VCCP: A Transparent, Coordinated Checkpointing System for Virtualization-based Cluster Computing

Hong Ong[#1] Natthapol Saragol[*2] Kasidit Chanchio[*3] Chokchai Leangsuksun[+4]

[#]Computer Science and
Mathematics Division
Oak Ridge National Laboratory,
Oak Ridge, TN 37831, USA
[1]e-mail: hongong@ornl.gov

[*]Department of Computer Science
Thammasat University
Patumtani, Thailand 12121
[2]e-mail: off_natoff@yahoo.com
[3]e-mail: kasidit@cs.tu.ac.th

[+]Department of Computer Science
Louisiana Tech University
Ruston, LA 71272, USA
[4]e-mail: box@latech.edu

***Abstract*—Virtual machine, which typically consists of a guest operating system (OS) and its serial applications, can be checkpointed, migrated to another cluster node, and restarted later to its previous saved state. However, to date, it is nontrivial to provide checkpoint-restart mechanisms with the same level of transparency for distributed applications running on a cluster of virtual machines. To address this particular issue, we have created the Virtual Cluster CheckPointing (VCCP) system, a novel system for transparent coordinated checkpoint-restart of virtual machines and its distributed application on commodity clusters. In this paper, we detail the design and implementation of the VCCP system. Our VCCP prototype extends the open source QEMU system with kqemu module by implementing hypervisor-based Coordinated Checkpoint-Restart protocols. To verify and validate our prototype, we measured its performance using the NAS parallel benchmark. Our experimental results indicate that VCCP generates less than 1% of additional execution overhead for non-communication intensive parallel applications. Furthermore, our correctness analysis shows that VCCP does not cause message loss or reordering, which is a necessary property to ensure correctness of checkpoint-restart mechanism. Finally, we believe that VCCP is a promising checkpoint-restart alternative for legacy applications that have implemented traditional process-level checkpoint-restart.**

Keywords-coordinated checkpointing; virtualization; fault tolerance; high performance computing

I. INTRODUCTION

While commodity High Performance Technical Computing (HPTC) systems continue to scale in size of computing elements and overall computing powers, their performance/cost benefit is subject to its abilities to provide high reliability, availability, and transparency in utilizing the underlying computing resources. Recently, it has been shown that the number of system interruptions increases, due to hardware or/and software faults during application execution, as the number of processor cores increases [1]. Checkpoint-restart has been used in HPTC to provide fault tolerance by migrating application off faulty cluster nodes and restarting from the last checkpoint instead of from scratch. Other benefits are to improve resource utilization by being able to checkpoint resource-intensive jobs when load is high and restarting such jobs again later when the load is lower and improve service availability by checkpointing application processes before cluster node maintenance and restarting them on other cluster nodes so that applications can continue to run with minimal downtime. However, many scientific applications are distributed and run on multiple cluster nodes. For these applications, a checkpoint-restart mechanism needs to not only save and restore the application state associated with each cluster node, but it must also ensure that the state saved and restored across all participating nodes is globally consistent. Checkpoint and restart must be coordinated across all participating nodes to ensure that application processes running on each node are synchronized correctly. To date, it is nontrivial to provide highly portable as well as efficient coordinated checkpoint-restart mechanisms. This is primarily due to difficulties to integrate fault tolerant mechanisms to existing distributed applications and software systems. Many supportive software components must be created and substantial software modifications must be made to facilitate fault-tolerant mechanisms. Another reason is that the traditional checkpoint-restart approach is often non-transparent to applications and system software. Fault-tolerant mechanisms may have to be redesigned or reintegrated into the distributed systems when the applications or system software change. Consequently, it makes fault resiliency supports for HPTC environments become more complicated and harder to extend and use.

Contemporary virtualization technology has built-in mechanisms to save and restore virtual machine (VM) state. As a result, existing solutions took advantage of these built-in mechanisms to checkpoint-restart a virtual machine [2, 3, 4, 5, 6]. However, there are only a handful of these efforts directly address the issue of checkpointing distributed applications running on a cluster of VMs [7, 22]. To address this issue, our work takes a novel approach toward providing checkpoint-restart for distributed applications executing inside virtual machine on virtualization-based HPTC systems. Our solution includes leveraging contemporary virtualization software and implementing a separate management layer that aims to provide high-level of transparency to applications and operating systems.

In this paper, we present a novel *Virtual Cluster CheckPointing (VCCP)* system for performing coordinated checkpoint-restart of virtual machines. The VCCP system is

978-1-4244-5012-1/09/$25.00 ©2009 IEEE

based on the *Virtual Cluster Architecture (VCA)*, which defines the software stack of a virtualized cluster-computing environment. Our prototype extends the open source QEMU [2] system with the kqemu module by implementing the Hypervisor-based Coordinated Checkpoint-Restart protocols. The QEMU system with kqemu module is more efficient than the traditional QEMU system [23]. It is also worth to note that work is undergoing to port VCCP to Kernel-based Virtual Machine (KVM) [3]. As our implementation is realized at the hypervisor level, guest OS and applications do not require further modifications. To verify and validate our VCCP system prototype, we measured its checkpointing overhead performance using the NAS parallel benchmark. Our experimental results indicate that the VCCP system only generates less than 1% of additional execution overhead for non-communication intensive parallel applications. Furthermore, our correctness analysis shows that the VCCP does not cause any message loss or reordering, a necessary property to ensure correctness of checkpoint-restart mechanisms.

This paper makes the following contributions:

- A generalized layering VCA: We divide coordinated checkpointing task into subtasks and assign them to three layers: the virtual network, VCCP, and the virtual machine. Since our coordinated checkpoint-restart mechanism requires reliable and FIFO data transmission, the virtual network guarantees this property. The VCCP system implements the hypervisor-based coordinated checkpoint-restart protocols. Unlike the traditional process-level coordinated checkpointing approach, our implementation coordinates transmission of Ethernet frames rather than MPI messages on a checkpointing event. Finally, the virtual machine layer consists of the hypervisor, guest OS, and applications. The hypervisor is modified to interact with VCCP.
- A transparent checkpoint-restart system: We provide a fully transparent checkpoint-restart mechanism for distributed application executing on a cluster of virtual machines. The VCCP system provides checkpoint-restart services without modifying existing applications and guest OS's.
- Correctness analysis: We show that our protocols and the virtual network do not cause frame loss or reordering. Additionally, checkpoint-restart events usually generate clock skews, which may interfere with applications that require extensive clock synchronization or have short application-specific data transmission timeouts. We discuss solutions and show that our approach and the traditional process-level coordinated checkpoint-restart approach share similar clock synchronization and timeout constraints. Therefore, applications that work in the traditional approach could also work on our implementation.

The rest of this paper is organized as follows: We describe representative related works in Section II. Sections III presents the VCA design. We detail our prototype implementation and discuss our mechanism to save and restore virtual machine state in Section IV, and present preliminary experimental results in Section V. Section VI presents our correctness analysis. Finally, Section VII concludes the paper and indicates future works.

II. Related Works

There are various approaches for checkpointing a process. In general, it can be classified as user-level or system-level checkpointing.

User-level checkpointing is supported through either manually inserting the checkpointing logic inside an application source code or implicitly calling user-level checkpointing library routines. In the latter, developers, for example, can recompile their source code with libckpt [10] library or relink object files with the Condor library [11]. User-level checkpointing can also be used in conjunction with source code analysis tools [8, 9] to determine the appropriate places to insert checkpointing codes. System-level checkpointing is supported through kernel module or virtual machines. Software such as BLCR [12] and CRAK [13] provide kernel-level checkpoint-restart capabilities without modifying application executables. However, they typically rely on a particular version of Linux and thus less portable.

Checkpointing can also be accomplished via virtual machines through saving and restoring VM state. Software, such as QEMU [2] and the Kernel-based Virtual Machine (KVM) [3], supports full system-level virtualization. QEMU allows the execution of an unmodified guest OS, e.g., Windows or Linux, and all its applications. KVM is a modified version of QEMU to utilize hardware virtualization supports, i.e., Intel VT or AMD-V. Neither of them requires guest OS or application modifications. Xen [5], on the other hand, provides para-virtualization and requires its guest OS to be patched, i.e., Xenolinux, in order to interact with the Xen hypervisor. In our previous work [6], we proposed a checkpoint-restart mechanism based on full-virtualization. The work in [19] proposed the concept of virtual cluster for managing resources in Grid computing environments, but did not focus on the issue of fault tolerance. Our current work compliments their design in that we focus on providing fully transparent fault tolerance supports for virtual clusters.

Coordinated checkpointing is developed based on the global distributed snapshot concept [14] and is commonly used as a basic mechanism to provide fault tolerance for parallel processing. However, most checkpointing tools for parallel applications perform coordinated checkpointing at the *process-level* and rely on a particular version of operating system and/or MPI implementation. Exemplar works are such as CoCheck [15], BLCR [12], and MPICHV [17]. CoCheck implemented coordinated checkpointing protocols on top of MPI. The work in [16] implemented coordinate checkpointing protocol on top of LAM/MPI and uses BLCR to save and restore process state. The MPICHV system integrated blocking and non-blocking coordinated checkpointing into their MPI implementation. A difference between the coordinated checkpointing used in LAM/MPI system and MPICHV is the management of communication connections among processes during checkpointing events. In MPICHV, the connections among processes must be disconnected before each process saves its local state to a checkpoint file, and later re-established their connections before the processes resume computation, which could be costly. The LAM/MPI checkpoint-restart mechanism, on the other hand, leaves the connection on during a checkpointing event. It uses bookmarking mechanism to manage two communication end-points to guarantee message reliability during checkpointing coordination. Our approach also does not close connection during the coordination phase of checkpointing events. However, unlike [16], we further consider the negative effects of transmission timeout and clock skew problems (see Section VI).

Another major difference between our approach and the process-level approach is the flushing of in-flight data. Since the process-level approaches modify MPI to implement checkpointing coordination, all complete in-flight messages sent before checkpointing must be flushed. On the other hand, our approach flushes in-flight Ethernet frames rather than the whole message. Thus, our approach can start creating local checkpoint file earlier. Neither does our approach require large buffer to hold the flushed frames nor spend large amount of time storing them to disk (see Section VI).

The checkpoint-restart mechanism presented in [18] performs coordinated checkpointing on a cluster of Xen virtual machines connecting to one another via an Infiniband network. Its design is different from ours in that they need to run several Global Coordinator processes on the guest OS of every virtual machine to control the coordination of application processes over the infiniband network. Our VCCP system, on the other hand, implements coordinated checkpoint-restart mechanism inside the hypervisor hidden from applications and guest OSes. Therefore, VCCP do not requires any additional process on the guest OS.

The work in [22] intends to take a snapshot of the virtual network system that connects Xen virtual machines, namely the VIOLIN system. Since the mechanisms to take a snapshot of the virtual network are implemented inside the virtual machine and the virtual network, their work can perform checkpointing for parallel computation without applications or guest OS modifications.

Although this work and VCCP are both highly transparent to applications and system software, their designs are very different. While [22] bases on a non-blocking distributed snapshot algorithm and do not require reliable, FIFO data transmission, VCCP uses a blocking coordinated checkpointing algorithm and requires reliable, FIFO transmission. Due to its coordination algorithm, VCCP would have the overheads of capturing in-transit Ethernet frames and virtual machine coordination *before checkpointing*. The work in [22] does not suffer these overheads. However, since they sidestep the capturing of in-transit frames and do not require reliable transmission, their system seriously incurs the retransmission overheads *after checkpointing*. Such retransmission overheads would not occur in VCCP because in-transit frames are received and stored in VCCP's receive queue during the coordination.

Moreover, due to the non-blocking distributed snapshot algorithm used in [22], the checkpointing operations on every virtual machine will occur at different times. Therefore, clock skews among the virtual machines could be high depending on the occurrence of a checkpointing event on each virtual machine. On the other hand, clock skews in VCCP are relieved by its adoption of the two-phase commit protocol in its coordination mechanism. Based on our analysis in Section VI, clock skews occurred due to checkpointing can affect application-specific timeout conditions. We further discuss the relationship of clock skew values to the correctness of checkpoint-restart mechanisms in distributed environments in Section VI.

III. Virtual Cluster Architecture

This section describes our *Virtual Cluster Architecture*, a computing environment upon which VCCP operates. We detail the VCCP system and its protocols in Section IV. Each node of the *virtual cluster* consists of three communication layers as illustrated in Fig. 1. They are the *Virtual Network*, the *VCCP*, and the *Virtual Machine* layers.

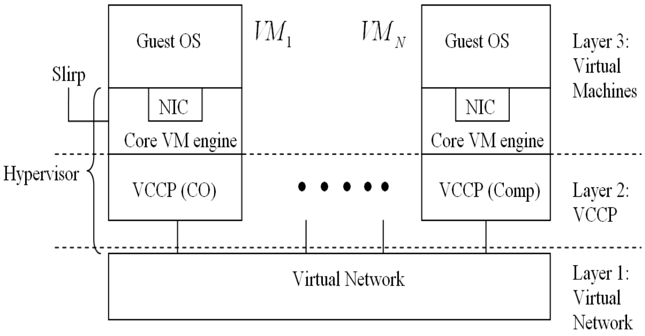


Fig. 1: Software layers in a virtual cluster environment

A. The Virtual Network Layer

The first layer is the *virtual network* layer. The virtual network layer must guarantee the reliability and FIFO property in order to work with the VCCP system. (Beside VDE, other network technology is also possible for implementing the virtual network layer.)

Fig. 4 shows our implementation of the *virtual network* layer using VDE switches [21], connected together using a star topology. Each VDE switch runs as a daemon process on every host computer in the virtual cluster environment. These switches work together to route messages between virtual machines. We configured one of the host computers to act as a central switch. All other VDE switches are connected to the central switch. A message must pass through this central switch before being forwarded to its destination. The connections between VMs and the virtual network stay the same throughout the virtual cluster's lifetime and remain intact during checkpoint-restart events.

We consider frame transmission between any two VMs to be reliable and FIFO for the following reasons. First, Ethernet frame transmission between VCCP and VDE are through the TAP interface and uses file descriptors, which are considered reliable and FIFO. Second, we configured every VDE switch not to drop data frames and forwards them over a pair of its input and output ports in order. Third, the VDE switches are connected to one another using *ssh* protocol. Finally, there is no loop or duplicate end-to-end paths between any two VMs in the VDE switch topology (a star). Thus, reliability and FIFO frame delivery between any pair of VMs are guaranteed.

B. The VCCP Layer

The second layer consists of our VCCP system. The VCCP implements the *hypervisor-level* checkpoint-restart mechanism, as opposed to the process-level ones. The VCCP monitors Ethernet frame transmission into and out of a virtual machine and handles message coordination for checkpoint-restart. The VCCP component is part of a hypervisor and serves as a gateway between a guest OS and the virtual network. We assume that every VCCP component knows the IP addresses of every virtual machine connecting to the virtual network. In a virtual cluster, one of the virtual machine is designated to be a head node of the cluster, while the rests are treated as compute nodes. There are two kinds of VCCP: the coordinator *VCCP(CO)* and the computing node *VCCP(Comp)*. The *VCCP(CO)* is installed at the head node of the virtual cluster and responsible for initiating and administrating checkpoint-restart operations. On the other hand, the *VCCP(Comp)* receives instructions from *VCCP(CO)* to perform checkpoint-restart operations on the compute nodes.

C. The Virtual Machine Layer

The third layer defines a set of virtual machines ($VM_1 ... VM_N$ where $N \geq 1$). Each VM has a hypervisor along with its guest OS and applications. In our design, the hypervisor consists of a *core VM engine* and the VCCP. The core VM engine performs original VM functions, while VCCP handles checkpoint-restart tasks. The guest OS sends and receives data though a virtual Network Interface Card (NIC). The head node requires two NICs: one connecting to the virtual network, and the other connecting to Internet via Slirp protocol.

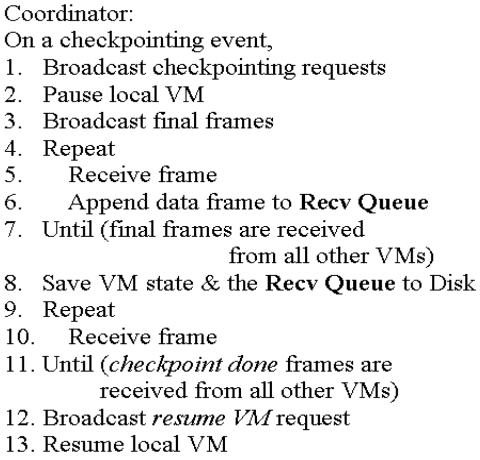
IV. Hypervisor-based Checkpoint-restart Protocols

This section details the VCCP system and its hypervisor-based checkpoint-restart protocols. VCCP maintains a *Recv* queue data structure to store incoming data frame from the virtual network. The *Recv* queue is an FIFO queue, i.e., the incoming data frames are appended to the end of the queue. Under normal operations, the hypervisor (with a modified I/O module to work with VCCP) will first obtain data frame from the *Recv* queue before obtaining new ones from the virtual network. During checkpoint-restart events, VCCP uses this data structure to ensure correct data communication (to be discussed in Section IV).

VCCP assumes the followings failure conditions: 1) the hypervisor could crash when failures occur, 2) the virtual network could crash and stop deliver frames if any of its components failed, 3) if crashes are detected, the virtual network and all hypervisors would eventually be terminated, and the virtual cluster would be restarted from the last set of checkpoints.

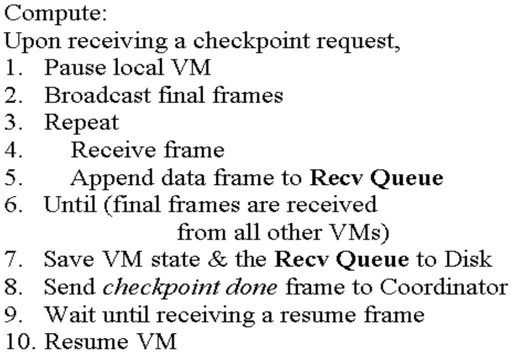
A. VM Checkpointing Protocol

Fig. 2a and Fig. 2b illustrate the pseudo-code of the VCCP checkpointing protocol. The algorithms show the operations of the coordinator and the compute node, labeled as *VCCP(CO)* and *VCCP(Comp)* in Fig. 1.



```
Coordinator:
On a checkpointing event,
Broadcast checkpointing requests
Pause local VM
Broadcast final frames
Repeat
    Receive frame
    Append data frame to Recv Queue
Until (final frames are received
                    from all other VMs)
Save VM state & the Recv Queue to Disk
Repeat
    Receive frame
Until (checkpoint done frames are
          received from all other VMs)
Broadcast resume VM request
Resume local VM
```

Fig. 2 (a): VM Checkpointing Protocol for VCCP (CO)



```
Compute:
Upon receiving a checkpoint request,
Pause local VM
Broadcast final frames
Repeat
    Receive frame
    Append data frame to Recv Queue
Until (final frames are received
                    from all other VMs)
Save VM state & the Recv Queue to Disk
Send checkpoint done frame to Coordinator
Wait until receiving a resume frame
Resume VM
```

Fig. 2 (b): VM Checkpointing Protocol for VCCP (COMP)

The VCCP(CO) initiates a checkpointing event by broadcasting a checkpointing request to every node in a virtual cluster. Upon receive of a checkpointing event, the VCCP(Comp) instructs its hypervisor to pause VM computation in order to preserve the VM internal state as well as to stop sending more data frames out to the network. The VCCP(CO) also does the same after the broadcast. Note that although the pausing helps synchronize every hypervisor, we are investigating another design that allows checkpointing to perform without pausing the VM in our future work.

Next, the checkpointing protocol flushes in-flight data frames off the network by allowing each hypervisor to broadcast its final data frame. Every hypervisor would then collect all incoming data frames until the final data frames from all other hypervisors arrive. Due to the reliability and FIFO property of our virtual network, every in-flight data frame would arrive at its destination before the final frames. The VCCP(Comp) at the destination then appends incoming data frames to the end of the *Recv* queue to prevent frame loss and preserve FIFO frame delivery.

After each hypervisor received final data frames from all other VMs, it performs a local VM checkpointing. Firstly, it saves the VM's internal state, as well as contents of the *Recv* queue to a checkpoint file. Secondly, it makes a snapshot copy of disk image that is consistent with the saved state. The size of the disk image is large in general. However, the stack file system can be used to substantially reduce the size of the disk image to be saved [20]. We briefly discuss the checkpointing of virtual machine state and disk image in the next section.

Finally, the VCCP(Comp) sends a "checkpoint done" frame to the VCCP(CO). Upon receiving all "checkpoint done" frames, the VCCP(CO) broadcasts the "resume VM" frames to instruct all VCCP(Comp) to resume VM computation. We will discuss the correctness of the checkpointing protocol in Section VII.

B. Saving and loading the Virtual Machine State

To checkpoint a virtual machine, we must store its internal state (including the VM's memory contents, caches, and operational data associated with every virtual devices) and disk image snapshot when a checkpointing event takes place. Most virtualization software such as Vmware, KVM, Xen, and QEMU has features to save the internal state of a virtual machine to a file and restore the state from a file back to the virtual machine. However, they do not provide a function to make a copy of the disk image. Making a copy of the entire disk image would also be impractical since the size of a disk image is usually large (hundreds of gigabytes or several terabytes).

Based on the work in [20] and our previous work [6], stack or union file systems can be used to solve the disk image replication problems. Following this approach, we configure the virtual machine's disk image to consist of two parts: the base image and the overlay image. The base image stores the original data that do not change over time. The overlay image keeps all changes made to data in the base image. This stackable disk image approach has already been in uses in QEMU's qcow disk image format.

Then, we have modified the hypervisor to perform the following VM checkpointing and restoration operations. When the hypervisor checkpoints a VM, it first pauses the VM, and then saves the internal state of the VM to a file, and finally makes a copy of the overlay image. The copied overlay image is basically the image containing all changes the VM made so far that is consistent with the saved internal state. The saved internal state and overlay image together represents a checkpoint.

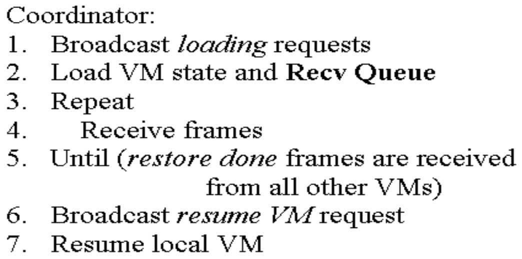
The local VM checkpointing performance would depend on the sizes of the internal state and the overlay image. The size of the internal state is dominated by the size of the physical memory of the VM. It could range from few hundred megabytes to few gigabytes. Methods such as Diskless checkpointing [24] and the work in [6] may be used to further reduce the cost of saving VM memory contents. However, they are not the focus of this work.

On the other hand, the size of the overlay image depends on disk usage characteristics of applications running on each VM. While the size may be large for disk intensive applications, it may be small for computational intensive ones. From experiences, we found the size and copying cost of the overlay image to be acceptable unlike those of the base disk image, which are too high.

To restore the VM, the hypervisor first copies the saved overlay image of a checkpoint over the current overlay image. Then, it loads data in the saved internal state stored in the checkpoint to the virtual machine and finally continue VM execution.

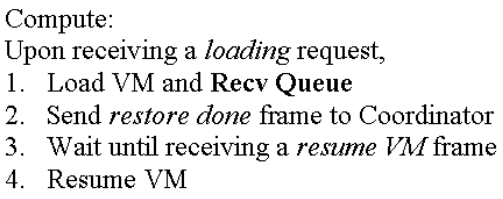
C. VM Restart Protocol

Once failure is detected, the virtual cluster's computation is stopped and made ready for recovery. We assume that the *Recv* queue is empty at this stage and the virtual network is operating properly. Each virtual machine would proceed to invoke the restart operation. Beyond this point, VCCP will append any incoming data frame to the end of the queue even though the VM state is not yet loaded or resumed. The pseudo-codes of the restart algorithms are shown in Fig. 3(a) and 3(b).



```
Coordinator:
Broadcast loading requests
Load VM state and Recv Queue
Repeat
    Receive frames
Until (restore done frames are received
            from all other VMs)
Broadcast resume VM request
Resume local VM
```

Fig. 3a: VM Restart Protocol for VCCP(CO)



```
Compute:
Upon receiving a loading request,
1. Load VM and Recv Queue
2. Send restore done frame to Coordinator
3. Wait until receiving a resume VM frame
4. Resume VM
```

Fig. 3b: VM Restart Protocol for VCCP(Comp)

The VCCP(CO) initiates the restart operation by broadcasting a *loading* request to all nodes (including that of the VCCP(CO)) in the virtual cluster. Upon receiving the request, every hypervisor loads it's VM state and the *Recv* queue from its last checkpoint. The saved contents of the *Recv* queue would be inserted in front of the existing *Recv* queue of the hypervisor. Since the VMs were paused before saving its state, the loaded VM would also be paused initially. Then, all VCCP(Comp)s send a "restore done" frame to the VCCP(CO). Upon receiving the "restore done" frame from all VCCP(Comp)s, the VCCP(CO) broadcasts a "resume VM" frame to all VCCP(Comp) to resume VM computation. The VCCP(CO) also resumes its VM computation locally.

V. EXPERIMENTAL RESULTS

We installed the modified QEMU (with the VCCP integration) on a 9-node Intel Xeon cluster running Rocks version 4.2.1 cluster toolkit. Each node has a dual Intel Xeon 2.6 GHz CPUs, equipped with 2 GB memory and a 40 GB hard disk. Nodes are interconnected via 1 Gigabit Ethernet network. We run a QEMU/w kqemu hypervisor on 8 nodes. We configured a hypervisor to run on each node and assigned 512 MB memory and 300 MB of disk image (using qcow2 format with "Copy On Write" mode enabled). We install Damned Small Linux version 3.4.1 as the guest OS on every VM.

In our virtual network configuration, each of the eight nodes contains a VDE switch daemon and a hypervisor. A central VDE switch is installed on the ninth node. Each VDE switch is configured to make a connection with the centralized switch as illustrated in Fig. 4.

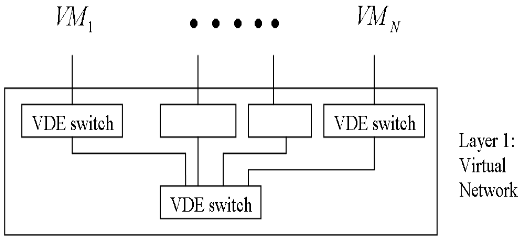


Fig. 4: A network of VDE switches

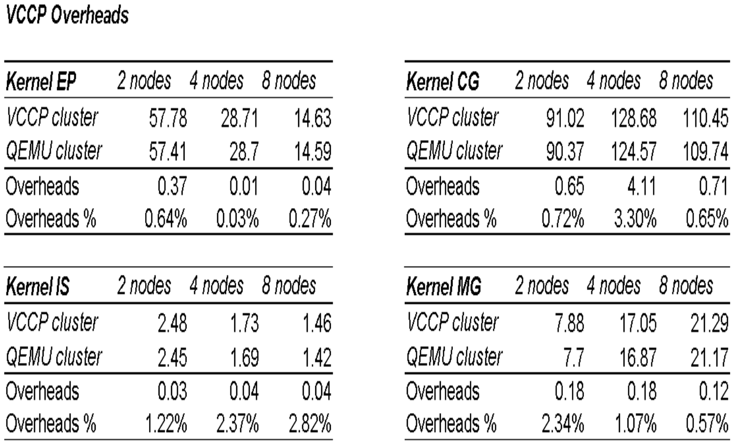
We use four application kernels from the NAS parallel benchmark and OpenMPI version 1.2.3 in our experiments. For simplicity, the binary executables of the benchmark and OpenMPI are copied to local disk images of each VM. Two sets of experiments have been conducted to measure performance and monitor operations of VCCP. Every data point is an average of 10 runs.

A. Overheads

The first set of experiment aims to investigate the overheads of VCCP by comparing the NAS benchmark execution time on a virtual cluster with VCCP implementation to another virtual cluster running traditional QEMU. Both configurations use a VDE network. There is no checkpointing or recovery event in this set of experiments.

Fig. 5 shows that the overheads of running four computation kernels in the VCCP cluster are not much different from that of the QEMU cluster. The execution times of each kernel running on VCCP cluster demonstrate similar characteristics as compare to those running on the QEMU cluster. It shows that the overheads are at most 3.3 percent and as low as 0.57 percents.

For kernel EP, VCCP cluster spent less than 1 percent additional execution time. This low overhead is due to minimum data communication. As the number of processor increases, the execution time of kernel EP decreases. This result shows similar behavior as running kernel EP on real machines. On the other hand, for those communication intensive kernels, the current implementation of the virtual network reveals the following two problems:



VCCP Overheads

Kernel EP	*2 nodes*	*4 nodes*	*8 nodes*
VCCP cluster	57.78	28.71	14.63
QEMU cluster	57.41	28.7	14.59
Overheads	0.37	0.01	0.04
Overheads %	0.64%	0.03%	0.27%

Kernel CG	*2 nodes*	*4 nodes*	*8 nodes*
VCCP cluster	91.02	128.68	110.45
QEMU cluster	90.37	124.57	109.74
Overheads	0.65	4.11	0.71
Overheads %	0.72%	3.30%	0.65%

Kernel IS	*2 nodes*	*4 nodes*	*8 nodes*
VCCP cluster	2.48	1.73	1.46
QEMU cluster	2.45	1.69	1.42
Overheads	0.03	0.04	0.04
Overheads %	1.22%	2.37%	2.82%

Kernel MG	*2 nodes*	*4 nodes*	*8 nodes*
VCCP cluster	7.88	17.05	21.29
QEMU cluster	7.7	16.87	21.17
Overheads	0.18	0.18	0.12
Overheads %	2.34%	1.07%	0.57%

Fig. 5: Overheads of Kernel EP, CG, IS, and MG of the NAS parallel benchmark

Resources contention between VM and VDE daemon: While running communication intensive kernels, the VM and VDE daemon reports that socket resources are temporarily unavailable periodically. This occurs because VM or VDE send data frames to sockets too quickly and thus overload the network buffers. Since the VM consumes most CPU time during the execution of these kernels, VDE may not get enough CPU time to drain the network buffer.

Hotspot at the central VDE switch: Due to the star topology, the bottleneck occurs at the central VDE switch while running communication intensive applications.

Fig. 5 shows the execution time of both CG and MG grows as the number of VM node increases. This is because of the inefficiency of VDE network configuration used in our current implementation. As a result, the communication overheads of the CG and MG kernels, which are communication intensive, become unusually high as the number of nodes increase. In particular, the CG uses irregular long distance communication, while the MG uses short and long distance, regular communications. On the other hand, the IS performance does not exhibit such behavior. In fact, the IS kernel execution time is shown to be decreasing. This is because the amount of data frame transmission among VM nodes is not as high as in the cases of CG and MG. Work is underway to improve our virtual network design and configuration.

B. *Checkpointing Performance*

We instruct the VCCP(CO) to perform coordinated checkpointing on 2, 4, and 8 VM nodes, respectively. We measure every stage of the checkpointing protocol based on the progress of the coordinator. We define the checkpointing delay time as follow:

Checkpointing Delay = Flush Time + Save VM + Save Frames + Wait

The *Flush time* denotes the time interval since the coordinator broadcasts checkpointing requests until it receives final frames from every VM node. The *Save VM* denotes the amount of time the coordinator used to save its VM state to a checkpoint file. The *Save Frames* indicates the amount of time the coordinator used to save frames in *Recv* queue to the checkpoint file. Finally, we denote *Wait* to represent the waiting time after the coordinator finishes saving the *Recv* queue until it receives the checkpoint done frames from every compute VM and resume its VM computation.

Figs. 6 (a)-(d) show the measurements of the NAS benchmarks. The X-axis indicates the number of VM nodes used while the Y-axis shows the checkpointing delay in seconds. The flushing time of every kernel demonstrates similar behavior. The flushing time increases as the number of VMs in the virtual cluster increases.

Majority of time is spent in saving VM state to a checkpoint file. The amount of time ranges from 7.54 seconds to 10.9 seconds. It represents approximately 75% to 82% of the overall checkpointing delays. This is because the time used to save VM state depends mostly on the physical memory size of the VM. Since we assigned only 512MB, their saving times in our experiments are not much different from one another. The time used to save flushed frames (*Save Frame*) from the *Recv* queue to a checkpoint file is hardly noticeable as compare to *Save VM* since there are not too many frames in the queue. Among the kernels, EP has the smallest number of frames in the queue. MG, on the other hand, spends 0.09 seconds time saving frames due to its communication intensive characteristic. Since our approach stores Ethernet frames rather than the whole message in the *Recv* queue, it only takes small amounts of time to save the queue's contents.

The *Wait* time roughly represents the amount of time the coordinator VM waits (after local checkpointing finished) for the last VM to finish its local checkpointing, plus the time *checkpoint done* frame travels from the last saved VM to the coordinator VM. It can roughly represent the timing discrepancies between the coordinator and other VMs. Fig. 6 shows that the wait time is higher as the number of VM nodes in the cluster increases. This is because the amount of time used to save local VM on each node varies more as the number of VM nodes increases.

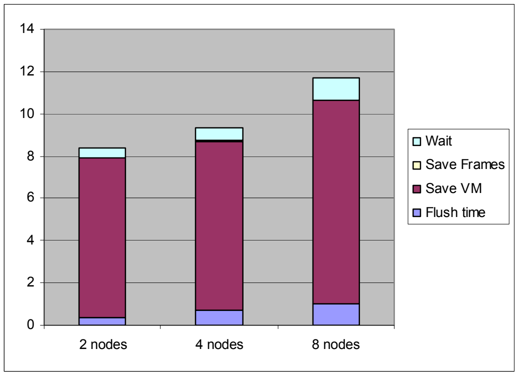


Fig. 6 (a): Kernel EP

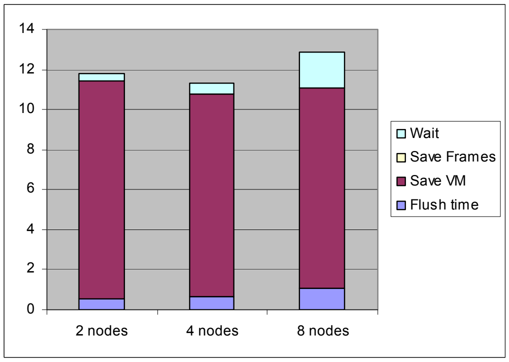


Fig. 6 (b) Kernel CG

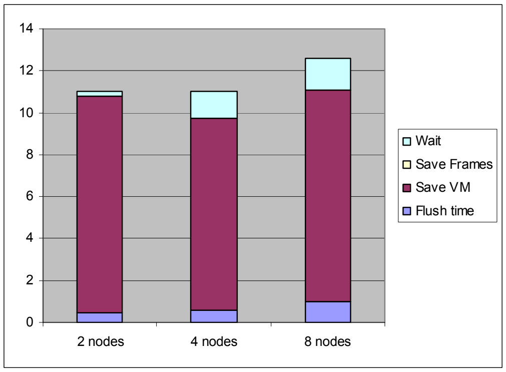


Fig. 6 (c) Kernel IS

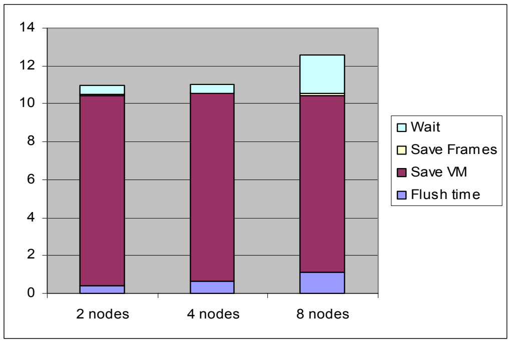


Fig. 6 (d) Kernel MG

VI. Correctness and Limitations

This section discusses the correctness of our checkpoint-restart protocols, presented in Section IV. Our aim is to assert that their operations are correct and do not violate the computation logic of the guest OS or applications of any virtual machine in the virtual cluster. In other words, the virtual machines must produce correct results in an environment where our coordinated checkpoint-restart protocols are enabled. Our correctness analysis also leads us to conclude certain facts, which in turn defines the limitations of our protocol for some classes of applications.

Considering the effects of VCCP on virtual machines' computation, there are potentially two possibilities that may lead to unexpected results: data communication and clock skew among virtual machines.

A. Reliable and FIFO Communication

Although the virtual network guarantees reliable and FIFO data transmission in a normal VM computation, we show here that the same conditions hold in the presence of checkpointing or recovery events. In this section, we show:

The checkpointing and recovery protocols do not cause a frame loss or violation of FIFO property.

The checkpointing protocol may lose data frame from any VM either prior to and/or after a checkpointing event. We will show here that none of that is true.

The protocol does not lose data frames *prior to* checkpoint-restart events. This is because (a) the checkpointing protocol stores every in-flight frame in *Recv* queue and (b) the FIFO property of the virtual network guarantees that frames sent to a particular VM must arrive at its destination before the final data frame. Moreover, after the final frames are broadcasted by every VM, the VM computation would be paused. Thus, there are no frames sent to network *while* the VMs perform checkpointing.

The checkpointing protocol also does not lose data frames *after* a checkpointing event. After the event, every VM would resume its computation and a VM could start sending frames. Since the *resume VM* frame may arrive at compute VM nodes at different time, VM computation on those nodes would resume at different times as well. If one VM sends data frames to another that has not yet resumed, those frames could be lost. However, this situation could not happen because VCCP always receives incoming data frames and stores them in the *Recv* queue throughout the checkpointing event. The destination VM will eventually get the data frames after it resumes. Thus, no data frame can be lost.

Likewise, the recovery protocol do not cause a frame loss *when* it operates because (a) the protocol loads the contents of the *Recv* queue from the checkpoint file before resuming VM computation and (b) every loaded VM would initially be in a pause state waiting for a *resume VM* frame from the coordinator. Thus, there would be no data frame transmission during the recovery operation. Also, there is no loss of data frame *after* a recovery operation. It is worth to note that in our design the hypervisor would always append new data frames from the network to the end of the *Recv* queue. After the recovery protocol operates, every VM node in the virtual cluster may restart at different time depending on the arrivals of the *resume VM* frames. In case some VM nodes send data to other nodes that have not resumed computation yet, the transmitted data will always be stored in the *Recv* queue and, thus, no data loss occurs.

Additionally, our protocols guarantee data frames, sent before and after a checkpoint or restart event, are always received in order. For a checkpointing event, the FIFO property holds because VCCP always appends in-flight frames to the *Recv* queue in order during a checkpointing event. Likewise, on a recovery event, the property holds because data frames transmitted before the checkpointing event (stored in the checkpoint file) would be loaded and inserted in front of the *Recv* queue, while data frames transmitted among VMs during or after the recovery event will be appended to the end of the *Recv* queue. Thus, the VM will always receive data in order.

B. Clock Skew and Communication Timeout

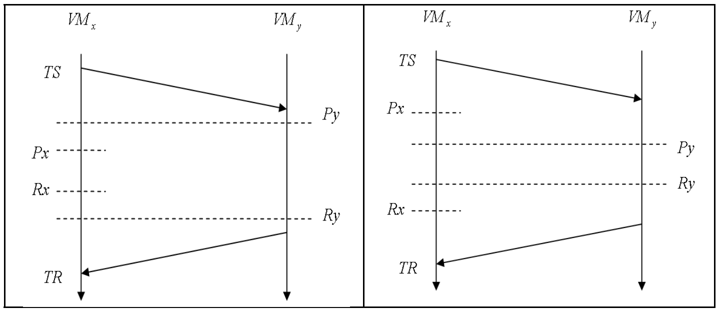
This section discusses how our protocols affect applications that rely on communication timeout and suggests a policy to deal with the clock skews issue. We assume that every VM has its own independent *virtual clock* that stops when the VM pauses and starts when the VM resumes.

The checkpoint and restart protocols can cause clock skew among virtual machines. The threes control frames of the checkpoint and restart protocols, 1) the checkpointing request frame, 2) the resume VM frames (in both protocols), and 3) the loading VM frames, can arrive at the destination VMs at different time. Thus, the VMs would pause or resume at different time causing clock skews among them. Although we base our protocol on the two-phase-commit protocol to keep the clock skews low, the clock skews would always exist.

There is another important fact about the interference of checkpoint and restart events to normal program execution that must be discuss. We show that:

A checkpointing or recovery event can affect Round Trip Time (RTT) between a pair of VM (based on the clock of the sender VM) and may cause communication timeout.

Supposed that there are two VMs, namely VM_x and VM_y, communicating with one another within a virtual cluster. Each VM starts computing at the same time with the same initial clock value and clock progression rate.



(a) (b)

Fig. 7 Effects of our protocols on round trip time and timeout decision

When VM_x sends out a frame to VM_y and wait for an acknowledgement, if a checkpointing event occurs while the sender is waiting, clock skew would occur. Let T_Δ represents the time difference between the two VMs. We will show that there exist situations where T_Δ can affect the round trip time reported by the sender VM (denoted by RTT_S) and may cause communication timeout. We will also show that T_Δ, RTT_S, and RTT_{real} (the real amount of time that the two VMs spent to send a frame and response with an acknowledgement) are related. Note that the RTT_{real} can also be considered as the round trip time when the two VMs run normally without checkpoint-restart events. Fig. 7 shows two of such situations. Let TS be the time a frame is sent from VM_x, TR be the time an acknowledgement frame is received, Px be the time VM_x is paused, Py be the time VM_y is paused, Rx be the time VM_x is resumed, and Ry be the time VM_y is resumed. All of them are real time. From Fig. 7(a), we can conclude the followings.

$$RTT_s = RTT_{real} + T_\Delta \tag{1}$$

where $RTT_{real} = (TR - Ry) + (Py - TS)$, (2)

$$T_\Delta = (Ry - Rx) + (Px - Py) \tag{3}$$

Thus, if T_Δ has a positive value, the RTT base on the sender's clock or RTT_S would be greater than the real RTT of the message or RTT_{real}. On the other hand, if T_Δ is negative, the RTT_S would be lesser than RTT_{real}. The T_Δ would have positive value in Fig. 7(a), and negative value in Fig. 7(b), respectively. It could also be 0 which indicates there is no clock skew occurring between VM_x and VM_y.

Certain applications running on VM_x may consider communication with VM_y to be timed out if RTT_S is greater than a timeout threshold value. Therefore, when T_Δ is high enough, timeout may occur as the result of a checkpointing event. Similar analysis can be applied to the recovery protocol.

C. Limitations and Solutions

The correctness analysis has led us to discover certain limitations of our VCCP protocols. It is worth to note that these limitations also exist in the traditional process-level checkpointing.

VCCP may not be suitable for applications that are sensitive to clock skews. This limitation also occurs in process-level checkpointing. However, the impact is much less in VCCP. This is because the increasing clock values on every process, due to the occurrence of a checkpoint or restart event, could be higher than in the VCCP approach since the whole checkpointing or recovery delays are included in application execution time.

There are techniques to avoid the clock skew limitation. For example, applications may use Lamport's logical clock for its computation rather than the VM clock. Alternatively, NTP daemons could be deployed to synchronize clocks on every VM in the virtual cluster. These solutions are, however, application-specific and out of the scope of this paper.

The limitation regarding the effects of checkpoint-restart events on communication timeouts also applies to process-level checkpointing approach. In fact, the RTT_S under the process-level approach would include the whole checkpointing delay time on the sender, which could be much more than T_Δ in VCCP approach.

We consider two different ways to implement timeout detection scheme. First, programmers may incorporate timeout detection schemes into their codes. Thus, the timeout threshold value is application-specific. On the other hand, applications may use communication software such as TCP and MPI that have timeout mechanisms built-in. In this case, the threshold values are those defined by guest OSes or the communication software.

A solution to avoid the timeout problem is to adjust the application-specific or guest OS timeout threshold values to be higher than the ones used in the non-checkpointing environment. In order to anticipate possible time increase due to checkpointing or restart operation, the new values should be at least T_Δ higher in theory. The same technique to increase timeout threshold is also used for process-level checkpointing. Due to time limitation, we leave the estimation and optimization of the T_Δ value for future works.

Finally, with the proposed solution, we argue that, a large class of HPC applications that work with traditional process-level checkpointing would also work under VCCP. This is because both our approach and the process-level checkpointing approach share similar sets of clock synchronization and timeout constraints, which can be managed using above solutions.

VII. Conclusions

Our work takes a new direction toward the design and implementation of fault tolerance systems for HPTC. In this paper, we proposed a layering Virtual Cluster Architecture and the VCCP software system. Our design goal is to provide high transparency by hiding coordinated checkpoint-restart mechanism from OSes and applications.

We implemented a VCCP prototype based on QEMU, and run experiments using the four kernels from the NAS parallel benchmark. The experimental results showed low overheads under a VCCP enabled cluster as compared to a QEMU cluster without checkpointing operations. We found that the checkpointing performance largely depends on the number of VM nodes in the cluster, the local VM state saving time and the imbalance of local saving state operations on different nodes. The local saving time is large as compared to other delaying factors, and affected by the sizes of VM's memory rather than by the number of VM nodes. In terms of parallel processing performance, we observed that our current VCCP implementation works well with computation intensive applications, while performing poorly on the communication-intensive ones. Our investigation revealed that the poor performance was due to the inefficiency of our VDE network configuration rather than the VCCP implementation.

We also discussed the correctness of our work and its limitations. Although clock skews may cause problems to certain applications, our analysis shows that the VCCP approach is applicable to applications that work under process-level coordinated checkpointing.

For future work, we plan to improve the communication among VMs by creating direct connections among VMs instead of using VDE with the star topology.

Acknowledgments

The submitted manuscript has been authored by a contractor of the U.S. Government under Contract No. DE-AC05-00OR22725 and DE-FG02-08ER25836 and National Science Foundation Contract No. CNS-0834483.

[1] Schroeder, B., Gibson, G.A., "Understanding failure in petascale computers." SciDAC 2007. To appear in the Journal of Physics: Conf. Ser. 78.

[2] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In Proceedings of the USENIX 2005 Annual Technical Conference, FREENIX Track, pp. 41-46, 2005.

[3] Avi Kiviti et al., kvm: the linux virtual machine monitor, In Proceedings of the Linux Symposium, 2007.

[4] VMware, Inc. http://www.vmware.com.

[5] B. Dragovic et al., Xen and the Art of Virtualization. In Proceedings of the ACM Symposium on Operating Systems Principles, October 2003.

[6] K. Chanchio et al., An Efficient Virtual Machine Checkpointing Mechanism for Hypervisor-based HPC systems, in Proc. of the High Availability and Performance Computing Workshop (HAPCW), Mar. 2008 (http://ft.cs.tu.ac.th/CEVM/)

[7] Huang et al. High Performance Virtual Machine Migration with RDMA over Modern Interconnect, in the proceedings of IEEE Cluster Computing, 2007.

[8] K. Chanchio and X. H. Sun, Data collection and restoration for heterogeneous process migration", SOFTWARE--PRACTICE AND EXPERIENCE, 32:1-27, April 15, 2002.

[9] Ferrari, A., Chapin, S. J., and Grimshaw, A. 2000. Heterogeneous process state capture and recovery through Process Introspection. Cluster Computing 3, 2 (Apr. 2000), 63-73.

[10] J. S. Plank, M. Beck, G. Kingsley, and K. Li, "Libckpt: Transparent Checkpointing under Unix", In Proceedings of the 1995 Winter USENIX Techincal Conference, 1995.

[11] M. Litzkow and M. Solomon, "The Evolution of Condor Checkpointing", 1998.

[12] J. Duell, P.Hargrove, and E. Roman, "The Design and Implementation of Berkeley Lab's Linux Checkpoint/Restart", 2002.

[13] H. Zhong and J. Nieh, "CRAK: Linux checkpoint / restart as a kernel module", Technical Report CUCS-014-01, Department of Computers Science, Columbia University, 2001.

[14] K.M. Chandy and L. Lamport, "Distributed Snapshots: Determining Global States of Distributed Systems", ACM Transactions on Computing Systems, 3(1): 63-75, 1985.

[15] G. Stellner, "CoCheck: Checkpointing and Process Migration for MPI", In Proceedings of the 10th International Parallel Processing Symposium, Honolulu, HI, 1996.

[16] Sriram Sankaran et al.. "The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing", In LACSI Symposium, October 2003. (publication LBNL-53808 Proc.).

[17] Camille Coti et al., "Blocking vs. non-blocking coordinated checkpointing for large scale fault tolerant MPI", In Int. Conf. For High Performance Computing, Networking, Storage and Analysis (SC2006), Tampa, USA, November 2006, IEEE/ACM, 2006.

[18] Daniele Paolo et al., "Transparent System-level Migration of PGAS Applications using Xen on InfiniBand", IEEE Cluster 2007 conference, 2007.

[19] Foster, I. et al., "Virtual Clusters for Grid Communities", Cluster Computing and the Grid, 6th International Symposium, 2006.

[20] Geoffroy Vallee et al., "Checkpoint/Restart of Virtual Machines Based on Xen," in Proc. of the High Availability and Performance Computing Workshop (HAPCW), 2006

[21] Renzo Davoli, "VDE: Virtual Distributed Ethernet," the IEEE 1st Intl. Conf. on Testbeds and Research Infrastructures for the Development of Network and Communities, 2005

[22] Ardalan Kangarlou et al, "Takings Snapshots of Virtual Networked Environments", in Proc. of 3^{rd} international workshop on Virtualization technology in distributed computing, 2007

[23] http://en.wikipedia.org/wiki/Comparison_of_platform_virtual_machines

[24] James S. Plank, Kai Li and Michael A. Puening, "Diskless Checkpointing", IEEE Transactions on Parallel and Distributed Systems, 9(10), October, 1998, pp. 972-986.