HotSnap: A Hot Distributed Snapshot System for Virtual Machine Cluster

Lei Cui, Bo Li, Yangyang Zhang, Jianxin Li
Beihang University, Beijing, China
{*cuilei, libo, zhangyy, lijx*}*@act.buaa.edu.cn*

Abstract

The management of virtual machine cluster (VMC) is challenging owing to the reliability requirements, such as non-stop service, failure tolerance, etc. Distributed snapshot of VMC is one promising approach to support system reliability, it allows the system administrators of data centers to recover the system from failure, and resume the execution from a intermediate state rather than the initial state. However, due to the heavyweight nature of virtual machine (VM) technology, applications running in the VMC suffer from long downtime and performance degradation during snapshot. Besides, the discrepancy of snapshot completion times among VMs brings the TCP backoff problem, resulting in network interruption between two communicating VMs. This paper proposes HotSnap, a VMC snapshot approach designed to enable taking *hot* distributed snapshot with milliseconds system downtime and TCP backoff duration. At the core of HotSnap is transient snapshot that saves the minimum instantaneous state in a short time, and full snapshot which saves the entire VM state during normal operation. We then design the snapshot protocol to coordinate the individual VM snapshots into the global consistent state of VMC. We have implemented HotSnap on QEMU/KVM, and conduct several experiments to show the effectiveness and efficiency. Compared to the live migration based distributed snapshot technique which brings seconds of system downtime and network interruption, HotSnap only incurs tens of milliseconds.

1 Introduction

With the increasing prevalence of cloud computing and IaaS paradigm, more and more distributed applications and systems are migrating to and running on virtualization platform. In virtualized environments, distributed applications are encapsulated into virtual machines, which are connected into virtual machine cluster (VMC) and coordinated to complete the heavy tasks. For example, Amazon EC2 [1] offers load balancing web farm which can dynamically add or remove virtual machine (VM) nodes to maximize resource utilization; CyberGuarder [22] encapsulates security services such as IDS and firewalls into VMs, and deploys them over a virtual network to provide virtual network security service; Emulab [12] leverages VMC to implement on-demand virtual environments for developing and testing networked applications; the parallel applications, such as map-reduce jobs, scientific computing, client-server systems can also run on the virtual machine cluster which provides an isolated, scaled and closed running environment.

Distributed snapshot [13, 27, 19] is a critical technique to improve system reliability for distributed applications and systems. It saves the running state of the applications periodically during the failure-free execution. Upon a failure, the system can resume the computation from a recorded intermediate state rather than the initial state, thereby reducing the amount of lost computation [15]. It provides the system administrators the ability to recover the system from failure owing to hardware errors, software errors or other reasons.

Since the snapshot process is always carried out periodically during normal execution, transparency is a key feature when taking distributed snapshot. In other words, the users or applications should be unaware of the snapshot process, neither the snapshot implementation scheme nor the performance impact. However, the traditional distributed systems either implement snapshot in OS kernel [11], or modify the MPI library to support snapshot function [17, 24]. Besides, many systems even leave the job to developers to implement snapshot on the application level [3, 25]. These technologies require modification of OS code or recompilation of applications, thus violating the transparency from the view of implementation schema.

The distributed snapshot of VMC seems to be an ef-

fective way to mitigate the transparency problem, since it implements snapshot on virtual machine manager (VMM) layer which encapsulates the application's running state and resources without modification to target applications or the OS. Many systems such as VNSnap [18] and Emulab [12] have been proposed to create the distributed snapshot for a closed network of VMs. However, these methods still have obvious shortcomings.

First, the snapshot should be non-disruptive to the upper applications, however the state-of-the-art VM snapshot technologies, either adopt stop-and-copy method (e.g., Xen and KVM) which causes the service are completely unavailable, or leverage live migration based schema which also causes long and unpredictable downtime owing to the final copy of dirty pages [26].

Second, the distributed snapshot should coordinate the individual snapshots of VMs to maintain a global consistent state. The global consistent state reflects the snapshot state in one virtual time epoch and regards causality, implying the VM before snapshot cannot receive the packets send from the VM that has finished the snapshot to keep the consistent state during distributed snapshot (further explanations about global consistent state can be referred in appendix A). However, due to the various VM memory size, variety of workloads and parallel I/O operations to save the state, the snapshot start time, duration time and completion time of different VMs are always different, resulting in the TCP back-off issue [18], thereby causing network interruption between the communicating VMs. Figure 1 demonstrates one such case happened in TCP's three-way handshake. Worse still, for the master/slave style distributed applications, the master always undertake heavier workloads so that cost more time to finish the snapshot than the slaves, therefore, the slaves which finish the snapshot ahead cannot communicate with the master until the master snapshot is over, causing the whole system hung. As a result, the master snapshot becomes the *short-board* during distributed snapshot of master/slave systems.

Third, most distributed snapshot technologies adopt the coordinated snapshot protocol [13] to bring the distributed applications into a consistent state. This requires a coordinator to communicate snapshot-related commands with other VMs during snapshot. In many systems, the coordinator is setup in the customized module such as VIOLIN switch in VNSnap [18] and XenBus handler used in Emulab [12], thus lack of generality in most virtualized environments.

To mitigate the problems above, we propose HotSnap, a system capable of taking *hot* distributed snapshot that is transparent to the upper applications. Once the snapshot command is received, HotSnap first suspends the VM, freezes the memory state and disk state, creates a *transient snapshot* of VM, and then resumes the VM. The

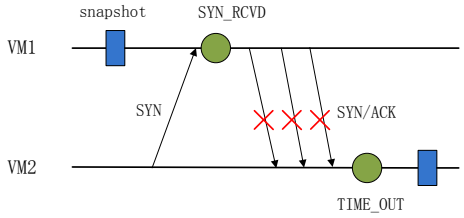


Figure 1: A TCP handshake case during distributed snapshot. VM_2 first sends SYN to VM_1 to request a TCP connection, at this moment VM_2 has not begin its snapshot; VM_1 receives this request, turn its own state into SYN_RCVD, and then sends SYN/ACK back to VM_2. We notice that now VM_1 has finished snapshot, and based on the coordinated protocol, packets sent from VM_1 will not be accepted by VM_2 until VM_2 has finished its own snapshot. If VM_2's snapshot duration exceeds TCP timeout, connection will fail.

transient snapshot only records the minimum instantaneous state, including CPU and device states, as well as two bitmaps reserved for memory state and disk state, bringing only milliseconds of VM downtime, i.e., *hot* for upper applications. The *full snapshot* will be acquired after resuming the VM, it saves the entire memory state in a copy-on-write (COW) manner, and create the disk snapshot in the redirect-on-write (ROW) schema; the COW and ROW schemas enable creating the *full snapshot* without blocking the execution of VM, i.e., live snapshot. Because the *transient snapshot* introduces only milliseconds of downtime, the discrepancy of downtime among different VM snapshots will be minor, thereby minimizing the TCP backoff duration.

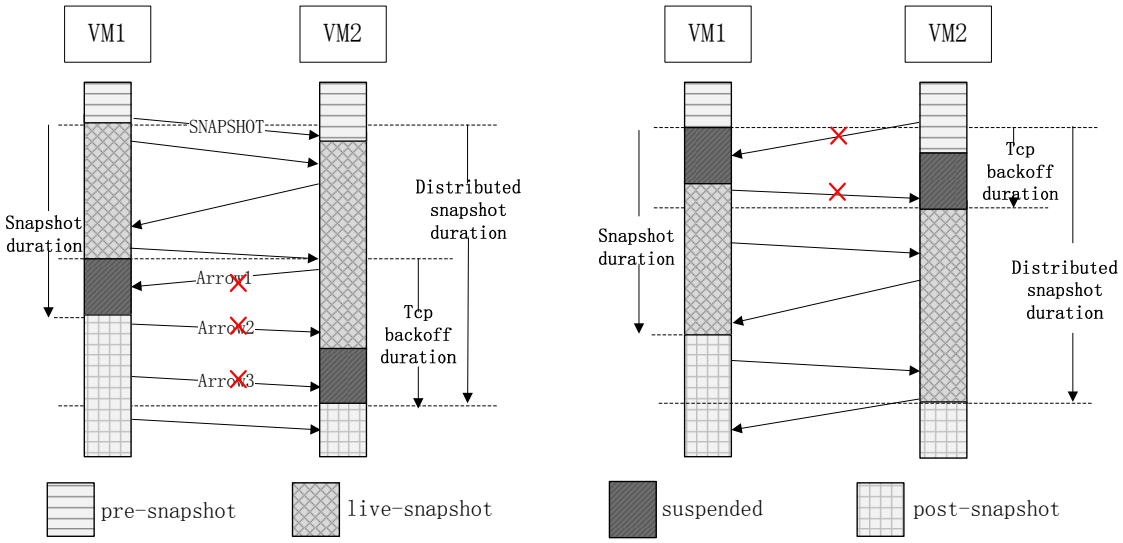
HotSnap is completely implemented in VMM layer, it requires no modification to Guest OS or applications, and can work without other additional modules. The major contributions of the work are summarized as follows:

1) We propose a VM snapshot approach combined of *transient snapshot* and *full snapshot*. The approach completes snapshot transiently, enables all VMs finish their snapshots almost at the same time, which greatly reduces the TCP backoff duration caused by the discrepancy of VMs' snapshot completion times.

2) A classic coordinated non-blocking protocol is simplified and tailored to create the distributed snapshot of the VMC in our virtualized environment.

3) We implement HotSnap on QEMU/KVM platform [20]. Comprehensive experiments are conducted to evaluate the performance of HotSnap, and the results prove the correctness and effectiveness of our system.

The rest of the paper is organized as follows. The next section provides an analysis of the traditional dis-



a) VNSnap distributed snapshot.

b) HotSnap distributed snapshot.

Figure 2: Comparison of VNSnap and HotSnap.

tributed snapshot and their problems. Section 3 introduces the HotSnap method, describes the *transient snapshot*, *full snapshot* and coordinated protocol. Section 4 describes the implementation-specific details on QEMU/KVM platform. The experimental results are shown in Section 5. Finally we present the previous work related to HotSnap in section 6 and conclude our work in Section 7.

2 An Analysis of Distributed Snapshot

The distributed snapshot includes independent VM snapshot and the coordinated protocol. Stop-and-copy schema is a simple way to create snapshot of individual VM, but this schema introduces long downtime of Guest OS and the upper applications running inside the VM, thus is impractical in many scenarios that deliver services to users. The live snapshot technologies leverage pre-copy based migration to achieve live snapshot by iteratively saving the dirty pages to the snapshot file [12, 18]. In this section, we will analyze the live migration based distributed snapshot proposed in VNSnap [18], and explain how it results in TCP backoff problem.

Figure 2(a) demonstrates the procedure of VNSnap distributed snapshot. Although VNSnap exploits the VIOLIN [12] switch to execute the coordinated protocol, we treat VM_1 as the coordinator for clarity. Upon distributed snapshot, the coordinator, i.e., VM_1, will send SNAPSHOT command to VM_2, and then create the snapshot of VM_1 itself. VNSnap leverages live migration to iteratively save the dirtied pages into stable storage or reserved memory region until some requirements are satisfied, such as the amount of dirty pages are minor enough, or the size cannot be further reduced even more iterations are conducted. Then VNSnap suspends the VM, stores the final dirty memory pages, saves other devices' state and creates the disk snapshot. After these steps, the snapshot of VM_1 is over and VM_1 is resumed. Upon receiving the SNAPSHOT command from VM_1, VM_2 follows the same procedure as VM_1 to create its own snapshot. VNSnap drops the packets send from the post-snapshot VM to pre-snapshot VM, to keep the global state consistent.

Take this tiny cluster which consists of two VMs as an example, the distributed snapshot duration time is from the start time of VM_1 snapshot to the end time of VM_2 snapshot (suppose VM_2 finishes snapshot later than VM_1), the TCP backoff duration is from the start of VM_1 suspend to the end of VM_2 suspend. The packets result in TCP backoff fall into three categories: 1) VM_1 is suspended while VM_2 is in live-snapshot, the packets send from VM_2 to VM_1 will not arrive, as $Arrow_1$ illustrates; 2) VM_1 finishes snapshot and then turns into post-snapshot state, but VM_2 is before or during snapshot. In this situation, packets send from VM_1 will be dropped to keep the consistent state of distributed snapshot. $Arrow_2$ shows such a case. 3) VM_1 is in post-snapshot, but VM_2 is suspended, VM_2 cannot receive the packets send from VM_1, as $Arrow_3$ shows.

Based on the three types of packets, we can conclude that two aspects affect the TCP backoff duration in dis-

tributed snapshot. One is the downtime of individual VM snapshot; the longer downtime implies more lost packets, thereby causing longer TCP backoff duration. Another is the discrepancy of the snapshot completion times, as the $Arrow_2$ illustrates, the packets send from VM_1 which has finished snapshot ahead will be dropped until VM_2 completes the snapshot.

According to the above analysis, the VNSnap distributed snapshot method has three drawbacks: First, the pre-copy based live migration method need to iteratively save the dirtied pages, thus the snapshot downtime is directly related to the workloads inside the VM and I/O bandwidth; it may last seconds in memory intensive scenarios [26]. Second, VNSnap proposes a VNSnap-memory method to reduce the TCP backoff duration, it saves the memory state into a reserved memory region whose size is the same to the VM memory size; this is wasteful and impractical in the IaaS platform which aims to maximize resource utilization. Third, the snapshot duration time is proportional to the memory size and workload, therefore the discrepancy of snapshot completion times would be large for VMs with various memory sizes, further leads to long TCP backoff duration. Even for the VMs with identical memory size and same applications, the snapshot completion times are still various owing to the parallel disk I/O for saving large amount of memory pages. Besides, the experimental results in VNSnap [18] also show this live migration based snapshot method brings seconds of TCP backoff duration.

3 Design of HotSnap

The design of HotSnap includes a new individual VM snapshot method and a coordinated non-block snapshot protocol. We firstly describe the design of the HotSnap method, then introduce the procedure of HotSnap for individual VM, and lastly describe the coordinated snapshot protocol to acquire a global consistent state of the virtual machine cluster.

3.1 Overview of HotSnap

Figure 2(b) illustrates our HotSnap approach. Different from VNSnap which suspends the VM at the end of the snapshot, HotSnap suspends the VM once receiving the SNAPSHOT command, takes a *transient snapshot* of VM and then resumes the VM. The *full snapshot*, which records the entire memory state, will be completed during the subsequent execution. Note that the VM actually turns to post-snapshot state after *transient snapshot* is over. In this approach, the TCP backoff duration is from the start of VM_1 *transient snapshot* to the end of VM_2 *transient snapshot*, and the entire distributed snapshot duration starts from the start of VM_1 *transient snapshot* to the end of VM_2 *full snapshot*.

In HotSnap approach, we suspend the VM and create the *transient snapshot* in milliseconds, thus the downtime during individual VM snapshot would be minor. Besides, in the nowadays IaaS platform or data center which are always configured with high bandwidth and low latency network, the round trip time is always less than 1ms, so the VMs can receive the SNAPSHOT command and then start to create snapshot almost simultaneously. As a result, the *transient snapshot* of VMs can start almost simultaneously and finish in a very short time, consequently minimizing the TCP backoff duration.

The individual VM snapshot combined of *transient snapshot* and *full snapshot*, as well as the coordinated protocol are two key issues to create the *hot* distributed snapshot, and will be described in detail in the following parts.

3.2 Individual VM Snapshot

A VM snapshot is a point-in-time image of a virtual machine state; it consists of memory state, disk state and devices' states such as CPU state, network state, etc. Our snapshot consists of two parts, one is a *transient snapshot* which contains the devices state, disk state and metadata of memory state; another is *full snapshot* which actually records the memory state. We divide the individual VM snapshot procedure into three steps as shown in Figure 3.

Step 1, Suspend VM and Create Transient Snapshot. We suspend the VM, store the devices state, set write-protect flag to each memory page, create two bitmaps to index the memory state and disk state, and create a new null disk file. We adopt the redirect-on-write method to create disk snapshot, therefore the disk snapshot is completed after the bitmap and disk file are created. This step only involves lightweight operations, i.e., bitmap creation, flag setting and device state saving, thus bringing only a few dozens of milliseconds downtime.

Step 2, Resume VM and Create Full Snapshot. We resume the VM to keep the Guest OS and applications running during *full snapshot*. The running applications will issue disk writes as well as dirty memory pages during fault-free execution. For the disk write operation, the new content will be redirected to a new block in the new disk file by the iROW block driver [23]. For the write operation on one memory page which is write-protected, page fault will be caught in the VMM layer. HotSnap for handling page fault will block the memory write operation, store the original page content into the snapshot file, remove the write-protect flag, and then allow the guest to continue to write the new content into the page. Meanwhile, a thread is activated to save memory pages ac-

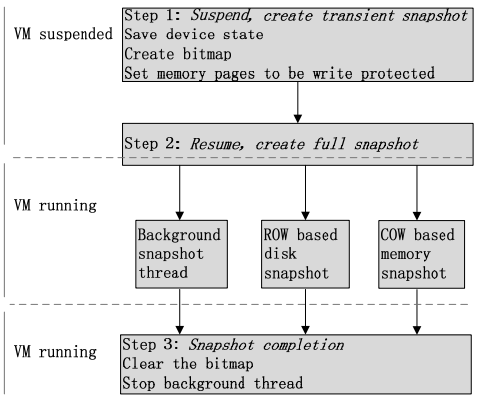


Figure 3: Steps of individual VM snapshot.

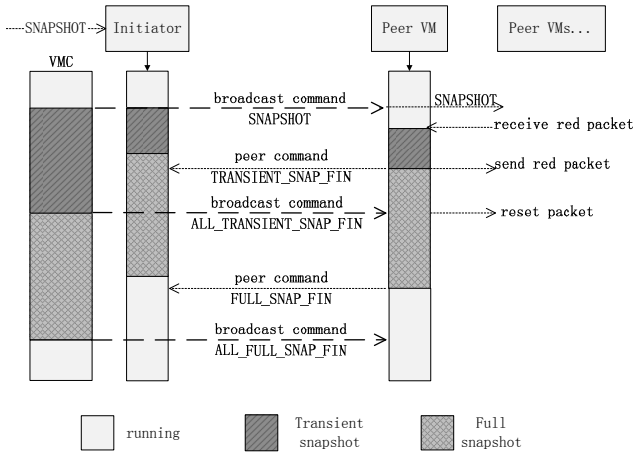


Figure 4: Global distributed snapshot protocol of HotSnap.

tively in background. The copy-on-write based memory snapshot only saves each memory page once and thus keeps the memory snapshot size to be the same to the VM memory size.

Step 3, Snapshot Completion. After all the memory pages are stored into the stable storage, the snapshot process is completed. In the end, we clear the bitmap, and stop the background thread.

3.3 Global Coordinated Non-block Snapshot Protocol

We design a global coordinated non-block snapshot protocol to coordinate the individual VM snapshot processes. Unlike Emulab [12] which synchronizes the clocks of all VMs to ensure these VMs are suspended for snapshot simultaneously, we deploy VMs on high bandwidth and low latency network so that the VMs can receive the message and start the snapshot at the same time. It is worth noting that Emulab's clock synchronization protocol can be utilized to extend the scope of HotSnap.

The pre-snapshot, live-snapshot and post-snapshot are both running state of individual VM, but they need to be distinguished from the view of VMC distributed snapshot for consistency requirement. Note that VNSnap suspends VM at the end of snapshot, so the live-snapshot can be regarded as pre-snapshot. Similarly, in HotSnap which suspends VM at the start, we consider the live-snapshot as post-snapshot, i.e., the state after *transient snapshot*. We leverage the message coloring [21] method to achieve the state distinction in the coordinated protocol, that is, we piggyback the white flag to the packet which is send from the VM in pre-snapshot state and represent the packets from the post-snapshot VM with red flag. If one pre-snapshot VM receives a packet piggybacked with a red flag, it will create its own snapshot first, and then receive and handle the packet.

There exist two roles in HotSnap system: initiator and peer. Unlike the VNSnap [18] or Emulab [12] systems that use separate modules such as VIOLIN switch or XenBus as the initiator, each VM in HotSnap can be either the initiator or a peer. And there is only one initiator during a failure-free VMC snapshot process. Each peer records its snapshot states including *transient snapshot* state and *full snapshot* state. The initiator not only records the snapshot states, but also maintains these states of the whole VMC distributed snapshot. Figure 4 illustrates how the coordinated protocol works, the initiator after receiving the SNAPSHOT command from the user or administrator, will first broadcast this command to all peers, and then takes its own *transient snapshot*. The peers will trigger the snapshot process when receiving two kinds of packets, the SNAPSHOT message from the initiator, or the packet piggybacked with red flag. Once finishing the *transient snapshot*, the peer VM will send a TRANSIENT_SNAP_FIN message to the initiator, and color the transmitted packets with the red flag to imply the peer is in post-snapshot state. After finishing the snapshot itself and receiving all peers' TRANSIENT_SNAP_FIN messages, the initiator will broadcast the ALL_TRANSIENT_SNAP_FIN message to all peer VMs to notify the completion of *transient snapshot* of the whole VMC. The peers who receive this message will cancel packet coloring and reset the packet with the white flag immediately. The distributed snapshot procedure continues until the initiator receives all VMs' FULL_SNAP_FIN message which marks the completion of the *full snapshot*. The initiator will finally broadcast ALL_FULL_SNAP_FIN message to all peer VMs, to declare the completion of the distributed snapshot of the VMC.

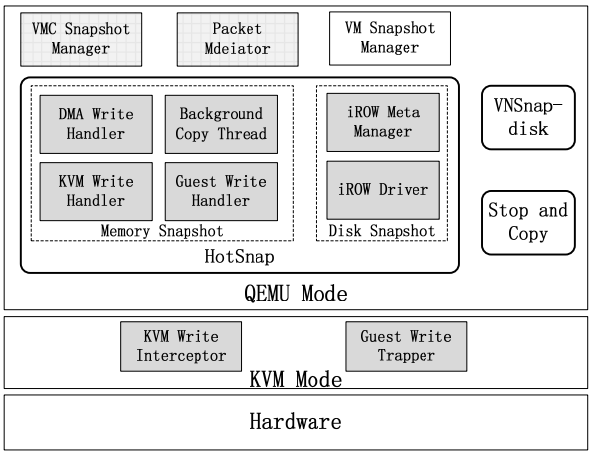


Figure 5: HotSnap system architecture.

4 System Implementation

This section presents the implementation issues in HotSnap. We start by describing the overall architecture, and then go on sub-level details and optimizations.

4.1 System architecture

We implement HotSnap on qemu-kvm-0.12.5 with Linux kernel 2.6.32.5-amd64. The system architecture is illustrated in Figure 5. HotSnap consists of two main components: the coordinated snapshot protocol component and VM snapshot component.

The coordinated protocol component includes the VMC Snapshot Manager and Packet Mediator. The VMC Snapshot Manager acts as the initiator during the distributed snapshot; it will firstly broadcast the SNAPSHOT command to other VMs, and then notify the VM Snapshot Manager to take the snapshot of VM itself. Packet Mediator has two functions: change the color of the sending packets according to the VM snapshot progress; decide whether or not to accept a packet by comparing packet's color with its own state.

VM Snapshot Manager is in charge of taking the individual VM snapshot; it supports three snapshot schemas, the HotSnap method, the Stop-and-copy snapshot method which is default adopted in QEMU/KVM, and the VNSnap-disk snapshot proposed in VNSnap [18] based on the pre-copy live migration in QEMU. In our HotSnap system, VM Snapshot Manager calls the Disk Snapshot Module to create the disk snapshot in the *transient snapshot* procedure, and exploits the Memory Snapshot Module to save the memory state during both *transient snapshot* and *full snapshot*. The details about Disk Snapshot Module can be referred in our iROW work [23], thus are omitted in this paper.

4.2 COW Based Memory Snapshot

We create the memory snapshot in the copy-on-write (COW) manner. Writing large amount of memory state into a file within the kernel is terrible and may crash the system, so we save the memory state in user space when taking the snapshot. However, in the QEMU/KVM platform, the guest memory pages will be written by not only the guest OS or applications inside, but also by the simulated DMA devices and the KVM module. In the following, we will describe how we save the entire memory state in detail.

Guest Write Handler: During the *transient snapshot*, we call the function *cpu_physical_memory_set_dirty_tracking* in QEMU to set the write protect flag for each guest physical memory page, so that the VMM layer can trap the write page fault triggered by the running applications or the guest OS. Once page fault occurs, the execution of Guest OS will hang and exit to KVM module. Then we handle the page fault in the function *handle_ept_violation*, which is defined to handle memory access violation for the specific CPU with the Extended Page Tables (EPT) feature. If the trapped page fault is owing to be write protected, we record the guest frame number (gfn) of the page, set the exit reason as EXIT_REASON_HOTSNAP, and then exit to QEMU for handling the exception. The QEMU component when encountering the exit whose reason is EXIT_REASON_HOTSNAP, will save the memory page indexed by gfn into the snapshot file, notify KVM to remove the write protect flag of the page, and then issue the function *kvm_run* to activate the VM. Afterwards, the resumed VM will continue to run and write that memory page without triggering page fault again. In addition, each page fault incurs exit from Guest to KVM kernel then to QEMU user space, as well as entry in the opposite direction, resulting in performance loss. Thus, we save dozens of memory pages and remove their associated write protect flags when handling one page fault. This can benefit from the feature of memory locality, thereby reducing the frequency of page fault and occurrences of context switch between Guest OS and VMM layer.

DMA Write Handler: DMA is a widely-used technology to accelerate the transfer speed of I/O related devices. QEMU also simulate the DMA schema for IDE driver and virtio network driver in qemu-kvm-0.12.5. In their implementation, a reserved memory region is mapped between the guest and the simulated DMA driver. For read or write operations, the driver just map the data to or from the guest directly instead of I/O operations. This method dirties the guest memory pages with-

out triggering the page fault and thus cannot be caught in the Guest Write Handler way. Therefore, we intercept the DMA write operations directly in QEMU. Take disk I/O as an example, the function *dma_bdrv_cb* in QEMU implements the DMA write operations, it first fetches the address and length of the data to be written from the scatter/gather list, maps this address to the guest physical page by the function *cpu_physical_memory_map* and finally writes the data into the guest disk file. Therefore, we intercept the write operation in the function *dma_bdrv_cb*, save the original memory page content, and then resume the execution of DMA write. One thing to be noted is that only the disk device and network device in qemu-kvm-0.12.5 support the DMA schema, but the newer versions such as qemu-kvm-1.4.0 add the DMA feature to more devices including disk device, sound card and video card. However, the methodologies are the same in dealing with DMA interception.

KVM Write Handler: The KVM kernel set value to the registers such as MSR for task switch operations, key-board operations, thus causing the guest memory pages dirtied. Similar to DMA Write Hanlder, we intercept KVM write in the function *kvm_write_guest* which is implemented in KVM to assign value to certain address space. The KVM kernel always repeatedly writes the same memory pages and the page count is only a little, so we first store the intercepted pages into a kernel buffer without blocking the execution of *kvm_write_guest*, then copy these pages in buffer to the user space asynchronously. In this way, all the memory pages dirtied by KVM will be saved to stable storage in the user space.

Background Copy: To accelerate creating the memory snapshot, a background copy thread is issued to store the guest memory pages concurrently. It traverses all the guest memory pages and saves the pages that have not been saved by the other three memory snapshot manners. Since all these four kinds of memory snapshot manners may save the same page simultaneously, a bitmap is reserved for indexing whether the page is saved or not, to guarantee completeness and consistency of the memory state. The bitmap is shared between QEMU and KVM. All the four manners should first store the page, then set the associated bit value and remove the write-protect flag. Or else, there will be concurrency bugs. Let us consider an improper case that set bit and removes flag first. Upon saving a page, the Background copy thread firstly set the associated bit and removes the write-protect flag; however, before the thread saving the page content, the Guest OS dirties this page since write protect flag has been removed, causing the thread to save the false page content. Thus, when taking memory snapshot, the interceptions on DMA write and KVM write will first check the associated bit value of the page about to write, save the original page if the bit value is not set, or ignore otherwise. The Guest Write Handler also takes the same procedure, for the bit value has been set, it allows the guest to continue running without exit to QEMU.

4.3 Log and Resend On-the-fly Packets

Dropping the on-the-fly packets send from post-snapshot VM to pre-snapshot VM during TCP backoff duration is a simple way to obtain the global consistent state. However, this way will increase the TCP backoff duration, the reason is as follows: TCP or other upper level protocols will retransmit the packets that are not acknowledged in a certain time named Retransmit Timeout (RTO), to achieve correctness and reliability of message passing. That means, if the the packets are lost, these reliable protocols will delay resending the packet until timeout. The default RTO value is 3 seconds in Linux 2.6.32 kernel, it is always larger than the TCP backoff duration which lasts tens of milliseconds in HotSnap system (in Section 5.3). Thus, if the packets are dropped, the actual network interruption time will be the RTO value at the minimum, i.e., 3 seconds. Worse still, the RTO value will increase manyfold until receiving the acknowledgement.

Instead of dropping the packets directly, the Packet Mediator component intercepts the read/write operations issued by the tap device which is connected to the virtual network interface card (VNIC) of VM, logs the on-the-fly packets send from the post-snapshot VM to the pre-snapshot VM, and then stores the packets into a buffer. After completing the *transient snapshot*, the Packet Mediator will first fetch the packets from the buffer, send them to the VNIC of the VM, and finally resume the normal network communication.

5 Experimental Evaluation

We apply several application benchmarks to evaluate HotSnap. We begin by illustrating the results for creating snapshot of individual VM, and then compare the TCP backoff duration between three snapshot modes under various VMC configurations, lastly we characterize the impacts on performance of applications in VMC.

5.1 Experimental Setup

We conduct the experiments on four physical servers, each configured with 8-way quad-core Intel Xeon 2.4GHz processors, 48GB DDR memory, and Intel 82576 Gigabit network interface card. The servers are connected via switched Gigabit Ethernet. We configure 2GB memory for the virtual machines unless specified otherwise. The operating system on physical servers and

virtual machines is debian6.0 with 2.6.32-5-amd64 kernel. We use qemu-kvm-0.12.5 as the virtual machine manager. The workloads inside the VMs includes:

Idle workload means the VM does nothing except the tasks of OS self after boot up.

Kernel Compilation represents a development workload involves memory and disk I/O operations. We compile the Linux 2.6.32 kernel along with all modules.

Matrix Multiplication multiplies two randomly generated square matrices, this workload is both memory and CPU intensive.

DBench [4] is a well known benchmark tool to evaluate the file system, it generates I/O workloads.

Memcached [8] is an in-memory key-value store for small chunks of data, the memcached server when receiving a request containing the key, will reply with the value. We set memcached server in one VM, and configure mcblaster [7] as client in another VM to fill the data in the memcached instance and then randomly request the data from the memcached server.

Distcc [5] is a compilation tool that distributes the compilation tasks across the VMs connected in the VMC. It contains one Distcc client and several servers. The client distributes the tasks to servers, and the servers after receiving the task will handle the task and then return the result to the client. This workload is memory and network intensive. We use Distcc to compile the Linux 2.6.32 kernel with all modules in the VMC.

BitTorrent [2] is a file transferring system, the peers connect to each other directly to send and receive portions of file. Different from distcc that is centralized, BitTorrent is peer-to-peer in nature.

We compare the three snapshot methods, all these methods save the snapshot file in local host.

Stop-and-copy. The default snapshot method used in QEMU/KVM, it suspends the VM while creating snapshot.

VNSnap-disk. We implement the VNSnap-disk snapshot method based on live migration in QEMU/KVM, save the memory state into the stable storage directly.

HotSnap. Our snapshot method that suspends the VM first, then create the *transient snapshot* and *full snapshot*.

QEMU/KVM optimizes taking snapshot by compressing the zero pages with one byte, thus reduce the amount of saved state. This incurs unfairness in experiments, the reason is, the VM after long time running may dirty more zero pages, and experience longer snapshot duration than the new booted VM which contains large number of zero pages, thus leading to unpredictable TCP backoff duration between the two VMs. As a result, we abandon the compression codes, save the whole zero page instead of only one byte to eliminate the impact of zero pages.

5.2 Snapshot of Individual VM

We start by verifying the correctness of saved snapshot state for individual VM, and then evaluate the snapshot metrics including downtime, duration and snapshot file size. When calculating the snapshot file size, we count all devices' state, memory state and bitmaps, as well as the disk snapshot which is either a bitmap file in HotSnap or multi-level tree file in the other two modes.

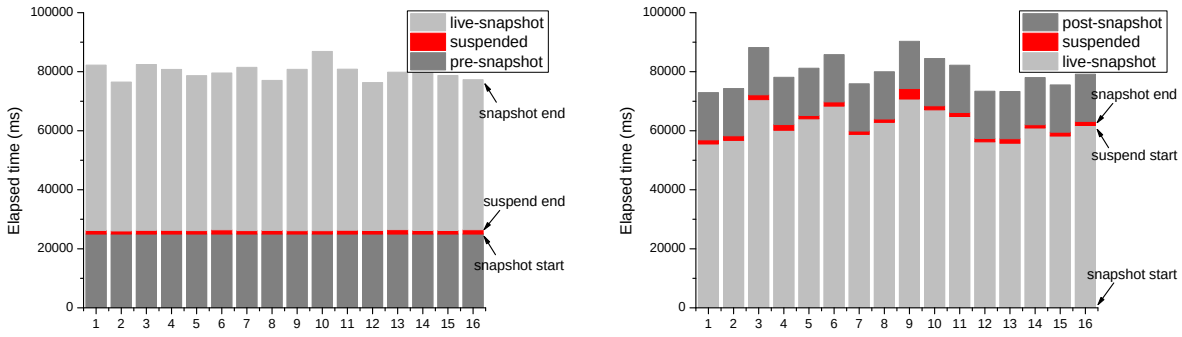
Correctness: Correctness means the snapshot correctly records all the state of the running VM, so that the VM can rollback to the snapshot time point and continue to run successfully. To verify the correctness of HotSnap, we compile the Linux kernel and take several snapshots during normal execution. We pick the snapshots and continue running from these snapshot points, the compiled kernel and modules can execute successfully. Besides, we take snapshot in the Stop-and-copy manner, and then create the HotSnap snapshot. The content of the two snapshots are identical and thus demonstrate the correctness of our system.

Snapshot Metrics: We run Kernel Compilation, Matrix Multiplication, Memcached and Dbench applications to evaluate the performance when taking snapshot of individual VM. We compare HotSnap with Stop-and-copy and VNSnap-disk in terms of snapshot duration, downtime and snapshot file size. As shown in Table 1, the VM only experiences about 35 milliseconds downtime during HotSnap, because this step is to create the *transient snapshot* which only involves lightweight operations. The downtime in VNSnap-disk is various, e.g., 381ms for kernel compilation and 36.8ms when idle, it is related to the workload and I/O bandwidth. HotSnap also achieves shorter snapshot duration and smaller file size than VNSnap-disk. This is because HotSnap saves only one copy for each memory page, while the live migration based VNSnap-disk needs to iteratively save dirty memory pages to snapshot file. The Stop-and-copy method, obviously, incurs dozens of seconds downtime. The bitmap file size is only a little, e.g., 128KBytes to index 4GBytes memory of VM, so that the snapshot file size in HotSnap and Stop-and-copy are both about 2.02GBytes.

5.3 Snapshot of VMC

In this section, we evaluate HotSnap in the virtual machine cluster and focus on the TCP backoff duration. We compare the TCP backoff duration in three snapshot modes, while changing the VMC configurations. The VMC configurations include: VMC under various workload, VMC of different scales, VMC with different VM memory size and disk size, and VMC mixed of VMs with different memory size.

We will first illustrate the details on snapshot progress



(a) HotSnap snapshot details

(b) VNSnap snapshot details

Figure 6: Comparison of TCP backoff details.

that lead to TCP backoff. We build a VMC with 16 VMs, with Distcc running inside. When creating the individual VM snapshot, we record the snapshot start time, VM suspend start time, VM suspend end time and snapshot completion time. The suspend end time equals snapshot completion time in VNSnap-disk snapshot method because VNSnap-disk suspends the VM at the end of snapshot, the TCP backoff duration between two VMs is from the first VM suspend start time to the last VM snapshot completion time. The snapshot start time is identical to the suspend start time in HotSnap because HotSnap suspends the VM at the start of snapshot, and the TCP backoff duration is from the suspend start time to the last suspend end time. Figure 6 shows the detailed results of 16 individual VM snapshot progresses in the VMC. The Stop-and-copy downtime last dozens or even hundreds of seconds, thus are not given in the figure.

We can see from Figure 6(a) that the duration from snapshot start to suspend end of VMs are almost identical and are minor, and the average suspend duration (downtime) is 116ms. The maximum TCP backoff duration between two VMs is 273ms. For VNSnap-disk snapshot method shown in Figure 6(b), although the snapshots start simultaneously, their suspend start time are various owing to iteratively saving the dirtied memory pages. The suspend duration are also different, ranges from tens of milliseconds to hundreds of milliseconds. The VM_9 which is the Distcc client even suffers from 2.03 seconds downtime, because it undertakes the heaviest task and generates large amount of memory during final transfer. VNSnap-disk brings 359ms VM downtime in average, only a little more than that of HotSnap. However, due to the discrepancy of snapshot completion times, the TCP backoff duration is much longer, e.g., the maximum value is 15.2 seconds between VM_1 and VM_9. This result suggests that the TCP backoff in VNSnap-disk snapshot method is much severe in the master/slave style distributed applications. The master always suffers from heavier workloads, costs longer time to finish the snapshot than the slaves, resulting in longer network interruption between master and slaves. However, the HotSnap method can effectively avoid this short-board affect because the downtime to create the *transient snapshot* is regardless of the workload, and lasts only tens of milliseconds.

The TCP backoff duration between two VMs is easy to acquire, but the backoff duration for the whole VMC

Metrics	**Duration(s)**			**Downtime(ms)**			**Snapshot Size(GBytes)**		
Benchmarks	**Stop-and-copy**	**VNSnap-disk**	**HotSnap**	**Stop-and-copy**	**VNSnap-disk**	**HotSnap**	**Stop-and-copy**	**VNSnap-disk**	**HotSnap**
Idle	50.64	51.66	51.57	50640	36.83	31.88	2.02	2.04	2.02
Compilation	52.50	61.11	51.96	52500	381.72	34.16	2.02	2.38	2.02
Matrix Multiplication	49.34	51.75	52.31	49340	55.73	35.93	2.02	2.19	2.02
Memcached	53.09	69.43	54.72	53090	150.85	33.80	2.02	2.41	2.02
Dbench	56.93	60.76	50.18	56930	79.36	39.36	2.02	2.17	2.02

Table 1: Comparison of individual VM snapshot techniques.

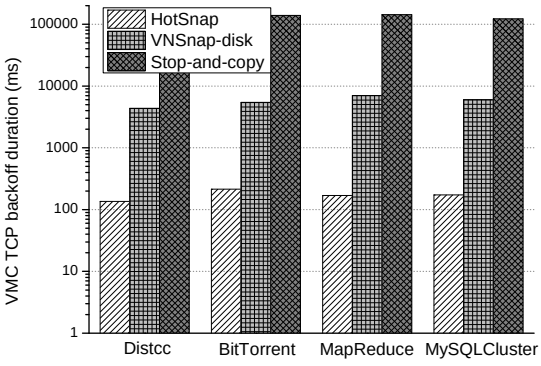


Figure 7: TCP backoff in VMC under various workloads.

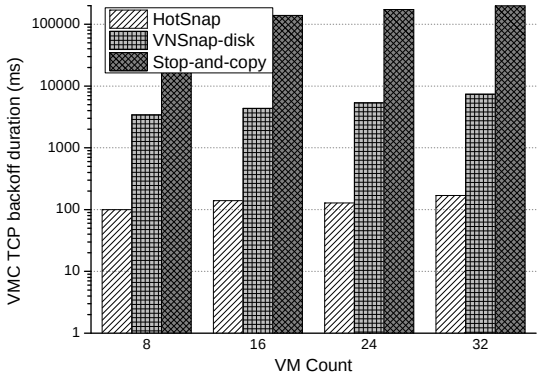


Figure 8: TCP backoff in VMC with different scales.

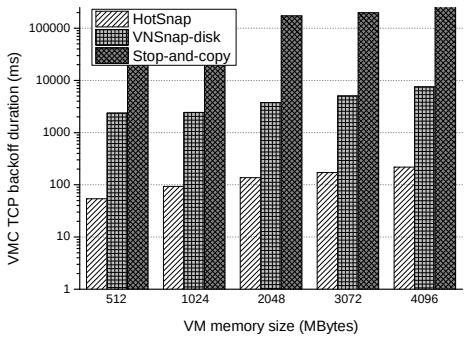
is hard to depict, we approximate the value by the average of TCP backoff durations between each two VMs in the VMC. Take Figure 6 as an example, the VMC TCP backoff duration are 137ms and 8.6s for HotSnap and VNSnap-disk methods respectively. In the following, we will compare the results of VMC TCP backoff duration under different VMC configurations in Figures 7-10. The VMC TCP backoff duration is log transformed on y-axis for clear comparisons:

VMC under various workloads. In this configuration, we build the VMC with 16 VMs, which are deployed evenly on four physical servers. The workloads are Distcc, BitTorrent, MapReduce and MySQL Cluster. For BitTorrent, we set one tracker in one VM, set two VMs as clients to download files from other VMs as seeds. We set up the Hadoop MapReduce [6] to count the key words in the short messages data set from China unicom, which contains over 600 million messages. Besides, we use the MySQL Cluster [9] to build a distributed database across the VMs, we configure one VM as management node, 13 VMs as database nodes, and exploit Sysbench [10] set up in two VMs to query the data in parallel. Figure 7 compares the VMC TCP backoff duration under different snapshot modes. As expected, the HotSnap distributed snapshot only suffers from about 100 milliseconds backoff duration under all the workloads, while the VNSnap-disk method incurs as many as 7 seconds in the MapReduce and MySQLCluster workloads, owing to the different snapshot completion times among VMs.

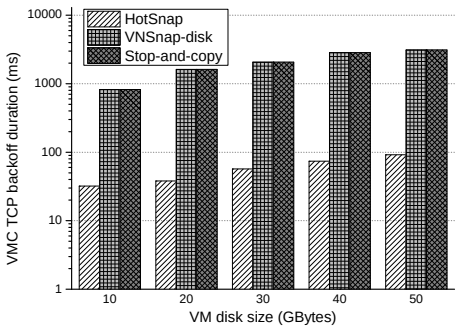
VMC of different scales. We set up the VMC with 8, 16, 24 and 32 VMs with Distcc running inside. Same as above, the VMs are deployed evenly on four physical servers. Figure 8 shows that the VMC TCP backoff duration in HotSnap method keeps almost constant, i.e., less than 200ms regardless of the number of VMs in the VMC. That's because the *transient snapshot* in HotSnap only save a few megabytes data, involves CPU state, bitmap files of disk and memory state. The VMC TCP backoff duration in VNSnap-disk and Stop-and-copy rises with the increase of VM number; the reason is the parallel execution of writing more large files.

VMC with increased VM memory and disk size. We configure the VMC with 16 VMs under the Distcc workload. Figure 9 compares the VMC TCP backoff duration while increasing the VM memory size and disk size. The VMC TCP backoff duration in all the three modes increase while increasing the VM memory size and disk size. The increase in HotSnap method is because HotSnap need to reserve larger bitmap files for larger disk and memory size, and more operations to set write protect flags to memory pages. The increase in Figure 9(a) in VNSnap-disk may come from two aspects: the first is the parallel execution of writing larger files, which is also the reason for the increase in the Stop-and-copy method; the second is owing to longer time to save more memory pages, which will further generate more dirtied pages during iteration time. Because the disk snapshot is created in VM suspend phase, and different disk size affects the VM downtime, so we show the downtime instead of VMC TCP backoff duration in Figure 9(b). As estimated, the redirect-on-write based snapshot method cost only tens of milliseconds, achieves the reduction by more than 20x compared to the other two modes. Besides, the VNSnap-disk downtime reaches several seconds, and is proportional to the VM disk size, because most of the downtime is consumed to index the disk blocks in the multi-level tree structure. The Stop-and-copy takes the same method to create disk snapshot, thus incurs the same downtime to VNSnap-disk.

VMC mixed of VMs with different memory size. In this VMC configuration, we set up the VMC with two



(a) TCP backoff in VMC with increasing VM memory size



(b) Downtime in VMC with increasing VM disk size. The disk is filled with data

Figure 9: TCP backoff in VMC with different VM configurations.

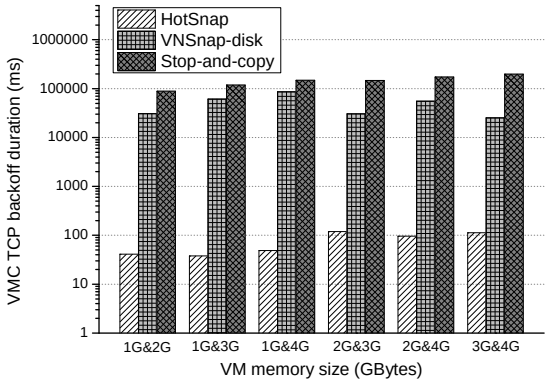


Figure 10: TCP backoff in VMC mixed of VMs with different memory size.

VMs of different memory size. As illustrated in Figure 10, compared to the HotSnap TCP backoff duration that keeps almost constant regardless of the discrepancy of VM memory size, the TCP backoff duration of VNSnap-disk increases proportionally to the raised discrepancy between memory size, i.e., every 1G memory difference will incur additional 27 seconds backoff duration. As expected, the VMC which consists of 1G memory VM and 4G memory VM obtains the largest backoff duration. The Stop-and-copy method, on the other hand, increases with the increasing memory size of VMs, this is easily understand because more time will be consumed to write larger files into the snapshot file.

5.4 Performance Impact on Individual VM

In the above experiments, we save the whole zero page instead of only one byte to avoid the impacts of zero pages. However, saving multiple large snapshot files simultaneously during distributed snapshot will degrade performance seriously, and even cause write operation timeout for I/O intensive applications. We consider the optimization as our future work, but in this paper, we simply resume the zero page compression mode to reduce the saved page count during snapshot.

As stated in the previous section, during the *full snapshot* step, we trap the write page fault in KVM and turn to QEMU to handle the write fault, so that the memory operations of Guest OS and user applications are affected. Besides, we intercept the DMA write operations, which may affect the guest I/O speed. So we first give the statistic on the four memory page saving manners during HotSnap, and then evaluate the performance on applications.

Page count of different manners in HotSnap. HotSnap saves the memory pages in four manners: Guest Write Handler, DMA Write Handler, KVM Write Handler and Background Copy. The saved page number of these four types under various workloads are listed in Table 2. The Guest Write pages always account more than that of DMA Write and KVM Write, but the amount is still minor even under memory intensive workloads, e.g., 2.5% of all memory pages under Memcached. This is possible because HotSnap saves dozens of neighbouring pages when handling one page fault and thus it benefits from the memory locality feature. Handling one Guest Write page cost about 60us, including the time to trap write page fault, exit to QEMU to save memory pages, remove write protect flag and resume the execution. As a result, the total cost incurred by saving

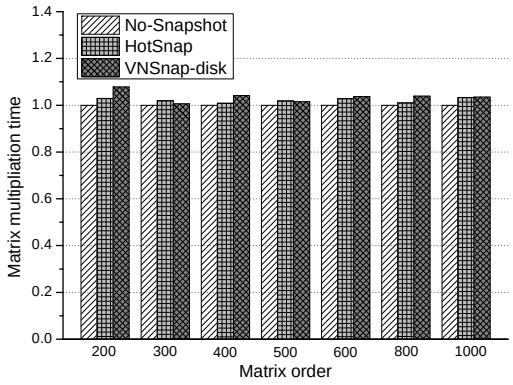


Figure 11: Matrix multiplication calculation time.

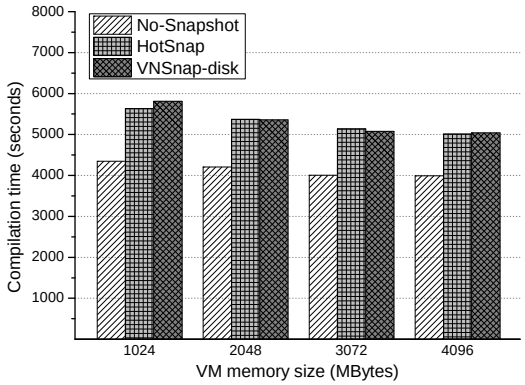


Figure 12: Kernel compilation time.

page in Guest Write is affordable. DMA Write operations and KVM Write operations always repeatedly access the same memory pages, therefore the count is minor. Besides, the interception on DMA write and KVM write lasts about 12us, making the impact can be negligible.

Workloads	**Background Copy**	**Guest Write**	**DMA Write**	**KVM Write**
Idle	528301	133	44	2
Compilation	524506	3912	59	3
Matrix Multiplication	520496	7916	65	3
Memcached	514814	13270	394	2
Dbench	526831	987	660	2

Table 2: Count of four page types during HotSnap.

Matrix multiplication time. Matrix Multiplication involves large amount of memory and CPU operations. We calculate the multiplication of two matrices while increasing the matrix order, and obtain the calculation time during No-Snapshot (i.e., normal execution), HotSnap and VNSnap-disk snapshot. Figure 11 compares the results of HotSnap and VNSnap-disk to the completion time of No-Snapshot as baseline. Both the two live snapshot methods bring less than 5% additional time to finish the computation, implying no obvious performance penalty during distributed snapshot for this kind of workload.

Kernel compilation time. Kernel compilation involves both memory operations and disk IO operations, we compile Linux-2.6.32.5 kernel with all modules during continuous VM snapshot. Figure 12 compares the compilation duration in No-Snapshot, HotSnap and VNSnap-disk modes. The time during HotSnap and VNSnap-disk distributed snapshot are almost equal, and both consume about 20% more time to compile the kernel. The increase is owing to the CPU utilization and I/O bandwidth consumed by the VM snapshot. The 20% reduction maybe unacceptable in many performance-critical systems, and we leave the performance optimization as our future work.

5.5 Performance Impact on VMC

We first build the VMC with different number of virtual machines, and run Distcc to compare the completion time of normal execution, HotSnap distributed snapshot and VNSnap-disk snapshot. Then we set the VMC with 16 virtual machines running on four physical servers evenly, and install BitTorrent to evaluate the download speed during distributed snapshot.

Distcc compilation time. Distcc client distributes the compilation task to servers, if it loses connection with one server, the client will do the task in local; and the client can continue to distribute the task once the server is connected again. Figure 13 depicts the compilation time during continuous distributed snapshot while increasing the number of VMs in the VMC. Compared to the No-Snapshot mode, the compilation duration during HotSnap distributed snapshot increases by about 20%. The increase are mainly due to the snapshot overhead such as I/O operations and CPU utilization, the similar results are also illustrated in Figure 12. The duration during VNSnap-disk is much longer, it cost about 7% to 10% more time to finish compilation than that of HotSnap. Obviously, this is due to the TCP backoff which incurs network interruption between client and servers.

BitTorrent download speed. We set up the BitTorrent to evaluate the network performance loss incurred during distributed snapshot. We build the tracker on

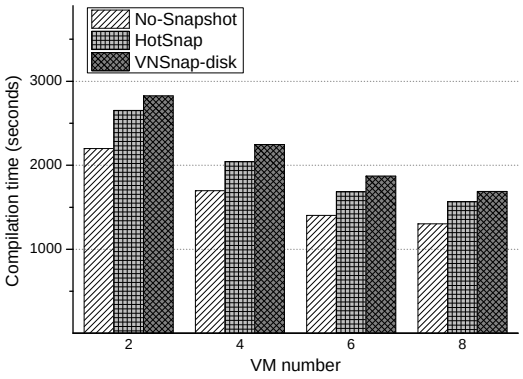


Figure 13: Distcc compilation time.

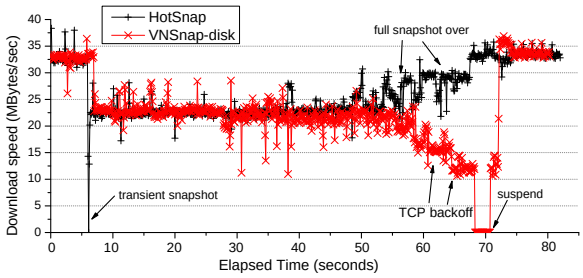


Figure 14: BitTorrent download speed.

one VM, treat one VM as the client to download a large video file from other VMs as seeds, and record the download speed every 100 milliseconds. Note that the practical download speed varies significantly even in adjacent epoches, we average the speed by twenty samples. Figure 14 compares the download speed between normal execution and distributed snapshot. The download speed reduces from 33.2MBytes/sec during normal running to 22.7MBytes/sec when taking snapshot. The difference of impact between HotSnap and VNSnap-disk is also illustrated in the figure. HotSnap incurs a sharp decrease at about the 6th seconds, which is actually the distributed snapshot downtime to create the *transient snapshot*. Then the download speed will reach about 22.7MBytes/sec and keep the speed until the 55th second. From this time point, many VMs finish the *full snapshot* and resume to normal execution, so that the download speed will increase and finally reach the normal download speed, i.e., about 33.2MBytes/sec. The download speed during VNSnap-disk distributed snapshot shows opposite result from the 55th second, it decreases and reaches 0MBytes/sec at the 68th second. The reason is that the BitTorrent client experiences about 65 seconds to finish the snapshot, and will not receive the packets from seed VMs that finish the snapshot ahead, therefore decrease the download speed. After the client resumes the execution from the 71st second, the download speed also returns to normal.

6 Related Work

Distributed snapshot has been widely studied in the past thirty years, and many techniques have been proposed to create snapshot for distributed systems. The earlier works mainly focus on designing the snapshot protocol between the peers to maintain a global consistent state. Chandy and Lamport [13] assume the message channel is reliable and FIFO, and propose the coordinated protocol. Lai [21] proposes the message coloring based protocol to achieve consistency for non-FIFO channel. Kim [19] blocks the peer running until all the distributed snapshot finishes to reserve the consistent state. In contrast to optimize the snapshot protocol, we implement the coordinated, non-blocking protocol in the high bandwidth and low latency local area network, the protocol is simple and suitable for taking the distributed snapshot in the virtualized environments.

Another key aspect of distributed snapshot is the snapshot technology. Copy-on-write, redirect-on-write, split mirror are common methods to save the system state, and are implemented on kernel level [11], library level [24, 17, 25] or application level [3, 16] to create the snapshot. However, these methods require modification of the OS kernel or applications, thus is unacceptable in many scenarios.

The virtualization technology encapsulates the whole application as well as the necessary resources; it contributes to create the snapshot in an transparent manner. Emulab [12] leverages live migration technique to enable taking snapshots of the virtual machine network. By synchronizing clocks across the network, Emulab suspends all nodes for snapshot near simultaneously, and implements a coordinated, distributed snapshot. The synchronization will block the execution of the VMs, thus interfere with the applications running in the VMs. Besides, Emulab requires modifications to the Guest OS, and is hard to support legacy and commodity OS. VNSnap [18] also leverages Xen live migration [14] function to minimize system downtime when taking VMN snapshots. Unlike Emulab, VNSnap employs non-blocking coordination protocols without blocking VMs, and VNSnap requires no modification to the guest OS. Our HotSnap proposal shares a similar manner to VNSnap, but we design a *transient snapshot* manner to suspend the VM first and then create the *full snapshot*, which reduces the discrepancy of snapshot completion time. Besides, we log and resend the packets send from post-snapshot VM to pre-

snapshot VM instead of dropping them which is adopted by VNSnap. Both these two technologies achieve the notable reduction in the TCP backoff duration. Moreover, we treat one VM as the initiator to avoid setting up a customized module, which makes HotSnap to be easily portable to other virtualized environments.

7 Conclusions

This paper presents a distributed snapshot system HotSnap, which enables taking *hot* snapshot of virtual machine cluster without blocking the normal execution of VMs. To mitigate TCP backoff problem and minimize packets loss during snapshots, we propose a transient VM snapshot approach capable of taking individual VM snapshot almost instantaneously, which greatly reduces the discrepancy of snapshot completion times. We have implemented HotSnap on QEMU/KVM platform, and conduct several experiments. The experimental results illustrate the TCP backoff duration during HotSnap distributed snapshot is minor and almost constant regardless of the workloads, VM memory size and different VM-C configurations, thus demonstrate the effectiveness and efficiency of HotSnap.

There still exists several limitations in HotSnap. First, the newer QEMU version supports more DMA simulators such as sound card and video card, implementing DMA write interceptions for each simulated device are fussy. Second, creating distributed snapshot involves large amount of I/O operations, thus affect the I/O intensive applications running insides the VMs. Therefore, our ongoing works include designing an abstract layer to intercept DMA write operations, scheduling the I/O operations from Guest OS and HotSnap for applications' performance requirements. We also plan to evaluate HotSnap under real-world applications.

Acknowledgements

We thank Hanqing Liu, Min Li for their contributable work on disk snapshot. We thank our shepherd, Marco Nicosia, and the anonymous reviewers for their valuable comments and help in improving this paper. This work is supported by the National Grand Fundamental Research 973 Program of China under Grant No. 2011CB302602, National High Technology Research 863 Program of China under Grant No. 2011AA01A202, National Nature Science Foundation of China under Grant No. 61272165, No. 60903149, No. 91118008, and No. 61170294.

References

[1] Amazon elastic compute cloud (amazon ec2). `Http://aws.amazon.com/ec2/`.

[2] Bittorrent. `http://www.bittorrent.com/`.

[3] Ckpt library. `http://pages.cs.wisc.edu/~zandy/ckpt/`.

[4] Dbench. `http://dbench.samba.org/`.

[5] Distcc. `http://code.google.com/p/distcc/`.

[6] Mapreduce. `http://hadoop.apache.org`.

[7] Mcblaster. `https://github.com/fbmarc/facebook-memcached-old/tree/master/test/mcblaster`.

[8] Memcached. `http://memcached.org/`.

[9] Mysql cluster. `http://dev.mysql.com/downloads/cluster/`.

[10] Sysbench. `http://sysbench.sourceforge.net/docs/`.

[11] A. Borg, W. Blau, W. Graetsch, F. Herrmann, and W. Oberle. A survey of rollback-recovery protocols in message-passing systems. *ACM Transations Computer Systems*, 7(1):1–24, 1989.

[12] A. Burtsev, P. Radhakrishnan, M. Hibler, and J. Lepreau. Transparent checkpoints of closed distributed systems in emulab. In *Proceedings of EuroSys*, pages 173–186, 2009.

[13] K. M. Chandy and L. Lamport. Distributed snapshots: Determining global states of distributed systems. *ACM Trans. Comput. Syst*, 3(1):63–75, 1985.

[14] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proceedings of NSDI*, pages 273–286, 2005.

[15] E. N. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3): 375–408, 2002.

[16] A. Goldberg, A. Gopal, K. Li, R. Strom, and D. F. Bacon. Transparent recovery of mach applications. In *Proceedings of Usenix Mach Workshop*, pages 169–184, 1990.

[17] J. Hursey, J. M. Squyres, T. Mattox, and A. Lumsdaine. The design and implementation of checkpoint/restart process fault tolerance for open mpi. In *Proceedings of IPDPS*, pages 1–8, 2007.

[18] A. Kangarlou, P. Eugster, and D. Xu. Vnsnap: Taking snapshots of virtual networked environments with minimmal downtime. In *Proceedings of DSN*, pages 534–533, 2009.

[19] J. Kim and T. Park. An efficient protocol for checkpointing recovery in distributed systems. *IEEE Transactions on Parallel and Distributed Systems*, 4(8):955–960, 1993.

[20] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. Kvm: the linux virtual machine monitor. *Computer and Information Science*, 1:225–230, 2007.

[21] T. H. Lai and T. H. Yang. On distributed snapshots. *Information Processing Letters*, 25(3):153–158, 1987.

[22] J. Li, B. Li, T. Wo, C. Hu, J. Huai, L. Liu, and K. P. Lam. Cyberguarder: A virtualization security assurance architecture for green cloud computing. *Future Generation Computer Systems*, 28:379–390, 2.

[23] J. Li, H. Liu, L. Cui, B. Li, and T. Wo. irow: An efficient live snapshot system for virtual machine disk. In *Proceedings of ICPADS*, pages 376–383, 2012.

[24] C. Ma, Z. Huo, J. Cai, and D. Meng. Dcr: A fully transparent checkpoint/restart framework for distributed systems. In *Proceedings of CLUSTER*, pages 1–10, 2009.

[25] J. F. Ruscio, M. A. Heffner, and S. Varadarajan. Dejavu: Transparent user-level checkpointing, migration, and recovery for distributed systems. In *Proceedings of IPDPS*, pages 1–10, 2007.

[26] X. Song, J. Shi, R. Liu, J. Yang, and H. Chen. Parallelizing live migration of virtual machines. In *Proceedings of VEE*, pages 85–96, 2013.

[27] R. Strom and S. Yemini. Optimistic recovery in distributed systems. *ACM Transations on Computer Systems*, 3 (3):204–226, 1985.

A Global Consistent State

The virtual machine cluster is a message-passing system; therefore the global state of VMC to be saved consists of the individual states of single VMs and the states of the communication channels. The global consistent state requires: 1) if the state of a process reflects a message receipt, then the state of the corresponding sender much reflect sending this message; 2) A packet can be in the sender, or flying in the channel, or is accepted by the receiver, but cannot exist in two at the same time. Figure 15 illustrates a consistent and inconsistent state [15]. Figure 15(a) is consistent even the P_1 do not receive m_1, because m_1 has been send from P_0, and is travelling in the channel in this case. On the other hand, Figure 15(b) describes an inconsistent state, this is because m_2 received by P_2 has not been send from P_1 in this snapshot state. In such a case, P_1 will resend m_2 after rollback to the inconsistent snapshot state, thus result in fault state of P_2. As a result, this kind of messages that send from post-snapshot process to pre-snapshot process are always dropped, or logged and then resend after snapshot is over.

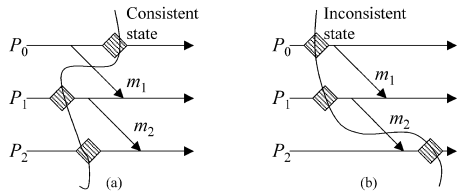


Figure 15: An example of consistent and inconsistent state. p stands for independent process, and m represents the message.