Taking Snapshots of Virtual Networked Environments

Ardalan Kangarlou
Department of Computer Science
Purdue University
ardalan@cs.purdue.edu

Paul Ruth
Department of Computer and Information Science
University of Mississippi
ruth@cs.olemiss.edu

Dongyan Xu
Department of Computer Science
Purdue University
dxu@cs.purdue.edu

Patrick Eugster
Department of Computer Science
Purdue University
p@cs.purdue.edu

ABSTRACT

The capture of global, consistent snapshots of a distributed computing session or system is essential to the system's reliability, manageability, and accountability. Despite the large body of work at the application, library, and operating system levels, we identify a void in the spectrum of distributed snapshot techniques: taking snapshots of the entire distributed runtime environment. Such capability has unique applicability in a number of application scenarios. In this paper, we realize such capability in the context of virtual networked environments. More specifically, by adapting and implementing a distributed snapshot algorithm, we enable the capture of causally consistent snapshots of virtual machines in a virtual networked environment. The snapshot-taking operations do not require any modification to the applications or operating systems running inside the virtual environment. Preliminary evaluation results indicate that our technique incurs acceptable overhead and small disruption to the normal operation of the virtual environment.

Categories and Subject Descriptors

C.4 [**Computer-Communication Networks**]: Miscellaneous

General Terms

Reliability

Keywords

Global snapshots, virtualization, virtual network, checkpoint

VTDC'07, November 12, 2007, Reno, NV, USA

1. INTRODUCTION

Distributed snapshots have been widely used as a means to achieve reliability and fault/failure recovery in distributed computing systems. During a distributed computing session, a snapshot can be taken to capture and preserve the session's instantaneous execution state to insure against failure at a later time. Upon failure, a recent snapshot can be used to restore the execution state instead of restarting from the beginning of the distributed computing session, thus saving computation time and resources while achieving failure recovery. Recently, reliability has become an increasingly desirable property in the emerging shared cyberinfrastructures [7]. In such an infrastructure, the dynamic resource availability and possible failure of individual components (e.g., a physical machine or cluster) call for the development of a spectrum of reliability and fault recovery techniques for the distributed computation activities on the infrastructure.

Meanwhile, significant research efforts have been made in the development of distributed snapshot techniques at the application, library (e.g., MPI library), and operating system levels. However, we notice a void in the spectrum of distributed snapshot techniques (more precisely, at one end of the spectrum): *taking snapshots of an entire distributed runtime environment*. For each of the networked machines in such an environment, a distributed snapshot captures its entire software execution state – including all applications and the operating system. Although distributed snapshots at this (low) level are coarse grained and heavy weight, they have their own applicability as discussed later in this section.

In this paper, we present a distributed snapshot capability for *virtual* distributed environments, based on our earlier virtual networking system called VIOLIN [8, 15]. More specifically, our technique realizes the capability of *taking a distributed snapshot of an entire VIOLIN virtual networked environment (or VIOLIN in short)* consisting of multiple virtual machines (VMs) connected by a virtual network (VN). We hasten to point out that the proposed capability is *not* a replacement of the existing higher-level distributed snapshot techniques. Instead, it completes the spectrum of solutions and provides a useful capability for the operation and management of VIOLIN virtual networked environments.

Residing below the VIOLIN virtual environment, the VIOLIN distributed snapshot capability does not require any modification to the applications or operating systems running inside the VIOLIN; nor does it require the installation of any additional library/middleware inside the VIOLIN. As such, it helps improve reliability for legacy applications, runtime libraries, and operating systems. We develop the VIOLIN distributed snapshot capability by adapting a classic distributed snapshot algorithm [12] and implementing the adapted algorithm on top of the Xen platform [3]. Our implementation addresses a number of technical challenges to guarantee that a distributed snapshot captures a *causally consistent* execution state of the VIOLIN virtual environment.

To the best of our knowledge, distributed snapshot capability for virtual networked environments is not yet widely available. Such capability benefits a number of application scenarios. For example, it can be used in network emulation experiments to take a snapshot of the entire emulated environment for future *replay*. In such a scenario, it is desirable to preserve every detail of the entire emulated environment, including all running processes and the operating system states of each VM. This is particularly useful in Internet malware emulation [9] where everything inside the emulated environment is subject to attack and contamination. Existing snapshot/checkpointing techniques at the application, library, or operating system level do not apply to this scenario. Another scenario is the execution of legacy parallel/distributed (binary) code without a built-in snapshot/checkpointing capability. In this scenario, it may be inconvenient, and potentially impossible, to apply existing library or operating system level snapshot/checkpointing techniques without modifying and re-compiling the source code of the application. For the above scenarios, the proposed VIOLIN distributed snapshot capability will exhibit unique convenience and effectiveness.

The rest of the paper is organized as follows: Section 2 discusses related works; Section 3 gives an overview of the proposed capability as well as a technical summary of the VIOLIN virtual networking technique; Section 4 presents the adapted distributed snapshot algorithm; Section 5 describes the implementation details of VIOLIN distributed snapshot on Xen; Section 6 presents our preliminary evaluation results using a real-world legacy parallel scientific application. Finally, Section 7 concludes this paper.

2. RELATED WORK

From the algorithm perspective, Chandy and Lamport [4] proposed the first global snapshot algorithm for systems with FIFO communication channels. In their algorithm, the snapshot initiator sends a control message to all neighboring processes after it records its local state. Upon receiving the first control message, a process that has not yet recorded its state saves its state and sends control messages to all neighboring processes. Eventually, control messages would reach all processes in a distributed system and they all would record their states. The FIFO property of communication channels ensures causal consistency as no message sent after a local snapshot can arrive before the control messages and hence no post-snapshot message can affect a recorded snapshot state.

For non-FIFO channels, it is shown that a snapshot algorithm has to be either inhibitory or has to piggy-back some control information on messages. In inhibitory algorithms, after a local snapshot, sending messages over a particular channel is suspended until a control message is received along that channel. In the piggybacking method, actions of the underlying application are never delayed. However, this comes at the expense of adding control information to the basic messages. Lai and Yang [11] and Mattern [12] have proposed algorithms for this family of snapshot algorithms.

From the system perspective, a variety of techniques have been proposed to checkpoint a distributed execution. These techniques can loosely be classified as application-level checkpointing, library-level checkpointing, and system-level checkpointing. In application-level checkpointing, application programmers implement checkpointing functionality as part of the application code. Therefore, limitations of this approach include the need for application source code as well as programmers' familiarity with distributed program execution and checkpointing.

In library-level checkpointing such as in LAM-MPI [16], FT-MPI [6] and CLIP [5], an application code is linked to the checkpointing library. In this approach, the checkpointing library is often part of the message passing library (e.g., MPI) for communications between distributed entities. One benefit of this approach is the lower burden on the application developer compared with the application-level approach. However, since it is coupled with a specific message passing library, it cannot be applied to parallel/distributed applications (e.g, an Internet worm emulation) that do not utilize the message passing library.

System level checkpointing techniques such as CRAK [18] and Zap [13] implement checkpointing functionality at the operating system level by either modifying the kernel or loading a special kernel module. Besides being able to save the execution state of a process, an application-transparent checkpointing technique needs to address challenges that arise during the restoration of a checkpoint. Two such challenges are maintaining open connections and handling the dependence of checkpoints on system level resources (e.g., specific processor identifiers or file descriptors) that may not be reusable upon checkpoint restoration. In light of these challenges, *virtualization-based* solutions such as ZapC [10], Xen on InfiniBand [17], and our proposed technique offer greater flexibility and portability. ZapC is an efficient, thin virtualization layer that decouples a distributed application from resource dependencies on the host and enables transparent and coordinated application checkpoint/restart on commodity clusters. Similar to our solution, Xen on InfiniBand uses consistent VM states as checkpoints for Partitioned Global Address Space (PGAS) programming models. Moreover, to support VM migration and checkpointing on InfiniBand networks, it adds global coordination functionality to the ARMCI one-sided communication library. However, it targets a specific programming model (PGAS) and a specific network platform (Infiniband) and thus cannot be readily applied to legacy applications running on general-purpose networked infrastructures (e.g., a commodity cluster).

Finally, we do not address questions such as when to take snapshots and where/how to store them. However, related solutions have been proposed in [14] which can potentially be integrated with the VIOLIN snapshot capability.

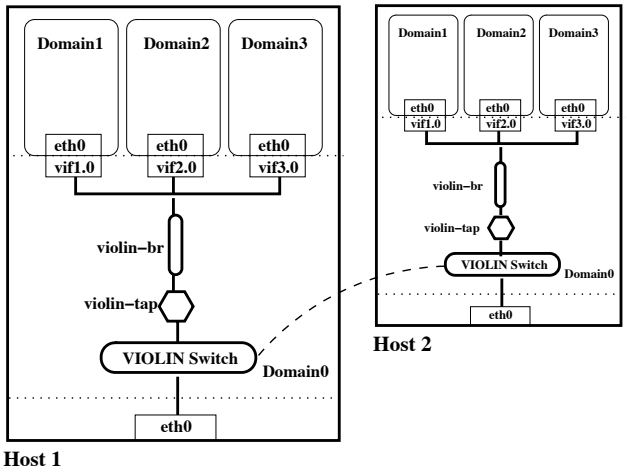

Figure 1: An overview of VIOLIN execution environment consisting of six virtual machines on two hosts

3. SYSTEM BACKGROUND

In this section we give a brief introduction to the VIOLIN virtual networked environment and its virtual networking implementation, which influences the VIOLIN distributed snapshot design. Further, we give an overview of the VIOLIN distributed snapshot capability.

A VIOLIN environment consists of multiple VMs connected by a virtual network. Created on top of a shared distributed infrastructure, multiple mutually-isolated VIOLINs can co-exist, each with its own IP address space, administrative privilege, and customized runtime and network services. Each VIOLIN is owned by a user or user group as a virtual private distributed computing environment with the same "look and feel" of a physical networked environment. VIOLIN supports the execution of unmodified parallel and distributed applications as well as the emulation of network and distributed systems. Virtual networking is the key enabling technique behind VIOLIN as shown in Figure 1. The VMs (i.e. the domains) in a VIOLIN are implemented on top of the Xen virtual machine monitor and are connected by VIOLIN switches running in domain 0 of the respective physical hosts. The VIOLIN switch intercepts traffic generated by VMs in the VIOLIN – in the form of layer-2 network frames – and tunnels them to their destinations using a transport protocol (e.g., UDP or TCP).

The distributed snapshot will be taken by the VIOLIN switches from outside of the VIOLIN environment. More specifically, a VIOLIN snapshot consists of the memory and disk images of each VM – taken at proper time instances – to capture a consistent execution state of the entire virtual environment. The main challenge is to ensure that the multiple distributed VIOLIN switches each record their local VM snapshots such that, when aggregated, they form a *causally consistent global snapshot*. To address this challenge, we adapt a distributed snapshot algorithm and implement it inside the VIOLIN switches (detailed in the next two sections). We point out that VIOLIN distributed snapshot does *not* require modification to or cooperation from the applications and operating systems inside the VIOLIN. The snapshot only incurs a small disruption to the VIOLIN's execution (Section 6), which will continue after the distributed snapshot is taken. The snapshot images will then be stored for future restoration.

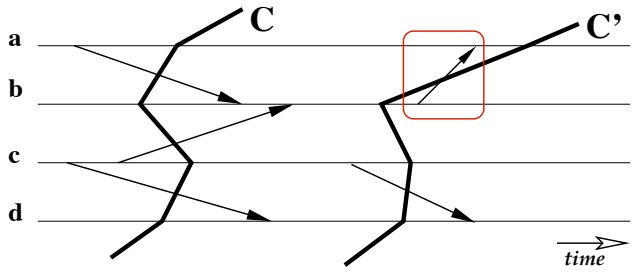

Figure 2: Consistent cut C and incosistent cut C'

4. SNAPSHOT ALGORITHM

The goal of a distributed snapshot algorithm is to record a causally consistent global state. If we could simultaneously snapshot all virtual machines and the virtual network, then we would have a causally consistent global state. However, given that we cannot always assume globally synchronized clocks, taking simultaneous snapshots is out of the question. For this reason we relax the state requirements and record a *possible* causally consistent state. A *possible* causally consistent state corresponds to a global state that may have been the states of the distributed entities (e.g., processes or VMs) and their communications, if they were captured simultaneously. Such a causally consistent state can be reached by means of finding a *consistent cut*. A cut is made of a sequence of events – one cut event at each distributed entity – that divide each entity's timeline into two parts. One part corresponds to events before the cut event (past) and one part corresponds to events after the cut event (future). If a cut is consistent, then there are no *messages passed from the future to the past*. Figure 2 illustrates a consistent cut C and an inconsistent cut C'. As a result, it is safe to assume all cut events in a consistent cut are taking place simultaneously and the local snapshots captured upon the cut events can represent a valid global snapshot of the system.

We adapt Mattern's distributed snapshot algorithm [12] for capturing VIOLIN snapshots. As discussed in Section 2, Mattern's algorithm is applicable to distributed systems with non-FIFO channels. As we will show, VIOLIN's snapshot algorithm is actually a simplification of Mattern's algorithm, due to VIOLIN's approach to virtual networking.

Similar to Mattern's algorithm, we use the concept of *message coloring* to distinguish between pre-snapshot and post-snapshot messages. By associating a color with each message, a distributed entity (in our case a VIOLIN switch) can be prevented from delivering a post-snapshot message before and while the snapshot is taken. As a result, global causal consistency is maintained by the VIOLIN switches across the VIOLIN. The adapted distributed snapshot algorithm works as follows:

1. The snapshot initiator – one of the VIOLIN switches – takes a local snapshot by saving the states of the VIOLIN's VMs residing in the same physical host. The initiator then sends a *TAKE_SNAPSHOT* message to all other VIOLIN switches to take their local snapshots. After the local snapshot, the initiator changes

its message color to the post-snapshot color and only accepts messages bearing that color.

2. Upon receiving the *TAKE_SNAPSHOT* message or a message bearing the post-snapshot color, a VIOLIN switch takes its local snapshot. It then notifies the initiator that it has taken its snapshot by sending a *SNAPSHOT_SUCCESS* message. While the snapshot is being taken, the VIOLIN switch will discard all messages bearing the post-snapshot color. After the local snapshot, it changes its message color to the post-snapshot color and only accepts messages bearing the same color.

3. The distributed snapshot procedure is complete when the initiator receives the *SNAPSHOT_SUCCESS* messages from all other VIOLIN switches.

The adapted VIOLIN distributed snapshot algorithm is a simplified version of Mattern's original algorithm. Due to the nature of communication channels in VIOLIN, we are *not* concerned with capturing in-transit messages at the time snapshot is taking place. In-transit messages are those messages of pre-snapshot color that arrive at their destinations after the local snapshots have been taken. In Mattern's algorithm a counter is used to keep track of the number of messages sent along each communication channel. When the count of messages sent is equal to the number of messages received, there will be no in-transit messages along that channel.

We can sidestep the capture of in-transit messages in VIOLIN snapshots because VIOLIN realizes network virtualization at layer-2 of the virtual network. As such, VIOLIN switches are not required to achieve reliable, in-order delivery of the layer-2 (virtual) network frames. If an application requires reliable packet delivery, a transport protocol (e.g., TCP) inside the virtual machines is expected to handle the re-sending of lost data. As a result, VIOLIN snapshot algorithm does not need to guarantee that in-transit messages reach their intended destinations upon the restoration of the snapshot. For a wide range of parallel/distributed applications, VIOLIN snapshots generated by the above algorithm will be able to restore their execution properly. For applications that employ a reliable transport protocol (e.g., TCP), the VIOLIN snapshot preserves the VM execution state to identify and re-transmit the lost packets upon restoration. For applications that use a best-effort transport protocol (e.g., UDP), packet loss is an *expected* behavior which will be handled by the application semantics. Hence, ignoring in-transit messages would not impede the execution of applications in either category.

Figure 3 illustrates the difference between the VIOLIN snapshot algorithm and Mattern's original algorithm. As shown in the figure, the two algorithms result in the same consistent cut for the participating VMs. However, the VIOLIN snapshot procedure can potentially end sooner than Mattern's algorithm because the VIOLIN algorithm does not wait for the last in-transit message to arrive at its destination. For the example in Figure 3, the VM on top of VS_c realizes upon restart that it has not received an acknowledgment for the packet it had sent to the VM on top of VS_a before the snapshot. As a result it will re-send the packet once it resumes. Note that in VIOLIN, all layer-2

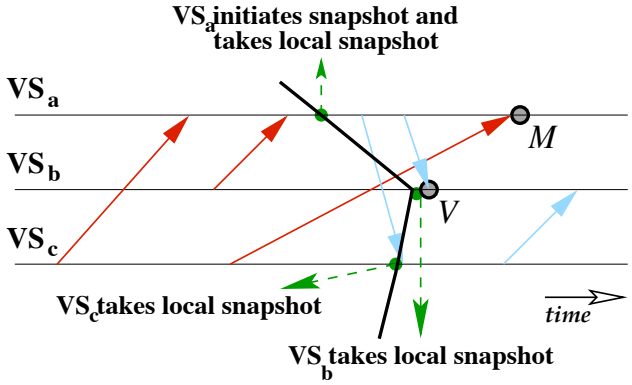

Figure 3: An illustration of VIOLIN snapshot procedure by three VIOLIN switches (VS_a, VS_b, and VS_c). Pre-snapshot messages are colored red and post-snapshot messages are colored light blue. Points *V* and *M* are the time instances when VIOLIN and Mattern snapshot algorithms complete the snapshot procedure, respectively.

frames, including those carrying TCP ACKs get colored. In Section 6, we will report the performance overhead incurred by packet retransmission upon the completion of a VIOLIN snapshot.

5. IMPLEMENTATION

In this section, we describe the details of implementing the VIOLIN distributed snapshot algorithm presented in the previous section. More specifically, we discuss the way VIOLIN snapshot achieves a consistent cut using the message coloring scheme. Our implementation is based on the Xen [3] VM platform, but it can easily be generalized to other VM platforms with similar live VM snapshot capability. A VM snapshot is live if the VM resumes execution upon the completion of a snapshot. A single VM snapshot consists of the states of its memory and disk at the time instance when the snapshot takes place. We leverage Xen's live snapshot mechanism to obtain a VM's memory image while we apply LVM snapshot mechanism [2] to capture the state of the disk.

In VIOLIN, UDP tunneling is used to carry the layer-2 traffic between VMs in the same VIOLIN. For the VMs (belonging to the same VIOLIN) in the same physical host, they communicate through a dedicated bridge as it is widely done in a typical Xen setup. Hereafter, we refer to these two types of communication as *inter-host* and *intra-host* communications, respectively. In vanilla VIOLIN, there is a VIOLIN switch that performs the tunneling inside domain 0 for each physical host participating in the VIOLIN. In the snapshot-enabled VIOLIN, each VIOLIN switch has a *color* and it assigns its color to the layer-2 frames originated from one of its associated VMs through packet encapsulation which incurs negligible overhead.

To handle inter-host communication, upon receiving a packet, a VIOLIN switch delivers the packet to the recipient VMs only if the color of the packet matches the color of the switch. If a switch receives a *TAKE_SNAPSHOT* message from the snapshot initiator or it receives the *first* packet with the post-snapshot color, the VIOLIN switch will

take the snapshot of all the associated VMs, after which the VIOLIN switch will change its color to post-snapshot and acknowledges the initiator. After the color change, the VIOLIN switch will only deliver packets of the post-snapshot color. In our implementation, we actually use three different colors to distinguish between two consecutive snapshots. More specifically, if the pre-snapshot and post-snapshot colors are colors 0 and 1 respectively in the first snapshot, they will be colors 1 and 2, 2 and 0, 0 and 1... in the subsequent snapshots.

To handle intra-host communication, VIOLIN uses a similar "bridge coloring" approach to preserve causal consistency. In bridge coloring, two bridges are used for connecting VMs residing on the same host: a pre-snapshot bridge and a post-snapshot bridge, as shown in Figure 4. Before a snapshot, VMs on the same host communicate through the same pre-snapshot bridge without any VIOLIN switch involvement. However, upon completion of a snapshot each VM switches to the post-snapshot bridge. What follows is that a packet on a post-snapshot bridge would never reach a VM that is still at the pre-snapshot stage. Hence causal consistency is maintained. Switching bridges is implemented by modifying the *xend* snapshot interface: Right after saving the VM memory state and right before resuming a VM, a bridge switch will be performed atomically. After the bridge switch, the pre-snapshot and post-snapshot bridges swap their roles: the pre-snapshot bridge becomes the post-snapshot bridge and the post-snapshot bridge becomes the pre-snapshot bridge (for the next snapshot in the future).

Based on the above switch/bridge coloring scheme, causal consistency can be preserved for both inter-host and intra-host communication, preventing the situation where a pre-snapshot VM receives a message from a post-snapshot VM in the VIOLIN virtual environment.

VIOLIN uses the Logical Volume Manager (LVM) snapshot feature [2] to capture the disk image of each VM. This requires having a VM store its file system on LVM partitions. LVM snapshot is a powerful feature that creates an exact replica of a volume without halting the execution of the system that is modifying the volume during snapshot. VIOLIN takes a volume snapshot at the same time a bridge switch operation is done. To minimize the disruption to VM execution, VM and LVM images can be processed after resuming the VM execution. Efficient handling and transfer of the images in the snapshot is part of our on-going work. The VIOLIN snapshot process is considered complete once all the VIOLIN switches have successfully captured the memory and disk images of their associated VMs.

We point out that there exist a number of limitations of the VIOLIN distributed snapshot capability: First, it is not suitable for distributed applications where one end of a (virtual) network connection cannot tolerate some downtime of the other end – while the VM is getting a snapshot. Second, to use the VIOLIN snapshot capability, an application must be self-contained within a VIOLIN. In other words, the execution of the application should not involve any communication with outside of the VIOLIN. Third, a VIOLIN switch has a very limited view of the computation activities inside the associated VMs. More precisely, it only knows the existence of the VMs and the layer-2 network frames that they generate. Therefore, VIOLIN snapshot by design cannot – and should not – provide explicit guarantee of event ordering between application processes in the VMs.

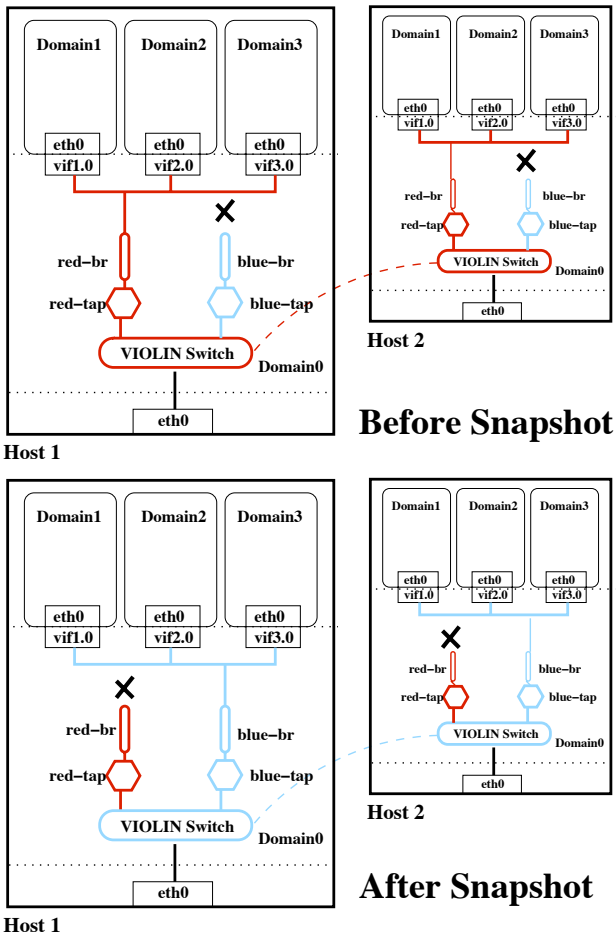

Figure 4: Packet coloring and bridge coloring in VIOLIN checkpointing. In this figure, red denotes the pre-snapshot color and blue the post-snapshot color.

In fact, a non-deterministic application, by nature, may traverse various paths to its termination even without taking any snapshots.

6. EVALUATION

In this section, we study the runtime overhead incurred by the VIOLIN snapshot procedure. The two major sources of overhead in our approach are (1) saving a VM memory image to the disk and (2) TCP backoff after the snapshot or upon restoring a snapshot (at a future time). Other modifications to the original VIOLIN such as coloring packets, switching bridges, and LVM snapshots incur negligible overhead. For experiments in this section, we use NEMO3D [1], a parallel MPI-based nanotechnology simulation program that performs nano-electric modeling of quantum dots. We choose this application for two reasons: First, given the same input parameters, the program generates a deterministic output; hence we can verify the *correctness* of execution results with and without snapshots. Second, the program is communication intensive thus we can study the impact of TCP backoff.

To study the overhead of VIOLIN snapshot under different settings, we run the NEMO3D application in VIOLINs of the same scale (each with the same input parameters) but

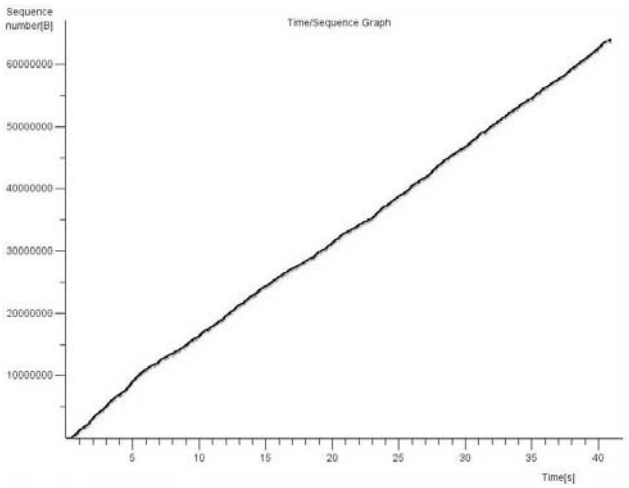

Figure 5: TCP sequence number trace for a NEMO3D run with no snapshot

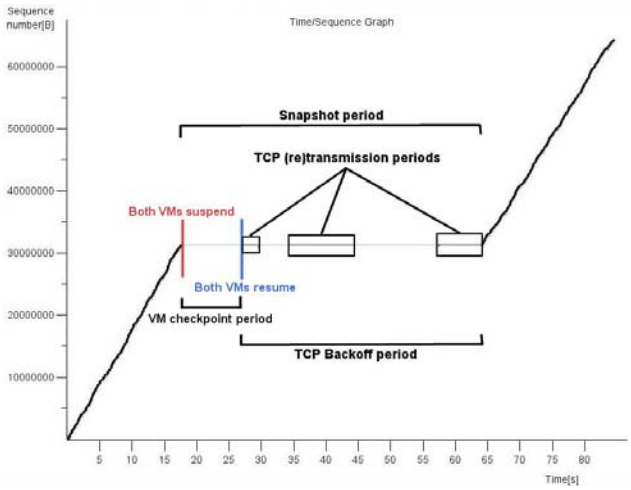

Figure 6: TCP sequence number trace for a NEMO3D run with snapshot

hosted by two different physical machine platforms. For the first platform, we use two Sunfire V20Z servers each with a single 2.6GHz AMD Opteron processor and we let each server host 1, 2 or 4 VMs for the 2, 4 or 8-node VIOLIN. For the second platform, we use 2, 4 and 8 PowerEdge 1750 Dell Servers each with 3.06GHz Intel Xeon processors. On this platform we let each server host one VM of the VIOLIN. On both platforms, each VM uses 700MB of RAM due to the high memory requirement of NEMO3D and is set up with MPICH2-1.0.5.

Before presenting the results, we use Figures 5 and 6 to illustrate our evaluation methodology. The two figures show the impact of a snapshot on the execution of a very short run of NEMO3D (subsequent experiments involve much longer runs). The figures are generated by plotting the TCP sequence number of packets transmitted in a 2-node VIOLIN as a function of time where each VM is hosted by a SunFire server. By using Xen logs, we can break the snapshot overhead into its two major components: the VM checkpoint period and the TCP backoff period. As shown in Figure 6, TCP backoff dominates the overall snapshot period. It only takes about 10 seconds to save the VM image whereas it takes about 37 seconds for TCP congestion control mechanism to recover from a time-out and/or receipt of triple ACKs that can happen following a snapshot and before both the sender and receiver VMs transition to the post-snapshot state.

In our experiments, we define the period between the initiation of a snapshot operation and the end of the TCP backoff as the snapshot overhead. This makes our evaluation different from (and in our view more complete than) the evaluations reported in related works in Section 2: We measure not only the overhead of the snapshot operation per se but also the side effects of snapshot on application execution following the snapshot. We note that the TCP backoff period is incurred by all system-level snapshot techniques (e.g., checkpointing a process, a pod [10], or a VM as in our work).

TCP backoff following a snapshot or upon a snapshot restoration mainly stems from the packet losses during the snapshot when the processes/pods/VMs become in-operational or when both the sender and receiver VMs are not in the same pre-snapshot or post-snapshot state. However, we do point out that the TCP backoff overhead varies for different applications and systems (mostly due to varying amount of application communication and secondary storage bandwidth) and should *not* necessarily be regarded as a part of the snapshot overhead. For the execution depicted in Figure 6, the overall snapshot period is about 47 seconds.

We also notice that in all our experiments, the TCP backoff overhead is only a few seconds apart for different TCP connections in the VIOLIN, so we decide to use one connection to measure the overhead as other connections experience a similar overhead at about the same time.

Tables 1 and 2 summarize the results we obtain from the 2, 4 and 8-node VIOLINs on the two physical platforms, respectively. From now on we will refer to the Sunfire and PowerEdge servers as Platform I and Platform II, respectively. In each of the two tables, the first two rows report the end-to-end execution time of NEMO3D (in minutes and seconds) without snapshot and with a snapshot halfway through its execution. The third row shows the total snapshot overhead, which is broken down to the VM checkpoint (involving copying VM image to disk) and TCP backoff overheads in the fourth and fifth rows, respectively. The last row measures the NEMO3D execution time from the start to the time by which the snapshot is completely written to the disk *plus* the time from the snapshot restoration to the completion of the execution.

From Table 1, we observe that the checkpoint duration for each VM linearly scales with the number of VMs in each server of Platform I. On Platform II, since each VM exclusively runs in a server, the VM checkpoint duration remains the same. In addition to the number of VMs per host, the amount of RAM allocated to a VM and the secondary storage bandwidth are the two main factors influencing the checkpoint overhead for a VM. Also, due to VM placement, the advertised receive window for TCP connections on Platform I are much smaller compared to the constant values seen on Platform II. This translates into larger advertised receive windows and higher transmission rates for connections on Platform II, which in turn leads to the larger TCP

Platform I	2 VMs	4 VMs	8 VMs
Exe. time w/o snapshot	19m 30s	17m 03s	44m 31s
Exe. time with snapshot	23m 1s	19m 40s	46m 37s
Total snapshot overhead	10s	50s	69s
VM checkpoint overhead	10s	19s	41s
TCP backoff overhead	0s	31s	28s
Exe. time with restore	23m 41s	18m 25s	47m 15s

Table 1: Snapshot overhead measurement results on Platform I

Platform II	2 VMs	4 VMs	8 VMs
Exe. time w/o snapshot	44m 43s	36m 18s	62m 36s
Exe. time with snapshot	47m 0s	36m 52s	63m 55s
Total snapshot overhead	73s	78s	122s
VM checkpoint overhead	20s	18s	20s
TCP backoff overhead	53s	60s	102s
Exe. time with restore	46m 29s	36m 56s	63m 18s

Table 2: Snapshot overhead measurement results on Platform II

backoff overhead in Table 2.

Interestingly, we notice that the TCP trace for the 2-VM VIOLIN on Platform I exhibits alternating CPU and network intensive periods[1]. And it happens that the VIOLIN snapshot is taken during one of the CPU intensive periods. As a result, there is almost no TCP backoff overhead in this case. For other applications with alternating computation and communication intensive periods, VIOLIN can potentially exploit this fact and minimize the snapshot overhead by avoiding taking a snapshot during the communication intensive periods. We note that some of the TCP backoff in the 4-VM and 8-VM experiments for Platform I is due to not setting the "forward delay" feature of the post-snapshot and pre-snapshot bridges to 0.

In Tables 1 and 2, the application execution times with (2nd row) and without (1st row) VIOLIN snapshot are reasonably close[2]. This observation indicates that for long-running applications (which are more likely to perform and benefit from snapshots than short ones), the latency incurred by the snapshot procedure tends to be insignificant in comparison with the long application execution time. To evaluate the efficiency of snapshot-based failure recovery, we also measure the time for VIOLIN to restore from its snapshot and execute the program to completion. As shown in the last rows of Tables 1 and 2, the end-to-end execution time based on VIOLIN snapshot restoration is similar to the execution time of the original run with snapshot (2nd row of Tables 1 and 2).

[1]We do not notice a similar pattern in other experimental traces.

[2]However, we note that the difference in execution time between the 1st and the 2nd rows is not exactly the snapshot overhead in the 3rd row. The reason is that other unreproducible factors such as network conditions also affect the execution time of individual runs. In fact, we observe such variation even between two runs – both without snapshot.

7. CONCLUSIONS

We have presented the design and implementation of a distributed snapshot capability for the VIOLIN virtual networked environment. By adapting a classic distributed snapshot algorithm, the VIOLIN distributed snapshot technique captures a causally consistent execution state of the entire virtual networked environment, which can be utilized for fault-recovery, management, and replay of the virtual environment. The capture of VIOLIN snapshots does not require modifications to the applications and operating systems running inside the VIOLIN, providing effective support for legacy applications as well as full system emulations. Preliminary evaluation results using a real-world legacy application indicate that our technique incurs reasonable overhead and small disruption to a VIOLIN's normal execution.

8. ACKNOWLEDGMENTS

We thank the anonymous reviewers for their helpful comments. We also thank Gerhard Klimeck, Hoon Ryu, and Rick Kennell for their timely help with the NEMO3D experiment setup. This work was supported in part by the National Science Foundation under grants OCI-0438246, CNS-0546173, CNS-0720665, OCI-0721680, OCI-0749140.

9. REFERENCES

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

[2] http://tldp.org/HOWTO/LVM-HOWTO/.

[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] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Transactions on Computer Systems*, 3(1):63–75, February 1985.

[5] Y. Chen, J. S. Plank, and K. Li. CLIP - a checkpointing tool for message-passing parallel programs. *In Preecedings of the Supercomputing, San Jose, California*, November 1997.

[6] G. E. Fagg and J. Dongarra. FT-MPI: Fault Tolerant MPI, supporting dynamic applications in a dynamic world. *In LNCS: Proceedings of the 7th European PVM/MPI User's Group Meeting. Springer-Verlag*, 1908:346–353, 2000.

[7] R. Figueiredo, P. Dinda, and J. Fortes. A case for grid computing on virtual machines. *Proc. International Conference on Distributed Computing Systems (ICDCS)*, May 2003.

[8] X. Jiang and D. Xu. VIOLIN: Virtual internetworking on overlay infrastructure. Department of Computer Science Technical Report CSD TR 03-027, Purdue University, July 2003.

[9] X. Jiang, D. Xu, H. J. Wang, and E. H. Spafford. Virtual playgrounds for worm behavior investigation. *Proceedings of 8th International Symposium on Recent Advances in Intrusion Detection (RAID 2005)*, September 2005.

[10] O. Laadan, D. Phung, and J. Nieh. Transparent checkpointing-restart of distributed applications on commodity clusters. *In Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2005)*, September 2005.

[11] T. H. Lai and T. Yang. On distributed snapshots. *Information Processing Letters*, (25):153–158, 1987.

[12] F. Mattern. Efficient algorithms for distributed snapshots and global virtual time approximation. *Journal of Parallel and Distributed Computing*, 18(4), 1993.

[13] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of Zap: A system for migrating computing environments. *In Proceedings of the Fifth Symposium on Operating System Design and Implementation (OSDI), Boston, MA*, December 2002.

[14] X. Ren, R. Eigenmann, and S. Bagchi. Failure-aware checkpointing in fine-grained cycle sharing systems. *In Proceedings of the 16th International Symposium on High Performance Distributed Computing*, June 2007.

[15] P. Ruth, X. Jiang, D. Xu, and S. Goasguen. Virtual distributed environments in a shared infrastructure. *IEEE Computer, Special Issue on Virtualization Technologies*, 38(5), 2005.

[16] S. Sankaran, J. M. Squyres, B. Barret, A. Lumsdaine, J. Duell, P. Hargrove, and E. Roman. The LAM/MPI checkpoint/restart framework: System-initiated checkpointing. *In Proceedings of the LACSI Sypsosium*, October 2003.

[17] 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. *In Proceedings of the IEEE International Conference on Cluster Computing (Cluster 2007)*, September 2007.

[18] H. Zhong and J. Nieh. CRAK: Linux checkpoint/restart as a kernel module. Technical Reports CUCS-014-01, Department of Computer Science, Columbia University, New York, November 2001.