A Fault-tolerant High Performance Cloud Strategy for Scientific Computing

Ekpe Okorafor
Computer Science and Engineering
African University of Science and Technology (AUST)
Abuja, Nigeria
Email: eokorafor@aust.edu.ng

***Abstract*—Scientific computing often requires the availability of a massive number of computers for performing large scale experiments. Traditionally, high-performance computing solutions and installed facilities such as clusters and super computers have been employed to address these needs. Cloud computing provides scientists with a completely new model of utilizing the computing infrastructure with the ability to perform parallel computations using large pools of virtual machines (VMs).**

The infrastructure services (Infrastructure-as-a-service), provided by these cloud vendors, allow any user to provision a large number of compute instances. However, scientific computing is typically characterized by complex communication patterns and requires optimized runtimes. Today, VMs are manually instantiated, configured and maintained by cloud users. These coupled with the latency, crash and omission failures in service providers, results in an inefficient use of VMs, increased complexity in VM-management tasks, a reduction in the overall computation power and increased time for task completion.

In this paper, a high performance cloud computing strategy is proposed that combines the adaptation of a parallel processing framework, such as the Message Passing Interface (MPI) and an efficient checkpoint infrastructure for VMs, enabling its effective use for scientific computing. By developing such a mechanism, we can achieve optimized runtimes comparable to native clusters, improve checkpoints with low interference on task execution and provide efficient task recovery. In addition, checkpointing is used to minimize the cost and volatility of resource provisioning, while improving overall reliability. Analysis and simulations show that the proposed approach compares favorably with the native cluster MPI implementations.

***Keywords*-Checkpointing; Cloud; MPI; Virtualization;**

I. Introduction

The unprecedented introduction of the commercial cloud infrastructure and services have allowed users to provision compute and storage resources, as well as applications in a dynamic manner on a pay per use basis. These resources are relinquished when not used and can be integrated within an existing infrastructure. Some of these infrastructures and services such as Amazon EC2/S3[1,2], Gogrid[3], Microsoft Azure [4], Google App Engine [5] and others, have the effect of improving resource utilization, decreasing waste, and improving energy conservation. In addition solutions employing cloud technologies such as MapReduce[6], Hadoop[7] and Dryad [8] have proved to be successful. The resources provisioned are transparent to the users. In some cases, the use of virtualized resources actually give the user much more control over the provisioned resources, including the ability to completely customize the Virtual Machine (VM) images, root/administrative access, etc.

With all these, an assumption can be made that access to computational power is no longer a barrier to scientific computation which usually requires the availability of a massive number of computers performing large scale data/compute intensive applications. Scientific computation typically involves the construction of mathematical models and numerical solution techniques to solve scientific, social and engineering problems. Many of these problems tend to be large and complex in scope that dedicated high-performance computing (HPC) infrastructures such as grids, clusters or network of workstations have traditionally been utilized to address these computing needs.

Grid and cluster computing have opened up many opportunities and created profound capabilities for innovative research with an extraordinary advantage of on-demand computing power as a utility much like electric power. These capabilities include dynamic resource or service discovery, and the ability to pool a large number of infrastructural resources belonging to different administrative domains. Additionally, the intrinsic ability to find the best set of compute and storage nodes to accomplish a specific scientific computational task is at core of the success of these HPC clusters and grid computing. Some of these grids for scientific computing [9] have become so successful that a world-wide infrastructure has been established and now available for computational science.

However, with all the promising features of the cloud, HPC has not been a good candidate due to its requirement for tight integration between compute nodes via low-latency interconnects. In addition, virtualization, which often is a prerequisite for migrating local applications to the cloud, does not allow for scalability and efficiency in an HPC context due, to its inherent performance overheads. HPC clusters usually run fully-utilized and therefore there are no benefits gained from consolidation.

In the context of scientific computation using HPC these three major preconditions need to be satisfied:

1) The need for tight coupling between the compute

nodes via low-latency interconnects
2) There should be an appropriate parallel runtime to implement it
3) The application needs to be parallizable to utilize the available resources

Therefore, in order to migrate scientific applications to the cloud, the aforementioned preconditions need to be satisfied. In a previous paper [10], we provided an assessment of the performance, overhead, system utilization and runtimes of HPC applications in the cloud environments. The paper also addressed the need for virtualization in HPC clouds. We concluded that HPC can use the cloud concept without an appreciable decrease in performance, especially in the case of a private cloud if virtualization is excluded. Achieving native application performance and scalability in cloud environments will enable HPC to effectively extend itself in such environments. This will certainly make it easier for research centers and universities to utilize high performance compute resources across the world for better research, education and product development.

The main challenge with using the cloud for HPC is the performance or productivity. HPC applications or scientific computations require high compute power, high performance interconnects and fast connectivity to the storage or file systems. Most clouds are built around non-homogeneous distributed systems, connected by low-performance interconnects. These solutions do not present optimal environments for HPC applications or scientific computing. In addition, virtualization, which enhances the provisioning process, increases the load on the compute infrastructure and further degrades cloud performance.

This paper proposes a method that adds fault-tolerance capabilities in a cloud environment for scientific computing and satisfies the conditions listed earlier for migrating scientific applications to the cloud. Specifically, we introduce a smart checkpointing infrastructure suitable to work with virtual machines (VMs), combined with a robust integration of a parallel processing framework and replication protocol within the cloud thereby optimizing its use for scientific computing.

Checkpoint mechanisms record the system state periodically to establish recovery points. Typically, making checkpoints can be very costly in terms of performance. It is especially noticeable in an environment consisting of virtual nodes. Checkpointing in such an environment will require dealing with huge VM images that will require large storage and efficient restore process.

A common mechanism for implementing parallel processing applications uses the Message Passing Interface (MPI) standard. Two very important features of MPI exploited in this regard include:

- API to simplify the message passing between application processing nodes
- process management environment to handle the launching of application processes across hosts

Considering that most clouds are connected by low-performance interconnects, latency then becomes a factor that degrades productivity and reduces performance. A node or link is considered failed if the latency associated with the node of link is large. A users set of VMs may not all be running on the same physical resources, so latency associated with the VMs could come from low network bandwidth, disk failures on the physical infrastructure and link unavailability. The probability of a node failure increases with the number of nodes. If many VMs can potentially be instantiated onto multiple physical resources, then we can conclude that the system needs a fault-tolerant mechanism to address these potential failures that can occur at the hardware level.

In this paper, we investigate the use of a modified MPI and a dynamic checkpoint method to achieve the goal of minimizing overall latency, tight coupling between the compute nodes while maximizing reliability. We show that a dynamic checkpointing strategy coupled with a parallel processing framework like MPI will further extend the cloud to HPC applications and scientific computation. The management load of the parallel framework is minimized while that of administering the cloud resources is rendered invisible to the user. The VMs can be instantiated or migrated in an impromptu manner onto alternate physical resources optimizing the checkpointing operation. We employ a method that replicates the checkpoint files on alternate physical resources and enables the user maintain a backup VM in case of a failure.

The rest of the paper is organized as follows. Section 2 describes the related work. Section 3 gives an overview of MPI architecture, cloud computing environment and checkpointing. Section 4 describes the architecture and functioning of our modified MPI and VM checkpoint/restart infrastructure. Section 5 presents the evaluation results. Finally, in section 6 we conclude.

II. Related Work

There have been many descriptions of cloud computing and virtualization strategies in the literature, including performance enhancements designed to reduce their overhead. Xen, in particular, has undergone considerable analysis [11].

However, little work exists that has adequately examined the varying cloud technologies and virtualization strategies for their use in HPC environments, particularly with regard to reliability and performance for scientific computations that require complex node interactions. Our design, evaluation and experimentation fills this gap and provides a better understanding of the use of the cloud and cloud technologies, especially virtualization for scientific computing.

VMWare ESX Server has been studied in regards to the architecture, memory management and the overheads of I/O processing [12, 13]. Further more, virtualization, which is the main thrust of VMWare, proves to adversely affect performance in a scientific computing environment.

Checkpointing at both the user-level and kernel-level has also received some considerable attention and studied extensively [14, 15]. However, there has been a focus on developing methods that work on single-process or multi-threaded checkpointing implementations. Developing a checkpointing mechanism in a distributed system requires additional considerations, including in-flight messages, sockets, and open files. In a recent study documented in Gropp, et al., a high-level overview of the challenges and strategies used in checkpointing MPI applications [16] is presented. The official LAM/MPI implementation includes support for checkpointing using the Berkeley Linux Checkpoint/Restart (BLCR) kernel-level checkpointing library [17]. A more recent implementation by Zhang et al. duplicates the functionality of LAMs kernel-level checkpointer, but implements checkpointing at the user-level [18].

The C3 system, considered the most widely published application-level checkpointing system for MPI programs [19], works for programs written in the C programming language. A a pre-processor/pre-compiler is used to first transform a users source code into checkpointable code. One advantage of the C3 system is that it allows for checkpointing and migration within heterogeneous cluster architectures [20]. However, application-level checkpointing also requires more effort on the part of the programmer, where checkpointing primitives must be inserted manually.

Checkpointing within virtual environments has also been studied, though typically not for the use of HPC applications. OpenVZ [21], Xen [22], and VMWare [23] all provide mechanisms to checkpoint the memory footprint of a running virtual machine. However, to date, the use of these checkpointing mechanisms have been limited to the area of "live migration" due to the lack of complete file system check pointing and/or a lack of checkpoint/continue functionality. By supporting only live migration, the virtualization tools avoid file system consistency issues.

Another application is in the use of preemptive migration [24]. However, the capability to heuristically predict node failures, does not offer much protection from sudden and unexpected node failures. Without a rollback-recovery strategy, the system must rely on accurately predicting failures prior to their occurrence. Our work does combines such proactive migration with a reliable recovery strategy in the case of node failure.

Our work differs from the previous work in VM checkpointing and storage in that we enable periodic checkpointing and rollback recovery for MPI applications executing within an virtualized environment. This requires cooperation with the existing VM-level checkpointing support that is already provided by the VM/VPS. We have added local disk checkpointing with replication to both reduce the overhead of checkpointing as well as to improve checkpoint resiliency in the presence of multiple simultaneous node failures. Our checkpointing solution does not rely on the existence of network storage for checkpointing. The absence of network storage allows for improved scalability and also shorter checkpoint intervals (where desired) [25].

III. MPI Architecture and Cloud Computing Environment

High performance clusters provide an ideal environment to run parallel applications, mostly due to the fact that they interconnect large amounts of computing resources and their ability to transparently schedule and allocate the resources that are most appropriate to run the application's processes. The cloud, utilizing virtualization techniques allow resources to be be harvested from physically distributed non-dedicated machines, contributing to the cost/benefit relation.

To put it in context, parallel programming in a cloud environment requires that the architecture be able to negotiate with a diverse set of resource managers to run the applications on an integrated set of resources that span different administrative domains. In addition to the issue of resource allocation, communication and fault-tolerance become more complicated. The current MPI architecture does not have this capability. Our solution is to modify the programming library to replace its process management mechanisms with those of the cloud or virtualization middleware, and incorporate a checkpoint-based fault-tolerant infrastructure. This maintains the original programming model, improves reliability and enables transparency to the parallel application developer. Another advantage our solution provides, is the ability to run legacy parallel programs on the grid. We have adopted this approach in our implementation.

A. MPI Architecture

MPI has become the *de facto* standard for parallel applications programming [26]. The open source LAM/MPI architecture combines portability and high performance is among the several implementations that exist. It has a modular, layered architecture which separates the implementation of the high level protocols and functions from the low level mechanisms used for *interprocess communication* and *process management*. This is very important because it makes it possible to build a new implementation by rewriting only the functions at the lowest level. The LAM layer provides a framework and run-time environment upon which the MPI layer executes. The MPI layer provides the MPI interface and an infrastructure for direct, process-to-process communication.

The MPI library consists of two layers. The upper layer is portable and independent of the communication sub-system (i.e., MPI function calls and accounting utility functions). The lower layer consists of a modular framework for collections of Systems Services Interfaces called SSI. One such collection is the MPI Request Progression Interface (RPI), which provides device dependent point-to-point message-passing between the MPI peer processes. Another is the checkpoint/restart (CR), which provides an interface to the back-end checkpointing system that does the actual checkpointing and restart functionality, using `blcr`.

At the start of execution of an MPI job, the CR SSI determines whether checkpoint/restart support was requested, and if so, the `blcr` module is selected to run. To support checkpointing, an RPI module must have the ability to generically *prepare* for checkpoint, *continue* after checkpoint, and *restore* from checkpoint. A checkpointable RPI module must therefore provide ***callback*** functions to perform this functionality.

B. Cloud Environment

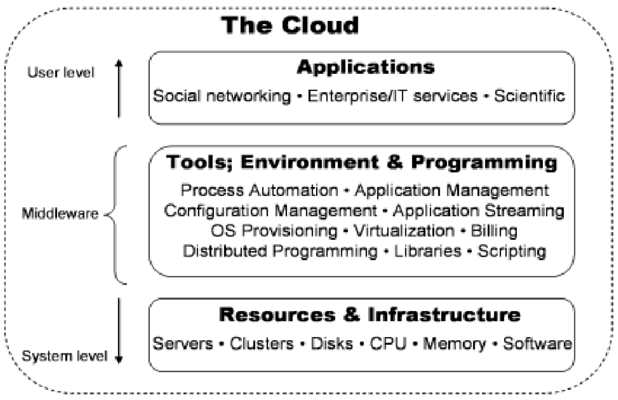
Cloud computing is a term used to describe the infrastructure, platform and type of application. A cloud computing environment encompasses the ability to dynamically provision, configure, reconfigure and de-configure servers and services as needed. The servers or nodes can be physical or virtual machines. The term cloud computing is too broad to be captured into a single definition. The key elements include software and hardware available on demand over the internet. Buyya et al. [27] gives a more structured definition as a “type of parallel and distributed system consisting of a collection of interconnected and virtualized computers that are dynamically provisioned and presented as one or more unified computing resources based on service-level agreement”. This definition is apt in that it adequately captures the ability of delivering both infrastructure and software as services. The resources that constitute the physical infrastructure include clusters, datacenters, servers or desktop machines. The services exposed by the cloud can be classified into three major offerings available to the end-user. These are; 1) Infrastructure-as-a-Service (IaaS), 2) Platform-as-a-Service (PaaS), and 3) Software-as-a-Service (SaaS)

Clouds can be internal (private), managed with an organization, providing compute resources to the organization’s employees. When these resources are managed by a cloud computing vendor, then it is external (public). Figure 1 shows the complete structure of the cloud, listing the different layers and components.

C. Virtual Machine Checpoint/Restart (VMCR) Architecture

The design of the VMCR architecture leverages the virtualization properties inherent in a cloud environment to support VM checkpointing mechanism. The objective is to trigger the VM checkpointing process by MPI with minimum overhead. We achieve this by enabling VM computation to perform concurrently on two different processors while replicating the checkpoint file to another node.

Our fault-tolerant VM checkpointing mechanisms can be described in four basic steps:



The Cloud
User level
Middleware
System level
Applications
Social networking • Enterprise/IT services • Scientific
Tools; Environment & Programming
Process Automation • Application Management
Configuration Management • Application Streaming
OS Provisioning • Virtualization • Billing
Distributed Programming • Libraries • Scripting
Resources & Infrastructure
Servers • Clusters • Disks • CPU • Memory • Software

Figure 1. Cloud computing layered architecture and components

1) **MPI checkpoint trigger:** The protocol proceeds in two main steps, first, checkpoint the MPI process and secondly, trigger checkpointing of the VM file system.
2) **VM migration:** Migrate a VM from one processor to another in order to let the VM on the original processor perform checkpointing while the VM on the new processor continue to execute. Live migration could be conducted quickly across different cores or processors on a multi-core or SMP machine. It could also be done across two separate computers over a network.
3) **Checkpointing and computation overlap:** Usually, after the VM migrated, the VM on the original processor (or the original VM) would be stopped. The new VM on the destination processor would continue execution. In our design, the original VM would perform checkpointing instead of being stopped. Moreover, in multi-core or SMP environments, we can assign the new VM and the original VM to run on any processor. Therefore, if users want to let the new VM continue running on the original processor and the checkpointing VM run on a new processor, they could readily do so by adjusting CPU scheduling parameters.
4) **Replication:** Select a randomly generated set of nodes to replicate checkpoint files.

IV. VM Checkpoint Enabled MPI

The cloud is a viable choice for HPC clusters. As earlier mentioned, one important attribute of cloud computing is virtualization. Virtualization is a common strategy for improving the utilization of existing computing resources, particularly within data centers. However, in order for large scientific research to be carried out on virtualized clusters, some form of fault tolerance/checkpointing and a parallel runtime must be present. Given that an operating system level virtualization provides the best overall performance with an MPI runtime, we propose a checkpointing solution based on an architecture that has an operating system level virtualization implementation with partition, checkpointing and live migration capabilities. In particular, we extend these capabilities to include support for checkpointing and fault-tolerance for a virtualized cloud environment using an MPI runtime for distributed scientific computing. The emphasis of this work is the development of a VM checkpoint/restart system that is triggered by MPI checkpointing. Our design includes the following:

1) Checkpoint-enabled LAM/MPI implementation
2) VM checkpoint/restart system (VMCR)
3) Operating system kernel-based VMCR deamon (vmcrd)

The model we adopt assumes that virtual instances or nodes fail when they crash or stop sending or receiving messages. In a typical virtualized environment as obtained in cloud computing, the nodes or virtual machines (VMs) exist in a SAN or some network storage. We reduce the checkpointing overheads by eliminating the use of a SAN or network storage. However to achieve this we adopt a mechanism that stores all checkpoints on a node's local disk. To add fault-tolerance we replicate these checkpoints using a random seed to store the checkpoints in peer nodes initiated by the VMs running MPI programs.

A. *Adaptation of LAM/MPI*

The vmcrd, which is our daemon implementation, runs as a single instance on the host system and acts as a relay between the VMs and the VM checkpoint/restart mechanism. In effect vmcrd is responsible for taking the checkpoint or snapshot of the running computations and file system. This design is based on the LAM/MPI with modification to its existing checkpoint support. The vmcrd now performs the checkpoint instead of the blcr module. The protocol proceeds in two main steps, first, checkpoint the MPI process and secondly, trigger checkpointing of the VM file system.

Since mpirun is the startup coordination point for MPI processes, it was the natural choice to serve as the entry point for a checkpoint request to be sent to a LAM/MPI job. At the start of execution, mpirun invokes the initialization function of the blcr checkpoint/restart SSI module to register both thread-based and signal-based callback functions with blcr. The thread-based callback is required to propagate the checkpoint requests to the MPI processes. This cannot be done in signal context because the propagation of the checkpoint request uses some non-reentrant C library calls, and the use of non-reentrant functions from signal context can cause deadlocks.

When a checkpoint request is sent by a user or batch scheduler, the vmcrd (by invoking the blcr utility cr-checkpoint with the process ID of mpirun), triggers the callbacks to start executing. The thread-based callback computes the names under which the images of each MPI process will be stored to disk and saves the process topology of the MPI job (called "application schema" in LAM) in mpiruns address space, that will be used for restoring the applications at restart. It then signals all the MPI processes about the pending checkpoint request by instructing the relevant lamds to invoke cr-checkpoint for every process that is a part of this MPI job. Once this is done, the callback thread indicates that mpirun is ready to be checkpointed.

When a checkpoint signal is delivered to all MPI process, each process exchanges bookmark information with all other MPI processes. This process is known as quiescing the network and is important in order to drain the in-flight data on the network. When all the bookmarks match, the RPI has drained all the in-flight data on network, and the callback thread in each process indicates that the process is ready to be checkpointed. This is already provided by LAM and we reuse it in our implementation. Once all messages have been accounted for, the VM checkpointing can commence.

V. VM Checkpointing & Replication Protocol

In this section we describe the VM checkpointing protocol and replication protocol; these are the two key components of VMCR system. Briefly, the checkpointing protocol coordinates activities among all parties involved in a checkpointing operation, while the replication protocol replicates checkpoint data throughout the cluster. during checkpointing.

A. *VM Checkpointing & Restart (VMCR) protocol*

When checkpointing starts, `vmcrd` first sends a `vm_init` request to the destination processor to load a new VM to memory. The `new_VM` would wait for the VM execution state from `old_VM`. Then, it sends a `vm_init_ack` to `vmcrd` to inform that the loading is successful. `vmcrd`, in turn, sends a `cp_init` message to instruct `old_VM` to migrate its state to `new_VM`.

After `old_VM` migrated, it creates a checkpoint file. At the same time, `new_VM` continues its normal execution. During the checkpointing period, the `old_VM` and `new_VM` would have to access the same disk image. The key idea of our solution is the creation of a temporary disk update file to hold all updates to disk images made by `new_VM` during checkpointing. When `old_VM` finishes checkpointing, it sends a `cp_end` message to `vmcrd`, which subsequently sends a integrate_store message to instruct `new_VM` to stop writing disk updates to the temporary file so that `old_VM` can integrate the temporary files contents to the original disk image.

In response, `new_VM` sends an `integrate_start` message to `old_VM` to start the integration. All disk updates occur on `new_VM` beyond this point are redirected to `old_VM`, which would schedule these updates to appropriate disk locations during the integration. At the end of the integration, three messages consisting of `integrate_end`, `last_update`, and `update_done` are exchanged between `old_VM` and `new_VM` to make sure all disk update requests from the `new_VM` have been served. Then, `new_VM` would discard the temporary file and send all disk updates to the original disk image (just like before checkpointing took place). Finally, `new_VM` sends a `resume_disk_access` message to tell `vmcrd` that it is now using the original disk image for normal VM execution.

B. *VM Replication Protocol*

If checkpoints are saved only to the host node's local disk, computations will be lost due to a node failure. One approach will be to save the VM files to a network storage or perhaps a dedicated checkpoint server. Checkpointing a virtual server will considerably increase the size of the checkpoints due to the need to checkpoint the file system of a virtual server. To mitigate this problem, we adopt a replication strategy that replicates checkpoints to randomly selected replica nodes within the cloud environment, given a user-specified number of nodes. The replication process is performed after the a checkpoint is done and the VM has been resumed. The replication process can then proceed concurrently with the VM computation.

When `new_VM` sends a `resume_disk_access` message to tell vmcrd that it is now using the original disk image for normal VM execution, the `vmcrd` sends a `cp_replicate_init` message to initiate the checkpoint replication process.

The replication process itself involves a randomly generated set of participating nodes within the cluster to function as replica nodes. The replication process is performed after checkpointing has completed and the VM computation has resumed, so can be done concurrently. This reduces the impact of checkpointing on computation and shared resources. By spreading the cost of checkpointing over the nodes included in the replica set, we reduce the probability of having network link saturation as in the case where a dedicated checkpointing server is used.

After a destination node receives the `cp_replicate_init` message, it sends a `cp_replicate_ack` message back to the source node. The `old_VM` then begins the transfer of the checkpoint file to the destination node. When the process is complete, `vncrd` sends a `terminate_old_VM` message to terminate `old_VM`.

VI. Analysis Evaluation & Performance Results

To study the feasibility of our design, we have measured the VM checkpointing performance and replication protocol and to see potential performance improvement. Since one of the main goals of our mechanism is to use live migration during the replication process, to hide the checkpointing delay, we consider the migration time to be the lower bound time and the checkpointing time to be the upper bound time.

These sets of experiments were conducted to measure the performance of our prototype system. The first set measures communication performance using NetPIPE [28] (A Network Protocol Independent Performance Evaluator). The second set measures overhead of checkpoint/restart using High-Performance Linpack [29]. The third set measures the application performance and the last set measures the resiliency of the system as nodes fail.

The test platform is a 4-node Linux mini-cluster. Each node has 2 dual core 2.2GHz AMD Opteron processors; 8 GB of memory and Fast Ethernet interconnect. We deploy Eucalyptus on Ubuntu 10.2 and provision 4 VMs per node.

A. *Analysis*

We model the upper bound time, lower bound time, and estimated execution performance of total VM execution time as follows. Supposed a VM creates n checkpoints throughout its lifetime, we can model its upper bound total computation time as $T_{vm} = T_{mpi} + nT_{cp}$, where T_{vm} is the total time to complete the execution of a program, T_{mpi} denotes the computation and T_{cp} is the time to accomplish checkpointing. The lower bound can then be expressed as $T_{vm} = T_{mpi} + nT_{mg}$, where T_{mg} is the average time it takes to migrate to another VM. We can then model the total execution time for our MPI triggered VM checkpointing and replication method as $T_{vm} = T_{mpi} + nT_{VMCP} + T_{rep}$, where T_{VMCR} is the average checkpointing delay using our approach and T_{rep} denotes the the overhead incurred while replicating the checkpoint files to a peer node. We would expect the following relationship: $T_{mg} \leq T_{VMCR} + T_{rep}/n \leq T_{cp}$.

This is an in-line $\int \frac{d\theta}{1+\theta^2} = \tan^{-1}\theta + C$ equation.

B. *Communication Performance*

NetPIPE is a program that performs pingpong tests, bouncing messages of increasing size between two processes across a network to measure communication performance. We ran NetPIPE on top of LAM/MPIs TCP RPI module (TCP point-to-point channel without and without checkpoint/restart support), and our VMCR prototype (TCP with checkpoint/restart). As seen in Figure 2, the TCP communication module of LAM/MPI with checkpoint/restart support causes slight overhead than the native LAM/MPI without checkpoint/restart support. The VMCR prototype has more overhead than the rest as expected, but the discrepancy is not large.

C. *Application Performance*

In this section, we evaluate the VMCR-based computing environment with actual HPC applications. We evaluate both NAS parallel benchmarks and HPL. The execution time for NAS was normalized based on the native LAM/MPI environment in Figure 3. We observe that our VMCR prototype performs comparatively with the native environment using LAM/MPI on physical nodes without CR support. The native LAM/MPI performs about 5% better for the NAS applications CG and FT due to a better utilization

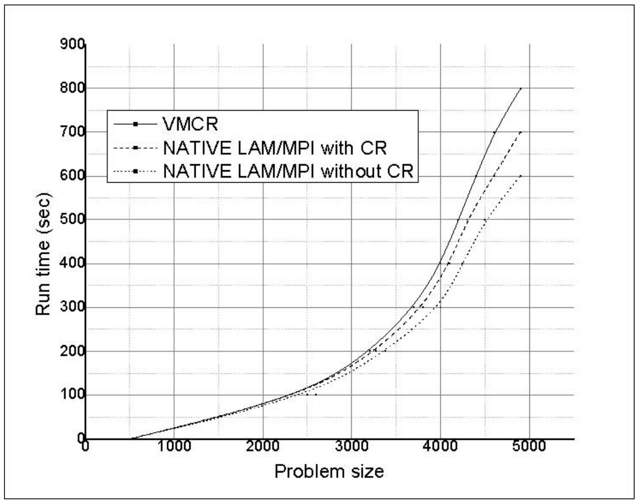


Figure 2. Checkpoint overhead

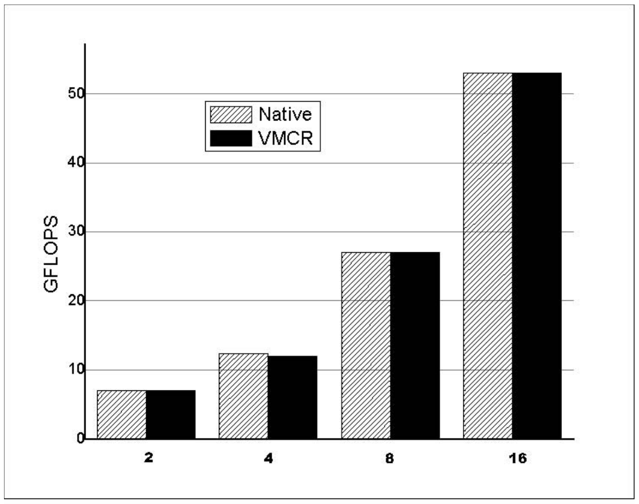


Figure 4. HPL on 2, 4, 8 and 16 processes

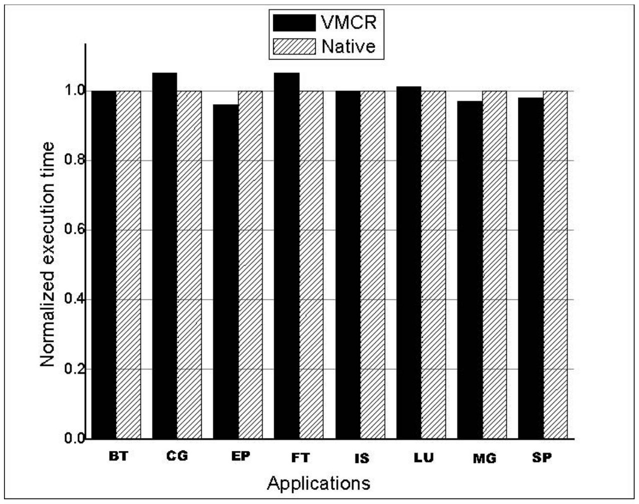


Figure 3. NAS parallel benchmarks

of the shared memory communication for processes on the same physical node. The performance achieved on HPL is also shown in Figure 4. Here, we observe that our VMCR prototype performs very well, with the native case outperforming by at most 1%.

VII. Conclusion

In this paper, we proposed a framework for cloud-based computing for scientific computing. We presented a case for combining the adaptation of MPI and a dynamic checkpoint method for VMs, enabling in effective use for HPC applications. We explained the infrastructure and the motivation behind our design. Our analysis and performance evaluation showed that HPC applications can achieve almost the same performance as those running in a native LAM/MPI environment.

References

[1] Amazon Elastic Compute Cloud (EC2), http://aws.amazon.com/ecs2/

[2] Amazon Simple Storage Services (S3), http://aws.amazon.com/s3/

[3] GoGrid Cloud Hosting, http://www.gogrid.com/

[4] Microsoft Azure, http://www.microsoft.com/windowsazure/

[5] Google App Engine, http://code.google.com/appengine/

[6] J. Ekanayake and S. Pallickara, ”MapReduce for Data Intensive Scientific Analysis,” Fourth IEEE International Conference on eScience, 2008, pp.277-284.

[7] Apache Hadoop, http://hadoop.apache.org/core/

[8] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, “*Dyrad: Distributed data-parallel programs from sequential building blocks*,” European Conference on Computer Systems, march 2007.

[9] M. J. Chin, S. Harvey, S. Jha, and P. V. Coveney, “*Scientific Grid Computing: The First Generation*,” Computing in Science and Engineering, vol. 7, 2005, pp. 24-32.

[10] E. Okorafor, “High Performance Cloud Computing: An Emerging Strategy for Scientific Computing,” *International Conference on Grid Computing and Applications*, GCA'10 Las Vegas, Nevada, USA, July 12-15, 2010.

[11] B. Clark, T. Deshane, E. Dow, S. Evanchik, M. Finlayson, J. Herne, and J. Matthews, “Xen and the Art of Repeated Research”, In *USENIX Technical Conference FREENIX Track*, pages 135144. USENIX Association, 2004.

[12] L. Cherkasova and R. Gardner. “Measuring CPU Overhead for I/O Processing in the Xen Virtual Machine Monitor”, In *USENIX 2005 Annual Technical Conference, General Track*, pages 387390. USENIX Association, 2005.

[13] J. E. Smith and R. Nair. The Architecture of Virtual Machines. Computer, 38(5):3238, 2005.

[14] J. Duell. “The Design and Implementation of Berkeley Labs Linux Checkpoint/Restart”, T*echnical Report LBNL-54941*, Lawrence Berkeley National Lab, 2002.

[15] J. S. Plank, M. Beck, G. Kingsley, and K. Li. Libckpt: “Transparent Checkpointing Under Unix”, Technical Report UT-CS-94-242, 1994.

[16] W. D. Gropp and E. Lusk, “Fault Tolerance in MPI Programs”, *International Journal of High Performance Computer Applications*, 18(3):363372, 2004.

[17] S. Sankaran, J. M. Squyres, B. Barrett, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman, “The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing”, *International Journal of High Performance Computing Applications*, 19(4):479493, 2005.

[18] Y. Zhang, D. Wong, and W. Zheng, “User-Level Checkpoint and Recovery for LAM/MPI”, *SIGOPS Oper. Syst*. Rev., 39(3):7281, 2005.

[19] G. Bronevetsky, D. Marques, K. Pingali, and P. Stodghill, “Automated Application-Level Checkpointing of MPI Programs”, In *PPoPP 03: Proceedings of the 9th Symposium on Principles and Practice of Parallel Programming*, pages 8494. ACM Press, 2003.

[20] A. Beguelin, E. Seligman, and P. Stephan, “Application Level Fault Tolerance in Heterogeneous Networks of Workstations”, *J. Parallel Distrib. Comput.*, 43(2):147155, 1997.

[21] SWSoft. OpenVZ - Server Virtualization, 2006. http://www.openvz.org/.

[22] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A.Warfield, “Xen and the Art of Virtualization”, In *SOSP 03: Proceedings of the 19th Symposium on Operating Systems Principles*, pages 164177. ACM Press, 2003.

[23] VMWare. VMWare, 2006. http://www.vmware.com.

[24] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive Fault Tolerance for HPC with Xen Virtualization. In ICS 07: Proceedings of the 21st annual International Conference on Supercomputing, pages 2332. ACM Press, 2007.

[25] J.P. Walters and V. Chaudhary, “Replication-Based Fault Tolerance for MPI Applications”, In *IEEE Transactions on Parallel and Distributed Systems*, 20(7):997-1010 (2009).

[26] The Message Passing Inteface (MPI)Standard. http://www-unix.mcs.anl.gov/mpi.

[27] R. Buyya, C.S. Yeo, and S. Venugopal, “Market-Oriented Cloud Computing: Vision, Hype, and Reality for Delivering IT Services as Computing Utilities”, Keynote Paper, in *Proc. 10th IEEE International Conference on High Performance Computing and Communications (HPCC 2008)*, IEEE CS Press, Sept. 2527, 2008, Dalian, China.

[28] Q. O. Snell, A. R. Mikler, and J. L. Gustafson, “NetPIPE: A Network Protocol Independent Performance Evaluator”, http://www.scl.ameslab.gov/netpipe.

[29] HPL A Portable Implementation of the High-Performance Linpack Benchmark for Distributed-Memory Computers. http://www.netlib.org/benchmark/hpl.