Using Cloud Constructs and Predictive Analysis to Enable Pre-Failure Process Migration in HPC Systems

J. Brandt*, F. Chen*, V. De Sapio*, A. Gentile°,
J. Mayo*, P. Pébay*, D. Roe°, D. Thompson*, and M. Wong°
Sandia National Laboratories
MS *9159 / °9152 P.O. Box 969, Livermore, CA 94551 U.S.A.
{brandt,fxchen,vdesap,gentile,jmayo,pppebay,dcroe,dcthomp,mhwong},
ovis@sandia.gov

***Abstract*—Accurate failure prediction in conjunction with efficient process migration facilities including some Cloud constructs can enable failure avoidance in large-scale high performance computing (HPC) platforms. In this work we demonstrate a prototype system that incorporates our probabilistic failure prediction system with virtualization mechanisms and techniques to provide a whole system approach to failure avoidance. This work utilizes a failure scenario based on a real-world HPC case study.**

***Keywords*-cloud computing; virtualization; migration; failure prediction; fault tolerance, failure avoidance; HPC;**

I. Introduction

Commercial cloud offerings rely heavily on virtualization technologies and redundancy to provide reliable customized compute environments tailored to a specific customer needs (e.g., [1]). These environments are well suited to software development work, web hosting, and even some embarrassingly parallel types of applications that have traditionally been run on high performance compute (HPC) platforms. The current interconnects and individual resource reliability in these environments, however, don't lend themselves to the kind of tightly coupled MPI applications that typify today's large scale scientific applications being run on HPC platforms which are specifically designed for these application's requirements (e.g. high reliability, fast processors, high bandwidth low latency interconnects, etc.).

High performance compute platforms, though they are tailored to the requirements of large scale scientific applications, are becoming increasingly less efficient due to stable failure rates of the large number of components. The problem is that no matter how reliable an individual component may be, as long as it is less than 100%, the reliability of a pool of such resources taken as an aggregate decreases as the number in the pool becomes large. This decrease in reliability ultimately implies a required increase in checkpoint frequency and corresponding decrease in efficiency as time spent in checkpointing doesn't contribute to the solution.

It is expected that further scaling of platforms (beyond peta-scale) will require mechanisms to deal with the projected short mean time to failures over such large pools of resources [2]. While there are efforts under way to design more fault tolerant programming methods [3] there is significant investment in current MPI based applications which will still be required to run at significant scale on platforms for the foreseeable future and hence be affected by these issues. To date most application related fault tolerance work has been in the form of more efficient checkpoint/restart schemes which has in some cases significantly increased application throughput [4].

The work presented in this paper is based on the premise that if failures can be reliably predicted with sufficient lead time, wasted time spent in speculative checkpointing could be dispensed with or largely reduced. Instead, affected resources could be replaced with good ones where possible and an application would checkpoint only when explicitly flagged to do so by the predicting system or speculatively based on a much longer mean time to interrupt driven by such things as human error and acts of nature. In this work we explore the use of some Cloud constructs (Infrastructure as a Service (IaaS) and virtualization) in conjunction with resource failure prediction to facilitate migration of traditional MPI based processes from failing to healthy resources without checkpoint/restart (though this mechanism is not precluded).

The reasons for utilizing virtualization are twofold: 1) the virtual machine provides a nice process state container that can be appropriately placed and dynamically migrated within the pool of real resources transparently to the application. 2) The computational infrastructure can be configured and maintained independent of the underlying platform configuration. While there is still overhead associated with the use of virtualized resources, it has been greatly reduced through hardware support and there is promise of further reduction in the future. With sufficiently reliable failure prediction mechanisms driving migration, the overhead of virtualization has only to be less than that of the checkpoint/restart mechanisms for this to be viable as a failure avoidance mechanism.

In this work we demonstrate use of a prototype system for

migrating running MPI processes from a resource (compute node in this case) for which failure is predicted to a known good resource. We include methodologies, constraints, and scaling concerns. Our failure scenario is based on previous work in which we identified precursor behavior for one of our production system's main failure modes (out of memory) [5]. In particular we utilize automated detection of such behavior to trigger the migration, in a way that is transparent to the application, of all MPI processes off a resource identified as likely to fail.

This paper is divided as follows: This section gave an introduction, background, and motivation for this work, Section II discusses our approaches and methodologies, Section III presents results using our prototype system applied to a failure scenario based on a real-world HPC cluster failure case study, Section IV discusses related work, and Section V summarizes.

II. Approaches and Methodologies

There has been work, as described in Section IV, on many aspects of resilience (e.g. component failure prediction, resource pool mean time to failure prediction, migration based on failure prediction both with and without virtualization, etc.). Our work in this area takes a systems approach which integrates failure prediction, both physical and virtual resource management, MPI, and a coordination system. Using our prototype system we demonstrate allocation of physical resources, deployment of virtual computational resources to meet an MPI application's needs, detection of pre-failure conditions, and migration of virtual resources to new physical resources in response. The migration is instigated and occurs transparently to the running application. To build this integrated capability we wrote a skeleton Resource Manager which carries knowledge of both physical and virtual resources and their relationships to running applications. We also constructed a Controller to interact with the Failure Prediction system, the Resource Manager, and the applications in order to orchestrate migration of virtual resources based on failure prediction. Additionally we wrote an MPI_Barrier wrapper to interact with the Controller in order to initiate migration at an appropriate time for an MPI application so as not to lose in-flight messages.

A. Prototype System

Our prototype system in Figure 1 is comprised of 1) a testbed of compute nodes identical to those used in our production systems, 2) a skeleton Resource Manager which manages both physical and virtual resources, 3) our OVIS data collection, analysis, visualization, and response tool [6], [7], and 4) a Controller responsible for orchestrating the migration of virtual machines (VMs) and their processes from one location to another on the basis of input from the OVIS monitoring system. In the next subsections we discuss in detail each of these blocks' functionality and known limitations.

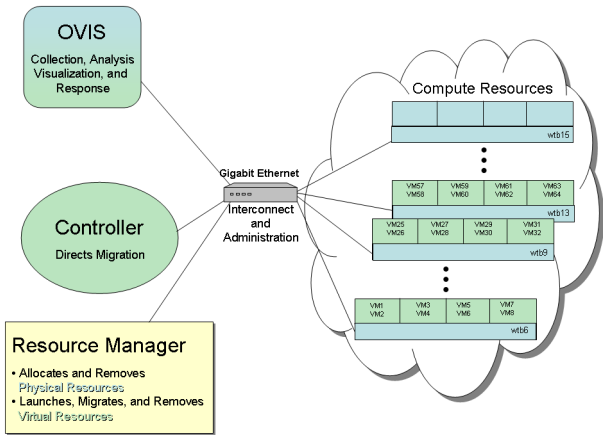

Figure 1. Components in a system to enable pre-failure process migration.

B. Base Testbed Components

Our testbed consists of ten diskless quad CPU 2.2Ghz AMD Istanbul (6 core) nodes. The interconnects are 4x DDR Infiniband (IB) and Gigabit Ethernet. In this prototype deployment we utilize the Gigabit Ethernet interconnect as we don't currently have the virtio drivers working for the IB network. This would be a limitation with respect to the performance of the MPI code but since this particular work is focused on a system to facilitate failure avoidance through migration and not performance (which we will address in the future) it is immaterial. On all hosts we run a Redhat linux 2.6.27.21 distribution.

With respect to the virtualized environment we are using Kernel Virtual Machines (KVM) [8] KVM-86, qemu-0.10.5, and libvirt-0.7.0. We boot the KVMs using a OneSis image (currently same as the host image) that resides on an NFS server. When launching or migrating the KVMs, numactl [9] is used to bind the processes associated with a KVM to a specific core and the memory region associated with that core's CPU for both performance and stability reasons. We have written a custom piece of code for launching, live migrating, and removing KVMs that also takes care of virtual network interface setup and teardown on the hosts as well as maintaining current state on all KVMs.

C. Software Components

The Resource Manager maintains a "free" resource pool, a mapping among {job, virtual resource allocation, and physical resources hosting the virtual resources}, and a "defunct" resource pool. It also performs initial launch of VMs onto physical resources on behalf of an application, starts an mpd ring [10] on the VMs, and launches an MPI application on the VMs. Details of management of the virtual resources are explicitly handled by the Virtual Resource

Manager which is a part of the Resource Manager. The Virtual Resource Manager, using the custom code mentioned above, performs the per-KVM launch, migrate, and tear-down on the specified resources, sets up and tears down the network environment, and maintains a state directory for each VM. The Virtual Resource Manager is used by the Resource Manager to launch or tear down a pool of KVMs and by the Controller to perform live KVM migration.

OVIS, originally built as a research tool for large scale computer system data collection and analysis, is able to scalably collect metric data, such as memory statistics, voltages, and resource utilizations and perform various types of analyses on this data in order to develop characteristic probabilistic models. Real-time data can then be compared against applicable models to drive inference about component health and failure prediction. When the OVIS analyses indicate impending failure for a particular component or aggregation of components, OVIS sends a notification message to the Controller. Additionally OVIS is informed (currently a manual operation) of the change in the resource pool membership being monitored on behalf of the application as soon as the migration to a new resource has been completed.

The function of the Controller is to coordinate migration of processes among physical resources (in this case from failing resources to a healthy ones). Once the Controller has been notified by OVIS of an impending resource failure it initiates migration of the affected MPI processes. In order to preclude loss of packets in flight this is done in several steps as described in Section II-D.

D. Migration Process

1) The Controller writes flag files on a per job id basis to a known location in the Controller's file system that will inform MPI processes (when checked at a barrier), which, if any, processes in a given job will need to be migrated.
2) The Controller contacts the Resource Manager to acquire new physical resources to host the affected MPI processes.
3) An MPI_Barrier wrapper directs each rank, upon passing a barrier for the nth time (n settable by the user and set to 1 for the purpose of this work), to check for notification of the need for a migration of any associated MPI process. The flag files written by the Controller are read by Rank0 at the barrier and shared with all processes in MPI_COMM_WORLD. For a given process, if no migration of any process in its job group is required, the process continues to its barrier. If migration of a process in a job group is required but the affected processes *do not* include the reading rank, then the process continues to its barrier. If migration of a process in a job group is required and the affected processes *do* include the reading rank, then the process notifies the Controller that the KVM hosting that process is ready to be migrated (this is a blocking call). Upon return the reading rank continues to its barrier.
4) When a Controller receives notification that a process is ready to be migrated, the Controller initiates migration of the KVM hosting the process that has notified it.
5) Upon successful migration of *all* affected KVMs in a job group, the Controller returns notification of completion to each affected MPI process (at which time the affected process continues to its barrier).
6) Upon successful migration of all KVMs from the affected resource(s) the Controller notifies the Resource Manager to remove the resource(s) from the "free" pool for inspection and repair.

E. Known Limitations and Issues

Scalability: In this deployment every MPI_Barrier call results in the rank0 process setting a semaphore, moving a file, releasing a semaphore and making two MPI_Bcast calls in order for each rank to determine if a migration is required. This can be made lower impact by increasing the ratio of barriers to checks (application dependent) as well as by having the Controller push the flag information to the appropriate KVM. Rank0's location would have to be sent to the Controller perhaps by an MPI_Init wrapper. Using semaphores around flag writes and reads generates extra overhead but is required in our prototype system to guarantee data corruption doesn't occur due to race conditions which we experienced before implementing the locks. In our prototype in-flight barrier messages could still occur during migration on large scale deployments due to propagation delays and barrier messages getting misrouted during the final phase of migration.

Other: The KVM images used in this deployment take 90 seconds to boot. Because we build the virtual infrastructure for an application at the time it is to be launched, each application must wait an initial 90 seconds, independent of the number of resources required or the time it will be run, to start. For small, short lived applications this is too much overhead and could be mitigated by maintaining a pool of VM's to be allocated as real resources currently are.

III. Proof of Concept Demonstration

To demonstrate the utility of our prototype system, we emulate, on our testbed, a precursor symptom to a major failure mode which occurs on one of our production systems. The failure scenario [5] is summarized below and is used here for illustrative purposes as it represents a real-world problem that can be detected and mitigated using the functionality of our prototype system.

In this scenario, active memory usage on a compute node becomes abnormally high, independent of memory being used by an application, resulting in less available memory

for future processes that are allocated this resource. When the memory usage on the node finally exceeds a threshold, the linux OOM Killer is invoked which may kill a user's application or system process(es) and cause the premature termination of a user's application. This can occur even when the application's memory usage is only a small fraction of the total memory on a host compute node.

In our emulation, we ran a background process that tied up half of the memory on one of the eight compute nodes used to host a 64 process MPI application. We choose not to detect this condition during idle time, as suggested in [5], as the threat it poses is application dependent. Within our system, OVIS continuously monitors memory utilization on the compute nodes in order to detect a plausible threat of application failure. We used a two factor analysis in OVIS to discriminate between this failure scenario and one in which a (well-balanced) application naturally requires a lot of memory: 1) Any single node with less than 75% Active memory utilization is not further evaluated for potential failure. This number was arbitrarily chosen for demonstration purposes; in an actual system such a threshold would most likely be determined empirically. 2) Nodes exceeding the threshold are further examined to determine if they are anomalous with respect to other compute nodes running the same application. For illustrative purposes we call anomalous anything exceeding a probabilistic threshold of two standard deviations above the mean for the group of nodes participating in the application run. This threshold is again a parameter that would in practice be set based on empirical measurements. Note that this assumes the application spans enough compute nodes that such a statistical evaluation is possible. This is a realistic assumption for many jobs on a reasonably large capacity HPC system.

Figure 2 is composed of the visual output of OVIS showing our testbed before, during, and after the migration process from top to bottom respectively. There are three types of components shown: nodes (the largest components in the rack), CPUs (4 per node), and cores (6 per CPU). Here only the nodes and cores are colored by a display attribute. For the nodes this attribute is Active memory scaled from 0GB at the red end to 32GB at the blue end (32GB is the total for a compute node). The cores are colored by CPU utilization with 0 again being red and full utilization being blue.

At the beginning of the scenario compute nodes cn[1-8] are being used to host 2 KVMs per CPU which each hosts one MPI process of a 64 process job. We use numactl [9] to tie each KVM to a particular core and the memory region associated with its core's CPU. This association can be seen in the figure where each CPU block shows 2 cores with relatively high utilization. The second node from the bottom (cn2) has the additional background process running that ties up memory and can be seen as a third active core in a CPU block in the top and bottom screen shots. This

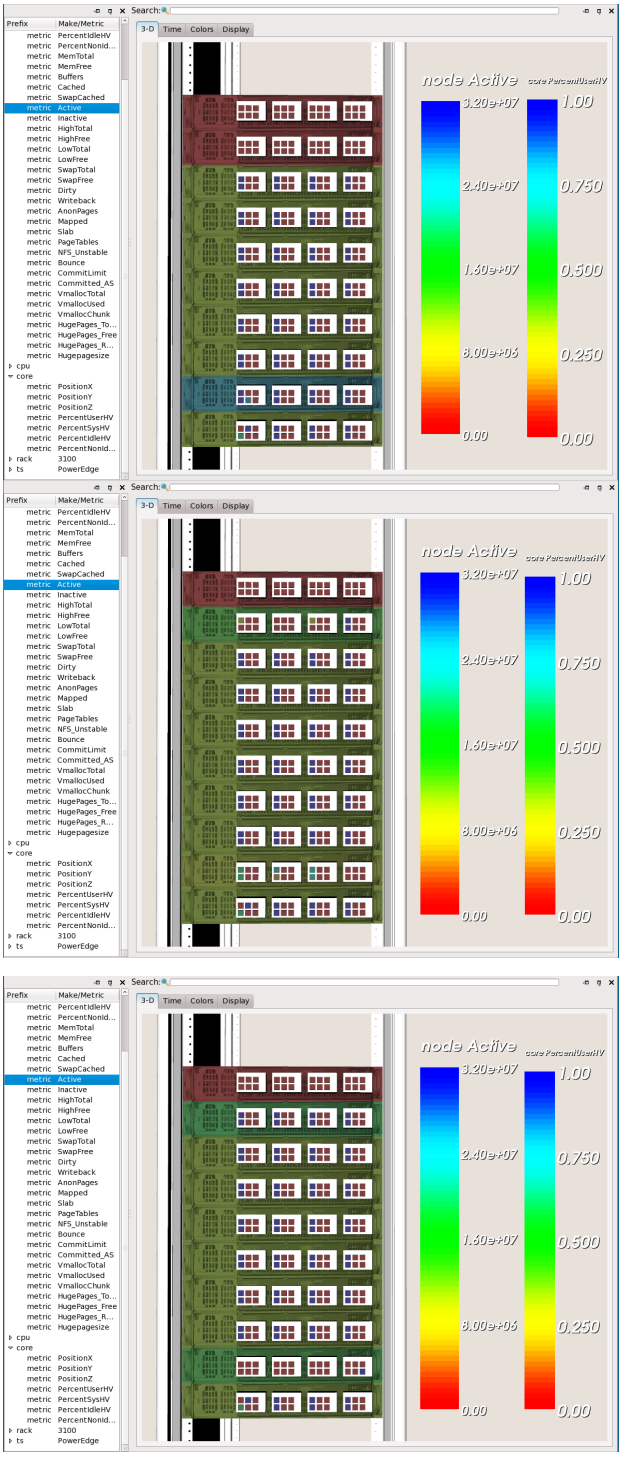

Figure 2. OVIS display showing Active Memory values on the nodes and CPU Utilization (scaled) on the cores. (t) OVIS discovery of abnormal memory utilization on a node (2nd from bottom) relative to the job group triggers OVIS to send a message to the Controller of impending node failure which in turn (m) instigates migration of the endangered resources to a new node (2nd from top). During live migration, both the original and the new cores are in use. Migration is complete (b).

process ran on the host and was not tied to a particular core. The blue color of cn2 (t) shows that it has passed the 75% threshold. It can be seen from the screen shots that cn2 is additionally using a good deal more memory than the other nodes in the job group. When OVIS determines that both of the aforementioned criteria have been met it informs the Controller which informs the affected processes, initiates new resource allocation and KVM setup and, when informed of readiness, coordinates migration on a per process granularity. The middle figure shows the system in transition with KVMs being migrated from cn2 to cn9. Finally all KVMs are successfully migrated to cn9 (b) and the affected MPI processes are allowed to drop through to their barriers at which time the barrier is complete across all MPI processes and the application continues. Note that the Active memory of cn9 is higher than for the others. This is due to how the live migration was carried out and that the migrated KVMs now occupy their maximum memory footprint. Though this could, under the right circumstances, trigger subsequent migrations without further checks, these are simple checks to implement and are not addressed here.

IV. Related Work

In this section we discuss both work related to specific components of our prototype system and other system approaches.

A. Failure Prediction

While there have been many papers about how reliable failure predictors could enhance the resilience of large scale HPC platforms, most of this work has been in the area of trying to model failure trends for use in checkpoint frequency calculations (e.g., [2], [11], [12]). Schroeder has done significant work on looking at disk failure trends and predictors [13] as well as those for DRAM [14]. There are also several bodies of work on using system log files (e.g., [15], [16]) to find predictors though these have been more successful in helping to identify the causes of failures than in discovering actionable precursor behaviors.

B. Checkpoint Restart Strategies

Though the work in this area may not be directly applicable to any of the system components discussed in this paper this is certainly complementary work and, until such time as accurate and effective prediction strategies are discovered, this will probably continue to be the main fault tolerance mechanism for some time. There are numerous methods (e.g. [4], [17]) for more efficient checkpointing using system memory to store the checkpoint state. Though this is fine for some applications, it requires the running application to have a substantially smaller footprint as checkpoint state can require significant memory resources. Oliner et. al. [18] take an interesting and related approach as they allow the system to work in cooperation with the application to take checkpoints at convenient places in the execution and dependent on the systems view of the likelihood of failure. This methodology would derive maximum benefit from good resource health metrics but in their absence degrades to checkpointing at places of convenience for the application. This has similarity to our system in that both provide mechanisms for cooperation between the system and application in taking action.

C. Virtualization Overhead and Migration Strategies in HPC Applications

There are a host of papers on this subject (e.g., [19], [20], [21]) as transparent targeted process migration is very appealing given the alternative of saving all state and performing application restarts. The overhead numbers seem to range from 60% [22] to actually achieving a speedup [23]. Consensus seems to be, however, that virtualization technology and the process mobility it provides is, or will be, viable for HPC applications in the near future. The component that is lacking in this work is an accurate and effective prediction component.

D. Complete System

Nagarajan et. al. [23] take a systems approach to this and even incorporate a "proactive fault tolerance daemon" which does health monitoring, load balancing, and decision making for virtual machine (VM) migration. What is missing in this work are viable health metrics beyond the known thresholds which suffer from the problem of typically having to be set so high that by the time they are crossed it is too late to react and if they are set low enough to allow reaction time there can be significant numbers of false positives which can result in unnecessary moves and associated overhead. The significant differences between this work and our own are our probabilistic approach to health monitoring and our system for notifying a MPI process of the desirability for migration but then allowing it to request a migration when it reaches an MPI_barrier. The latter requires relinking of the application code with our wrapper code. The Charm++ project [3] is a programming language approach that supports object migration for both load balancing and fault tolerance.

V. Summary

In this work, we have demonstrated a prototype system that utilizes some Cloud constructs (IaaS and virtualization) in conjunction with failure prediction to facilitate MPI-based application failure avoidance in a manner that is transparent to the application. Specifically, our system performs prediction-driven live migration of MPI processes running in virtual machines to healthy resources obtained from the resource manager. Our proof of concept work presented here used emulation of a real life failure scenario found in production systems in order to illustrate the utility of such a system for failure avoidance.

ACKNOWLEDGMENT

These authors were supported by the United States Department of Energy, Office of Defense Programs. Sandia is a multiprogram laboratory operated by Sandia Corporation, a Lockheed-Martin Company, for the United States Department of Energy under contract DE-AC04-94-AL85000.

REFERENCES

[1] "AMAZON WEB SERVICES," http://aws.amazon.com.

[2] Daly, "Performance challenges for extreme scale computing," http://www.pdsi-scidac.org/publications/, SDI/LCS Seminar Series, 2007.

[3] Zheng, Shi, and Kale, "FTC-CHARM++: An in-memory checkpoint-based fault tolerant runtime for CHARM++ and MPI," *IEEE Int'l. Conf. on Cluster Computing*, 2004.

[4] Oldfield, Arunagiri, Teller, Seelam, Varela, Riesen, and Roth, "Modeling the impact of checkpoints on next-generation systems," in *24th IEEE Conf. on Mass Storage Systems and Technologies*, 2007.

[5] Brandt, Gentile, Mayo, Pébay, Roe, Thompson, and Wong, "Methodologies for advance warning of compute cluster problems via statistical analysis: A case study," in *Proc. 18th ACM Int'l. Symp. on High Performance Distributed Computing (2009 Workshop on Resiliency in High-Performance Computing)*, 2009.

[6] "OVIS," http://ovis.ca.sandia.gov.

[7] Brandt, Debusschere, Gentile, Mayo, Pébay, Thompson, and Wong, "OVIS 2: A robust distributed architecture for scalable RAS," in *Proc. 22nd IEEE Int'l. Parallel & Distributed Processing Symp.(4th Workshop on System Management Techniques, Processes, and Services)*, 2008.

[8] "KVM," http://www.linux-kvm.org.

[9] "NUMACTL," see, for example, man page for numactl.

[10] "MPD RING," https://svn.mcs.anl.gov/repos/mpi/mpich2/tags/release/mpich2-1.2.1/README.vin.

[11] Gottumukkala, Liu, Leangsuksun, Nassar, and Scott, "Reliability analysis in HPC clusters," in *Proc. High Availability and Performance Computing Workshop*, 2006.

[12] Schroeder and Gibson, "Understanding failures in petascale computers," in *J. Phys.: Conf. Ser. 78 012022*. IOP Publishing, 2007.

[13] ——, "Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you?" in *Proc. 5th USENIX Conf. on File and Storage Technologies*, 2007.

[14] Schroeder, Pinheiro, and Weber, "DRAM errors in the wild: a large-scale field study," in *Proc. 11th Int'l. Joint Conf. on Measurement and Modeling of Computer Systems*, 2009.

[15] Sahoo, Oliner, Rish, Gupta, Moreira, Ma, Vilalta, and Sivasubramaniam, "Critical event prediction for proactive management in large-scale computer clusters," in *Proc. 9th ACM SIGKDD Int'l. Conf. on Knowledge Discovery and Data Mining*, 2003.

[16] Stearley and Oliner, "Bad words: Finding faults in SPIRIT'S syslogs," in *Proc. 8th IEEE Symp. on Cluster Computing and the Grid (2008 Workshop on Resiliency in High-Performance Computing)*, 2008. [Online]. Available: http://xcr.cenit.latech.edu/resilience2008

[17] Engelmann and Geist, "A diskless checkpointing algorithm for super-scale architectures applied to the fast fourier transform," in *Proc. 1st Int'l. Workshop on Challenges of Large Applications in Distributed Environments*, 2003.

[18] Oliner, Rudolph, and Sahoo, "Cooperative checkpointing: a robust approach to large-scale systems reliability," in *Proc. 20th Int'l. Conf. on Supercomputing*, 2006.

[19] Tikotekar, Vallée, Naughton, Ong, Engelmann, Scott, and Filippi, "Effects of virtualization on a scientific application running a hyperspectral radiative transfer code on virtual machines," in *Proc. 2nd Workshop on System-level Virtualization for High Performance Computing*, 2008.

[20] Wang, Mueller, Engelmann, and Scott, "A job pause service under LAM/MPI+BLCR for transparent fault tolerance," in *Proc. IEEE Int'l. Parallel and Distributed Processing Symposium*, 2007.

[21] ——, "Proactive process-level live migration in HPC environments," in *Proc. 2008 ACM/IEEE Conf. on Supercomputing*, 2008.

[22] Fenn, Murphy, and Goasguen, "A study of a KVM-based cluster for grid computing," in *Proc. 47th Annual Southeast Regional Conf.*, 2009.

[23] Nagarajan, Mueller, Engelmann, and Scott, "Proactive fault tolerance for HPC with XEN virtualization," in *Proc. ACM Int'l. Conf. on Supercomputing)*, 2007.