VNsnap: Taking Snapshots of Virtual Networked Infrastructures in the Cloud

Ardalan Kangarlou, *Member, IEEE,* Patrick Eugster, *Member, IEEE* and Dongyan Xu, *Member, IEEE,*

Abstract—A virtual networked infrastructure (VNI) consists of virtual machines (VMs) connected by a virtual network. Created for individual users on a shared cloud infrastructure, VNIs reflect the concept of "Infrastructure as a Service" (IaaS) as part of the emerging cloud computing paradigm. The ability to take snapshots of an entire VNI – including images of the VMs with their execution, communication and storage states – yields a unique approach to reliability as a VNI snapshot can be used to restore the operation of the entire virtual infrastructure. We present VNsnap, a system that takes distributed snapshots of VNIs. Unlike many existing distributed snapshot/checkpointing solutions, VNsnap does not require any modifications to the applications, libraries, or (guest) operating systems running in the VMs. Furthermore, by performing much of the snapshot operation concurrently with the VNI's normal operation, VNsnap incurs only seconds of downtime. We have implemented VNsnap on top of Xen. Our experiments with real-world parallel and distributed applications demonstrate VNsnap's effectiveness and efficiency.

Index Terms—Virtual Infrastructure, Infrastructure-as-a-Service (IaaS), Cloud Computing, Distributed Snapshots, Reliability

1 Introduction

A virtual networked infrastructure (VNI) consists of multiple virtual machines (VMs) connected by a virtual network. In a shared cloud infrastructure, VNIs can be created as private, mutually isolated "virtual computing facilities" 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; a virtual "playground" can be established to emulate computer malware infection and propagation. With the emergence of cloud computing [1], especially its "Infrastructure as a Service" (IaaS) paradigm, the VNI is expected to gain more attention in research and practice.

To bring reliability and resume-ability to VNIs, it is highly desirable that the underlying cloud infrastructure provides the capability of taking distributed snapshots of an entire VNI. Such a snapshot includes images of all VMs in the VNI, preserving their execution, communication, and storage states. The snapshot can later be used to restore the entire VNI, thus supporting fault/outage recovery, system suspension and resumption, as well as troubleshooting, audit, and forensics.

In this paper, we present VNsnap, a system capable of taking distributed snapshots of VNIs. Based on a virtual machine monitor (VMM), VNsnap runs *outside* of the target VNI. Unlike many 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 intended for virtual infrastructure hosting in the cloud, the main technique behind the IaaS paradigm where VMs or VNIs can be requested on demand as a service by cloud users. VNsnap allows an IaaS provider (e.g., Amazon EC2) to support VNI recovery or replay, *without* knowing the details of a cloud user's VM setup or customization. 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 in taking VNI 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 secondary storage. As shown in our previous work [2], such downtime can be tens of seconds long, which disrupts both human users and applications in the VNI. Second, the snapshots of individual VMs have to be coordinated to create a *globally consistent* distributed snapshot of the entire VNI. Such coordination is essential to preserving the consistency of the VM execution and the application state when the VNI snapshot is restored in the future.

To address the first challenge, VNsnap introduces an efficient 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 VNI snapshots. Furthermore, we develop system-level techniques to mitigate the perfor-

• *A. Kangarlou, P. Eugster and D. Xu are with the Department of Computer Science, Purdue University, West Lafayette, IN, 47907.*
E-mail: {ardalan,p,dxu}@cs.purdue.edu

Manuscript received December 1, 2010; revised March 27, 2011.

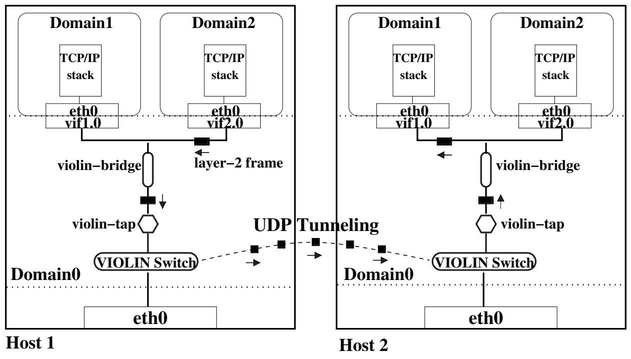


Fig. 1. A 4-VM VIOLIN based on Xen, hosted by two physical machines.

mance impact of VNsnap.

We have implemented a Xen-based [3] prototype of VNsnap for VIOLIN [4] – our instantiation of the VNI concept. To evaluate the VIOLIN downtime incurred by VNsnap and its impact on applications, we use two real-world parallel/distributed applications with no built-in checkpointing capability – one is a legacy parallel nanotechnology simulation 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 less than a second of VM downtime in all experiments.

2 VIOLIN Overview

For completeness, we give a brief overview of VIOLIN and a previous VIOLIN snapshot prototype presented in [2]. 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, network configuration, administrative privileges, and runtime support. VIOLIN has been deployed in a number of real-world systems: in the nanoHUB cyberinfrastructure (*http://www.nanoHUB.org*), VIOLINs run as virtual Linux clusters for executing a variety of nanotechnology simulation programs; in the vGround emulation testbed [5], 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 protocol. VIOLIN snapshots are taken by VIOLIN switches from outside the VMs. As such, there is no need to modify the application, library, or OS (including the TCP/IP protocol stack) that runs inside the VMs. A VIOLIN snapshot can be restored on any set of physical hosts without the need to reconfigure the VIOLIN's IP address space. This is due to the fact that VIOLIN performs layer-2 network virtualization. As a result, its IP address space is totally orthogonal to that of the underlying hosting infrastructure.

In our previous work [2], we presented the first prototype for taking VIOLIN snapshots. Unfortunately, that prototype has serious limitations: by leveraging Xen's live VM checkpointing capability, the system has to freeze each VM for a non-trivial period of time during which the VM's memory image is written to the disk. As a result, taking a VIOLIN snapshot causes considerable *downtime* to the VIOLIN, in the magnitude of tens of seconds. Moreover, due to TCP backoff incurred by the VM's long freeze, it will take extra 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 propose an optimized implementation that reduces network bandwidth consumption for periodic snapshots. Finally, we describe our solution to taking *distributed* snapshots of a VIOLIN with multiple communicating VMs.

3.1 Live VM Snapshots

3.1.1 Snapshot Daemon

VNsnap aims at minimizing the Xen live VM checkpointing downtime thus making the process of taking a VM snapshot *truly live*. 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. Our solution is inspired by Xen's *live VM migration* function [6]: instead of freezing a VM throughout the snapshot [2], we take a VM snapshot much the same way as Xen performs a live VM migration.

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 re-sent 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 re-sent to the destination host. After the completion of this "stop-and-copy" phase, 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* the switch of control (namely the same VM will keep running). To facilitate live snapshot, we introduce an entity called *snapshot daemon* that "impersonates" the destination host during live migration. 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.

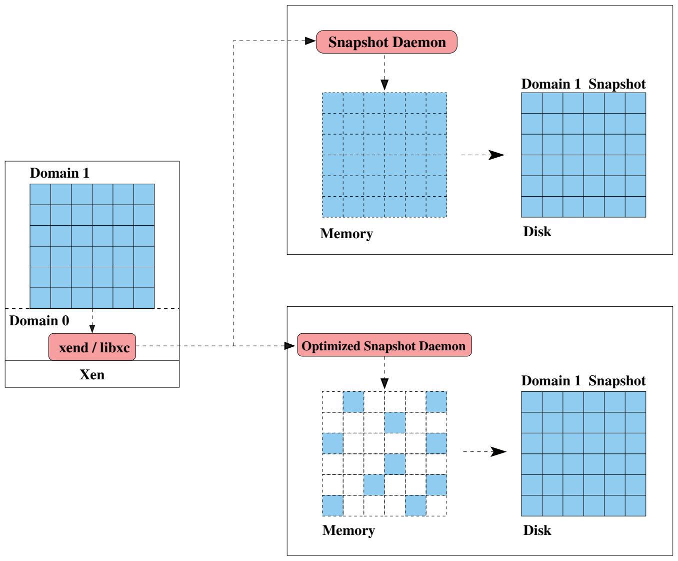


Fig. 2. Different snapshot daemon implementations.

The implementation of the snapshot daemon involves making modifications to the *xend* and *libxc* components of Xen that handle live VM migration. Our implementation is based on Xen 3.1, but it can be easily ported to other VMMs that support live migration (e.g., VMware ESX, KVM, etc.). The snapshot daemon can run either locally on the same host as where the VM is running or remotely on a different host. 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. On a single-core machine this can be done by the VMM which can enforce CPU capacity allocations to different domains, whereas 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 VM and snapshot daemon hosts for VM image transport. Next, we present a technique to improve the VM image transport efficiency.

3.1.2 Exploiting Snapshot Similarity

In this section, we present an optimization of the snapshot daemon implementation for a scenario where a long-running application runs in a VIOLIN while we *periodically* take its snapshots. In such a scenario, we have the opportunity to exploit the similarity between consecutive snapshots of each VM in the VIOLIN, with the goal of reducing the memory page transfer traffic between the VM and the snapshot daemon and improving network efficiency of the underlying cloud infrastructure. More specifically, in the design presented in Section 3.1.1, all memory pages of the VM will be transported to the snapshot daemon during the *first* iteration of the live VM snapshot operation. This will incur high traffic volume considering the typical size of a VM's memory image and the large number of VMs running in cloud data centers.

With the above background, we observe that there is a high degree of similarity between memory images of a VM at different times. There are two main reasons for such a similarity. (1) Most code pages of a VM (for both kernel code and user code) are read-only and thus do not change at runtime. (2) More importantly, for a range of long-running applications (including the ones presented in Section 4), we notice that the percentage of pages that are frequently dirtied is fairly low relative to the total number of memory pages of the VM. This property is characterized by the *writable working set (WWS)* concept utilized in live VM migration [6], where only a small subset of pages belonging to the WWS of a VM are frequently dirtied and have to be frozen and transported during the stop-and-copy phase of the VM migration. According to our profiling study on a number of applications, the locality revealed by the WWS during migration can also be exhibited over a longer period of time during the execution of these applications. As a result, we observe that a significant percentage of a VM's memory pages remain unchanged between two consecutive VIOLIN snapshot operations. If we skip those unchanged pages during the first iteration of the live VM snapshot operation (Figure 2, bottom – the white pages are skipped), a high volume of page transport traffic can be avoided in the underlying cloud infrastructure.

The first technical question for realizing the similarity-aware optimization is: Before taking a live VM snapshot, how to identify those pages that have remained unchanged since the previous snapshot? A body of previous work proposed content-based page sharing for VMs running on the same host in order to facilitate higher server consolidation (e.g., [7], [8]). Such solutions rely on computing the hash of memory pages of a VM in order to identify identical pages for potential sharing opportunities. We leverage a similar solution to identify identical pages across different snapshot rounds. More specifically, after completion of a VM snapshot operation, the snapshot daemon computes the hash values [9] of a VM's memory pages. It then sends the hash values back to the host where the VM is running. When the next snapshot starts, during the first iteration of the memory page transfer, *libxc* will compute the current hash of each page and compare it with the corresponding hash sent from the snapshot daemon. The page will be transported to the snapshot daemon only if the two hash values are distinct (which indicates that the page has been modified since the previous snapshot).

The above solution requires modification of the snapshot daemon and slightly changes the VM migration implementation. We will show in Section 4.1 that such a simple method can result in significant reduction in VM page transfer traffic. While hash computation is very fast (0.013 ms to compute the hash of a 4KB page), it can lengthen the snapshot operation for VMs with very large memory.

A more efficient solution, particularly for scenarios where very few pages get dirtied between two consecutive snapshot rounds is to identify modified pages by trapping writes to VM pages. Content-based page sharing systems, such as [8], also leverage such a mechanism to implement copy-on-write (COW) for shared pages among multiple VMs. Fortunately, Xen's *shadow mode logging* facilitates such a functionality if it is enabled *throughout* the VM execution. Once this mode is enabled, all VM pages become read-only so a write to a page results in a fault that can be tracked by the Xen hypervisor. In fact, shadow mode logging is used during live VM migration (and thus live snapshot) to keep track of the pages that have been modified since the previous iteration of migration. To prevent excessive write faults, once a dirty page is identified it can become writable again (only if the page is normally writable; e.g., it is not a page table) so that future writes do not result in faults. While handling faults does not incur any additional overhead during the snapshot operation, it slightly degrades VM execution between snapshots. Handling faults is slightly more expensive than computing hashes (it takes 0.160 ms for Xen to process a write fault), but for scenarios where very few pages change or when snapshots are very frequent, such a solution would be favorable to a hash-based approach.

The second technical question is: How to construct the complete snapshot image of a VM with partial memory transfer? Given that the snapshot daemon no longer receives the entire set of a VM's memory pages, the daemon has to merge the modified pages received in the current snapshot round with the unchanged pages received previously so that a new, complete snapshot can be generated. To facilitate easier assembly of the VM's snapshot, the snapshot daemon divides the snapshot image of a VM into three segments. The first segment contains metadata about the layout of the VM, such as the number of pages in the VM. The second segment which consists of all the memory pages of a VM is reused by the snapshot daemon across different snapshot rounds. Upon receiving a page, the snapshot daemon replaces the old copy of a page with its newly received copy. Finally, the third segment holds some execution state information such as the VM's virtual CPU context. Once the snapshot operation is complete, the updated three segments will be merged to create a complete VM snapshot image.

3.2 Taking Distributed VIOLIN Snapshot

3.2.1 Overview

With the individual VM snapshots achieving minimal downtime, we now present our approach to coordinating VM snapshots in order to obtain a globally consistent, distributed snapshot of a VIOLIN. We adopt a simplified version of Mattern's distributed snapshot algorithm [10] which is based on message coloring. 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 [11], [12], [13], [14], [15], [16] and thus *are not our contribution*. The contribution of VNsnap is the adaptation of a classic snapshot algorithm to the emerging cloud-based virtual infrastructures, as well as the proof of its *applicability*. The applicability is not straightforward for the following reasons. First, in previous application scenarios, the algorithm enforces causal consistency for the messages exchanged between the entities that execute the algorithm. However, in VNsnap, the algorithm is executed by VIOLIN switches *outside* the VMs, yet the goal is to guarantee causal consistency for the transport-level state *inside* the VMs. Second, Mattern's original algorithm assumes *reliable* communication channels, whereas in VNsnap, the VIOLIN switches forward layer-2 frames (encapsulating the TCP/UDP packets from the VMs) 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., [16]), the VIOLIN switches *do not* maintain any transport protocol state. Finally, previous works require modification to application, library, and/or OS when applying the algorithm, while VNsnap does not require any modification to the VMs' application and system software (including the network protocol stack).

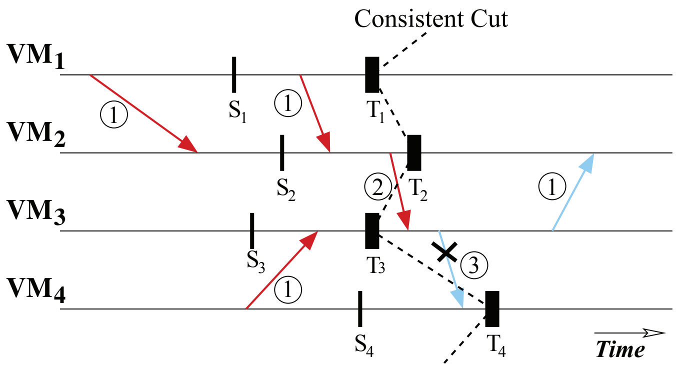


Fig. 3. Illustration of VNsnap's snapshot algorithm: The snapshot of VM_i begins at time S_i and ends at T_i.

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 for the VMs on the same physical host. While a VM snapshot is in progress, its underlying VIOLIN switch colors that VM and all the frames originating from that VM with the *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* of a VM, will preserve the semantics of application-level message passing communication via (unmodified) TCP or UDP *inside* of the VM. For space constraints, we will focus on the case of TCP while the proof for the UDP case is much simpler and will only be briefly discussed. 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. The proof 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, subsequently, 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, the 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 the 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 a result, 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 the 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 a result, the snapshot of VM_1 *does not* record the receipt of ACK_p. Upon snapshot

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.

restoration, VM_1 will, according to the 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 the TCP semantics.

In the second part of the proof, 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 [17]. The TCP state transitions are triggered 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. 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 will not 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 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. It is also very straightforward to show that VIOLIN's UDP tunneling design preserves fair-lossiness for UDP-based transport in applications. Our proof builds a "bridge" between the classic algorithm and practice – with particular relevance to the emerging virtual infrastructures in the cloud.

3.2.3 Mitigating Performance Impacts

Although the snapshot algorithm preserves the correctness of the transport and application-level semantics in a VIOLIN, it impacts VIOLIN's network transport performance.

For transport via TCP, the direct consequence of executing 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. TCP backoff can happen at either a pre-snapshot or post-snapshot VM. For a post-snapshot VM, TCP backoff is attributed to the dropping of packets transmitted to a pre-snapshot VM as these packets are encapsulated in category 3 frames. For a pre-snapshot VM, TCP backoff is also caused by the dropping of category 3 frames, but here the category 3 frames carry the ACKs from a post-snapshot VM acknowledging packets (in category 2 frames) from the pre-snapshot VM. The duration of the TCP backoff is therefore directly related to the *degree of discrepancy* among the VMs' snapshot completion times.

For transport via UDP, dropping category 3 frames means loss of the UDP packets carried by those frames. Although reliable packet delivery should not be expected based on UDP's "best-effort" semantics, excessive loss may exceed the tolerance level of some UDP-based applications, leading to undesirable consequence (e.g., abnormal exit). In addition, we note that category 2 frames can also cause UDP loss – during the *restoration* of a VIOLIN snapshot. Category 2 frames do not lead to any loss when the snapshot is taken. However, when the snapshot is restored in the future, the sender VM will "believe" that it had sent some UDP packets prior to the snapshot but the receiver VM will not "remember" receiving those packets (as they arrived after the receiver VM's snapshot operation). Although not semantically wrong, such loss will have negative impact on an application's performance.

To mitigate the above impacts on VIOLIN transport performance, we develop a technique called *frame buffering and injection* (FBI) to enhance VNsnap. The key idea is that category 2 and 3 frames can actually be buffered by their receiving VIOLIN switches during a snapshot and later re-injected into the relevant VMs to reduce packet loss. More specifically, a receiving VIOLIN switch will keep a copy of each category 2 or 3 frame received. Once a VM transitions to the post-snapshot state, the switch will inject the buffered category 3 frames *before* delivering any new frames. The buffered category 2 frames are saved as part of the VIOLIN snapshot (as snapshot of in-transit traffic) and when the snapshot is restored in the future, they will be delivered to destination VMs. Our experimental results indicate that FBI is effective in reducing UDP packet loss in a VIOLIN and, *depending on the timing of frame injection*, in alleviating TCP backoff. Detailed description and analysis of our evaluation results will be presented in Section 4.2.

3.2.4 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.

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 [18] partition 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 [19]. 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 compare Xen's and VNsnap's live checkpointing functionality. Second, we evaluate VNsnap's frame buffering and injection technique. Finally, we evaluate the impact of VNsnap on VIOLINs running real-world parallel/distributed applications – NEMO3D [20] and BitTorrent [21]. 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 the 2.6.18 Linux kernel.

4.1 Downtime Minimization for Live VM Snapshots

We first evaluate the true 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. For comparison, we experiment with (1) Xen's live VM checkpointing function (used in [2]) and (2) the VNsnap daemon implementation. For both implementations, we measure the metrics for the same VM running with 650MB of RAM. The tests are run both when the VM is idle and when it is executing the parallel application NEMO3D. NEMO3D is a long-running (tens of minutes to hours), legacy parallel simulation program without any built-in checkpointing support and it is widely used by the nanotechnology community for nano-electric modeling of quantum dots.

Table 1 shows the averages of ten runs. Since VNsnap daemon is based on Xen's live migration function, it involves multiple iterations of memory page transfer during the snapshot operation (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 rate at which the workload is dirtying the VM's memory pages. For instance, we observe that, during the NEMO3D execution, memory pages can get dirtied at a rate about 125MB/s.

The most important metric in Table 1 is the VM downtime. We have two main observations. First, the VNsnap daemon incurs significantly shorter downtime (ranging from 100 ms to 500 ms) than Xen's checkpointing function (around 9 seconds). Second, for Xen live checkpointing, the downtime remains almost the same for both the "idle" and "NEMO3D" runs. VNsnap daemon implementation, on the other hand, exhibit shorter downtime for the "idle" runs than the "NEMO3D" runs. The explanation for both observations lies in the fact that for VNsnap daemon the VM is down only during the stop-and-copy phase where only pages that belong to the WWS of a VM are transferred (as explained in Section 3.1.2). The duration of downtime is determined by the WWS of the VM or the number of dirty pages transferred in the *last* iteration – about 110 pages in the "idle" run and 11,000 pages in the "NEMO3D" run – out of the total 166,400 pages of the VM. This differs from Xen's VM checkpointing, where there is only one iteration during which the VM freezes and all 166,400 pages are written to disk.

Another important metric from Table 1 is the total snapshot duration. For both Xen checkpointing and VNsnap daemon, the duration represents the amount of time it takes for the snapshot image to be fully committed to disk. We observe that for the "NEMO3D" run, the VNsnap daemon incurs longer duration than Xen checkpointing because of its multi-iteration memory page transfer. It takes 15 seconds to transfer all pages to the snapshot daemon with an additional 10 seconds for the daemon to write the pages to disk. Given that writing pages to disk does not interfere with VM execution and is completely independent of the page transfer operation, we do not include it in future results.

	State	Duration(s)	Iterations	Downtime(ms)	Pages in Last Iteration	Size
Xen Live Checkpointing	Idle	10	1	9154	166400	650MB
	NEMO3D	10	1	9337	166400	650MB
VNsnap	Idle	4+10	4	119	105	650MB
	NEMO3D	15+10	25	468	11054	650MB

TABLE 1
Measurement results comparing Xen Live checkpointing with VNsnap.

	VNsnap			Optimized VNsnap				
Index	Duration(s)	Iter.s	Pg.s Transf.d	Duration(s)	Iter.s	Pg.s Transf.d	Saving (%)	Pg.s Skipped
NEMO3D$_1$	8	8	177052	7	15	49733	72%	139267
NEMO3D$_2$	21	30	287568	14	30	160442	44%	153117
NEMO3D$_7$	7	6	166685	7	4	31867	81%	134673
NEMO3D$_9$	8	8	183367	8	19	77243	58%	156125
NEMO3D$_{11}$	9	21	217029	9	26	108102	50%	156409

TABLE 2
Page transfer efficiency of the optimized VNsnap daemon.

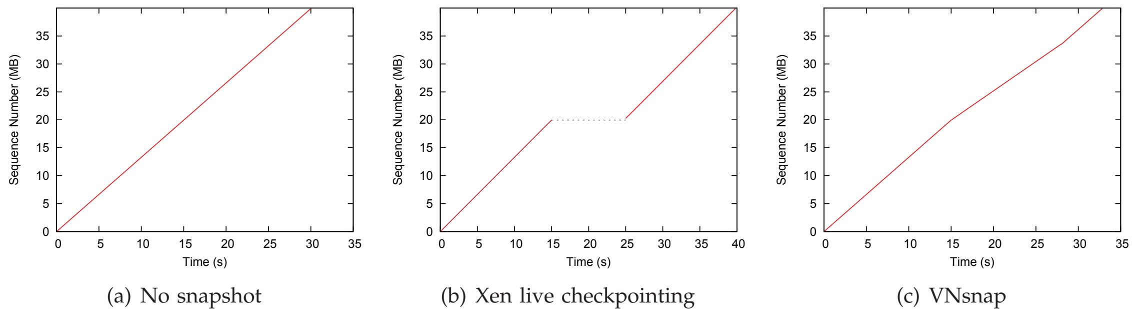


Fig. 4. The impact of different VM snapshot techniques on TCP throughput in a VIOLIN running NEMO3D. Traces are obtained from `tcpdump`.

Similarity-aware VM snapshot optimization. We also evaluate our similarity-aware optimization for VM snapshot (Section 3.1.2). The results are shown in Table 2. We compare the number of memory pages transferred during VIOLIN snapshot – with and without the optimization. In each case, we start with taking snapshot of an idle VIOLIN and focus on *one* of the VMs in a VIOLIN with a total of 166,400 pages (650MB). For the "with optimization" case, the VNsnap daemon uses this idle-time snapshot as the "base" snapshot image for the subsequent snapshot. After that we start running the NEMO3D application and take periodic snapshots of the VIOLIN every 10 minutes. Table 2 shows the results from *selected* (thus non-consecutive) snapshot operation instances (indicated by the index in the first column). The instances are selected as they represent varying degree of memory write intensity during the NEMO3D execution, ranging from the least intensive (NEMO3D$_7$) to the most intensive (NEMO3D$_2$) – indicated by the snapshot duration and the number of page transfer iterations.

There are two main observations from the results in Table 2. First, the optimized VNsnap system significantly reduces the number of memory pages transferred during each snapshot operation. The "Saving" column is the percentage of fewer pages transferred in comparison with the "without optimization" case. The degree of saving varies (from 44% to 81%) depending on the degree of memory write intensity during the snapshot operation: The higher the memory write intensity, the lower the page transfer saving. More interesting is the second observation: *Regardless* of the memory write intensity during those snapshot operations, the number of pages skipped from transfer (i.e., pages that are determined as unmodified since the last snapshot) does *not* vary as much – ranging from 134,673 to 156,409 pages. Relative to the total number of pages of the VM (166,400 pages), the percentage of pages skipped is consistently high (from 81% to 94%). This observation can be explained by the locality property exhibited by NEMO3D's execution: During most part of the execution, only a small number of memory pages are being modified at any time, though the memory write intensity varies over time. Even if we look at a time window that spans two snapshot operations, the subset of modified pages is still small relative to the total number of pages of the VM. Our profiling study shows that many long-running applications exhibit similar locality property.

Finally, our experiments also indicate that the overhead of the optimization is negligible. More precisely, our measurement results show that it takes no more than 2.0 seconds to generate and compare page hashes for each snapshot operation. We point out that, during hash calculation and comparison, the VIOLIN *is* running normally and the overhead is justified by the reduction of the page transfer overhead.

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 which may induce 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)), and VNsnap daemon (Figure 4(c)). We focus on one TCP connection between the two VMs. The flat, "no progress" period shown in Figure 4(b) 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 Xen live checkpointing (Figure 4(b)) incurs 2-3 seconds of TCP backoff, whereas the VNsnap daemon (Figure 4(c)) does not incur noticeable TCP backoff. More results and analysis will be presented in the next subsections.

4.2 Effectiveness of Frame Buffering and Injection

In this section we evaluate the effectiveness of the frame buffering and injection (FBI) technique (Section 3.2.3) in reducing UDP packet loss and shortening TCP backoff period and hence mitigating the impact of VNsnap algorithm on transport performance. To study the effects of frame buffering and injection under a controlled setting, we introduce a 5-second artificial delay before one of the VIOLIN switches issues a snapshot (by delaying the propagation of TAKE_SNAPSHOT message for five seconds). We note that there are many factors that can potentially influence the effectiveness of FBI, such as application semantics and the number of VMs and VIOLIN switches. Nonetheless, we focus here on a basic, "noise-free" scenario for a 2-node VIOLIN (consisting of two VMs where each VM resides on a different physical host; similar to the 2-node NEMO3D experiment of Section 4.3) to gain insights into the effectiveness of FBI in different settings.

4.2.1 *Effectiveness of FBI for UDP*

For applications using best-effort protocols such as UDP, the main purpose of FBI is to alleviate packet loss. We first show how FBI helps packet delivery for the ICMP protocol. Figure 5 shows the RTT measurements for ICMP (*ping*) packets with the default 1-second transmission interval. In this figure, despite the fact that receiver VM (xen2) completes its snapshot operation about 5 seconds later than sender VM (xen1), all packets are received by xen2. In particular, five ICMP packets (ICMP sequence numbers 4-8) are buffered and injected into xen2, as indicated by their longer RTTs (mostly time in the VIOLIN switch buffer). Had FBI not been used, the five packets would have been lost.

Our next experiment studies the effectiveness of FBI in reducing packet loss from the perspective of a UDP-based application. This experiment also involves two VMs in a VIOLIN snapshot operation. There are two

```
xen1: # ping xen2
PING xen2 (10.0.13.51) 56(84) bytes of data.
64 bytes from xen2 (10.0.13.51): icmp_seq=1 ttl=64 time=0.490 ms
64 bytes from xen2 (10.0.13.51): icmp_seq=2 ttl=64 time=0.449 ms
64 bytes from xen2 (10.0.13.51): icmp_seq=3 ttl=64 time=0.469 ms
64 bytes from xen2 (10.0.13.51): icmp_seq=4 ttl=64 time=4584 ms
64 bytes from xen2 (10.0.13.51): icmp_seq=5 ttl=64 time=3584 ms
64 bytes from xen2 (10.0.13.51): icmp_seq=6 ttl=64 time=2584 ms
64 bytes from xen2 (10.0.13.51): icmp_seq=7 ttl=64 time=1584 ms
64 bytes from xen2 (10.0.13.51): icmp_seq=8 ttl=64 time=584 ms
64 bytes from xen2 (10.0.13.51): icmp_seq=9 ttl=64 time=0.478 ms
64 bytes from xen2 (10.0.13.51): icmp_seq=10 ttl=64 time=0.319 ms
64 bytes from xen2 (10.0.13.51): icmp_seq=11 ttl=64 time=0.320 ms
```

Fig. 5. Impact of FBI on ICMP with 5-second snapshot completion time discrepancy.

scenarios in the experiment: (1) the sender completes the snapshot 5 seconds earlier than the receiver and FBI buffers and injects category 3 frames during the current execution; (2) the sender completes the snapshot 5 second later than the receiver and FBI re-injects category 2 frames during the restoration of the VIOLIN snapshot in the future. For each scenario, we perform the measurement under various UDP packet transmission intervals: 1 ms, 10 ms, and 100 ms. For comparison, we also repeat the experiment without FBI. Table 3 shows the average results for ten runs, which indicate that FBI significantly reduces UDP packet loss (by close to 100%) in both scenarios (1) (Table 3(a)) and (2) (Table 3(b)).

(a) Scenario 1: during current execution (FBI for category 3 frames)

Transmission Interval	Buffered Pkts.	Pkts. Lost w/o FBI	Pkts. Lost with FBI	Reduction in Loss
1 ms	465	473	8	98%
10 ms	241	246	5	98%
100 ms	42	43	1	98%

(b) Scenario 2: during snapshot restoration (FBI for category 2 frames)

Transmission Interval	Buffered Pkts.	Pkts. Lost w/o FBI	Pkts. Lost with FBI	Reduction in Loss
1 ms	473	486	13	97%
10 ms	227	236	9	96%
100 ms	44	45	1	98%

TABLE 3
Effectiveness of FBI in reducing UDP packet loss during snapshot and snapshot restoration.

There are three points to note here. First, for FBI to be effective, the VIOLIN switch buffer size for each VM and the application socket buffer size should be large enough to accommodate the buffered category 2 and 3 frames. Second, while FBI greatly reduces packet loss, it does not *always* completely prevent loss as shown in Table 3. The source of these packet losses are VM downtime and the detachment of I/O devices and bridge change that take place during snapshot. Third, FBI may not be useful for applications that are highly sensitive to timeliness of data arrival (e.g., real-time video conferencing). However, for application that do not have stringent timing requirement, use of FBI will significantly mitigate UDP packet loss both during the current VIOLIN operation and during the restoration of VIOLIN snapshot in the future.

4.2.2 Effectiveness of FBI for TCP

To study the impact of FBI on applications using TCP transport, we study TCP backoff in a VIOLIN during and after the snapshot. Before proceeding with the experiment setup, we point out the two main challenges associated with FBI for TCP. First, the delivery of buffered TCP packets requires stringent timing. To be of use, these packets need to be injected within a narrow window when the receiver VM has resumed normal execution after the snapshot (or snapshot restoration) but before the sender VM retransmits 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 the buffered packets. Otherwise, these packets still have to be retransmitted and FBI yields no benefits. 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 the TCP window.

In this experiment, the sender VM sends TCP packets to the receiver VM every millisecond. The sender VM completes its snapshot about 5 seconds earlier than the receiver VM. Figure 6 compares the network traces recorded *with* and *without* FBI at the sender. For viewing convenience, we move up the "with FBI" curve by 500000 units along the y-axis. Figure 6 shows that FBI shortens TCP backoff period – the flat segment of the curves in the figure – from 7.33 seconds (without FBI) to 4.44 seconds (with FBI) which represents a 40% reduction.

A closer examination of the results confirms that FBI's effectiveness varies based on the timing of the injection within a *"window of opportunity"*. The window of opportunity, as shown in Figure 6, refers to the period during which the sender VM has not successfully retransmitted the buffered packets following the completion of the receiver's snapshot operation. More specifically, after each failed retransmission, the sender doubles the timeout interval before attempting a new retransmission. Therefore, packet injection by FBI can effectively advance the congestion window of the connection sooner as the sender VM does not have to wait for the timeout to resend the unacknowledged packets during the exponential backoff period. The earlier the injection, the higher the benefits of FBI are going to be. In the "with FBI" case of Figure 6, the length of the window of opportunity is 4.24 seconds (starting when the receiver VM completes its snapshot at 13.67 seconds) and packets are injected 0.59 second after the start of the window. Our analysis above also indicates that a greater snapshot completion time discrepancy widens the window of opportunity. As such, FBI is particularly useful in scenarios where the discrepancy is large between the two end points of a TCP connection (more than 4 seconds). When discrepancy is small, the window of opportunity is also small and FBI does not yield much benefit. Fortunately, the TCP backoff period will also be short in that case, which is a favorable situation.

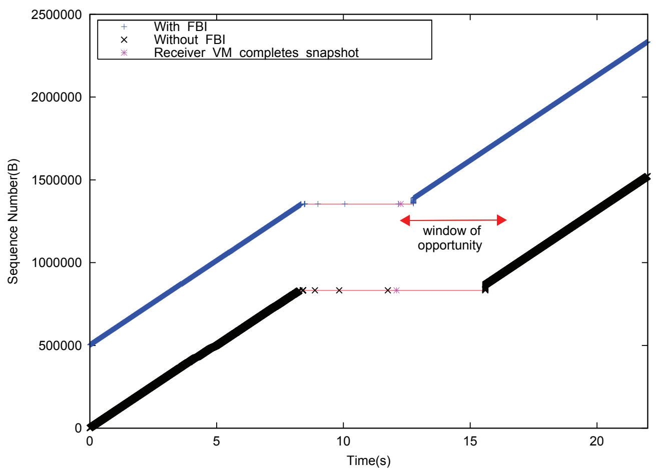
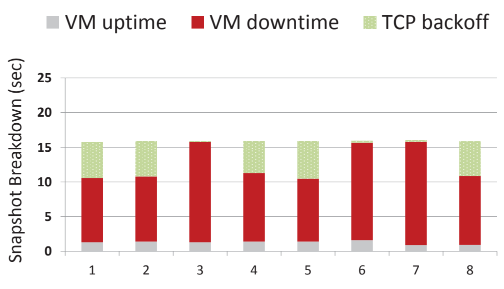


Fig. 6. Effectiveness of FBI in shortening TCP backoff.

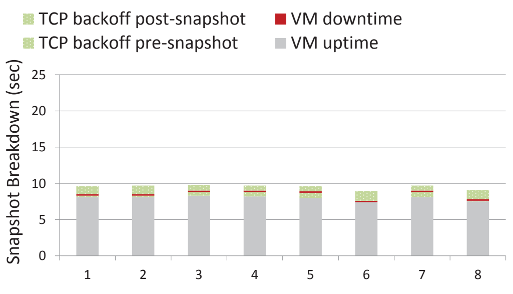
4.3 Taking Snapshot of a VIOLIN Running NEMO3D

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 Xen live checkpointing and the VNsnap daemon 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 7 shows the results. The times shown are averages of all VMs in a given VIOLIN from a given experiment. We observe that VNsnap incurs very low disruption (VM downtime + TCP backoff) – more specifically 0.05, 0.8, 1.4, and 3.8 seconds for the 2, 4, 8, and 16-node VIOLINs, respectively. On the other hand, Xen checkpointing 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 for VNsnap), but the downtime also scales with the number of VMs that are simultaneously being snapshotted on the same host (about 20 seconds with two VMs per host vs.



(a) Xen Live Checkpointing



(b) VNsnap

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

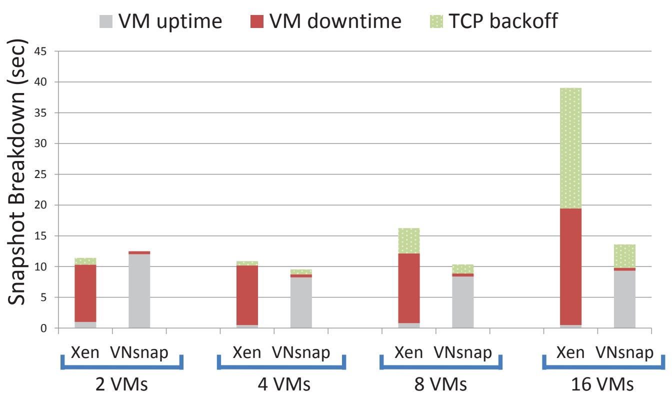


Fig. 7. The breakdown of snapshot timing for 2, 4, 8 and 16-node VIOLINs running NEMO3D.

about 10 seconds with one VM per host as in the 2, 4, and 8-node cases).

Figure 8 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 8, the discrepancy among the 8 VMs is very insignificant for VNsnap (less than 1 second – Figure 8(b)). 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 Xen live checkpointing. On the other hand, since for the VNsnap daemon disk writes are decoupled from the snapshot operation as far as the VMs are concerned, the VMs experience less discrepancy in snapshot completion time and much less TCP backoff.

In all experiments, we validated 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 generated the same program output.

4.4 Taking Snapshot of a VIOLIN Running BitTorrent

In this section we study the impact of VNsnap on a VIOLIN running the peer-to-peer BitTorrent application [21]. 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 9 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 server to run a remote VNsnap daemon. Of the remaining two servers, 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 350MB of memory) runs an Apache webserver and a BitTorrent tracker. In the CERIAS subnet, we use four servers each hosting a VM with 1GB of memory that runs as a BitTorrent client or seed. The remaining four servers host a VNsnap snapshot daemon. The 6 VMs – two in CS and four in CERIAS – constitute 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.

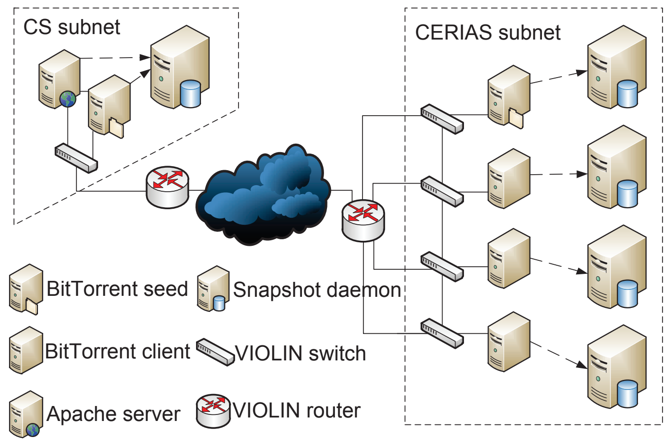
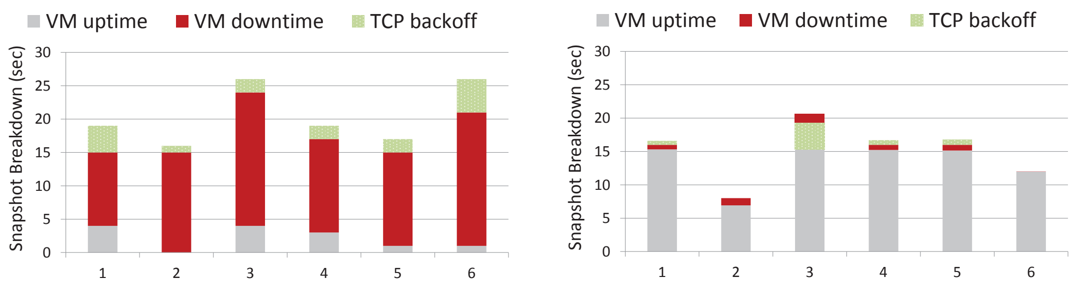


Fig. 9. The setup of the BitTorrent experiment.



(a) Xen Live Checkpointing (b) VNsnap

Fig. 10. Per-VM breakdowns of snapshot timing for the VIOLIN running BitTorrent.

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 10 compares the per-VM snapshot timing breakdown under Xen's live checkpointing and under VNsnap. 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 (all below 2 seconds except VM 3 – Figure 10(b)). The disruption periods under Xen's 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 10(a)).

When looking at the results for VNsnap (Figure 10(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 on the same server that runs a VNsnap snapshot 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 as a result it cannot receive ACKs from other post-snapshot VMs. Finally, we validate the correctness of VNsnap by comparing the checksum of the original file with the checksums of the files downloaded during the runs when the snapshot is taken and when it is restored.

5 Discussion

In this section, we discuss two main issues surrounding the overhead and applicability of VNsnap in a cloud setting. The first issue concerns the lack of synchrony in snapshot completion times of the individual VMs that make up a VNI. While our proposed optimized daemon and frame buffering and injection methods to some extent alleviate this problem by reducing the duration of snapshot (Section 4.1) and by reducing UDP packet loss and TCP backoff (Section 4.2), it does not make the snapshot operation completely transparent to applications running in the VNI. Heterogeneity in the memory size of VMs particularly exacerbates this problem. One simple solution to reduce the snapshot skew overhead is to modify the live VM migration implementation such that the migration/snapshot takes a *uniform* or *bounded* amount of time transferring VM memory pages to snapshot 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), the snapshot completion times for the VMs will be of low discrepancy. However, since VNsnap cannot completely eliminate the discrepancy without making any modifications to VMs, VNsnap requires applications to tolerate the short period of disruption incurred by the snapshot algorithm. We believe that many – though not all – cloud applications meet this requirement.

The second issue concerns the restorability of a VIOLIN snapshot. First, for a 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 for execution. This requirement exists because a snapshot may be restored at an arbitrary time in the future. As a result, the execution inside the VIOLIN should not depend on a connection to the outside that may time out by the time a snapshot is restored. This problem can pose complications for clients that connect to a cloud service running in a VIOLIN whose state is being captured by VNsnap. Two solutions to address this problem are: (1) extending the VIOLIN snapshot to client hosts or (2) requiring the execution state to completely reside in the cloud and have the client interact with cloud services via a *stateless* connection (e.g., a VNC session), which can be re-established upon snapshot restoration. Second, device virtualization which decouples the virtual devices in a VM from the physical devices in a host allows VM migration and snapshot restoration across different sets of hosts. For example, we have seen snapshots generated by VNsnap being restorable on two sets of hosts with different 64-bit Intel processors and NICs. Moreover, with VM migration (equivalent to VM snapshot as far as VNsnap is concerned) being supported across different processors (e.g., KVM supports VM migration between Intel and AMD hosts), snapshot restoration will become further insensitive to host-level differences.

6 Related Work

Many techniques have been proposed to checkpoint distributed applications, but few have addressed the need for checkpointing an entire networked infrastructure. Checkpointing distributed applications can be loosely categorized into application-level, library-level (e.g., [22], [11]), and OS-level (e.g., [23]) checkpointing. Although these techniques are beneficial in their own rights and work best in specific scenarios, they come with their own limitations. Application-level checkpointing requires access to application source code and is highly semantics-dependent. 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 it 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 [14] is a thin virtualization layer that provides checkpoint/restart functionality for a self-contained virtual machine abstraction, namely a *pod* (*Pr*O*cess* *D*omain), 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 [15] 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, many solutions have been proposed based on Xen migration to address fault-tolerance in virtualized environments [24], [25], [19], [26]. [24] 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 create checkpoints.

Remus [25] is a practical VM-transparent service that protects unmodified software against physical host failures. The focus of Remus is high availability of individual VMs whereas VNsnap focuses on the reliability of distributed VNIs. Remus leverages an enhanced version of Xen migration to efficiently transfer a VM state to a backup site at high frequencies (i.e., 40 times per second for Remus vs. every few minutes for VNsnap). It also implements a network buffering method similar to FBI. The main difference between the two methods is that in Remus buffering is done at the end host running the sender VM so that network activity corresponding to speculative execution would not reach the destination while the synchronization to backup is in progress. However, in VNsnap buffering takes place at the destination VIOLIN switch (as a sender VM or its corresponding physical host are unaware of the current epoch of a receiver VM) to mitigate the side effects of the snapshot algorithm. [27] is an effort similar to Remus with the goal of improving state synchronization between a VM and its backup site. It employs a hashing method similar to VNsnap that also operates at sub-page granularity.

The closest work to VNsnap from the application point of view (i.e., checkpointing distributed execution) is an advanced system [19] that enables frequent, transparent checkpointing of closed distributed systems in Emulab [28]. Being parallel efforts, VNsnap and [19] share similar goals with different system requirements: [19] requires high-accuracy clock synchronization to avoid the distributed snapshot algorithm and to achieve high fidelity and transparency for network experiments. Therefore, [19] requires modifications to the guest kernel of VMs. On the other hand, VNsnap is geared towards IaaS clouds where IaaS providers have no or minimal control over the custom VMs used by cloud users. Therefore, while clock synchronization helps VNsnap to synchronize individual VM snapshots (Section 5), an IaaS provider can safely use VNsnap without making any assumption about the hosted VMs – in particular, VMs can be unsynchronized, paravirtualized, fully virtualized, and distributed across multiple data centers.

While *pre-copy*ing memory pages to a backup site or a checkpoint file seems to be the dominant trend, a few techniques have been proposed to *post-copy* pages in order to reduce the VM downtime during migration [29], [30]. While post-copy approaches incur less downtime and result in fewer page transfers as only one copy of a page is transferred, they may not be suitable for checkpointing VMs based on the pace at which they lazily transfer the memory pages. One advantage of using pre-copying is that VNsnap can easily be generalized to other virtualization platforms with live migration support (e.g., VMware ESX, KVM) while checkpointing solutions based on post-copying rely on heavily modifying the hypervisor and possibly the guest kernel.

7 Conclusion

We have presented the VNsnap system to take consistent snapshots of an entire VNI, 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 adapt 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 VNI reliability for the emerging IaaS cloud computing paradigm.

Acknowledgments

We would like to thank the anonymous IEEE Transactions on Services Computing (TSC) reviewers, and the anonymous reviewers of a preliminary conference version of this paper [31] for their insightful comments and suggestions. We also thank Gerhard Klimeck, Hoon Ryu, and Rick Kennell for their earlier help with the NEMO3D experiment setup. This work was supported in part by the US National Science Foundation (NSF) under Grants 0546173, 0720665, 0721680, 0749140, and 0834529 and by the Purdue Research Foundation under Grant 204533. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the NSF.

References

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

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

[3] 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.

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

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

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

[7] C. A. Waldspurger, "Memory resource management in VMware ESX server," *USENIX OSDI*, 2002.

[8] D. Gupta, S. Lee, M. Vrable, S. Savage, A. C. Snoeren, G. Varghese, G. M. Voelker, and A. Vahdat, "Difference engine: harnessing memory redundancy in virtual machines," *USENIX OSDI*, 2008.

[9] http://www.azillionmonkeys.com/qed/hash.html.

[10] F. Mattern, "Efficient algorithms for distributed snapshots and global virtual time approximation," *Journal of Parallel and Distributed Computing*, 1993.

[11] S. Sankaran, J. M. Squyres, B. Barrett, and A. Lumsdaine, "The LAM/MPI checkpoint/restart framework: system-initiated checkpointing," *LACSI Symposium*, 2003.

[12] G. E. Fagg and J. J. Dongarra, "Lecture notes in computer science 1 FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world," 2000.

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

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

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

[16] J. F. Ruscio, M. A. Heffner, and S. Varadarajan, "DejaVu: Transparent user-level checkpointing, migration, and recovery for distributed systems," *IPDPS*, 2007.

[17] R. W. Stevens, *TCP/IP Illustrated Volume 1*. Reading, MA: Addison-Wesley, 1996.

[18] http://sources.redhat.com/lvm2.

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

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

[21] http://www.bittorrent.com.

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

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

[24] 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.

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

[26] Y. Tamura, "Kemari: Virtual machine synchronization for fault tolerance using DomT," *Xen Summit*, 2008.

[27] M. Lu and T. cker Chiueh, "Fast memory state synchronization for virtualization-based fault tolerance," *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-DCCS)*, 2009.

[28] B. White, J. Lepreau, L. Stoller, R. Ricci, S. Guruprasad, M. Newbold, M. Hibler, C. Barb, and A. Joglekar, "An integrated experimental environment for distributed systems and networks," *OSDI*, 2002.

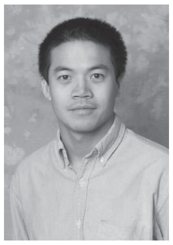
[29] M. R. Hines and K. Gopalan, "Post-copy based live virtual machine migration using adaptive pre-paging and dynamic self-ballooning," *ACM VEE*, 2009.

[30] H. A. Lagar-Cavilla, J. A. Whitney, A. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan, "SnowFlock: Rapid virtual machine cloning for cloud computing," *Eurosys*, 2009.

[31] A. Kangarlou, P. Eugster, and D. Xu, "VNsnap: Taking snapshots of virtual networked environments with minimal downtime," *IEEE/IFIP International Conference on Dependable Systems and Networks (DSN-DCCS)*, 2009.

Ardalan Kangarlou is a Ph.D. candidate in the Department of Computer Science at Purdue University. He received a B.S. degree from the University of Southern Mississippi in 2004 and an M.S. degree in Computer Science from Purdue University in 2006. His current research areas include virtualization, cloud computing, and distributed systems. He is a member of the ACM, IEEE, and USENIX.

Patrick Eugster is an assistant professor of computer science at Purdue University. His current research interests include programming abstractions, distributed algorithms, and middleware. Patrick holds M.S. and Ph.D. degrees from EPFL. He is recipient of a CAREER award from US NSF (2007), and a fellowship for experienced researchers from the Alexander von Humboldt foundation (2011). He is a participant of DARPA's 2011 computer science study group.



Dongyan Xu is an associate professor of computer science and electrical and computer engineering (by courtesy) at Purdue University. He received a B.S. degree from Zhongshan (Sun Yat-Sen) University in 1994 and a Ph.D. in Computer Science from the University of Illinois at Urbana-Champaign in 2001. His current research areas include virtualization technologies, computer malware defense, and cloud computing. He is a recipient of a US National Science Foundation CAREER Award.