CERIAS Tech Report 2008-11

VNsnap: Taking Snapshots of Virtual Networked Environments with Minimal Downtime

by Ardalan Kangarlou, Dongyan Xu, Patrick Eugster
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, Patrick Eugster, Dongyan Xu
Dept. of Computer Science, Purdue University, West Lafayette, IN 47907, USA
{ardalan,p,dxu}@cs.purdue.edu, 765-494-6182, 765-494-0739(fax)

Abstract

A virtual networked environment (VNE) consists of virtual machines (VMs) connected by a virtual network. It has been adopted to create "virtual infrastructures" for individual users on a shared cloud computing infrastructure. The ability to take snapshots of an entire VNE — including images of the VMs with their execution, communication and storage states — yields a unique approach to reliability as a snapshot can restore the operation of an entire virtual infrastructure. We present VNsnap, a system that takes distributed snapshots of VNEs. Unlike existing distributed snapshot/checkpointing solutions, VNsnap does not require any modifications to the applications, libraries, or (guest) operating systems running in the VMs. Furthermore, VNsnap incurs only seconds of downtime as much of the snapshot operation takes place concurrently with the VNE's normal operation. We have implemented VNsnap on top of Xen. Our experiments with real-world parallel and distributed applications demonstrate VNsnap's effectiveness and efficiency.

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, mutually isolated "virtual infrastructures" serving individual users or groups. For example, a virtual cluster can be created to execute parallel jobs with its own root privilege and customized runtime library; a virtual data sharing 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. With the emergence of cloud computing [5] and "infrastructure as a service" (IaaS) paradigms, the VNE is expected to receive more attention.

To bring reliability and resume-ability to VNEs, it is highly desirable that the underlying hosting infrastructure provide the capability of taking a distributed snapshot of an entire VNE, including images of the execution, communication, and storage states of all VMs in the VNE. 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 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 operating system (OS) levels, VNsnap does not require any modifications to software running inside the VMs and thus works with *unmodified* applications and (guest) OSes that *do not* have built-in snapshot/checkpointing support. VNsnap is especially useful for virtual infrastructure hosting in cloud computing, where the host is required to provide virtual

*This report, submitted in March 2009, supersedes an earlier version of the report (with the same title) submitted in April 2008.

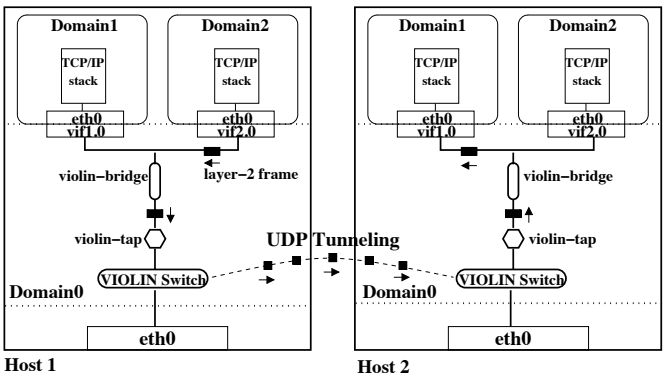


Figure 1: A 4-VM VIOLIN based on Xen, hosted by two physical machines.

infrastructure recoverability without knowing the details of guest VM setup. As such, VNsnap fills a void in the spectrum of checkpointing techniques and complements (instead of replacing) 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 [17], 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 introduces 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, we instantiate a classic global snapshot algorithm and show its *applicability* to taking VNE snapshots.

We have implemented a Xen [6] based VNsnap prototype for VIOLIN [15] – our instantiation of the VNE concept. To evaluate the VIOLIN downtime incurred by VNsnap and its impact on applications, we use two real-world parallel/distributed applications – one is a legacy parallel nanotechnology simulation *without* built-in checkpointing capability while the other is BitTorrent, a peer-to-peer file sharing application. 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 Overview

For completeness, we give a brief overview of the VIOLIN virtual networked environment and a previous VIOLIN snapshot prototype presented in [17]. Based on Xen, a VIOLIN virtual networked environment (or "VIOLIN" for 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 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 [14, 16], VIOLINs run as virtual "testing grounds" 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 running in domain 0 (the driver/management domain of Xen) of their respective physical hosts. Each VIOLIN switch intercepts link-

level traffic generated by the VMs – in the form of layer-2 Ethernet frames – and tunnels them to their destination hosts using the UDP transport protocol. VIOLIN snapshots are taken by VIOLIN switches from outside the VMs. As such, there is no need for modifying the application, library, or OS (including the TCP/IP protocol stack) 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 its IP address space is totally orthogonal to that of the underlying hosting infrastructure.

In the previous work [17], we presented the first prototype for taking VIOLIN snapshots. Unfortunately, that prototype has serious limitations: 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 regain its full execution speed, following a VIOLIN snapshot.

3 VNsnap Design and Implementation

In this section, we present the design and implementation of VNsnap. 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

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 VM *live migration* function [10]: instead of freezing a VM throughout the snapshot [17], 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 the source host to the 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 relatively 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. We adopted the same set of parameters used to determine the last iteration in Xen migration for VNsnap.

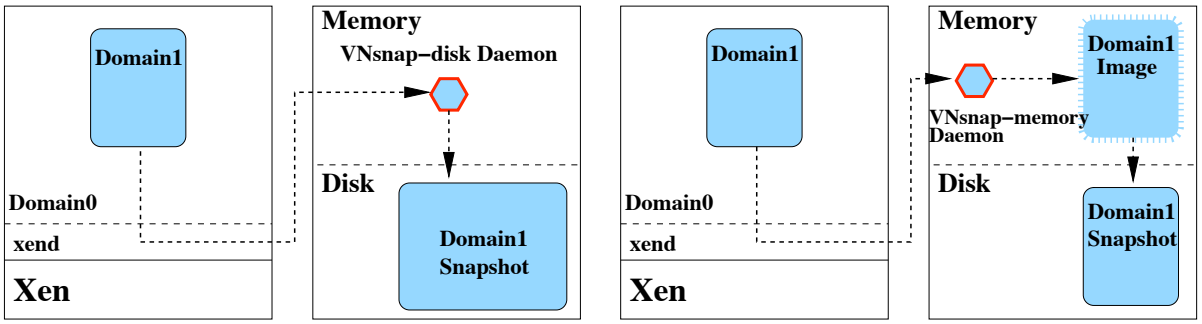
Following the same principle, our optimized live VM checkpointing technique effectively migrates a running VM's memory state to a local or remote snapshot file but *without* a switch of control (namely the same VM will keep running). To facilitate such migration, we create the *snapshot daemon* that "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 once 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 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, without any offline processing of the snapshot file, the restoration will take longer time compared with restoring a snapshot file generated by Xen's original live checkpointing function.



(a) VNsnap-disk

(b) VNsnap-memory

Figure 2: Designs of VNsnap-disk and VNsnap-memory for optimized live VM snapshot.

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 keeps only 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 further 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 explained and demonstrated in

Sections 3.2.3 and 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 in the disk 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 is not done by simply reusing some existing features of Xen. It might seem that a VM snapshot can 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, live migration in Xen 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 need the canonicalized pages so that the snapshot can be restored on any host in the future. 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. Third, a VM migration allocates and locks portions of the memory on the destination host to be used by the migrated VM. Such allocation can reduce memory available to domain 0 and potentially other domains in the future. VNsnap-memory avoids such fixed allocation of memory by allocating memory from the heap that can be swapped to disk.

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 (equivalent to Xen 3.1), but it can be easily ported to other VMMs that support live migration (e.g., VMware). We point out that both daemons can run locally or remotely. For the local run it would be helpful 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

3.2.1 Overview

With the individual VM snapshots achieving minimal downtime, we now present how to coordinate these VM snapshots in creating a consistent, distributed snapshot of a VIOLIN. We adopt a simplified version of Mattern's distributed snapshot algorithm which is based on message coloring [19]. In VNsnap, the algorithm is executed by the VIOLIN switches on the layer-2 Ethernet frames generated by the VMs.

We point out that distributed snapshot algorithms have long been proposed and applied [24, 13, 11, 18, 25, 23] and thus *are not our contribution*. The contribution of VNsnap is the application of a classic snapshot algorithm to the emerging virtualized environments, as well as the proof of its *applicability*. The applicability is not straightforward for the following reasons: First, in previous application scenarios, the message-passing layer is responsible for executing the snapshot algorithm. However, in VNsnap the algorithm is executed by VIOLIN switches *outside* the VMs yet the goal is to guarantee causal consistency for transport state *inside* the VMs in VIOLIN. Second, Mattern's original algorithm assumes *reliable* communication channels, whereas in VNsnap, the VIOLIN switches forward layer-2 frames (encapsulating the

TCP/UDP packages from the VMs) between each other through *non-reliable* (fair-lossy by assumption) UDP tunneling (recall Figure 1). Third, unlike some previous scenarios that require extra logging functions to ensure correct message delivery (e.g., [23]), the VIOLIN switches *do not* maintain any VM's internal transport protocol state. Finally, previous works require modification to application, library, and/or OS when applying the algorithm; whereas VNsnap does not require any modification to the VMs' application and system software (including the network protocol stack).

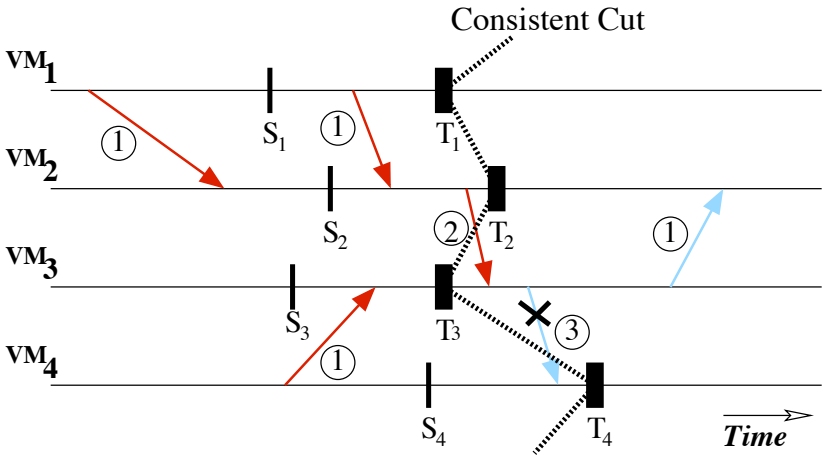


Figure 3: Illustration of VNsnap's snapshot algorithm: The snapshot of VM_i begins at time S_i and ends at T_i.

In VNsnap, the snapshot algorithm works as follows: One VIOLIN switch (or "switch") initiates a run of the algorithm by sending a TAKE_SNAPSHOT control message to all switches running for the same VIOLIN. This represents the initialization of an agreement protocol (e.g., 2PC). Upon receiving the TAKE_SNAPSHOT message or a frame from a post-snapshot VM, a VIOLIN switch starts the snapshot operations on the VMs in the same physical host. While a VM snapshot is in progress, its underlying VIOLIN switch colors that VM with a *pre-snapshot* color and prevents the delivery of frames from any *post-snapshot* colored VM. Once the VM's snapshot is completed, the switch will color the VM with post-snapshot color. When all VM snapshots in the same host are completed, the switch notifies the initiator via a SUCCESS message. If the initiator receives SUCCESS messages from all switches of the VIOLIN, the agreement protocol terminates by informing the switches to commit the snapshots (otherwise to discard them).

At the heart of the algorithm lie the different treatments of layer-2 frames transmitted between VIOLIN switches. Before describing the details, we first define the term "epoch": For a VM, an epoch is the continuous interval between the completion times of two consecutive snapshot operations. In Figure 3, time T_i is when the snapshot of VM_i completes and thus it marks the end of one epoch and the beginning of the next epoch for VM_i ($1 \leq i \leq 4$). A frame falls into one of the following three categories:

1. A frame whose source and destination VMs are in the same epoch (e.g., the frames labeled 1 in Figure 3). Category 1 frames will be delivered to the destination VMs.

2. A frame whose source VM is one epoch behind the destination VM (e.g. the frame labeled 2 in Figure 3). Category 2 frames will be delivered to the destination VMs.

3. A frame whose source VM is one epoch ahead of the destination VM (e.g., the frame labeled 3 in Figure 3). Category 3 frames are *dropped* by the destination VIOLIN switches.

3.2.2 Applicability of Algorithm

Our proof of applicability needs to show that the snapshot algorithm, executed *outside* a VIOLIN, will preserve the semantics of application-level message passing communication via (unmodified) TCP or UDP *inside* the VIOLIN. For space constraint, we will focus on the case of TCP while the proof for the UDP case is much simpler and thus omitted. Inside the VMs, the TCP transport protocol achieves reliable message delivery via acknowledgement, time-out and re-transmission semantics. Interestingly, we will show that it is TCP's semantics that preserve the correctness of application-level communications in the face of the snapshot algorithm.

Proof sketch. The proof sketch has two parts. In the first part, we will show that, when restoring a VIOLIN snapshot, the semantics of application-level message transport using TCP will be preserved as in the original execution during which the snapshot is taken[1]. Suppose, in the original execution, VM_1 sends a message m to VM_2 via TCP. Let P be the set of TCP packets that carry the content of message m. Let $VS(VM_i)$ be the VIOLIN switch running in the host of $VM_i (i = 1, 2)$. Let $T_i (i = 1, 2)$ be the time when the snapshot operation of VM_i completes and let the epoch before T_i be epoch e and the one after T_i be epoch $e + 1$. To show that message m will be successfully delivered in the execution restored from the VIOLIN snapshot, we will show that for each packet $p \in P$, following VIOLIN snapshot restoration, VM_2 will eventually see the receipt of p and VM_1 will eventually see the acknowledgment of p – denoted as ACK_p. Packet p is encapsulated in a layer-2 frame, which is then tunneled from $VS(VM_1)$ to $VS(VM_2)$. Let $f(p)$ be the frame that successfully arrives at $VS(VM_2)$ (recall the unreliable UDP tunneling). $f(p)$ falls into one of the following cases:

Case 1: $f(p)$ is a category 3 frame. This means that $f(p)$ is sent by $VS(VM_1)$ in epoch $e + 1$ and received by $VS(VM_2)$ in epoch e. According to the snapshot algorithm, category 3 frame $f(p)$ will be dropped by $VS(VM_2)$ and will not be delivered to VM_2. As a result, the snapshot of VM_2 *does not* record the receipt of p and the snapshot of VM_1 does not record the receipt of ACK_p. Upon VIOLIN snapshot restoration, VM_1 will, by TCP semantics, re-transmit p to VM_2.

Case 2: $f(p)$ is a category 2 frame. This means that $f(p)$ is sent by $VS(VM_1)$ in epoch e and received by $VS(VM_2)$ in epoch $e+1$. As such, the snapshot of VM_2 *does not* record the receipt of p but the snapshot of VM_1 *does* record the sending of p. We can further infer that the snapshot of VM_1 *does not* record the receipt of ACK_p – If it did, the layer-2 frame that encapsulates ACK_p would have been sent by $VS(VM_2)$ in epoch $e + 1$ and received by $VS(VM_1)$ in epoch e. This contradicts the snapshot algorithm which drops category 3 frames. Upon snapshot restoration, VM_1 will, by TCP semantics, re-transmit p to VM_2.

Case 3: $f(p)$ is a category 1 frame. Here we have two sub-cases:

Case 3.1: VM_1 transmits p and receives ACK_p in the same epoch. (*Case 3.1.1*:) If both happen in epoch e, the snapshot of VM_1 will record the transmission and acknowledgment of p. We further infer that the snapshot of VM_2 records the receipt of p: if not, ACK_p would have been carried by a category 3 frame, contradicting the algorithm. Right upon snapshot restoration, both VM_1 and VM_2 will consider p successfully delivered. (*Case 3.1.2*:) If both happen in epoch $e + 1$, the snapshots of VM_1 and VM_2 do not record p's transmission and p will be re-transmitted after snapshot restoration.

Case 3.2: VM_1 transmits p in epoch e and receives ACK_p in epoch $e+1$. As such the snapshot of VM_1 *does not* record the receipt of ACK_p. Upon snapshot restoration, VM_1 will, according to TCP semantics, re-transmit p to VM_2. Note that VM_2 may or may not have received p in epoch e. But in either case VM_2 will send ACK_p to VM_1 upon receiving the re-transmitted p, according to TCP semantics.

In the second part of the proof sketch, we show that, when restoring a VIOLIN snapshot, the semantics of TCP connection establishment and tear-down will be preserved as in the original execution. These semantics are specified by the well-known TCP state transition diagram [26]. The TCP state transitions are triggered

[1]We assume that there is no host, VM, or network failure during VIOLIN snapshot taking and restoration. The handling of failures is done outside of the snapshot algorithm.

by the receipt and/or transmission of a packet with its SYN or FIN control bit set and the receipt of its corresponding ACK. Conveniently, the transmission, acknowledgment, and possibly re-transmission of these control packets follow the same semantics as that of the TCP packet p in the first part of the proof sketch. As a result, we can basically follow the same logic in the first part to show that, following snapshot restoration, a control packet will eventually be transmitted and acknowledged, which will trigger the proper TCP state transitions on both sides of the TCP connection.

As an example, suppose in the original execution, VM_2 (client) is trying to establish a TCP connection with VM_1 (server). During TCP's three-way handshake, VM_1 completes its snapshot while its TCP state is SYN_RCVD. At that moment, VM_1 has sent control packet SYN,ACK to VM_2 but has not received the corresponding ACK. On the other side, VM_2 receives SYN,ACK, sends an ACK to the now *post-snapshot* VM_1, enters the ESTABLISHED state, and then completes its snapshot. Upon VIOLIN snapshot restoration, it may appear that the two VMs were in inconsistent states, with VM_1 stuck in SYN_RCVD state waiting for the ACK already sent by VM_2. However, such inconsistency won't last thanks to the TCP semantics: VM_1 will time-out and re-transmit SYN,ACK to VM_2, which will in turn re-send ACK to VM_1. After that both VMs are in ESTABLISHED state and the TCP connection is established.

The proof sketch above covers the entire life cycle of a TCP connection inside the VIOLIN. One can see that the TCP semantics play a critical role in showing the applicability of the snapshot algorithm, despite the differences between VIOLIN and previous application scenarios (Section 3.2.1). Using a similar proof logic, we can check the algorithm's applicability under other connection-oriented, reliable transport protocols. Our work builds a "bridge" between the classic algorithm and practice – with particular relevance to the emerging virtualized infrastructures.

3.2.3 Implementation

In our implementation, a VIOLIN switch enters the SNAPSHOT state when it starts the snapshot-taking operations for the local VMs connected to it. It exits the SNAPSHOT state when all the VM snapshots have completed. To handle the asynchronous completion of VM snapshots on the same host, VNsnap implements 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 frame from a post-snapshot VM can reach a pre-snapshot VM on the same 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. We also extend *xend* such that it will notify the VIOLIN switch whenever a VM finishes its snapshot operation. Specifically, we define a signal handler inside the VIOLIN switch which will receive a user-defined POSIX 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 state.

We point out that, although the snapshot algorithm preserves functional semantics in a VIOLIN, it does affect network performance in the VIOLIN. One direct impact of running the algorithm is the *TCP backoff* inside the VIOLIN. More specifically, since *not all* VMs finish their snapshot operations at the same time, the algorithm has to drop category 3 frames to enforce causal consistency between the VM snapshots. Such frame drop results in temporary backoff of active TCP connections inside the VIOLIN. The duration of the TCP backoff is directly related to the *degree of discrepancy* among the VMs' snapshot completion times. In fact, one of the motivations behind the design of VNsnap-memory daemon (Section 3.1.2) is to reduce such discrepancy by eliminating the impact of disk bandwidth on VM snapshot completion times. For UDP connections, Loss of UDP packets is "expected" from an application's perspective. If application semantics require recovery of lost UDP packets due to snapshot, it is the application's responsibility (e.g., through retransmission or erasure coding), while our system only preserves UDP's property (unreliable, non-FIFO). Similar argument can be made for other non-reliable protocols. Section 4 will present performance evaluation results for TCP connections.

So far we have 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 [1] 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 records only changes made to a logical volume after the snapshot and as a result is very fast. A more efficient way to use LVM snapshots can be found in [8]. 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 [2] and BitTorrent [3]. Throughout this section, we compare VNsnap with our previous work [17]. All physical hosts involved in our experiments are Sunfire V20Z servers with two 2.6GHz AMD Opteron processors and 4GB of RAM. In our setup, both domain-0 and guest domains run on the 2.6.18 Linux kernel.

4.1 Downtime Minimization for Live VM Snapshots

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

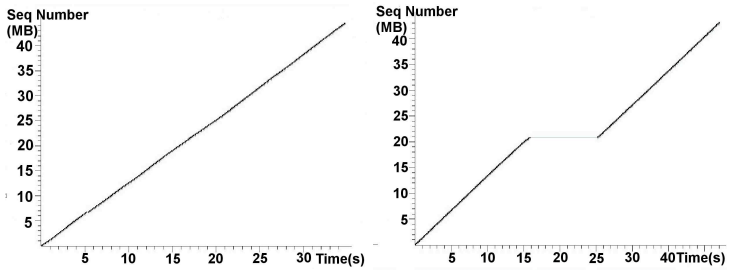
Table 1: Measurement results comparing three VM snapshot implementations for VNsnap.

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 [17]), (2) the VNsnap-disk daemon, and (3) the VNsnap-memory daemon. For each of the implementations we measure the metrics with the same VM 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) [10] 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 about 125MB/s.

The most important metric in Table 1 is the VM downtime. We have three main observations. 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. Third, VNsnap-memory achieves a much lower downtime for the "NEMO3D" run than VNsnap-disk. This is because the VNsnap-disk daemon directly writes the 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).



(a) No snapshot (b) Xen live checkpointing

(c) VNsnap-disk (d) VNsnap-memory

Figure 4: The impact of different VM snapshot techniques on TCP throughput in a VIOLIN running NEMO3D. Traces are obtained from tcpdump.

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 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 (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 on 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 3.1.2, 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 design, generate VM snapshots of the *same* size as the VM's memory image. A larger VM snapshot 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 generated by VNsnap-memory or Xen checkpointing.

Impact of VM snapshot on TCP throughput. As discussed in Section 3.2.3, individual VMs in a VIOLIN may complete their snapshots at different times and thus result in TCP backoff. Figure 4 shows such impact on a 2-VM VIOLIN executing NEMO3D, under no snapshot (Figure 4(a)), Xen live checkpointing (Figure 4(b)), VNsnap-disk (Figure 4(c)), and VNsnap-memory (Figure 4(d)). We focus on one TCP connection between the two VMs. The flat, "no progress" periods shown in Figures 4(b) and 4(c) each consist of two parts: (1) the downtime of the sender VM during snapshot and (2) the TCP backoff period due to the different snapshot completion times of the two VMs. We observe that both Xen live checkpointing (Figure 4(b)) and VNsnap-disk (Figure 4(c)) incur 2-3 seconds of TCP backoff, whereas VNsnap-memory (Figure 4(d)) does not incur noticeable TCP backoff. More results and analysis will be presented in the next two subsections.

4.2 Taking Snapshot of VIOLIN Running NEMO3D

NEMO3D is a long-running (tens of minutes to hours), legacy 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. 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 VIOLIN.

Figure 5 shows the results [2]. 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) – more specifically 0.0, 0.8, 1.4, and 3.8 seconds for the 2, 4, 8, and 16-node VIOLINs, respectively. 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 downtime as well as overall disruption period (from 10 to 35 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

[2] We would like to suggest color printing for viewing Figures 5, 6, and 8. We apologize for any inconvenience.

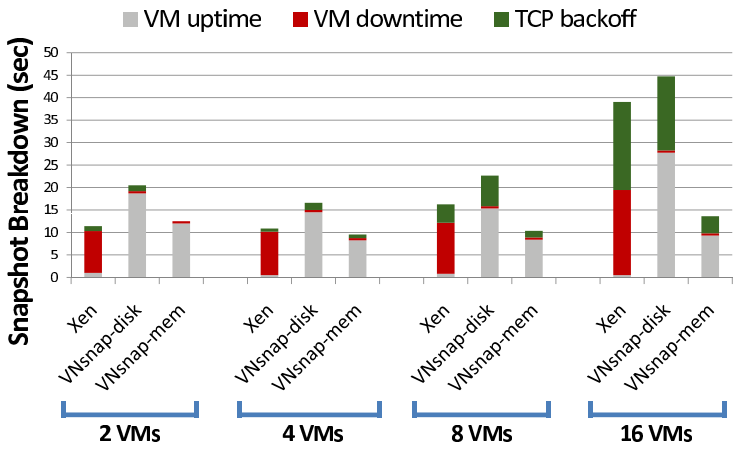


Figure 5: The breakdown of snapshot timing under different VM snapshot implementations for 2, 4, 8 and 16-node VIOLINs running NEMO3D.

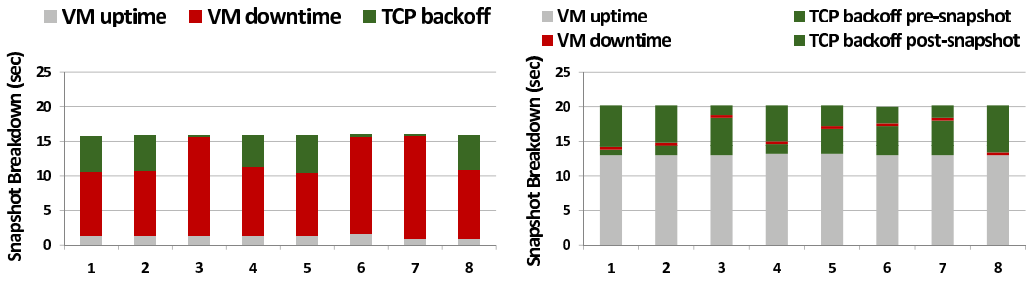
performing snapshot on the same host (about 20 seconds with two VMs per host vs. about 10 seconds with one VM per host as in the 2, 4, and 8-node cases).

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 6 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 6, the discrepancy among the 8 VMs is more significant for VNsnap-disk (up to 4 seconds – Figure 6(b)) than for VNsnap-memory (less than 1 second – Figure 6(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 and TCP backoff.

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

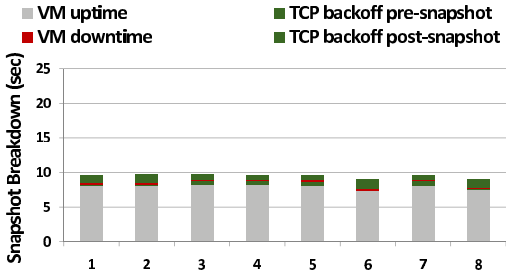
4.3 Taking Snapshot of VIOLIN Running BitTorrent

In this section we study the impact of VNsnap on a VIOLIN running the peer-to-peer BitTorrent application [3]. 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 7 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



(a) Xen Live Checkpointing

(b) VNsnap-disk



(c) VNsnap-memory

Figure 6: Per-VM breakdowns of snapshot timing for the 8-node VIOLIN running NEMO3D.

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 8 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 8(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 8(a)).

When looking at the result for VNsnap-memory (Figure 8(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

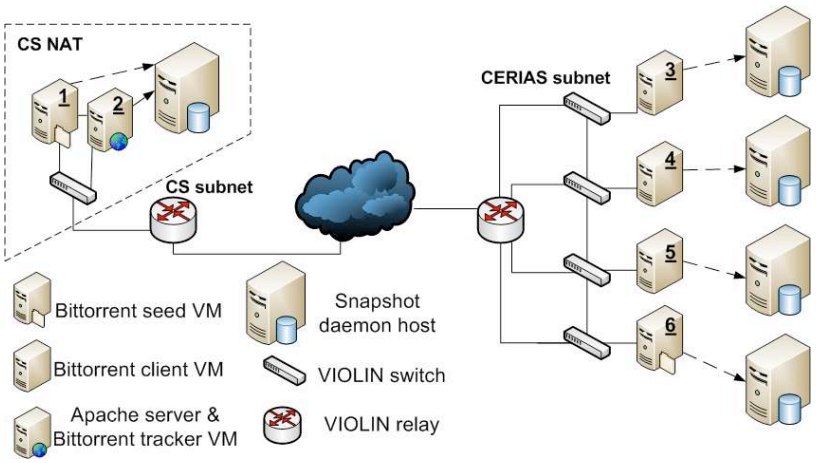
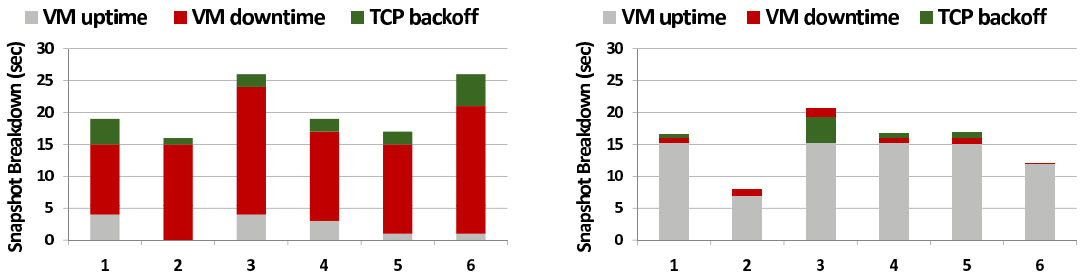


Figure 7: Setup of BitTorrent experiment



(a) Xen Live Checkpointing

(b) VNsnap-memory Remote

Figure 8: Per-VM breakdowns of snapshot timing for the VIOLIN running BitTorrent.

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 non-uniform host/VM conditions. Finally, we validate 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 improvement. 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 have *xend* spend a *uniform* or *bounded*

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 similarities between different yet similar VM snapshots can be exploited through efficient hash-based mass storage techniques (e.g. [27, 7]) and compression. 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. Such similarity can also be exploited during snapshot generation in order to reduce memory and network bandwidth utilization by VNsnap. A preliminary investigation using the *rsync* utility [4] shows that, for the 2-node NEMO3D experiment in Section 4.1, if two snapshots are taken 5 minutes apart, the difference between the two snapshots can be accounted for by 25MB of data (or 4% of the snapshot file).

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* 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 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. [24, 13, 11, 9]), and OS-level (e.g. [21, 29]) 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 [18] 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 [25] 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. [20] 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 does it restore distributed execution states in the event of such failures. Remus [12] 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 (e.g., every tens of minutes), which can be determined by existing solutions (e.g. [22]) based on mean-time to failure prediction. The most related work is an advanced system [8] that realizes a more powerful capability of highly transparent checkpointing of closed distributed systems in Emulab [28]. Being parallel efforts, VNsnap and [8] share similar goals with different system requirements: [8] requires high-accuracy clock synchronization and modifications to the guest OS, whereas VNsnap assumes VMs with unmodified software and no fine-grain clock synchronization.

7 Conclusion

We have presented the VNsnap 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 instantiate a distributed snapshot algorithm to enforce causal consistency across the VM snapshots and verify the algorithm's applicability. Our experiments with VIOLINs running unmodified OS and real-world parallel/distributed applications demonstrate the unique capability of VNsnap in supporting reliability for the emerging virtual infrastructures.

References

[1] `http://sources.redhat.com/lvm2`.

[2] `http://cobweb.ecn.purdue.edu/~gekco/nemo3D`.

[3] `http://www.bittorrent.com`.

[4] `http://samba.anu.edu.au/rsync`.

[5] M Armbrust et al. Above the clouds: A Berkeley view of cloud computing. *Technical Report No. UCB/EECS-2009-28, UC Berkeley*, 2009.

[6] 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. *ACM SOSP*, 2003.

[7] D. R. Bobbarjung, Jagannathan, and C. S., Dubnicki. Improving duplicate elimination in storage systems. *ACM Transactions on Storage (TOS)*, 2, 2006.

[8] A. Burtsev, P. Radhakrishnan, M. Hibler, and J. Lepreau. Transparent checkpoints of closed distributed systems in Emulab. *ACM EuroSys 2009*.

[9] Y. Chen, J. S. Plank, and K. Li. CLIP: A checkpointing tool for message-passing parallel programs. *SC97*, 1997.

[10] C. Clark, K. Fraser, S. Hand, and J. G. Hansen. Live migration of virtual machines. *USENIX NSDI*, 2005.

[11] A. Clematis and V. Ginuzzi. CPVM - extending PVM for consistent checkpointing. *IEEE PDP*, 1996.

[12] B. Cully, G. Lefebvre, D. Meyer, M. Freeley, N. Hutchinson, and A. Warfield. Remus: High availability via asynchronous virtual machine replication. *USENIX NSDI*, 2008.

[13] Graham E. Fagg and Jack J. Dongarra. Lecture notes in computer science 1 FT-MPI: Fault tolerant MPI, supporting dynamic applications in a dynamic world, 2000.

[14] X. Jiang and D. Xu. vBET: a VM-based emulation testbed. *ACM Workshop on Models, Methods and Tools for Reproducible Network Research*, 2003.

[15] X. Jiang and D. Xu. VIOLIN: Virtual Internetworking on Overlay INfrastructure. *Technical Report CSD TR 03-027, Purdue University*, 2003.

[16] X. Jiang, D. Xu, H. J. Wang, and E. H. Spafford. Virtual playgrounds for worm behavior investigation. *RAID 2005*.

[17] A. Kangarlou, D. Xu, P. Ruth, and P. Eugster. Taking snapshots of virtual networked environments. *2nd International Workshop on Virtualization Technology in Distributed Computing*, November 2007.

[18] O. Laadan, D. Phung, and J. Nieh. Transparent checkpoint-restart of distributed applications on commodity clusters. *IEEE International Conference on Cluster Computing*, 2005.

[19] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18:423–434, 1993.

[20] A. B. Nagarajan, F. Mueller, C. Engelmann, and S. L. Scott. Proactive fault tolerance for HPC with Xen virtualization. *ACM International Conference on Supercomputing (ICS)*, 2007.

[21] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. *USENIX OSDI*, 2002.

[22] X. Ren, R. Eigenmann, and S. Bagchi. Failure-aware checkpointing in fine-grained cycle sharing systems. *HPDC 2007*.

[23] J. F. Ruscio, M. A. Heffner, and S. Varadarajan. DejaVu: Transparent user-level checkpointing, migration, and recovery for distributed systems. *IEEE Parallel and Distributed Processing Symposium (IPDPS)*, 2007.

[24] Sriram Sankaran, Jeffrey M. Squyres, Brian Barrett, and Andrew Lumsdaine. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. In *in Proceedings, LACSI Symposium, Sante Fe*, pages 479–493, 2003.

[25] D. P. Scarpazza, P. Mullaney, O. Villa, F. Petrini, Tipparaju V., and J. Nieplocha. Transparent system-level migration of PGAS applications using Xen on Infiniband. *IEEE International Conference on Cluster Computing*, 2007.

[26] Richard W. Stevens. *TCP/IP Illustrated Volume 1*. Addison-Wesley, Reading, MA, 1996.

[27] A. Warfield, R. Ross, K. Fraser, C. Limpach, and S. Hand. Parallax: Managing storage for a million machines. *USENIX Workshop on Hot Topics in Operating Systems*, 2005.

[28] Brian White, Jay Lepreau, Leigh Stoller, Robert Ricci, Shashi Guruprasad, Mac Newbold, Mike Hibler, Chad Barb, and Abhijeet Joglekar. An integrated experimental environment for distributed systems and networks. In *OSDI 2002*, pages 255–270.

[29] H. Zhong and J. Nieh. Linux checkpoint/restart as a kernel module. *Technical Report CUCS-014-01, Department of Computer Science, Columbia University*, 2001.