of Hierarchical Data Staging Framework

staging nodes are able to quickly absorb the large amount of data thrust upon them by the client nodes, with the help of the scratch space provided by the staging servers. Once the checkpoint data has been written to the staging nodes, the application can resume. Then, the data transfer between the staging servers and the shared filesystem takes place in background and overlaps with the computation. Hence, this approach reduces the idling time of application due to the checkpoint. Regardless of which approach is chosen to write the checkpointing data, it eventually has to reach the same media.

We have designed and developed an efficient software subsystem which can handle large, concurrent snapshot writes from typical rollback recovery protocols and can leverage the fast storage services provided by the staging server. We use this software subsystem to study the benefits of hierarchical data staging in Checkpointing mechanisms.

Figure 2 shows a global overview of our Hierarchical Data Staging Framework which has been designed for use with these staging nodes. A group of clients, governed by a single staging server, represents a staging group. These staging groups are building blocks of the entire architecture. Our design imposes no restriction on the number of blocks that can be used in a system. The internal interactions between the compute nodes and a staging server are illustrated for one staging group in the figure.

With the proposed design, neither the application nor the MPI stack needs to be modified to utilize the staging service. We have developed a virtual filesystem based on FUSE [1] to provide this convenience. The applications that run on compute nodes can access this staging filesystem just like any other local filesystem. FUSE provides the ability to intercept standard filesystem calls such as open(), read(), write(), close() etc., and manipulate the data as needed at user-level, before forwarding the call and the data to the kernel. This ability is exploited to transparently send the data to the staging area, rather than writing to the local or shared filesystem.

One of the major concerns with checkpointing is the high degree of concurrency with which multiple client nodes write process snapshots to a shared stable storage subsystem. These concurrent write streams introduce severe contention at the Virtual Filesystem Switch (VFS) which impairs the total throughput. To

avoid this contention caused by small and medium-sized writes which is common in the case of checkpointing, we use the write-aggregation method proposed and studied in [17]. It allows to coalesce the write requests from the application/checkpointing library, and group them into fewer large-sized writes, which in turn reduces the number of pages allocated to them from the page cache. After aggregating the data buffers, instead of writing them to the local disk, the buffers are en-queued in a work-queue which is serviced by a separate thread that handles the network transfers.

The primary goal of this staging framework is to let the application which is being checkpointed proceed with its computation as early as possible, without penalizing it for the shortcomings of the underlying storage system. The InfiniBand network fabric has RDMA capability which allows for direct reads/writes to/from host memory without involving the host processor. This capability has been exploited to directly read the data that is aggregated in the client's memory, which then gets transferred to the staging node which governs it. The staging node writes the data to a high-throughput node-local SSD while it receives chunks of data from the client node (step A1 in Fig. 2). Once the data has been persisted in these Staging servers, the application can be certain that the checkpoint has been safely stored, and can proceed with its computation phase. The data from the SSDs on individual servers are then moved to a stable distributed filesystem in a lazy manner (step A2 in Fig. 2).

Concerning the reliability of our staging approach, we have to notice that, after a checkpoint, all the checkpoint files are eventually stored in the same shared filesystem as in the direct-checkpointing approach. So the both approaches provide the same reliability regarding the saved data. However, with the staging approach, the checkpointing operation is faster. This reduces the odds of losing the checkpoint data due to a compute node failure. During a checkpoint, the staging servers introduce additional points of failure. To counter effects of such a failure, we ensure that the previous set of checkpoint files are not deleted before all the new ones are safely transferred to the shared filesystem.

4 Experimental Evaluation

4.1 Experimental Testbed

A 64-node InfiniBand Linux cluster was used for the experiments. Each client node has eight processor cores on two Intel Xeon 2.33 GHz Quad-core CPUs. Each node has 6 GB main memory and a 250 GB ext3 disk drive. The nodes are connected with Mellanox MT25208 DDR InfiniBand HCAs for low-latency communication. The nodes are also connected with a 1 GigE network for interactive logging and maintenance purposes. Each node runs Linux 2.6.30 with FUSE library 2.8.5.

The primary shared storage partition is backed by Lustre. Lustre 1.8.3 is configured using 1 MetaData Server (MDS) and 1 Object Storage Server (OSS), and is set to use InfiniBand transport. The OSS uses a 12-disk RAID-0 configuration which can provide a 300 MB/s write throughput.

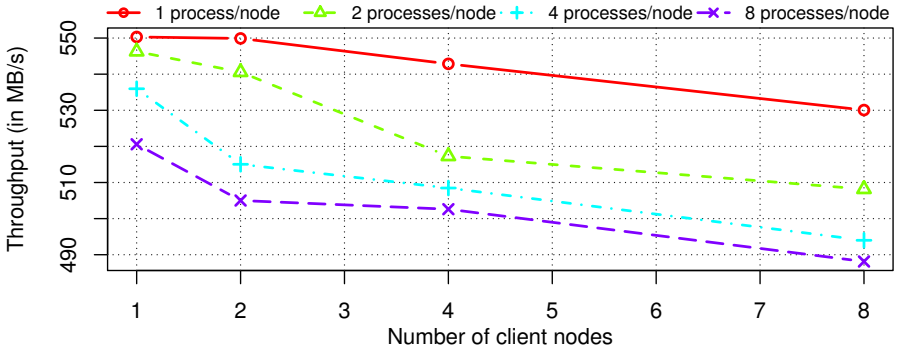


Fig. 3. Throughput of a single staging server with varying number of clients and processes per client (Higher is better)

The cluster also has 8 storage nodes, 4 of which have been configured to be used as the "staging nodes"(as described in Fig. 2) for these experiments. Each of these 4 nodes have PCI-Express based SSD cards with 80 GB capacity, two of them being Fusion-io ioXtreme cards (350 MB/s write throughput) and two others being Fusion-io ioDrive cards (600 MB/s write throughput).

4.2 Profiling of a Stand-Alone Staging Server

The purpose of this experiment is to study the performance of a single staging node with varying number of clients. The I/O throughput was computed using the standard IOzone benchmark [2]. Each client writes a file of size 1 GB using 1 MB records. Figure 3 reports the results of this experiment.

We see maximal throughput of 550 MB/s when a single client with 1 process writes data. This throughput matches the write throughput of the SSD used as the staging area (*i.e.* 600 MB/s). This indicates that transferring the files over the InfiniBand network does not prove to be a bottleneck. As the number of processes per client node (and the total number of processes in turn) increases, there is contention at the SSD which slightly decreases the throughput. For 8 processes per node and 8 client nodes, *i.e.* 64 client processes, the throughput is 488 MB/s, which represents only a 11% decline.

4.3 Scalability Analysis

In this section, we study the scalability of the whole architecture from the application's perspective. In these experiments, we choose to associate 8 compute nodes with a given staging server.

We measure the total throughput using the IOzone benchmark for 1 and 8 processes per nodes. Each process writes a total of 1 GB of data using 1 MB record size. The results are compared to the classic approach where all processes directly write to the Lustre shared filesystem.

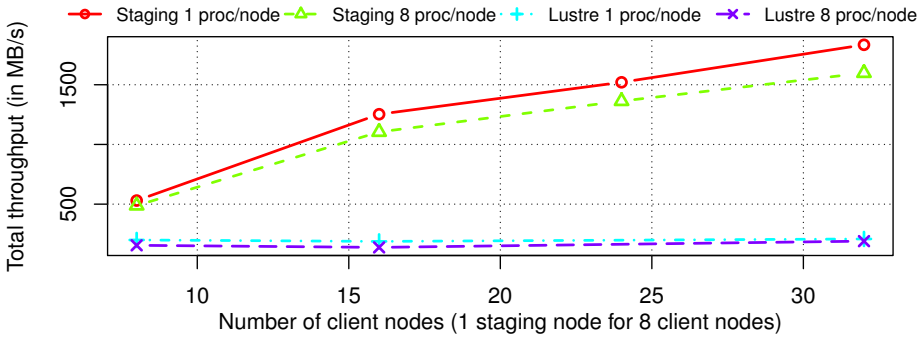


Fig. 4. Throughput scalability analysis, with increasing number of Staging groups and 8 clients per group (Higher is better)

Figure 4 shows that the proposed architecture scales even as we increase the number of groups. It is expected because it is designed in such a way that the I/O resources are added proportionally to the number of computing resources. Conversely, the Lustre configuration does not offer such a possibility, so the Lustre throughput stays constant. The maximal aggregated throughput observed for all the staging nodes is 1,834 MB/s, which is close to the sum of write throughput of the SSDs from these nodes (1,900 MB/s).

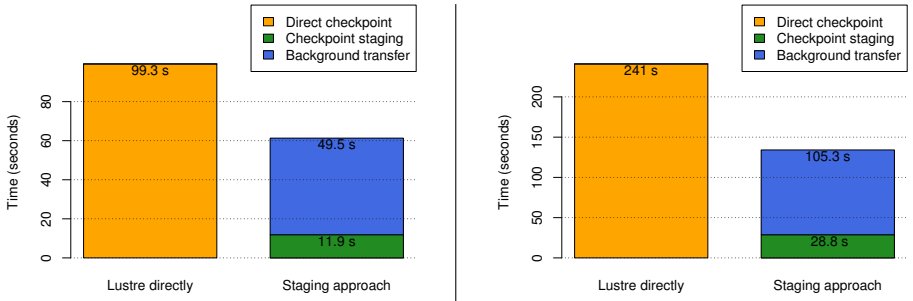
4.4 Evaluation with Applications

As explained in Figure 1, the purpose of the staging operation is to allow the application to resume its execution faster after a checkpoint. In the experiment, we measure the time required to perform a checkpoint from the application perspective, i.e. the time during which the computation is suspended because of the checkpoint. We compared this staging approach with the classic way in which the application processes directly write their checkpoints to the parallel Lustre filesystem. As a complement, we also measure the time required by the staging node to move the checkpointed data to Lustre in background once the checkpoints have been staged and the computation has resumed.

The next experiment used two applications (LU and BT) from the NAS Parallel Benchmarks. The class D input has a large memory footprint, and hence, big checkpoints files. These applications were run on 32 nodes with MVAPICH2 [8] and were checkpointed using the integrated Checkpoint/Restart support based on BLCR [10]. Table 1 shows the checkpoint size of these applications for the considered test cases.

	Average size per process	Total size
LU.D.128	109.3 MB	13.7 GB
BT.D.144	212.1 MB	29.8 GB

Table 1. Size of the checkpoint files



(a) LU.D.128 (b) BT.D.144

Fig. 5. Comparison of the checkpoint times between the proposed staging approach and using the classic approach (Lower is Better)

Figure 5 reports the checkpointing time that we measured for the considered application. For the proposed approach, two values are distinctly shown: the checkpoint staging time (step A1 in Figure 2) and the background transfer time (step A2 in Figure 2). The staging time is the checkpointing time as seen by the application, i.e. the time during which the computation is suspended. The background transfer time is the time to transfer the checkpoint files from the staging area to the Lustre filesystem, which takes place in parallel to the application execution once the computation resumes.

For the classic approach, the checkpoint is directly written to the Lustre filesystem, so we show only the checkpoint time (step B in figure 2). The application is blocked on the checkpointing operation for the entire duration shown.

The direct checkpoint time and the background transfer time both write the same amount of data to the same Lustre filesystem. The huge difference (twice faster or more) between these data transfer times is because, thanks to our hierarchical architecture, the contention on the shared filesystem is reduced. With the direct-checkpointing approach, 128 or 144 processes write their checkpoint simultaneously to the shared filesystem. With our staging approach, only 4 staging servers write simultaneously to the shared filesystem.

It is interesting to compare only the direct checkpoint time to the checkpoint staging time because they correspond to the time which is seen by the application (for classic approach and staging approach, respectively). Indeed, the background transfer is overlapped by the computation.

Our results show the benefit of using the staging approach which considerably reduces the time during which the application is suspended. For both our test cases, the checkpoint time, as seen by the application, appears to be 8.3 times faster. Then, the time gained can be used to make progress in the computation.

5 Related Work

Checkpoint/Restart is supported by several MPI stacks [8, 12, 6] to achieve fault tolerance. Many of these stacks use FTB [9] as a back-plane to propagate fault

information in a consistent manner. However, Checkpoint is well known for its heavy I/O overhead to dump process images to stable storage [18].

A lot of efforts have been conducted to tackle this I/O bottleneck. PLFS [5] is a parallel log-structured filesystem proposed to improve the checkpoint writing throughput. This solution only deals with N-1 scenario where multiple processes write to the same shared file, hence it cannot handle MPI system-level checkpoint where each process is checkpointed to a separate image file.

SCR [15] is a multi-level checkpoint system that stores data to the local storage on compute nodes to improve the aggregated write throughput. SCR stores redundant data on neighbor nodes to tolerate failures of a small portion of the system, and it periodically copies locally cached data to parallel filesystem to tolerate cluster-wide catastrophic failures. Our approach differs from SCR where a compute node stages its checkpoint data to its associated staging server, such that the compute node can quickly resume execution while the staging server asynchronously moves checkpoint data to a parallel filesystem.

OpenMPI [11] proposes a feature to store process images to node-local filesystem, and later copies these files to a parallel filesystem. Dumping a memory-intensive job to a local filesystem is usually bounded by the local disk speed, and it is hard to work on disk-less clusters where RAM disk is not feasible due to the high application memory footprint. Our approach aggregates node-local checkpoint data and stages it to a dedicated staging server, which takes advantages of high bandwidth network and advanced storage media such as SSD to achieve good throughput.

Isaila et al. [14] designed a two-level staging hierarchy to hide file access latency from applications. Their design is coupled with Blue Gene's architecture where dedicated I/O nodes service a group of compute nodes, and not all clusters have such a hierarchical structure.

DataStager [4] is generic service for I/O staging which is also based on InfiniBand RDMA. However, our work is specialized for the Checkpoint/Restart. Thus, we can optimize the I/O scheduling for this scheme. For example, we give the priority to the data movement from the application to the staging nodes to shorten the checkpoint time from the application perspective.

6 Conclusion and Future Work

As a part of this work, we explored several design alternatives to develop a hierarchical data staging framework to alleviate the bottleneck caused by heavy I/O contention at the shared storage when multiple processes in an application dump their respective checkpointed data. Using the proposed framework, we have studied the scalability and throughput of hierarchical data staging and the merits it offers when it comes to handling large amounts of Checkpoint data. We have evaluated the Checkpointing times of different applications, and have noted that they are able to resume their computation up to 8.3 times faster than what they would normally, in the absence of data staging. This clearly indicates that Checkpoint/Restart mechanisms can indeed benefit from hierarchical data staging. As part of the future work, we would like to extend this framework to

offload several other Fault-Tolerance protocols to the Staging server and relieve the client of additional overhead.

References

1. Filesystem in userspace. `http://fuse.sourceforge.net`
2. IOzone filesystem benchmark. `http://www.iozone.org`
3. Top 500 supercomputers. `http://www.top500.org`
4. Abbasi, H., Wolf, M., Eisenhauer, G., Klasky, S., Schwan, K., Zheng, F.: Datastager: Scalable data staging services for petascale applications. In: HPDC (2009)
5. Bent, J., Gibson, G., Grider, G., McClelland, B., Nowoczynski, P., Nunez, J., Polte, M., Wingate, M.: PLFS: a checkpoint filesystem for parallel applications. In: SC (2009)
6. Buntinas, D., Coti, C., Herault, T., Lemarinier, P., Pilard, L., Rezmerita, A., Rodriguez, E., Cappello, F.: Blocking vs. non-blocking coordinated checkpointing for large-scale fault tolerant mpi protocols. Future Generation Computer Systems (2008)
7. Cappello, F., Geist, A., Gropp, B., Kale, L., Kramer, B., Snir, M.: Toward exascale resilience. IJHPCA (2009)
8. Gao, Q., Yu, W., Huang, W., Panda, D.K.: Application-transparent checkpoint/restart for mpi programs over infiniband. In: ICPP (2006)
9. Gupta, R., Beckman, P., Park, B.H., Lusk, E., Hargrove, P., Geist, A., Panda, D.K., Lumsdaine, A., Dongarra, J.: Cifts: A coordinated infrastructure for fault-tolerant systems. ICPP (2009)
10. Hargrove, P.H., Duell, J.C.: Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters. In: SciDAC (2006)
11. Hursey, J., Lumsdaine, A.: A composable runtime recovery policy framework supporting resilient hpc applications. Tech. rep., University of Tennessee (2010)
12. Hursey, J., Squyres, J.M., Mattox, T.I., Lumsdaine, A.: The design and implementation of checkpoint/restart process fault tolerance for Open MPI. In: IPDPS (2007)
13. InfiniBand Trade Association: The InfiniBand Architecture, `http://www.infinibandta.org`
14. Isaila, F., Garcia Blas, J., Carretero, J., Latham, R., Ross, R.: Design and evaluation of multiple-level data staging for blue gene systems. TPDS (2011)
15. Moody, A., Bronevetsky, G., Mohror, K., Supinski, B.R.d.: Design, modeling, and evaluation of a scalable multi-level checkpointing system. In: SC (2010)
16. Ouyang, X., Gopalakrishnan, K., Gangadharappa, T., Panda, D.K.: Fast Checkpointing by Write Aggregation with Dynamic Buffer and Interleaving on Multicore Architecture. HiPC (2009)
17. Ouyang, X., Rajachandrasekhar, R., Besseron, X., Wang, H., Huang, J., Panda, D.K.: CRFS: A lightweight user-level filesystem for generic checkpoint/restart. In: ICPP (2011), to appear
18. Plank, J.S., Chen, Y., Li, K., Beck, M., Kingsley, G.: Memory exclusion: Optimizing the performance of checkpointing systems. In: Software: Practice and Experience (1999)
19. Schroeder, B., Gibson, G.A.: Understanding failures in petascale computers. Journal of Physics: Conference Series (2007)