Virtualization Aware Job Schedulers for Checkpoint-Restart

R. Badrinath, R.Krishnakumar and R.K. Palanivel Rajan
Hewlett-Packard
{badrinath, krishnakumar.r, palanivelrajan.rk}@hp.com

Abstract

Application checkpoint and restart has been a widely studied problem over the last several decades. Despite immense volume of theory and several research project level implementations, there is very little by way of working solutions for the case of parallel distributed applications (such as MPI programs on a cluster). We describe our experiences in enhancing a job scheduler to leverage mechanisms of a virtual machine environment to support checkpoint-restart. We also describe the basic coordinated checkpoint-restart framework that we implemented on which this solution is based.

1. Introduction

The problem of checkpointing and restarting of message passing parallel applications (e.g. MPI applications) on a cluster has been the focus of much research over the years. This is particularly relevant in the HPC world where long running scientific applications are not uncommon. It would be a huge waste to have to restart the computation from the beginning just because one of the nodes of the cluster failed at some point of time during the application's running. Such failures are a reality in large clusters of servers where the possibility that any one node goes down increases with the size of the cluster and duration of the running of the application. The idea of checkpoint-restart (or CR for short) is to save one or more intermediate states in order to be able to restart from an intermediate state should there be a failure some time after that state has been saved. Typically, the restart would be from the latest such saved state. CR may also be used to implement preemptive job scheduling. Since the job scheduler provides access to cluster resources to the end user, any CR mechanism designed for usability should be integrated with the job scheduler. This paper describes our experience in enhancing a job scheduler to make it virtualization aware thereby leveraging mechanisms provided by the virtualization layer to provide an application transparent CR solution.

1.1 Background to CR

The reference [1] is an excellent introduction to the topic and survey of the work in the field of CR. We will briefly mention some aspects to provide context to this work. Our work falls under the class called *Backward Error Recovery*. The idea is that when a failure occurs, we move the state of some or all of the participating entities *back* to an earlier saved state. In our approach, called *coordinated checkpointing,* we take a snapshot of all the participating entities for each checkpoint. [1] discusses other approaches.

Despite there being rich study on the issues in CR for distributed parallel applications, in practice, however, very few successful solutions exist. Most solutions are not *application transparent*, i.e., they require changes to the user code or they require the user code to be linked with specific libraries. Solutions such as [16, 17] are single node, supporting at best multi-threaded applications; these also require some application or middleware modification, re-linking against a library, or are solutions based on modified kernels. The engineering complexity of application transparent solutions for MPI applications such as [6] has prevented its proliferation. An important reason for this state of affairs is that recovery has been difficult given the fact that there is dependency on data that is shared by the application with the OS. The solution described in [2] builds on the OS level virtualization solution described in [3]. That paper shows that virtualization can help resolve several of the issues traditionally considered difficult in practical CR. The solution in [2], though it does not require OS modifications, does use OS specific kernel modules, requiring reimplementation of the mechanism for different operating systems. In the serious HPC user community, authors of most long running application programs who wish to have the CR feature across platforms or on common operating systems such as Linux, Windows or standard Unixes build in code to save sufficient information regularly on persistent storage. This information is explicitly read back when the program is started again in a resume mode to

978-1-4244-1890-9/07/$25.00 ©2007 IEEE

continue computation from that point. Clearly the code for this is not always trivial and it is not the ideal solution for one focused on the application in hand. Hence, there is a need for an *application transparent* CR solution that works for cluster applications. This paper proposes such a solution that uses physical machine virtualization. This implies that our work in no way limits itself to a particular programming paradigm such as MPI or a specific operating system.

1.2 Virtualization and CR

It has been predicted [4] that virtualization of the physical hardware will help make CR a practical reality. In this paradigm instead of checkpointing and restarting application processes, one just checkpoints and restarts the whole virtual machine including the OS. Thus all issues of state which is shared between the OS and the application, become non-issues.

There are several reasons why we expect virtualization to play an increasing role in HPC. The ability of virtualized environments to enable operating systems specialized for particular workloads (called library OSes), having full control over the hardware resources, is beneficial to the HPC users[13]. It has also been projected that virtualization can enable scale testing with fewer resources. Virtualization provides us a means of guaranteeing the availability of computing resources, with a finer granularity of control, such as fraction of CPU cycles. Virtualization also provides better introspection and correctness verification capabilities [13] that are relevant to HPC users who require high levels of security.

A common perception is that bringing virtualization to HPC systems will hamper performance. As typical HPC applications do not call privileged operations frequently, the CPU virtualization overhead does not pose significant overhead for the HPC domain [18]. Additionally I/O virtualization overhead can be reduced by employing techniques such as VMM-bypass I/O [7]. In benchmarking studies[14,15] on virtualization and HPC, the authors employ various micro benchmarks from HPC challenge and LLNL ASCI Purple suites to evaluate several performance characteristics. With only a single exception, the authors report that Xen para-virtualization system poses no statistically significant overhead over other OS configurations currently in use at the LLNL for HPC clusters. With commodity platform support for virtualization (stemming from technologies such as VT-x and Pacifica) one expects that HPC will benefit from the advantages without significantly compromising performance.

We have used the Xen [20] para-virtualization environment for our implementation. Para-virtualization offers good performance and incurs low overhead. We use the term *Guest OS* for the OS running in a virtual machine on top of the virtualizing layer, which we call *Host OS*.

1.3 Our Contribution

The central idea of this paper is to engineer the job scheduler to make it virtualization aware. The job scheduler is thus capable of performing basic control operations on virtual machines. We use this enhanced job scheduler to implement a solution to the checkpoint-restart problem for parallel applications on a cluster. The solution is application and OS transparent. Aspects of disk checkpointing are deferred to a later work. Similarly, while we focus on the design of the solution, enhancing the performance of the method is out of the scope of this paper.

For our implementation we enhance an open source job scheduler, SLURM[9], to support application transparent CR.

In section 2, we explain the architecture design and implementation of a coordinated CR mechanism for clusters that leverages the virtual machine infrastructure. In section 3, we describe the architecture design and implementation of the enhancements we did to a job scheduler in order to support application transparent CR, again leveraging virtualization. Following that in section 4, we detail the experiments we performed in order to validate our implementation. We also present timing measurements to shed light on the feasibility of the method. Following this, we conclude and present future directions.

2. Basic CR mechanism - Coordinated CR

2.1 Architecture

A primary component of virtual machine technology is the concept of saved virtual machine state. Among other features, *suspending* a virtual machine as well as the *live migration* feature use this idea. Using this as the basic mechanism to save machine state for CR is our main idea. This ensures that our architecture can be implemented on a variety of virtualization environments such as Xen and VMWare. We use coordinated (a.k.a. synchronized) CR strategy described in [1], modified to work in a

virtual machine environment. Thus we arrive at an off-the shelf portable solution.

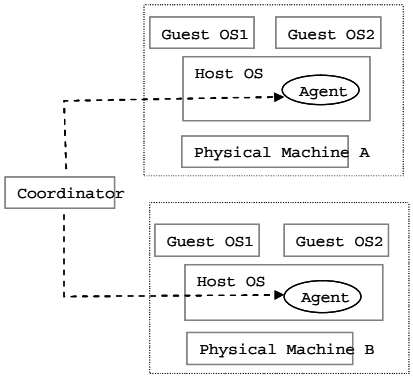

Figure 1:Coordinated checkpointing with virtual machines

Figure 1 shows the overall architecture. In this instance we have two physical servers A and B. For each physical server we have two Guest OSes running in different virtual machines(VMs) on the server. On another physical server we are running a *Coordinator* program. On each Host OS there is an *Agent* which communicates with the coordinator to facilitate the checkpoint and restart. The typical parallel MPI program runs inside the Guest OSes. The objective is to do two things: checkpoint the guest VMs in a coordinated manner and to restart the checkpointed VMs together when needed.

Checkpoint*:* When the coordinator is asked to checkpoint it sends a request to all relevant Agents requesting each to checkpoint a set of VM Guests of interest. When successful, the agent returns a checkpoint completion message back to the coordinator. The coordinator does any bookkeeping required and then tells the agents to continue the guests from where they were checkpointed. All this is done in a synchronized fashion and the bookkeeping tracks the versions of the checkpoints taken. Notice that we are doing nothing that is application specific. The entire state of the application along with the OS is checkpointed by the virtual machine platform onto stable storage using existing state saving mechanisms available in virtualization solutions.

Restart*:* When the coordinator is told to restart the application from a given checkpoint version, and a given bunch of physical servers to start the VM Guests on, it contacts the corresponding Agents and requests them to restart the guest OSes from checkpointed images. The coordinator ascertains that all restarts were successful. Existing state restart mechanisms available with virtualization solutions are used for this purpose.

2.2 Implementation

The above-mentioned coordinated CR architecture has been implemented on top of Xen 3.0. The current implementation only does memory checkpointing, as our focus has been compute intensive jobs; we do not implement file-system checkpointing. We refer the reader to [5] for a detailed description of our implementation.

3. Virtualization aware Job Scheduler for CR

3.1 Architecture

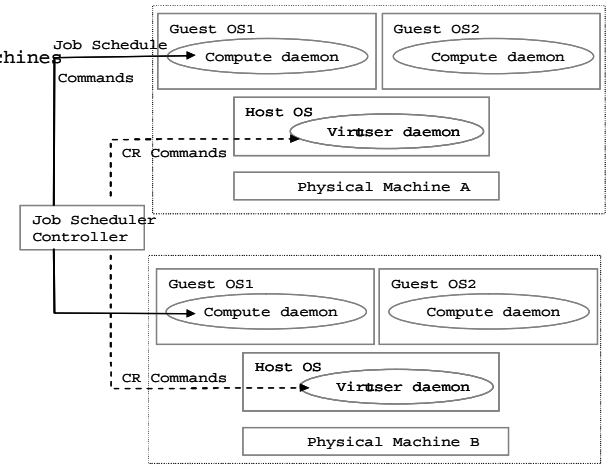

Figure 2: Job Scheduler Communication Paths for CR

Typical job schedulers have an architecture where there is a central *job scheduler controller* which resides on the node responsible for the administrative activities of the cluster. User can interact with the controller by means of tools. There will be agents of the job scheduler called *compute daemons* running on each of the compute nodes. These daemons are responsible for all the job scheduling activities with respect to the compute nodes such as running of jobs/compute workloads, cleaning up of the jobs, cancelling of the jobs etc.

Our architecture is shown in Figure 2. The Guest OS maps to a normal compute node in a cluster. In such a cluster environment a parallel (MPI) application consists of tasks that run on several Guest OSes. The c*ompute daemons* are the job scheduler daemons responsible for the job scheduling activities on the Guest OS. The interaction between the job scheduler controller and the compute daemons are indicated by a set of solid lines. As shown in the figure there is another agent that runs on the Host OS called the *virtual user daemon,* (corresponding to the *Agent* mentioned in the previous section) which in turn

communicates with the job scheduler controller on the one hand and the Host OS on the other. The channel between the job scheduler controller and the virtual user daemon is for the communicating requests to checkpoint and restart Guest OS. Next, we describe the sequence of events and corresponding actions during a CR of a parallel job running on multiples nodes (Guest OSes) on the cluster.

Checkpoint: An action to checkpoint a particular job can be initiated either by manual intervention (using job control commands) each time a checkpoint is needed or by using any support for periodic checkpointing. The following are the steps involved in the checkpoint action:

1. *The user makes a checkpoint request to the Job scheduler controller indicating the JobID of the job to checkpoint.*
2. *The job scheduler controller determines the Guest OSes corresponding to the job in question and the corresponding Host OSes from the configuration.*
3. *Job controller daemon creates a checkpoint ID from the the JobID and the NodeName(ie Guest OS) and passes it to the virtual user daemon on each node with a request to checkpoint the Guest OS.*
4. *Individual Virtual machine (i.e., Guest OS) checkpoints are performed by the corresponding virtualization daemons and stored in a shared external location with the corresponding identifier.*
5. *The virtualization daemons reply success/failure to the job scheduler controller.*
6. *The job scheduler controller signals the virtual user daemons to continue executing the VMs (hence the jobs).*

The job scheduler normally tracks the availability status of nodes in the cluster. In our environment this is the set of Guest OSes. When one of the nodes is determined to be down there are two approaches to deal with this. First, the scheduler may simply inform the user of the failure of the node and hence the job. The user may then initiate restart using commands provided by the scheduler. Alternately, it is possible to arrange for the scheduler to automatically restart the GuestOSes from the most recently checkpointed state. In either case the solution is application transparent. While system failure detection is not the immediate concern of this paper, it would be reasonable to integrate any failure detection mechanism with the job scheduler.

Restart: The following steps are involved in the restart mechanism

1. *The user makes restart request to the Job scheduler controller indicating the JobID of the job to be restarted.*
2. *The job scheduler controller determines all the corresponding Guest OS images that form the checkpoint.*
3. *The job scheduler controller then determines the physical nodes on which to run these Guest OSes as per the scheduling policy in place.*
4. *The job scheduler controller contacts the virtual user daemons running on respective Host OSes.*
5. *If a Guest OS is already running corresponding to the node, the virtual user daemon kills it in-order to ensure that, the image corresponding to the checkpoint may be restarted without conflict.*
6. *The virtual user daemons determine paths to the corresponding images by using the {JobID, Nodename} information.*
7. *Virtualization daemons restart each of the Guest OS on each Host OS across all the physical nodes involved in the job.*
8. *The virtual user daemons reply success/failure of restart to the job scheduler controller.*
9. *The job scheduler controller signals the virtual user daemons to continue executing the VMs (hence the jobs).*

3.2 Implementation

We implemented the above ideas by enhancing SLURM (Simple Linux Utility for Resource Management)[9], which is an open source scheduler used in HPC environments. Figure 2 presents the high level architecture of SLURM job scheduler with virtualization based checkpoint restart support. We use SLURM version 1.1.9.

During the initialization of the job scheduler controller and other job scheduler daemons, the mapping of Guest OSes to the Host OSes specified in the configuration is imported into the job scheduler controller data structures. At this point the job scheduler is aware of the virtualization infrastructure present on the cluster. The location of the data storage to hold the checkpoint image is also specified in the configuration settings. Keeping this storage shared facilitates migration of virtual machines among physical nodes.

We use the SLURM mechanisms to detect node failure. If a Guest OS does not respond for pre-configured duration of time, the machine is moved into a "DOWN" state from an "ALLOC" state. The action of restart can be initiated when the "DOWN" state is detected for a node automatically or by administrator intervention.

Job scheduling commands, configuration change requests etc. are submitted by the user to the job scheduler controller - *slurmctld* in SLURM environment by means of user land tools such as *srun, scontrol* etc. The job scheduler controller contacts the job scheduler daemons – *slurmd,* running on respective nodes (Guest OSes) for spawning various processes required for job execution. A Xen CR plug-in – *xencr* – has been written for the job scheduler controller as part of this work. Once this plug-in is loaded and configured, one could send CR commands (*scontrol checkpoint [create|restart] <jobid>)* to the job scheduler controller. The CR plug-in, i.e., *xencr*, interprets these checkpoint commands and performs required actions in accordance with the protocols mentioned earlier. *Xencr* communicates requests to *xend* on the respective physical nodes. Note that the *xend* runs in Xen dom0 on these physical machines. We use *libvirt* APIs[12] (version 0.1.4) to communicate from *xencr* to the corresponding *xend*s.

4. Experimentation

4.1 Experiments with the job scheduler

The experimental setup consisted of 3 nodes - a node running the *slurmctld*, two nodes running one Xen Guest OS on respective Host OSes. The *slurmd* – the compute daemon was run on each of the Guest OSes. HPLinpack[11] was compiled with the PMI libraries of SLURM in order to use the SLURM daemons for MPI communications.

An HPLinpack job requiring two compute resources(nodes), was spawned from the system running the *slurmctld* using the *srun* command. The job was checkpointed at a certain point of time and allowed to continue running; some time before the job completed, one of the nodes was killed manually (by using *xm destroy*), to simulate a node failure. The *slurmctrld* detects the node failure within a pre-configured amount of time. At this point a restart of the job is performed by using the *scontrol* command. We verified that the job gets properly run after re-start and also that the job reached completion correctly by looking into the correctness of the HPLinpack test results.

4.2 Timing the CR mechanism

We ran experiments without a scheduler to measure checkpoint times during failure-free runs. The times we report here are the total times to checkpoint – including time to coordinate, do the `xm save` and the `xm restore`, averaged over six checkpoint events each.

Linux FC4 was the Guest OS. The application that was run in all our experiments was HPLinpack built with MPICH-1.2.7. We used two workstations with 2.8GHz Dual Xeon processors and with 1GB RAM each and stored checkpointed images on local disks.

Table 1: *Observations on checkpoint times*

Guest Physical Mem Size(MB)	*768*	*512*	*256*
Checkpoint time(sec) with N=6490	*30.50*	*21.33*	*11.33*
Checkpoint time(sec) with 80% memory occupancy	*30.66*	*21.50*	*11.33*

Table 1 summarizes our observations. We note that the time to checkpoint was independent of the memory that the application occupied (in this case the HPLinpack problem size *N*) for a fixed guest virtual machine physical memory size. We find that the time to checkpoint a virtual machine was a simple linear function of the physical memory of the guest OS. Note that the overall time to checkpoint an application distributed over several nodes in the cluster would be only marginally more than this (to account for increased coordination time).

4.3 Discussion

We observed that it would have been nice to have a c*heckpoint and pause* type of facility rather than a *checkpoint and halt* that is implemented by Xen – VM state save facility. We currently use the available facility to first save and halt the VM followed by a VM restore to do the checkpointing. Similarly, during restart from a checkpoint it would help if we have a *restore and pause* facility, which will bring up the domU, but put it in a paused state so that the agent could wait for a final go-ahead from the coordinator before continuing. This is to reduce the probability that the application times out on communication during restart, generating unnecessary error messages. This is especially useful when we have large clusters.

During restart we observed that the *slurmd* running on the Guest OSes performs a sanity check of whether the daemon which actually spawns the process (i.e., *slurmstepd*) is running. This is done by communicating via Unix sockets. This is normal *slurmd* functionality. These sockets become stale after we resume from a checkpointed image. This makes it impossible for the *slurmd* to register the job with *slurmctrld*. This results in the jobs get killed by the controller during the restart. Our analysis is that this

is a side effect of bringing a checkpointed image back to life at a future point in time. We currently worked around this issue by avoiding the inter process communication during registration phase of node across *slurmd* and *slurmstepd*. The job scheduling and other actions did not get affected by this modification. In fact the cleanup of jobs worked appropriately after the logical end of the job. This is one topic which we hope to investigate further in the future.

During checkpoint and during restart the coordinator stops and resumes the guest OSes synchronously. Yet, it is likely that not all the guest OSes are halted/restarted at the same instant. This leads to a situation where some packets are lost (not received). This does not pose an immediate problem because the inter-node communication protocol we used is based on TCP/IP which retransmits unacknowledged packets.

5. Summary and Future Work

Our work shows that indeed it is possible to develop practical CR solutions for HPC applications using standard virtualization technology and to integrate it with a job scheduler. Our observations show that the cost of coordinated checkpointing is acceptable in the case of long running applications.

There are at least three important directions for our future work. First, we have plans to enhance the scalability of the solution to large clusters. This is possible with a combination of solutions – tree-like communication mechanisms for coordination, a high bandwidth parallel storage system such as Lustre [19] to hold the checkpoint images, incremental checkpointing and lazy transfer of checkpoint images. Second, we intend to integrate support for RDMA infrastructure. [7,8,21] indicate efforts underway to support Infiniband on Xen and we intend to integrate our CR infrastructure once that stabilizes. Third, several HPC applications do write regularly to storage, and it is necessary that the state of the file system is also checkpointed along with the application state – initial investigations show that one can use ideas from [10] or a variety of proprietary storage technologies to address this.

References

[1] M. Elnozahy, L. Alvisi, Y-M Wang, and D.B. Johnson. *A survey of rollback-recovery protocols in message-passing systems*. Technical Report CMU-CS-99-148, Canigie Mellon University, June 1999.

[2] G. Janakiraman, R. J. Santos; D. Subhraveti, Y. Turner; *Cruz: Application-Transparent Distributed Checkpoint-Restart on Standard Operating Systems*. Tech Report HPL-2005-66, 2005.

[3] S. Osman, D. Subhraveti, G. Su, and J. Nieh, *The Design and Implementation of Zap: A System for Migrating Computing Environments.* in Proceedings of the 5th OSDI Symposium, Boston, MA, December 2002.

[4] R. Figueiredo, P. Dinda, J. Fortes, J. *Resource Virtualization Renaissance.* Guest Editors' Introduction: IEEE Computer, Volume: 38(5), May 2005.

[5] V. R. P. Gokul, Y. Krishnakumar, P. Rajan, and D. Sukumar. *Checkpoint-Restart of Distributed Applications using Xen.* Masters–Thesis Report, IIIT, Bangalore, June 2006.

[6] IBM*; Workload Management with LoadLeveler*. http://www.redbooks.ibm.com/abstracts/sg246038.html

[7] J. Liu, W. Huang, B. Abali, D. K. Panda. *High Performance VMM-Bypass I/O in Virtual Machines.* Usenix Conference, May2006. http://nowlab.cse.ohio-state.edu/publications/conf-papers/2006/usenix06.pdf

[8] Xensource; *Xen-IB support implementation.* http://wiki.xensource.com/xenwiki/XenSmartIO.

[9] M. Jette, M. Grondona; *SLURM – Simple Linux Utility for Resource Management*. Proceedings of ClusterWorld Conference and Expo, San Jose, California, June 2003.

[10] G Vallee, T Naughton, H Ong, S L Scott; *Checkpoint/Restart of Virtual Machines Based on Xen*. HAPC Workshop, Santa Fe, October 2006.

[11] A. Petitet, R. C. Whaley, J. Dongarra, A. Cleary*; HPL - A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers*. January 2004, http://www.netlib.org/benchmark/hpl/

[12] *Libvirt - The Virtualization API*, http://libvirt.org/

[13] Mark F. Mergen, V. Uhlig, O. Krieger, J. Xenidis; *Virtualization for High-Performance Computing,* in ACM SIGOPS OS Review, 40(2), April 2006. http://www.research.ibm.com/K42/papers/osr06-virt.pdf.

[14] Lamia Youseff, Rich Wolski, Brent Gorda, Chandra Krintz. *Evaluating the Performance Impact of Xen on MPI and Process Execution for HPC Systems*, in the Proceedings of the First International Workshop on Virtualization Technology in Distributed Computing (VTDC), held in conjunction with SC06.

[15] L. Youseff, R. Wolski, B. Gorda, and C. Krintz. *Paravirtualization for HPC Systems*. Technical Report Number 2006-10, Computer Science Department University of California, Santa Barbara, Aug. 2006.

http://www.cs.ucsb.edu/research/tech_reports/reports/2006-10.pdf

[16] J. S. Plank, M. Beck, G. Kingsley, K. Li; *Libckpt: Transparent Checkpointing under Unix*, Conference Proceedings, Usenix Winter 1995 Technical Conference, January, 1995.

[17] P. H. Hargrove and J. C. Duell *Berkeley Lab Checkpoint/Restart (BLCR) for Linux Clusters* In Proceedings of SciDAC 2006: June 2006.

[18] W. Huang, J. Liu, B. Abali and D.K. Panda, *A Case for High Performance Computing with Virtual Machines*. The 20th ACM International Conference on Supercomputing (ICS '06), Cairns, Queensland, Australia, June 2006.

[19] Cluster File Systems Inc., *Lustre: A scalable high performance file system*, Whitepaper available at http://www.lustre.org/docs/whitepaper.pdf

[20] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauery, I. Pratt, A. Wareld. *Xen and the Art of Virtualization.* SOSP'03, October 19-22, 2003.

[21] W. Huang, J. Liu, M. Koop, B. Abali and D.K. Panda. *Nomad: Migrating OS-bypass Networks in Virtual Machines.* In Proceedings of the 3rd International ACM SIGPLAN/SIGOPS Conference on Virtual Execution Environments (VEE'07), San Diego, CA, June 2007.