CERIAS Tech Report 2008-25
The Social Structure and Construction of Privacy in Sociotechnological Realms
by Lorraine Gayle Kisselburgh
Center for Education and Research
Information Assurance and Security
Purdue University, West Lafayette, IN 47907-2086

VNsnap: Taking Snapshots of Virtual Networked Environments with Minimal Downtime

Ardalan Kangarlou, Dongyan Xu, and Patrick Eugster

Department of Computer Science and CERIAS
Purdue University
West Lafayette, Indiana, USA
{ardalan,dxu,p}@cs.purdue.edu

Abstract. A virtual networked environment (VNE) consists of multiple virtual machines (VMs) connected by a virtual network. It has been adopted to create private "workspaces" for individual users or communities on a shared physical infrastructure. The ability to take a distributed snapshot of the whole virtual environment — including images of the VMs with their execution, communication and storage states — yields a unique, practical approach to VNE reliability. The snapshot can then be used to bring the entire VNE back up in the event of a failure or outage. In this paper, we present VNsnap, a middleware system that takes distributed snapshots of VNEs. Unlike existing distributed snapshot/checkpointing solutions, VNsnap does not require any modifications to the applications, libraries, and (guest) operating systems running in the VNE. Furthermore, VNsnap incurs only seconds of downtime as much of the snapshot operation takes place concurrently with the normal operation of the VNE. We have implemented VNsnap on top of the Xen virtual machine monitor. Our experiments with real-world parallel and distributed applications demonstrate the effectiveness and efficiency of VNsnap.

Key words: Virtual Networks, Reliability, Distributed Snapshots

1 Introduction

A virtual networked environment (VNE) consists of multiple virtual machines (VMs) connected by a virtual network. In a shared physical infrastructure, VNEs can be created as private, isolated "workspaces" serving individual users or communities. For example, a virtual cluster can be created to execute parallel/distributed jobs with its own root privilege and customized runtime library; a virtual data network can be set up across organizational firewalls to support seamless file sharing; and a virtual "playground" can be established to emulate computer virus infection and propagation.

To bring reliability and resume-ability to VNEs, it is highly desirable that the underlying infrastructure provide the capability of taking a distributed snapshot of the entire virtual environment, including images of the execution, communication, and storage states of all the VMs. The snapshot can later be used to

restore the entire VNE, thus supporting fault/outage recovery, system pause and resume, as well as troubleshooting and forensics.

In this paper, we present VNsnap, a middleware system capable of taking distributed snapshots of VNEs. Based on the virtual machine monitor (VMM), VNsnap runs *outside* of the target VNE. Unlike existing distributed snapshot (checkpointing) techniques at application, library, and OS levels [15–17, 19], VNsnap does not require any modifications to software running inside the VNE and thus works with *unmodified* applications and (guest) OSes that do not have built-in snapshot/checkpointing support. As such, VNsnap fills a void in the spectrum of snapshot/checkpointing techniques and complements (instead of replaces) the existing solutions.

There are two main challenges to taking VNE snapshots. First, the snapshot operation may incur significant system *downtime*, during which the VMs freeze all computation and communication while their memory images are being written to disks. As shown in our previous work [1], such downtime can be tens of seconds long, which disrupts both human users and applications in the VNE. Second, the snapshots of individual VMs have to be coordinated to create a *globally consistent distributed* snapshot of the entire VNE. Such coordination is essential to preserving the consistency of the VM execution and communication states when the VNE snapshot is restored in the future.

To address the first challenge, VNsnap involves an optimized technique for taking individual VM snapshots where much of the VM snapshot operation takes place concurrently with the VM's normal operation thus effectively "hiding" the snapshot latency from users and applications. To address the second challenge, VNsnap modifies and instantiates a classic global snapshot algorithm so that it can handle the complexities arising from both the virtual network design and the optimized VM snapshot technique. As such, VNsnap is capable of taking snapshots of a VNE with only minor downtime.

We have implemented a Xen [3] based VNsnap prototype for our virtual networked environment VIOLIN [2]. To evaluate the VIOLIN downtime incurred by VNsnap and its impact on applications, we use two real-world parallel/distributed applications – one is an MPI-based parallel nanotechnology simulation *without* built-in checkpointing capability while the other is the peer-to-peer file sharing application BitTorrent. Our experiments show that VNsnap is able to generate semantically correct snapshots of VIOLINs running these applications, incurring about 1 second (or less) of VM downtime in all experiments.

2 VIOLIN Background

In this section, we give a brief introduction to the VIOLIN virtual networked environment and a previous distributed VIOLIN snapshot system we presented in [1]. VIOLIN is our instantiation of the VNE concept described in Section 1. Based on Xen, a VIOLIN virtual networked environment (or "VIOLIN" in short) provides the same "look and feel" of its physical counterpart, with its own IP address space, administrative privileges, runtime services and libraries, and network

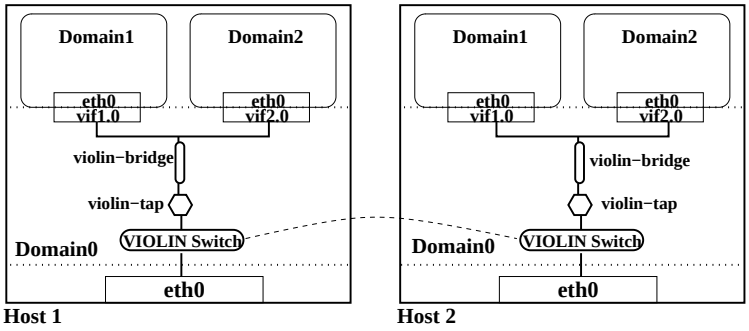


Fig. 1. A 4-VM VIOLIN based on Xen, running on two physical hosts

configuration. VIOLIN has been deployed in a number of real-world systems: In the nanoHUB cyberinfrastructure (http://www.nanoHUB.org, with more than 20,000 users worldwide), VIOLINs run as virtual Linux clusters for the execution of a variety of nanotechnology simulation programs; In the vBET/vGround emulation testbed [4, 5], VIOLINs run as virtual internetworking environments for the emulation of distributed systems and malware attacks.

As shown in Figure 1, a VIOLIN consists of multiple VMs connected by a virtual network. In our implementation, VMs (i.e. guest domains) are connected by VIOLIN switches that run in domain 0 of their respective physical hosts. Each VIOLIN switch intercepts link-level traffic generated by the VMs – in the form of layer-2 network frames – and tunnels them to their destination hosts using the UDP transport protocol. VIOLIN snapshots are taken at the VIOLIN switch level from outside of the VMs. As such, there is no need for modifying the application, library, or OS that runs inside the VMs. Another benefit of VIOLIN snapshots is that such a snapshot can be restored on any physical machine and network without requiring reconfiguration of the VIOLIN's IP address space. This is due to the fact that VIOLIN performs layer-2 network virtualization, and as such any set of network addresses can be used in a VIOLIN without any conflict with the underlying physical infrastructure.

In our previous work [1], we presented the first prototype for taking VIOLIN snapshots. Unfortunately, that prototype has a serious limitation: By simply leveraging Xen's live VM checkpointing capability, the system has to freeze each VM for a non-trivial period of time during which the entire memory image of the VM is written to the disk. As a result, taking a VIOLIN snapshot causes considerable downtime to the VIOLIN (in the magnitude of ten or tens of seconds). Moreover, due to TCP backoff incurred by the VM's long freeze, it will take extra amount of time for an application to fully resume its operation following a VIOLIN snapshot.

3 VNsnap Design and Implementation

In this section, we present the design and implementation of VNsnap. More specifically, we first describe our solution to minimizing VM downtime during the VIOLIN snapshot operation. We then describe our solution to taking distributed snapshot of a VIOLIN with multiple communicating VMs.

3.1 Optimizing Live VM Snapshots

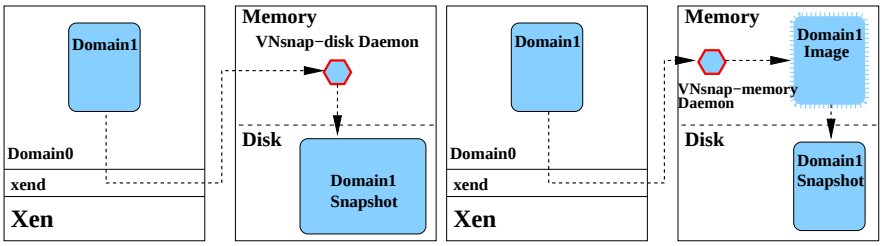
3.1.1 Overview of Approach Our solution in VNsnap aims at minimizing the Xen live VM checkpointing downtime thus making the process of taking a VM snapshot truly *live*. Interestingly, the solution is inspired by Xen's live VM *migration* function [6]: Instead of freezing a VM throughout the snapshot, we take a VM snapshot much the same way as Xen performs a live VM migration. As such we hide most of the snapshot latency in the VM's normal execution time leading to a negligible (usually less than a second) VM downtime.

Xen's live migration operates by incrementally copying pages from a source host to a destination host in multiple iterations while a VM is running. In every iteration, only the pages that have been modified since the previous iteration get resent to the destination. Once the last iteration is determined (e.g., when a small enough number of pages are left to be sent, the maximum number of iterations are completed, or the maximum number of pages are sent), the VM is paused and only the few remaining dirty pages are resent to the destination host. Once this "stop-and-copy" phase is completed, the VM on the source host is terminated and its copy on the destination host is activated. As a result, during live migration a VM is operational for all but a few tens/hundreds of milliseconds.

Following the same principle, our optimized live VM checkpointing technique effectively migrates a running VM's memory state to a local or remote snapshot file but *without* a switch of control (namely the same VM will keep running). To facilitate such migration, we create the *snapshot daemon* that "impersonates" the destination host during a live snapshot. The snapshot daemon interacts with the source host in obtaining the VM's memory pages, which is, to the source host, just like a live migration. However, the snapshot daemon does *not* create an active copy of the VM. Instead, the original VM resumes execution the snapshot has been taken.

3.1.2 Detailed Design and Implementation We have implemented two versions of the snapshot daemon, each with different advantages. Both versions can run either locally on the same host where the VM is running or remotely on a different host. For the rest of the paper we will refer to these two versions as the *"VNsnap-disk"* and *"VNsnap-memory"* daemons. We next describe their implementations and compare their performance and effectiveness.

VMsnap-disk Daemon: The VNsnap-disk daemon operates by recording the stream of VM memory image data generated by the source host VMM during a



(a) VNsnap-disk (b) VNsnap-memory

Fig. 2. Designs of VNsnap-disk and VNsnap-memory for optimized live VM snapshot. VNsnap-disk may result in larger snapshot files; whereas VNsnap-memory generates snapshots of the same size as the VMs at the cost of memory reservation.

live migration. In this simple design, bytes received by the VNsnap-disk daemon are grouped into chunks (32KB in our implementation) and as soon as a chunk is full it is immediately written to the disk (Figure 2(a)). As such the daemon is oblivious to the nature of data it receives and is only concerned with recording the data stream as is. When the snapshot file is restored on a host in the future, the stream is played back and the host perceives the operation as receiving a VM memory image during live migration.

The VNsnap-disk daemon has two main advantages. First, it does not require a large amount of memory as the daemon writes small chunks of VM memory image data directly to the disk (Figure 2(a)). Second, by the time the (fake) VM migration is completed, the snapshot file is readily available on the disk. However, the VNsnap-disk daemon does have a number of weaknesses. First, the snapshot file it generates can potentially be much larger than the actual VM memory image as *multiple* copies of the same memory page may have been received and recorded during migration. The larger snapshot size translates into more writes to the disk and consequently a lengthier duration of the snapshot operation. Second, during a future snapshot restoration, a host will have to go through multiple iterations to obtain the final image of a memory page. As a result, the restoration will take longer when compared with restoring a snapshot file generated by Xen's original live checkpointing function.

VNsnap-memory Daemon: The VNsnap-memory daemon overcomes the weaknesses of the VNsnap-disk daemon, at the cost of reserving a memory area equal to the size of the memory image of the VM it checkpoints (Figure 2(b)). The VNsnap-memory daemon is "conscious" of the nature of data it receives from the source host and only keeps the *most recent* image of a page – in the reserved memory area. As a result, the final snapshot it generates is the same size as the VM's memory image. The snapshot will not be written to disk until the VM snapshot operation is complete and the VM has resumed normal execution. Compared with VNsnap-disk, this design better hides the snapshot operation duration by postponing disk writes until the VM snapshot is completed. It also

leads to shorter VM downtime with only memory writes. Moreover, VNsnap-memory causes much less TCP backoff than VNsnap-disk, as to be demonstrated and explained in Section 4. On the other hand, the postponed snapshot dump in VNsnap-memory does lead to the disadvantage that the snapshot file is not immediately available after the snapshot operation.

Although the operation of the VNsnap-memory daemon resembles that of a live VM migration, the implementation of the VNsnap-memory daemon involves modifications to Xen's live VM migration function. It might seem that a VM snapshot can simply be done by performing a live migration followed by (1) the restart of the original VM and (2) the freeze and dump of the new copy on the destination host using Xen's live VM checkpointing function. However, our experience indicates that this is not as simple as it sounds. First, Xen by design does not allow checkpointing a VM that has not started or resumed execution (which is the case for the new VM). Second, Xen live migration involves translating the VM's memory page addresses that are specific to the source host (i.e. page tables that reference *machine frame numbers*) into some host-independent representation (i.e. *pseudo-physical frame numbers*) through what is known as canonicalization. Upon receipt of such pages on the destination host, these pages have to be mapped to the machine frame numbers specific to the destination host (or get un-canonicalized). However, for VM snapshots we only need the canonicalized pages so that the snapshot can be restored on any host. In our implementation, the VNsnap-memory daemon intercepts and maintains the most recent image of any canonicalized page. Once the VM memory image transfer is complete, the daemon writes all memory pages in batches to a snapshot file as if the snapshot file were generated by Xen's live checkpointing function.

The implementation of VNsnap-disk and VNsnap-memory daemons involved making modifications to the *xend* component of Xen that handles VM live migration. Our implementation is based on a recent unstable release of Xen (May 2007). We point out that both daemons can run locally or remotely. For the local run it is desirable to reserve a certain amount of CPU capacity for the daemon in order to prevent a snapshot from affecting the VMs' execution. In a uni-core machine this can be done by enforcing CPU capacity allocations to different domains, while in a multi-core machine this can be done by assigning the daemon and the VMs to different cores. For a remote run, the daemons consume much less resources of the source host but will depend on a high speed network between the source and destination hosts for VM image transport.

3.2 Taking Distributed VIOLIN Snapshot

With the individual VM snapshots achieving minimal downtime, we now present our approach to the coordination of these snapshots in creating a distributed consistent snapshot for a VIOLIN. Our distributed snapshot algorithm is based on Mattern's global snapshot algorithm [7] for non-FIFO communication channels. This algorithm was also adopted in our earlier work [1]. However, we will show that the algorithm is particularly suited to the optimized live VM snapshot

technique (Section 3.1) and not so much to the "freeze and dump" VM snapshot technique used in [1].

3.2.1 Overview of Snapshot Algorithm In the context of VIOLIN, the main motivation behind the use of Mattern's algorithm is to prevent a post-snapshot network frame (i.e. a layer-2 frame generated by a VM whose snapshot has been taken) from affecting the state of a pre-snapshot VM (i.e. a VM whose snapshot has not been completed). Therefore, the VMs' snapshots will form a *consistent cut* [7] which are guaranteed to be *causally consistent* and safely restore-able. To ensure causal consistency, we use Mattern's *message coloring* technique to distinguish between pre-snapshot and post-snapshot frames. The VNsnap distributed snapshot algorithm works as follows:

1. One of the VIOLIN switches initiates the distributed snapshot by sending a *TAKE_SNAPSHOT* message to all other switches. It then starts the snapshot-taking operations for the local VMs that belong to the same VIOLIN. Once these VM snapshots are completed, the switch starts tainting all outgoing frames with the post-snapshot color.
2. Upon receiving the *TAKE_SNAPSHOT* message or a frame with the post-snapshot color, a VIOLIN switch starts the VM snapshot-taking operations as done by the initiator switch. When the VM snapshots are completed, the switch notifies the initiator via a *SNAPSHOT_SUCCESS* message and colors all outgoing frames with the post-snapshot color.
3. While a VM snapshot is in progress, the underlying VIOLIN switch colors frames originating from that VM with the pre-snapshot color and prevents the delivery of frames bearing the post-snapshot color to that VM.
4. The distributed snapshot operation is complete when the initiator receives the *SNAPSHOT_SUCCESS* messages from all other VIOLIN switches.

Figure 3 shows an example of the above algorithm for a VIOLIN consisting of four VMs. Frames exchanged between VMs are denoted by arrows going from the sender to the receiver where pre-snapshot frames are colored red and post-snapshot frames are colored blue. The VIOLIN switch for VM_a initiates the snapshot at time S_{1a} and the snapshot is completed at S_{2a}. At S_{2d}, the distributed snapshot operation is completed.

It is important to note the difference between Mattern's algorithm and the VNsnap algorithm: Mattern's snapshot algorithm was proposed for distributed systems connected by non-FIFO, *reliable* communication channels (i.e. no message loss thus no message retransmissions); whereas the VNsnap algorithm involves VIOLIN switches that forward VM-generated layer-2 network frames via UDP tunneling. As such there may be message losses – namely losses of UDP packets carrying the layer-2 frames (though no message retransmissions). Moreover, the VNsnap algorithm may induce additional message drops, as just explained, to maintain causal consistency across VM snapshots. Despite the differences, we point out that the VNsnap algorithm is able to guarantee the correct

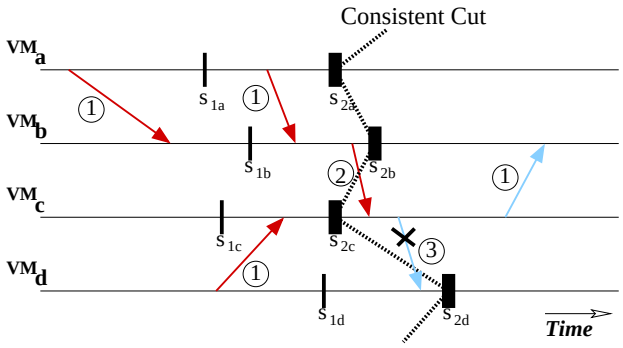


Fig. 3. An illustration of the VNsnap distributed snapshot algorithm: For each VM, S_1 is the point when the VM starts the snapshot operation and S_2 is when the VM finishes the snapshot and resumes normal execution. This figure also demonstrates the frame-coloring scheme and the three categories of frames. In particular, Category 3 frames are dropped to preserve a consistent cut.

operation of transport protocols in the VIOLIN. For best-effort transport protocols such as UDP, packet losses are *expected* and if needed should be handled by the application. For reliable transport protocols such as TCP, the VNsnap algorithm enforces causal consistency of layer-2 frame delivery so that the transport protocol state – in the face of frame losses – remains correct in the VIOLIN snapshot.

We use the example in Figure 3 for further explanation. Given the asynchronous nature of VM snapshot operations, messages (i.e. layer-2 frames) in Figure 3 fall into the following three categories:

1. Frames where the source and destination VMs are both in the pre-snapshot state or both are in the post-snapshot state (e.g. the frames labeled 1 in Figure 3). Such frames can be safely delivered to the destination VMs.
2. Frames where the source VM is in the pre-snapshot state and the destination VM is in the post-snapshot state (e.g. the frame labeled 2 from VM_b to VM_c). Such frames can also be safely delivered.
3. Frames where the source VM is in the post-snapshot state and the destination VM is in the pre-snapshot state (e.g. the frame labeled 3 from VM_c to VM_d). Such frames are *dropped* by the VIOLIN switches.

Suppose a TCP packet is encapsulated in the Category 2 frame from VM_b to VM_c. Although the packet is delivered to VM_c, its acknowledgement, encapsulated in a frame from VM_c to VM_b, will *not* be delivered to VM_b as long as VM_b is still in the pre-snapshot state. As a result, VM_b does not advance its TCP window and keeps retransmitting the packet until it completes its snapshot and can receive the acknowledgement from VM_c. Thus, VNsnap's handling of Category 3 messages guarantees the correct TCP state across VM snapshots. This prevents *incorrect* transport states from being exhibited when the snapshot is restored. For example, it will never be the case that VM_b thinks that it

has *successfully* sent the TCP packet to VM_b prior to its snapshot while VM_c, whose snapshot was taken *before* the TCP packet arrival, has no indication of having received this packet.

3.2.2 Detailed Design and Implementation In our implementation, a VIOLIN switch (or switch for short) enters SNAPSHOT mode when it starts the snapshot-taking operations for the the local VMs that are connected to it. It exits SNAPSHOT mode when all these VM snapshots are completed. When the switch is not in SNAPSHOT mode, it taints all outgoing frames with the same color. However, when the switch is in SNAPSHOT mode, it may taint a frame with either the pre-snapshot or post-snapshot color depending on the status of the VM generating the frame. This is due to the fact that the various VMs on the same host may not complete their live snapshot operations at exactly the same time.

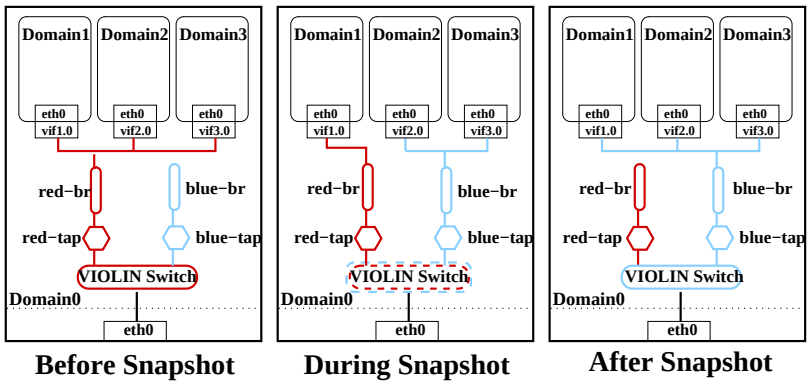


Fig. 4. An illustration of asynchronous VM snapshot progress in a machine hosting three VMs of the same VIOLIN. Red is the pre-snapshot color and blue is the post-snapshot color. In the middle figure, Domains 2 and 3 have transitioned to the post-snapshot state while Domain 1 is still in the pre-snapshot state.

Figure 4 illustrates such a situation. To handle the asynchronous completion of VM snapshots on the same host, VNsnap uses two pairs of bridges and tap devices: one pair for the pre-snapshot VMs and the other pair for the post-snapshot VMs. As a result, it is guaranteed that no post-snapshot frames can reach a pre-snapshot VM on the same host. Furthermore, the underlying VIOLIN switch is able to determine if a frame comes from a pre-snapshot or post-snapshot tap device and color the frame accordingly before sending it to another host. We modify Xen's *xend* to transition a VM from the pre-snapshot bridge to the post-snapshot bridge at the end of the stop-and-copy phase and right before the VM resumes normal execution.

For a VIOLIN switch to exit SNAPSHOT mode, we have to extend *xend* such that it will notify the switch whenever a VM finishes its snapshot operation. Specifically, we define a signal handler inside the VIOLIN switch which will receive a user-defined signal from *xend* when a VM completes its stop-and-copy phase. Once the VIOLIN switch has received the signals for all local VMs belonging to the same VIOLIN, the switch will exit SNAPSHOT mode and taint all outgoing frames with the post-snapshot color.

To mitigate the frame drops by the VNsnap algorithm, we have also implemented a frame buffering scheme that preserves the frames dropped by the algorithm. In this scheme, a VIOLIN switch will buffer (instead of drop) post-snapshot frames that are destined for a pre-snapshot VM. These buffered frames will later be delivered when the receiver VM finishes its snapshot or when the VIOLIN snapshot is restored in the future. This scheme proves useful for applications using UDP transport in a VIOLIN. However, it turns out that the scheme does not fare well with applications using TCP transport. There are three main reasons for this. First, the delivery of buffered TCP packets requires stringent timing: To be of use, these packets need to be injected within the narrow window when the receiver VM has resumed normal execution after the snapshot but before the sender VM resends the buffered packets. In the case of VIOLIN snapshot restoration, the sender VM also needs to be fully operational first so that it can receive the ACKs that indicate the successful delivery of these packets. Otherwise, these packets will still be retransmitted. Second, many of the buffered packets are retransmitted packets to begin with. As a result, only a small percentage of the buffered packets are of real "help" to the progress of a TCP connection and their repeated delivery should be avoided. Third, Given that these buffered packets have been time-stamped at the time of snapshot, the *TSval* and *TSecr* fields (used for Round-Trip Time Measurement (RTTM) and Protect Against Wrapped Sequence numbers (PAWS) mechanisms) in the TCP headers would most likely not match the TCP clock time when a VM finishes a snapshot or when the VIOLIN snapshot is restored. As a result, these packets are likely to get discarded by TCP [8].

So far we have only discussed the different ways VNsnap captures the VM state and maintains causal consistency. For a VIOLIN snapshot to be useful, it should also include the file system state. To meet this goal, we store a VM's file system on an LVM [9] logical volume and use the LVM snapshot capability to capture the state of the file system at the time of snapshot. The main advantages behind LVM snapshots are availability and speed. LVM snapshots do not require a system using the logical volume to be halted during the snapshot. It also does not work by mirroring a logical volume to some other partition. Instead, it only records changes made to a logical volume after the snapshot and as a result is very fast. In VNsnap, LVM snapshots are taken during the (very short) stop-and-copy phase when a VM is suspended. The snapshot partitions can be processed after the VM resumes normal execution.

4 Evaluation

In this section, we evaluate the effectiveness and efficiency of VNsnap. First, we focus on testing the optimized live VM snapshot technique. Then, we evaluate the impact of VNsnap on VIOLINs running real-world parallel/distributed applications – NEMO3D [10] and BitTorrent [11]. Throughout this section, we compare VNsnap with our previous work [1]. All physical hosts involved in our experiments are Sunfire V20Z servers with a single 2.6GHz AMD Opteron processor and 4GB of RAM.

4.1 Downtime Minimization for Live VM Snapshots

We first evaluate the optimized live VM snapshot technique (Section 3.1) for individual VMs in a VIOLIN. The evaluation metrics include the total *duration* and VM *downtime* of an individual VM snapshot operation as well as the *size* of the VM snapshot generated. For comparison, we experiment with all of the following VM snapshot implementations: (1) Xen's live VM checkpointing function (used in [1]), (2) the VNsnap-disk daemon, and (3) the VNsnap-memory daemon. For each of the implementations we measure the metrics from the same VM (in a VIOLIN) with 600MB of RAM. The tests are run both when the VM is idle and when it is executing the parallel application NEMO3D.

Table 1 shows the results. Since both VNsnap-disk and VNsnap-memory daemons are based on Xen's live migration function, they both involve multiple iterations of memory page transfer during the snapshot (the "iteration" column) while the VM is running. It is during the very last iteration that the VM freezes and causes the downtime (the "pages in last iteration" column). The number of iterations is proportional to the application's Writable Working Set (WWS) [6] or the rate at which the application is dirtying its memory pages. For instance, we observe that during the NEMO3D execution memory pages get dirtied at a rate above 125MB/s.

The most important metric in Table 1 is the VM downtime. Three main observations can be made from these results. First, both VNsnap-disk and VNsnap-memory incur significantly shorter downtime (ranging from tens of milliseconds to just above one second) than Xen's checkpointing function (around 8.6 seconds). Second, for Xen's live checkpointing function, the downtime remains almost the same for both the "idle" and "NEMO3D" runs. VNsnap-disk and VNsnap-memory, on the other hand, exhibit shorter downtime for the "idle" runs than the "NEMO3D" runs. This is because for VNsnap-disk and VNsnap-memory, the downtime is determined by the number of dirty pages transferred in the *last* iteration – about 100 pages in the "idle" run and 11,000 pages in the "NEMO3D" run – out of the total 153,600 pages of the VM. This differs from Xen's VM checkpointing, where there is only one iteration during which the VM freezes and all 153,600 pages are written to disk. Finally, we observe that VNsnap-memory achieves a much lower downtime for the "NEMO3D" run than VNsnap-disk. This is because the VNsnap-disk daemon directly writes the

Xen Live Checkpointing					
Application	Duration(s)	Iterations	Downtime(ms)	Pages in Last Iteration	Size
Idle	9	1	8583	153600	1.00
NEMO3D	12	1	8626	153600	1.00
VNsnap-disk Daemon					
Application	Duration(s)	Iterations	Downtime(ms)	Pages in Last Iteration	Size
Idle	12	4	65	104	1.00
NEMO3D	72	30	1025	11102	1.55
VNsnap-memory Daemon					
Application	Duration(s)	Iterations	Downtime(ms)	Pages in Last Iteration	Size
Idle	8	4	68	104	1.00
NEMO3D	18	30	258	11094	1.00

Table 1. Measurement results comparing three VM snapshot implementations for VNsnap.

page images to the disk (which is slow) while the VNsnap-memory daemon keeps them in the RAM during the snapshot operation (which is fast).

Another important metric from Table 1 is the total snapshot duration. For both Xen checkpointing and VNsnap-disk, the duration represents the amount of time it takes for the snapshot image to be fully committed to disk. For VNsnap-memory, the duration represents the amount of time it takes for the daemon to construct a VM's full image in memory and consequently does not include the hidden disk write latency *after* the snapshot. We observe that for the "NEMO3D" run, both VNsnap-disk and VNsnap-memory incur longer duration than Xen checkpointing because of their multi-iteration memory page transfer. The duration for VNsnap-disk is particularly long compared to the other two implementations (72 seconds vs. 12 seconds for Xen checkpointing and 18 seconds for VNsnap-memory) as the daemon competes with the local VM for both disk bandwidth and CPU cycles. Such a contention can be mitigated by running the VNsnap-disk daemon in a remote host, which will reduce the snapshot duration to 33 seconds as our experiment shows.

Table 1 also shows the size of the VM snapshot relative to the amount of memory allocated to the VM. As discussed in Section 3.1, the VM snapshot generated by the VNsnap-disk daemon can be larger than the VM's memory size. In fact, the VM snapshot file is 1.55 times the size of the VM's memory image for the "NEMO3D" run. Both Xen checkpointing and VNsnap-memory, by their respective design, generate VM snapshots of the *same* size as the VM's memory image. A larger VM snapshot file consequently results in longer time in *restoring* the VM. Our experiments confirm that it takes 20 seconds to restore a snapshot generated by VNsnap-disk whereas it takes 8 seconds to restore a VM snapshot file generated by VNsnap-memory or Xen checkpointing.

Impact of VM Snapshot on TCP Throughput As discussed in Section 3.2, the individual VM snapshot operations for the same VIOLIN may have different completion times. Specifically, not all VMs transition from the pre-snapshot

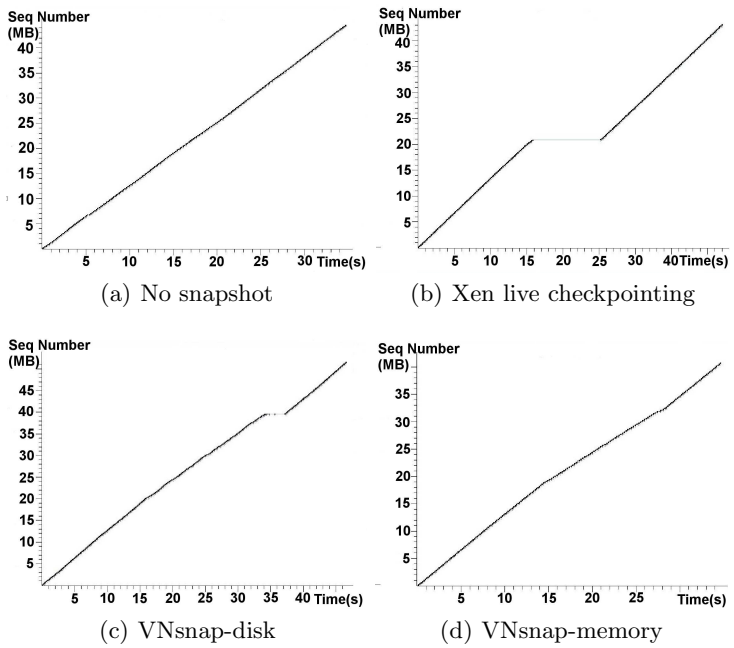


Fig. 5. The impact of different VM snapshot techniques on TCP throughput in a VIOLIN running NEMO3D. Traces are obtained from tcpdump.

to post-snapshot state at exactly the same time and the VNsnap distributed algorithm may have to drop certain frames to enforce causal consistency between the VM snapshots. Such frame drop results in temporary backoff for the TCP connections during and after snapshots. As one would expect, the duration of TCP backoff is directly related to the degree of discrepancy in individual VM snapshot completion times.

Figure 5 shows such impact on a 2-VM VIOLIN executing NEMO3D, under no snapshot (Figure 5(a)), Xen live checkpointing (Figure 5(b)), VNsnap-disk (Figure 5(c)), and VNsnap-memory (Figure 5(d)). We focus on one TCP connection between the two VMs. The flat, "no progress" periods shown in Figures 5(b) and 5(c) each consist of two parts: (1) the downtime of the sender VM during snapshot and (2) the TCP backoff period due to the varying snapshot completion times of the sender and receiver VMs. We observe that both Xen live checkpointing (Figure 5(b)) and VNsnap-disk (Figure 5(c)) incur 2-3 seconds of TCP backoff, whereas VNsnap-memory (Figure 5(d)) does not incur noticeable TCP backoff. More results and analysis will be presented in the next two subsections.

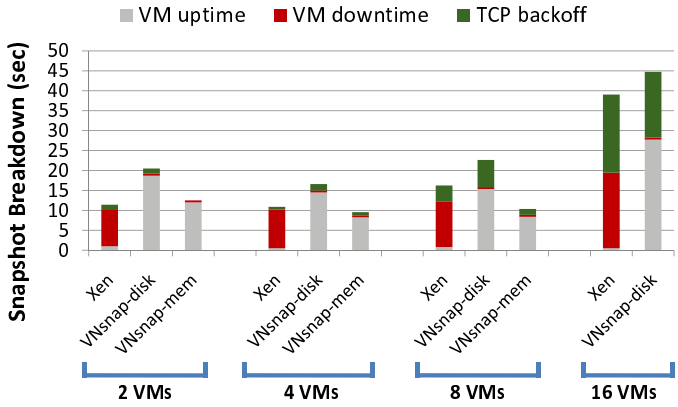


Fig. 6. The breakdown of snapshot timing under different VM snapshot implementations for 2, 4, 8 and 16-node VIOLINs running NEMO3D.

4.2 Snapshot of VIOLIN Running NEMO3D

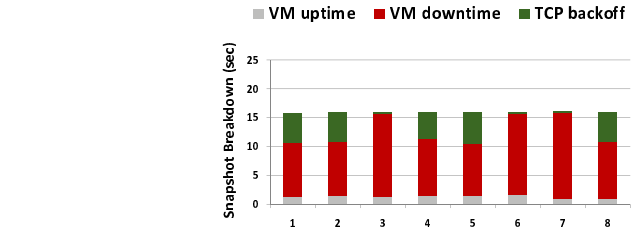
NEMO3D is a long-running (tens of minutes to hours), MPI-based parallel simulation program without any built-in checkpointing support. It is widely used by the nanotechnology community for nano-electric modeling of quantum dots. To execute NEMO3D, we create VIOLINs as virtual Linux clusters of varying size (with 2, 4, 8, and 16 VMs). The underlying physical infrastructure is a cluster of 8 Sunfire V20Z servers connected by Gigabit Ethernet. For the 2, 4, or 8-VM VIOLIN, each VM runs in a distinct physical host and is allocated 650MB of memory. For the 16-VM VIOLIN, there are two VMs per host each with 650MB of memory (due to the limited availability of 8 hosts). For each VIOLIN, we run NEMO3D with the same input parameters and trigger the snapshot algorithm at exactly the same stage of NEMO3D execution for the Xen checkpointing, VNsnap-disk, and VNsnap-memory implementations. For each implementation, we measure, on a per VM basis, the VM uptime and VM downtime during the snapshot operation as well as the TCP backoff experienced by the VM due to snapshot completion time discrepancy. We note that the VM downtime plus the TCP backoff constitute the actual *period of disruption* to application execution inside the VM.

Figure 6 shows the results. The times shown are averages of all VMs in a given VIOLIN from a given experiment. We observe that VNsnap-memory always incurs the least disruption (VM downtime+TCP backoff) to a VIOLIN – more specifically 0.0, 0.8, and 1.4 seconds to the 2, 4, and 8-node VIOLINs, respectively[1]. VNsnap-disk also incurs minimal VM downtime but incurs higher TCP backoff than VNsnap-memory (to be explained shortly). Still, it performs much better than Xen checkpointing, which incurs significantly higher VM down-

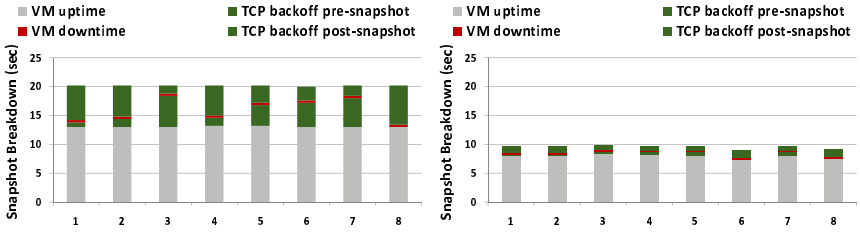
[1] We were not able to evaluate VNsnap-memory for the 16-node VIOLIN as there was not enough memory in each physical host to keep the snapshots of two local VMs.

time as well as overall disruption period (from 10 to 15 seconds). The 16-node experiment further indicates that Xen live checkpointing not only suffers from longer downtime (about 20 seconds vs. less than 1 second in VNsnap-disk), but the downtime also scales with the number of VMs that are simultaneously being checkpointed on the same host (about 20 seconds with two VMs per host vs. about 10 seconds with one VM per host as in the 2, 4, and 8-node cases).

To explain why VNsnap-memory leads to a smaller TCP backoff than VNsnap-disk, we present the detailed results from the 8-VM VIOLIN experiment. Figure 7 shows the individual result for *each* of the 8 VMs in the VIOLIN. As discussed in Section 4.1, differences in VM snapshot completion times (shown by the upper edges of the "VM downtime" bars) lead to TCP backoff. As can be seen in Figure 7, the discrepancy among the 8 VMs is more significant for VNsnap-disk (up to 4 seconds – Figure 7(b)) than for VNsnap-memory (less than 1 second – Figure 7(c)). Our investigation reveals that some of the hosts (e.g. the ones hosting VMs 3, 6, and 7) have longer disk write latency than the others, leading to a noticeable difference in VM snapshot completion times for VNsnap-disk. On the other hand, VNsnap-memory does not involve disk writes (only memory writes) during snapshot and thus results in much less discrepancy.



(a) Xen Live Checkpointing



(b) VNsnap-disk (c) VNsnap-memory

Fig. 7. Per-VM breakdowns of snapshot timing for the 8-node VIOLIN running NEMO3D.

In all experiments, we verify the *semantic correctness* of NEMO3D execution by comparing the outputs of the following: (1) an uninterrupted NEMO3D

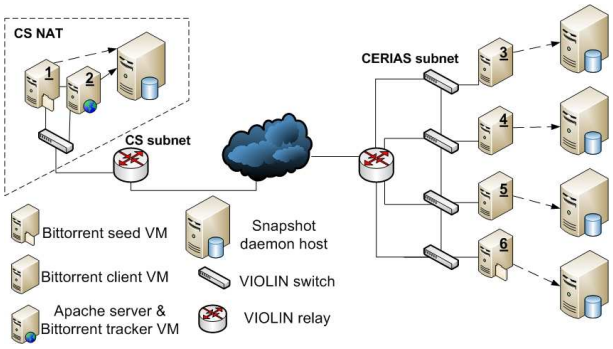


Fig. 8. The setup of the BitTorrent experiment

execution, (2) a NEMO3D execution during which a VIOLIN snapshot is taken, and (3) a NEMO3D execution restored from the VIOLIN snapshot. We confirm that all executions generate the same program output.

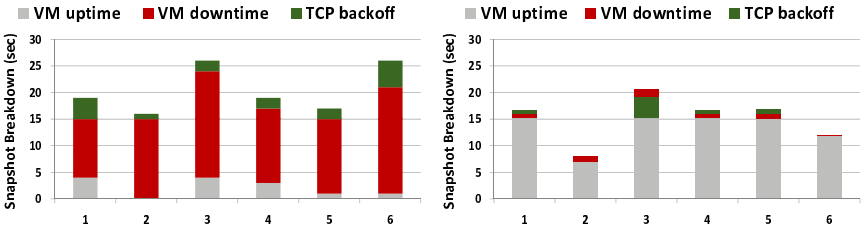
4.3 Snapshot of VIOLIN Running BitTorrent

In this section we study the impact of VNsnap on a VIOLIN running the peer-to-peer BitTorrent application [11]. The reason for choosing this application is to demonstrate the effectiveness of VNsnap for a VIOLIN running a communication and disk I/O-intensive application that spans multiple network domains. Figure 8 shows the experiment setup, where the VIOLIN spans two different subnets at Purdue University. Our testbed consists of 3 Sunfire servers in our lab at the Computer Science (CS) Department and 8 servers at the Center for Education and Research in Information Assurance and Security (CERIAS). In the CS subnet, we dedicate one host to run a remote VNsnap-memory daemon. Of the remaining two hosts, we use one to run a VIOLIN relay daemon (explained shortly) and the other one to host two VMs: VM 1 (with 700MB of memory) runs as a BitTorrent seed while VM 2 (with 350 MB of memory) runs an Apache webserver and a BitTorrent tracker. In the CERIAS subnet, we use four hosts each hosting a VM with 1GB of memory that runs as a BitTorrent client or seed. The remaining four hosts each run a VNsnap-memory daemon. The 6 VMs – two in CS and four in CERIAS – form the BitTorrent network. To overcome the NAT barrier between the two subnets, we deploy two software-based VIOLIN relays operating at the same level as the VIOLIN switches. The VIOLIN relays run in hosts with both public and private network interfaces so that they can tunnel VIOLIN traffic across the NAT.

The goal of the BitTorrent network is to distribute a 650MB file from two seeds (VMs 1 and 6) to all participating clients (VMs 3, 4, and 5). The experiment starts with the two seeds, one in CS and one in CERIAS. We trigger the VIOLIN snapshot when all clients have downloaded almost 50% of the file. At that time,

the average upload and download rates for each client are about 1350KB/s and 3200KB/s, respectively.

Figure 9 compares the per-VM snapshot timing breakdown under Xen's live checkpointing and under VNsnap-memory. We observe that the total disruption caused by the snapshot operation (i.e. VM downtime+TCP backoff) is considerably less – and at times negligible – for VNsnap-memory (all below 2 seconds except VM 3 – Figure 9(b)). The disruption periods under Xen live checkpointing range from 15 seconds to 25 seconds. Moreover, the slower disk bandwidth on some hosts (i.e. those hosting VMs 3 and 6) causes large discrepancy (up to 10 seconds) among the VMs' snapshot completion times, leading to non-trivial TCP backoff (Figure 9(a)).



(a) Xen Live Checkpointing (b) VNsnap-memory Remote

Fig. 9. Per-VM breakdowns of snapshot timing for the VIOLIN running BitTorrent.

When looking at the result for VNsnap-memory (Figure 9(b)), one notices that the VM snapshot completion times are *less* uniform than those in the NEMO3D experiments. There are three reasons behind this observation: First, as described in the experiment setup, not all VMs are configured with the same amount of memory. For instance, given that VM 2 has only 350MB of memory, it completes snapshot before other VMs. Second, unlike the NEMO3D experiment where all VMs are equally active, some VMs in the BitTorrent experiment are more active than others (i.e. they have larger WWS). For example, at the time of the snapshot, the three client VMs (VMs 3, 4, and 5) are mostly communicating with VM 1, leaving the other seed (VM 6) mostly idle and thus a shorter snapshot duration for VM 6. Third, the workloads of the hosts are not uniform, which can have an impact on the VM snapshot times. For example, due to resource constraints of our testbed, we have to run the CERIAS VIOLIN relay in the same server that runs a VNsnap-memory daemon. As a result, it takes VM 3, which is served by that daemon, longer time to finish its snapshot despite the fact that VM 3 is just as busy as other clients (VMs 4 and 5). The longer duration of VM 3 snapshot manifests itself as the TCP backoff during which VM 3 becomes the only pre-snapshot VM in the VIOLIN. Overall, the BitTorrent results demonstrate the effectiveness of VNsnap even under less than favorable conditions and a non-uniform configuration. Finally, we verify the correctness of

VNsnap by comparing the checksum of the original file with the checksums of the files downloaded during the run when the snapshot is taken and during a run restored from the snapshot.

5 Discussion

In this section, we discuss some issues with VNsnap and propose future enhancements. The first issue is the negative impact of VM snapshot completion time discrepancy on TCP throughput – especially for VNsnap-disk. This problem can be substantially alleviated if we further modify the VM live migration implementation in *xend*. As part of our future work, we plan to do so such that *xend* spends a *constant* amount of time transferring VM memory pages to the VNsnap daemons. As such, all VMs in a VIOLIN will start their "stop and copy" phase at about the same time. Considering the very short duration of this phase (i.e. the VM downtime), their completion times for the VMs will be of low discrepancy.

The second issue is the size of VIOLIN snapshots. We note that, through efficient hash-based mass storage techniques (e.g. [12, 13]), similarities between different yet similar VM snapshots can be exploited. For instance, in a VIOLIN running NEMO3D, the VMs share many pages for the OS, library, and application code. Meanwhile, the similarity between consecutive snapshot images of the same VM can also be exploited for improved storage efficiency.

Finally, for a VIOLIN snapshot to be restorable, the VIOLIN has to be self-contained. This means that any application inside the VIOLIN should not depend on any connections to *outside* of the VIOLIN. In addition, VNsnap requires that applications running inside a VIOLIN be able to tolerate the short period of disruption incurred by VNsnap. We believe that many – though not all – applications meet such requirements.

6 Related Work

Many techniques have been proposed to checkpoint distributed applications, but very few have addressed the need for checkpointing an entire execution environment, including the applications, OS and file system. These techniques can be loosely categorized into application-level, library-level (e.g. [15–17]), and OS-level (e.g. [14, 18]) checkpointing. Although these techniques are beneficial in their own rights and work best in specific scenarios, each comes with limitations: Application-level checkpointing requires access to application source code and is highly semantics-specific. Similarly, only a certain type of applications can benefit from linking to a specific checkpointing library. This is because the checkpointing library is usually implemented as part of the message passing library (such as MPI) that not all applications use. OS-level checkpointing techniques often require modifications to the OS kernel or require new kernel modules. Moreover, many of these techniques fail to maintain open connections and accommodate application dependencies on local resources such as IP addresses,

process identifiers (PIDs), and file descriptors. Such dependencies may prevent a checkpoint from being restorable on a new set of physical hosts. VNsnap complements the existing techniques yet is not without its own limitations (Section 5).

Virtualization has emerged as a solution to decouple application execution, checkpointing and restoration from the underlying physical infrastructure. ZapC [19] is a thin virtualization layer that provides checkpoint/restart functionality for a self-contained virtual machine abstraction, namely a pod (PrOcess Domain), that contains a group of processes. Due to the smaller checkpointing granularity (a pod vs. a VM), ZapC is more efficient than VNsnap in checkpointing a group of processes. However, ZapC does not capture the *entire* execution environment which includes the OS itself. Xen on InfiniBand [20] is a Xen-based solution with a goal similar to VNsnap. But it is designed exclusively for the Partitioned Global Address Space programming models and the InfiniBand network. Hence, unlike VNsnap, it does not work with legacy applications running on generic IP networks.

Recently, two solutions have been proposed based on Xen migration. [21] advocates using migration as a proactive method to move processes from "unhealthy" nodes to healthy ones in a high performance computing environment. Though this method can be used for planned outages or predictable failure scenarios, it does not provide protection against unexpected failures nor restore distributed execution states in the event of such failures. Remus [22] is a practical, guest transparent high-availability service that protects unmodified software against physical host failures. The focus of Remus is individual VMs whereas VNsnap focuses on distributed VNEs. Remus leverages an enhanced version of Xen migration to efficiently transfer a VM state to a backup site at high frequency (i.e. 40 times per second); whereas VNsnap is triggered at a much lower frequency, which can be determined by existing solutions (e.g. [23]) based on mean-time to failure prediction.

7 Conclusion

We have presented VNsnap as a middleware system to take snapshots of an entire VNE, which include images of the VMs with their execution, communication, and storage states. To minimize system downtime incurred by VNsnap, we develop optimized live VM snapshot techniques inspired by Xen's live VM migration function. We also implement a distributed snapshot algorithm to enforce causal consistency among the communicating VMs. Our experiments with VIOLINs running unmodified OS and real-world parallel/distributed applications demonstrate the unique usability of VNsnap in achieving reliability for an entire VNE.

References

1. Kangarlou, A., Xu, D., Ruth, P., Eugster, P.: Taking Snapshots of Virtual Networked Environments. In: 2nd International Workshop on Virtualization Technology in Dis-

tributed Computing (2007)
2. Jiang, X., Xu, D.: VIOLIN: Virtual Internetworking on Overlay Infrastructure.: Technical Report CSD TR 03-027, Purdue University (2003)
3. Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I., Warfield, A.: Xen and the Art of Virtualization. In: ACM SOSP (2003)
4. Jiang, X., Xu, D.: vBET: a VM-Based Emulation Testbed. In: ACM Workshop on Models, Methods and Tools for Reproducible Network Research (2003)
5. Jiang, X., Xu, D., Wang, H. J., Spafford, E. H., Virtual Playgrounds for Worm Behavior Investigation. In: 8th International Symposium on Recent Advances in Intrusion Detection (RAID'05) (2005)
6. Clark, C., Fraser, K., Hand, S., Hansen, J. G.: Live Migration of Virtual Machines. In: USENIX NSDI (2005)
7. Mattern, F.: Efficient Algorithms for Distributed Snapshots and Global Virtual Time Approximation. Journal of Parallel and Distributed Computing, 18 (1993)
8. RFC1323, `http://www.ietf.org/rfc/rfc1323.txt`
9. LVM2, `http://sources.redhat.com/lvm2/`
10. NEMO3D, `http://cobweb.ecn.purdue.edu/~gekco/nemo3D/`
11. BitTorrent, `http://www.bittorrent.com`
12. Warfield, A., Ross, R., Fraser, K., Limpach, C., Hand, S.: Parallax: Managing Storage for a Million Machines. In: USENIX Workshop on Hot Topics in Operating Systems (2005)
13. Bobbarjung, D. R., Jagannathan, S., Dubnicki, C., Improving Duplicate Elimination In Storage Systems. In: ACM Transactions on Storage (TOS), vol. 2 (2006)
14. Osman, S., Subhraveti, D., Su, G., Nieh, J.: The Design and Implementation of Zap: A System for Migrating Computing Environments. In: USENIX OSDI (2002)
15. Sankaran, J., Squyres, J. M., Barret, B., Lumsdaine, A. Duell, J. Hargrove, P. Roman, E.: The LAM/MPI Checkpoint/Restart Framework: System-Initiated Checkpointing. In: Proceedings of the LACSI Symposium (2003)
16. Fagg, G. E., Dongarra, J.: FT-MPI: Fault Tolerant MPI, Supporting Dynamic Applications in a Dynamic World. In: The 7th European PVM/MPI User's Group Meeting. LNCS, vol. 1908 (2000)
17. Chen, Y., Plank, J. S., Li.: CLIP - A Checkpointing Tool for Message-Passing Parallel Programs. In: IEEE Supercomputing (SC97) (1997)
18. Zhong, H., Nieh, J.: Linux Checkpoint/Restart As a Kernel Module.: Technical Report CUCS-014-01, Department of Computer Science, Columbia University (2001)
19. Laadan, O., Phung, D., Nieh, J.: Transparent Checkpointing-Restart of Distributed Applications on Commodity Clusters. In: IEEE International Conference on Cluster Computing (2005)
20. Scarpazza, D. P., Mullaney, P., Villa, O., Petrini, F., Tipparaju, V., Nieplocha, J.: Transparent System-Level Migration of PGAS Applications Using Xen on Infiniband. In: IEEE International Conference on Cluster Computing (2007)
21. Nagarajan, A. B., Mueller, F., Engelmann, C., Scott, S. L.: Proactive Fault Tolerance for HPC with Xen Virtualization. In: ACM International Conference on Supercomputing (ICS) (2007)
22. Cully, B., Lefebvre, G., Meyer, D., Freeley, M., Hutchinson, N., Warfield, A., : Remus: High Availability via Asynchronous Virtual Machine Replication. In: USENIX NSDI (2008)
23. Ren, X., Eigenmann, R., Bagchi, S.: Failure-Aware Checkpointing in Fine-Grained Cycle Sharing Systems. In: IEEE International Symposium on High Performance Distributed Computing (2007)