

Checkpoint/Restart of Virtual Machines Based on Xen

Geoffroy Vallée, Thomas Naughton, Hong Ong and Stephen L. Scott *
Oak Ridge National Laboratory
Computer Science and Mathematics Division
Oak Ridge, TN 37831, USA
{vallegr, naughtont, hongong, scottsl}@ornl.gov

Abstract

System level virtualization provides several advantages: (i) customization is eased since virtual machines may be based on different systems; (ii) virtual machines are isolated from hardware, subsequently applications are isolated via the virtual machines; (iii) basic fault tolerance mechanisms – pro-active fault tolerance through virtual machine migration and virtual machine snapshot/restore; and (iv) basic load balancing mechanisms – the capability to move and stop virtual machines running in the system. However, the current Xen implementation does not natively provide mechanisms for virtual machine checkpoint/restart.

This document presents the design of a reactive fault tolerant system, based on a checkpoint/restart mechanism for Xen virtual machines. We present the infrastructure for the management of virtual machines' checkpoint data as well as challenges for the implementation of a virtual machine checkpoint/restart mechanism based on Xen.

1 Introduction

System level virtualization provides several advantages: (i) *customization* is eased since virtual machines may be based on different systems; (ii) virtual machines are *isolated* from hardware, subsequently applications are isolated via the virtual machines; (iii) basic *fault tolerance* mechanisms – pro-active fault tolerance through virtual machine migration and virtual machine snapshot/restore; and (iv) basic *load balancing* mechanisms – the capability to move and stop virtual machines running in the system.

The Xen Hypervisor provides mechanisms that allow users to save a Xen virtual machine's (VM) exe-

cution state, stop a VM, and move a VM to a remote node [2]. However, Xen currently assumes that the disk image, i.e., the VM file system, is unique for a given VM and is accessible by compute nodes participating in VM migration. This assumption implies that Xen is not suitable for providing a real checkpoint/restart mechanism today, i.e., it is not possible to take a complete snapshot of a VM (the file system is not checkpointed). Therefore, after saving a checkpoint, the VM restart requires the checkpoint file and VM disk image be at a consistent state to avoid corruption/errors.

This paper presents the design of a *reactive fault tolerant system*, based on the checkpoint/restart of Xen virtual machines. We present the infrastructure for the management of virtual machines' checkpoints as well as challenges for the implementation of a VM checkpoint/restart mechanism.

The remainder of this paper is organized as follows: Section 2 presents the terminology used in the document; Section 3 presents an overview of Xen; Section 4 presents the infrastructure for the management of Xen VM checkpoints; Section 5 presents an approach for the checkpoint/restart of Xen VMs; and finishes with concluding remarks in Section 6.

2 Terminology

The execution of a virtual machine (VM) implies that one or more virtual systems are running concurrently on top of the same hardware, each having its own view of available resources. The virtualization may take place at different levels in the software stack, e.g., high-level language VMs (Java Virtual Machine), system-level VMs (Xen, Qemu) [5]. The level in the software hierarchy where the virtualization occurs influences the transparency and performance overhead.

The system-level virtualization incorporates a management facility called a *Hypervisor*, which oversees VMs on a *host machine*. The VMs run on the host

*ORNL's work was supported by the U.S. Department of Energy, under Contract DE-AC05-00OR22725.

machine with the Hypervisor managing the concurrent VM execution and resource mapping between virtual and real hardware. The Hypervisor is also called a *Virtual Machine Monitor* (VMM).

The Hypervisor is typically a small system running alongside of the VMs that does not include device drivers and specific mechanisms for access to the physical hardware (i.e. network cards, hard drives, etc.). Therefore, the Hypervisor is coupled with a traditional operating system (OS), referred to as the *host OS*. Users may log into the host OS to manage the physical machine and for management of the VMs (via a Hypervisor supported interface). The OS running in the VMs is called the *guest OS*. The guest OS only has access to resources that have been allocated to the VM by the Hypervisor.

Two approaches are possible to implement a virtualization solution: (i) full virtualization, for which the system is *fully virtualized*, VM are similar to real machines, no modification of host OSes and guest OSes are necessary; and (ii) *para-virtualization*, for which both host OSes and guest OSes are modified in order to improve scalability and performance. Furthermore, new generations of processor also include hardware virtualization support, e.g., VT Intel technology and AMD-V technology [1, 4, 6], and in that case, para-virtualization solutions may be extended to become a full-virtualization support (no modifications of OSes).

For instance, Xen is initially a para-virtualization solution but may be used without modifications of OSes with the virtualization support at the hardware level.

3 Xen Overview

As we saw in the previous section, Xen is initially a para-virtualization solution. That typically means that it requires the OS be “ported” to run on the para-virtual machine platform [1, 7]. The motivation being to improve scalability and performance. For instance, a new “software architecture” has been added to the Linux kernel in order to interface with the Xen Hypervisor.

To guarantee resource isolation, the guest OS does not directly access physical resources. For that, Xen changes the management of the execution of privileged instructions at the processor level. For instance, the x86 architecture supports four different privileged execution modes on the processor, called rings. Ring 0 allows full access to the hardware (privileged mode); ring 3 avoids the execution of privileged instructions (typically used for application execution). In a traditional operating system, the kernel is running in the

first ring (ring 0) for full access to the hardware. In the case of Xen, the Hypervisor runs in ring 0 and other operating systems (both guest OSes and host OS) are run in ring 1. This means that if an OS wants to execute a privileged instruction (e.g., during a page fault) the kernel has to go through the Hypervisor.

The Hypervisor, in conjunction with the host OS, provides system management utilities for users to manage VMs. Take VM creation for instance, when the command is sent to the Hypervisor, it sets up the virtual hardware for the VM, the Hypervisor allocates memory to the VM, loads the kernel into the virtual address space and then the VM boots like a real machine. It is important to note that the boot sequence is not complete, the hardware detection by the BIOS and all associated functionalities (like PXE) are not supported.

Xen provides several mechanisms that can be leveraged for fault tolerance techniques: (i) VM migration and (ii) VM pause/unpause.

The implementation of VM migration [2] uses the following algorithm:

1. check if the image of the VM is accessible on the remote node (through a distributed file system for instance),
2. reserve resources on the remote node,
3. start a lazy copy of the VM memory to the remote node (the VM is still running),
4. when most of the memory is copied, stop the VM and end the transfer of the VM to the remote node.

It is important to note that the disk image of the VM (its file system) is not transferred during the migration; Xen assumes the image is available from all the host machines.

The implementation of the VM pause/un-pause mechanism is quite similar:

1. change the VM identifier to avoid conflicts with other VM management task,
2. stop the VM,
3. dump the VM memory into a file,
4. flush I/Os.

As with the standard migration approach, the VM’s disk image is not saved during the process, instead a reference to the initial image is used when un-pausing (resuming) the stopped VM on the new node.

4 Virtual Machine Checkpoint/Restart Manager

The hypervisor is responsible for the checkpoint and restart of a virtual machine. However, the hypervisor works in concert with the host OS to access resources to actually carry out the process, i.e., writing checkpoint to disk, send/receive data on network. Therefore the checkpoint/restart mechanism for virtual machines is composed of two parts: (i) the hypervisor checkpoint mechanism, and (ii) the checkpoint-manager and resource-manager that run on the host OS (see Figure 1).

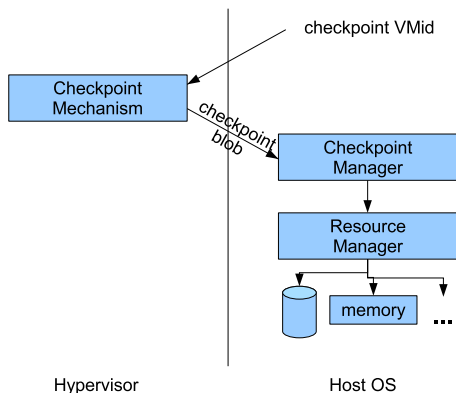


Figure 1. Design of VM Checkpoint Mechanism

4.1 Checkpoint Manager

The checkpoint coming from the Hypervisor is a raw checkpoint saving the entire VM image. Before storing a checkpoint, it may be interesting to modify this checkpoint. For example, if we want to store a checkpoint on a remote disk, we can compress the checkpoint in order to speed up the transfer to the remote node. The *Checkpoint Manager* (CM) is responsible for preparing the checkpoint for storage, to include modifications as mentioned above.

Typically the CM uses standard Xen interfaces to checkpoint the VM’s data. However, it uses additional mechanisms for file system checkpointing, which is detailed in Section 5.

4.2 Resource Manager

Once a checkpoint is ready for storage, the system has to access a hardware resource. The *Resource Manager* (RM) is responsible for these aspects of the system. The storage may be local (e.g., local disk, local

memory) or remote (e.g., remote disk, remote memory) to the VM that is being checkpointed. Additionally, checkpoints may have to be stored on several resources, e.g., remote memory for performances and remote stable storage for persistence. Furthermore, checkpoints may come from the local CM but also from remote CM when the request is a remote memory storage or a remote disk storage. Therefore the CM has to manage checkpoints coming from both the local host OS and remote host OSes. For that, a Resource Manager will abstract the storage method from the CM (see Figure 2).

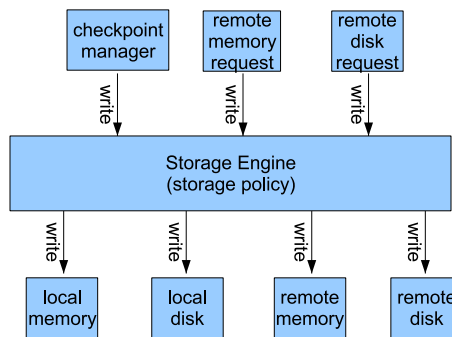


Figure 2. Design of resource Manager

For that, the RM is composed of multiple components, each of them being dedicated to a specific resource access (for instance local vs. remote, memory vs. disks). When a checkpoint is received, in order to identify the correct component that can store the checkpoint, the RM sends a request to all components. If a checkpoint characteristics match a component requirements, the component saves the checkpoint. For instance, for a checkpoint coming from a remote node and that has to be stored in memory, only the “local memory” component replies to the request; the RM stores then the checkpoint in memory. This approach enables dynamic management of resources since RM components may be activated/deactivated according the resource availability.

That means that at the RM level a checkpoint is identified by:

- a VM identifier (VMid),
- a checkpoint identifier (checkpoint_ID),
- an hypervisor identifier (Hypervisor_ID), which is actually similar to a node identifier since only one hypervisor is running per node and each of them as a unique identifier,
- a source component identifier (component_ID), used to identify a component during a remote re-

Function Name	Description
save_checkpoint_in_memory	Save a checkpoint in the local memory
save_checkpoint_on_disk	Save a checkpoint on the local disk
send_checkpoint_to_remote_node	Send a checkpoint to a remote node
get_checkpoint_vm_id	Get the VM id from a checkpoint
get_checkpoint_hypervisor_id	Get the Hypervisor id from a checkpoint
get_checkpoint_component_id	Get the component id from a checkpoint
get_local_hypervisor_id	Get the local Hypervisor id
get_local_component_id	Get the local component ID

Table 1. Interface for the Implementation of New Policies for Checkpoint Management

quest, i.e., identify what we have to do with a checkpoint coming from the network.

The definition of such characteristics eases the implementation of new checkpoint policies. For that, an interface allowing developers to access a checkpoint’s information but also local information has been defined and may be used to implement new checkpoint policies. Table 1 presents this API.

Example The following shows a checkpoint coming from the network:

```
VMid = 1
checkpoint_ID = 1
hypervisor_ID = 1 (also called
  checkpoint_hypervisor_ID)
component_ID = REMOTE_MEMORY_STORAGE (also called
  checkpoint_component_ID)
```

The RM then opens the following component:

```
hypervisor_ID = 2 (ID of local hypervisor)
component_ID = MEMORY_STORAGE
policy: see pseudo-code of Listing 1
```

Listing 1. Example of Policy for Checkpoint Management

```
hypervisor_id = get_checkpoint_hypervisor_id ();
cpt_id = get_checkpoint_component_id ();
if (hypervisor_id != get_local_hypervisor_id ())
  && (cpt_id == REMOTE_MEMORY_STORAGE)
then
  save_checkpoint_in_memory ();
else
  refuse_checkpoint ();
```

In example Listing 1, the checkpoint is coming from a remote node and is saved to the host machine’s local memory.

4.3 Summary

We presented the design of a framework for the management of Xen virtual machines in a distributed environment. The standard Xen VM checkpoint is primarily the dumping of a VM’s memory image. Section 5 presents additions to the basic Xen mechanism, which enable more effective checkpoint/restart mechanisms for Xen VMs.

The framework allows users to access multiple checkpoint storages (typically memory or disks), which allows users to take benefit of different checkpointing policies, e.g., memory for volatile checkpoints, disk for stable storage. In order to take full benefit of this capability, the framework also enables policy customization, with which it is possible to specialize the policy to the user’s available storage resources.

5 Implementation Challenges

As discussed in Section 3, Xen provides mechanisms for checkpoint/restart of the VM memory. However, these mechanisms are insufficient for the implementation of a complete VM checkpoint/restart mechanism, as Xen makes assumption about the VM’s disk image regarding accessibility and concurrent access. This leads to two key issues regarding the checkpoint/restart of VM disk images (file systems): (i) size/amount of data, and (ii) consistency/synchronization with other Xen checkpoint mechanisms (e.g., memory snapshot).

The size of the checkpoint influences the time it takes to write the data to local or remote storage. A differential checkpointing approach could be used to reduce the size by only capturing the data that has changed from a prior disk state (possibly the original installation state). The differential checkpoint would (typically) be less than the entire image size (ignoring the worst case disk scenario of upgrades or re-installations). Additionally, the ability to synchronize disk and memory state during a checkpoint is critical to avoid problems with inconsistency or corruption, e.g., see Figure 3.

A potential solution that encompasses both of these aspects is the use of stackable filesystems with snapshot support, e.g., unionFS [8]. The copy-on-write semantics would be employed to allow a VM to checkpoint disk state and then move this incarnation of the stackable filesystem to a lower read-only level. The top-most filesystem being the current (live) filesystem,

where changes are accrued through copy-on-write semantics. Once the VM's disk and memory state have been recorded a full rollback mechanism is possible without the potential for inconsistency during checkpoint/restart, as outlined by Figure 3.

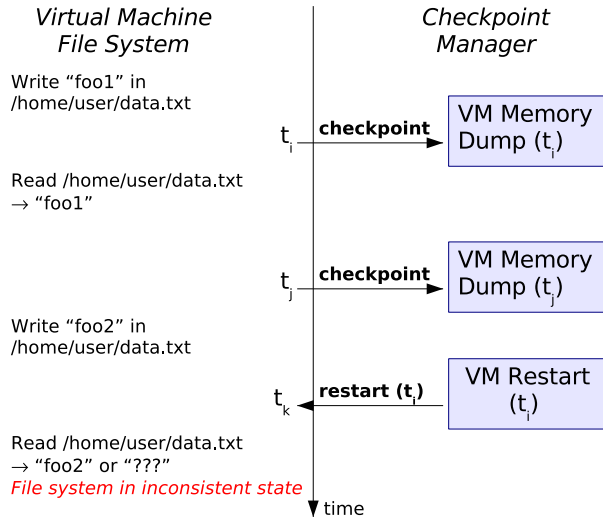


Figure 3. Example of an Inconsistent VM File System Using the Xen Save/Restore Mechanism

5.1 Discussion

In order to extend Xen mechanisms for the implementation of a complete checkpoint/restart mechanism, the whole virtual machine has to be checkpointed: the memory, as Xen is already doing, but also the file system.

The checkpoint/restart of the file system may be an expensive task especially since a Xen VM's file system represents the complete OS file system, which may include several gigabytes of data. However, in recent years a new kind of file system called *stackable file systems* [9] have emerged, e.g., unionFS [8].

These file systems, are widely used for the creation of LiveCDs, and allow several file systems to appear unique to applications within the VM. For instance, it is possible to give the illusion of a writable file system, while it is actually read-only. Stackable file systems also provide (typically) the ability to merge the different file systems together and redirect the result to a separate file system.

These capabilities may be used for the implementation of a checkpoint/restart file system for Xen VMs (see Figure 4). Virtual machines never directly use the

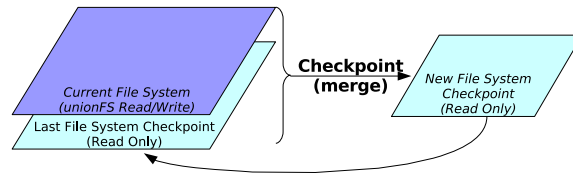


Figure 4. Checkpoint of a VM File System Using a Stackable File System

“real” file system contained within the image; VMs always use a union of the “basic file system”, which is in read-only mode. The read-write access is done via the stackable file system, e.g., unionFS. The writable file system captures file system accesses and modifications. The initial file system of the disk image is considered as the first file system checkpoint, then the following algorithm is used:

1. mount the last checkpoint in read-only mode,
2. create a union with a writable file system,
3. when a checkpoint operation is received do:
 - (a) merge the two file systems to a new file system checkpoint,
 - (b) associate the file system checkpoint with the memory dump created by Xen,
 - (c) consider the new file system checkpoint as the last checkpoint and restart at step 1.

This algorithm allows us to have an automatic rotation of file system checkpoints. Moreover, the checkpoint of the file system does not imply a checkpoint of the complete file system since the merge mechanism of stackable file system typically merges only differences. For the example of a LiveCD, the differences between the CD-ROM file system and the writable file system are merged into a copy on the local hard drive (or USB drive, etc).

Moreover, Xen does not have to be modified for the implementation of a checkpoint/restart mechanism since unionFS may be used transparently if integrated directly into the image used by the VM. In that case, two images are created: (i) a basic image for the standard file system and (ii) an updated image which includes unionFS.

It is then simple to restart a VM from a checkpoint, using the following algorithm:

1. identify the checkpoint to restart,
2. switch the current VM image to the checkpointed file system,

3. restore the memory checkpoint (Xen memory dump).

6 Conclusion

This document presents the design of a checkpoint/restart infrastructure for Xen virtual machines. The infrastructure aims to: (i) provide flexible management of VM checkpoints; and (ii) provide an efficient way to checkpoint VMs, including their file system.

The solution proposed in this document does not need any modifications of Xen since the infrastructure uses the existing Xen interface, and the checkpoint/restart of VMs can be accomplished by updating the default image of virtual machines (to make them use unionFS instead of a traditional file system such as ext3).

A prototype of the checkpoint manager is currently under development. The development does not include unionFS by default for Xen images. Integration with OSCAR is planned, in order to take advantage of OSCAR's [3] capabilities to define system images to ease Xen VM creation and deployment. An OSCAR LiveCD incorporating the efforts described in this paper is also planned.

References

- [1] 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 *Proceedings of the nineteenth ACM symposium on Operating System s Principles (SOSP19)*, pages 164–177. ACM Press, 2003.
- [2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live Migration of Virtual Machines. In *Proceedings of the 2nd Symposium on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2-4, 2005. USENIX.
- [3] J. Mugler, T. Naughton, S. L. Scott, B. Barrett, A. Lumsdaine, J. M. Squyres, Benot des Ligneris, Francis Giraldeau, and C. Leangsuksun. OSCAR Clusters. In *Proceedings of the 5th Annual Ottawa Linux Symposium (OLS'03)*, Ottawa, Canada, July 23-26, 2003.
- [4] I. Pratt, D. Magenheimer, H. Blanchard, J. Xenidis, J. Nakajima, and A. Liguori. The Ongoing Evolution of Xen. In *Proceedings of the Ottawa Linux Symposium (OLS)*, volume 2, pages 255–266, Ottawa, Ontario, Canada, July 19-22, 2006.
- [5] J. E. Smith and R. Nair. The Architecture of Virtual Machines. *IEEE Computer*, 38(5):32–38, May 2005.
- [6] R. Uhlig, G. Neiger, D. Rodgers, A. L. Santoni, F. C. M. Martins, A. V. Anderson, S. M. Bennett, A. Kägi, F. H. Leung, and L. Smith. Intel Virtualization Technology. *IEEE Computer*, 38(5):48–56, May 2005.
- [7] A. Whitaker, M. Shaw, and S. D. Gribble. Denali: Lightweight Virtual Machines for Distributed and Networked Applications. Technical Report 02-02-01, University of Washington, Feb. 2002.
- [8] C. P. Wright, J. Dave, P. Gupta, H. Krishnan, E. Zadok, and M. N. Zubair. Versatility and Unix Semantics in a Fan-Out Unification File System. Technical Report FSL-04-01b, Computer Science Department, Stony Brook University, October 2004.
- [9] E. Zadok and I. Bădulescu. A stackable file system interface for Linux. In *LinuxExpo Conference Proceedings*, pages 141–151, Raleigh, NC, May 1999.