CloudVS: Enabling Version Control for Virtual Machines in an Open-Source Cloud under Commodity Settings

Chung Pan Tang, Tsz Yeung Wong, Patrick P. C. Lee
Department of Computer Science and Engineering, The Chinese University of Hong Kong, Hong Kong
{tangcp, tywong, pclee}@cse.cuhk.edu.hk

***Abstract*—Open-source cloud platforms provide a feasible alternative of deploying cloud computing in low-cost commodity hardware and operating systems. To enhance the reliability of an open-source cloud, we propose CloudVS, an add-on system that enables version control for virtual machines (VMs). CloudVS targets a commodity cloud platform that has limited available resources. It exploits content similarities across different VM versions using redundancy elimination (RE), such that only non-redundant data chunks of a VM version are transmitted over the network and kept in persistent storage. Using RE as a building block, we propose a suite of performance adaptation mechanisms that make CloudVS amenable to different commodity settings. Specifically, we propose a tunable mechanism to balance the storage and disk seek overheads, as well as various I/O optimization techniques to minimize the interferences to other co-resident processes. Using a 3-month span of real VM snapshots, we experiment CloudVS in an open-source cloud testbed built on Eucalyptus. We demonstrate how CloudVS leverages RE to save the storage cost and the VM operation time than simply keeping full VM images. More importantly, we show how CloudVS can be parameterized to balance the performance trade-offs between version control and normal VM operations.**

I. Introduction

With the advent of cloud computing, people can pay for computing resources from commercial cloud service providers in a pay-as-you-go manner [1]. Open-source cloud platforms, such as Eucalyptus [11] and OpenStack [12], provide an alternative of using cloud computing with the features of *self-manageability*, *low deployment cost*, and *extensibility*. Using open-source cloud software, one can deploy an in-house private cloud, while preserving the inherent features of existing public commercial clouds such as virtualization and resource management. In addition, an open-source cloud is deployable in low-cost commodity hardware and operating systems that are readily available to general users. Its open-source nature also provides flexibility for developers to extend the cloud implementation with new capabilities.

To deploy an open-source cloud (as a private cloud) in practice, a major challenge is to ensure its *reliability* toward software/hardware failures, especially with the fact that the cloud infrastructure is now self-managed. Here, we propose to *enable version control for virtual machines (VMs)*, in which we take different snapshots of individual VM images launched within the cloud and keep different VM versions. Applying version control for VMs enables users to save work-in-progress jobs in persistent storage. From a reliability perspective, one can roll-back to the latest VM version due to software/hardware crashes, and perform forensic analysis in the past VM versions should malicious attacks happen.

However, there are several design challenges of enabling version control for VMs in an open-source cloud platform:

- *Scalability to many VM versions.* We need to store and maintain a large volume of VM versions within a cloud, given that the cloud may handle the VMs of many users, and each VM may create many VM versions over time.
- *Limited network bandwidth.* There could be a huge network transmission overhead of transmitting VM snapshot versions from different compute nodes in the cloud to the repository that stores the snapshots.
- *Compatibility with commodity settings.* The version control mechanism must be compatible with the cloud infrastructure, such that the performance of the normal cloud operations is preserved. Since an open-source cloud is deployable in commodity hardware and operating systems, we require that the version control mechanism be able to take into consideration the performance constraints of storage, computation, and transmission bandwidth.

In this paper, we propose *CloudVS*, a practical version control system for storing and managing different versions of VMs designed for an open-source cloud deployed under commodity settings. CloudVS incrementally builds different VM snapshot versions using *redundancy elimination (RE)*, such that only the new and modified chunks of the current VM image are transmitted over the network and stored in the backend. A particular VM version can be constructed from prior VM versions. We demonstrate how RE can be used to minimize the overheads of transmission and storage in the version control for VMs in an open-source cloud platform.

On top of RE, we propose a suite of adaptation mechanisms that make CloudVS amenable to different commodity settings. One major challenge of using RE is that it introduces fragmentation, i.e., the content of a VM image is scattered in different VM versions. Fragmentation increases the disk seek overhead. Thus, we propose a simple *tunable mechanism* that can trade between storage and fragmentation overheads via a single parameter. Also, we propose various *I/O optimization* techniques to mitigate the performance interferences to other co-resident processes (e.g., VM instances) during the version-

This research is supported by project #MMT-p1-10 of the Shun Hing Institute of Advanced Engineering, The Chinese University of Hong Kong. C. Tang and P. Lee are affiliated with SHIAE in this research.

978-1-4673-0269-2/12/$31.00

ing process. In short, CloudVS exploits different performance adaptation strategies to easily make performance trade-offs between version control and normal VM operations.

We implement CloudVS and integrate it into Eucalyptus [11] as an add-on system. The current open-source implementation of Eucalyptus (whose version is 2.0.3 at the time this paper being written) does not provide version control for VMs, so all changes made to a VM will be lost if the VM is shut down. As a proof of concept, we show how CloudVS remedies this limitation with minor modifications of the Eucalyptus source code, such that the original semantics of Eucalyptus are completely preserved.

We conduct extensive experiments for CloudVS on a Eucalyptus-based cloud testbed. We evaluate CloudVS using a 3-month span of snapshots of a regularly updated VM. We show how CloudVS uses RE to reduce the storage cost and VM operation times when compared to simply keeping VM versions with full VM images. Also, we show that CloudVS can be parameterized to address different performance trade-offs and limit the interferences to co-resident processes.

The remainder of the paper proceeds as follows. In Section II, we overview the cloud architecture considered in this paper. In Section III, we explain the design of CloudVS and propose several practical optimization techniques. In Section IV, we experiment CloudVS in a cloud testbed built on Eucalyptus. In Section V, we review related work, and finally, Section VI concludes.

II. Background

Figure 1 shows a simplified cloud architecture that we consider in this paper. It consists of three types of nodes: (i) the *controller node*, which processes VM-related requests from users and manages the lifecycles of VM instances, (ii) the *compute node*, which runs VM instances, and (iii) the *storage node*, which provides an interface that accesses the VM images in the persistent storage backend. Note that existing open-source cloud platforms such as Eucalyptus [11] and OpenStack [12] are designed based on the same layout as in Figure 1. To aid our discussion, in this paper, we mainly focus on a simplified cloud platform that has only one controller node, one storage node, and multiple compute nodes.

To launch a VM instance, a user first issues a start request through the controller node, which selects an available compute node on which the VM instance runs. The selected compute node then retrieves the corresponding VM image from the storage node. It also allocates local disk space for running the VM instance. Note that the compute node can cache the image in the local disk for subsequent use, and this feature is supported in current open-source cloud platforms.

Similarly, the user can issue a stop request to the controller node to stop the VM instance. The controller node then instructs the compute node to destroy the VM instance and recycle the resources.

We examine how the current implementations of Eucalyptus (version 2.0.3) and OpenStack (version 2011.2) handle the lifecycle of a VM instance. In Eucalyptus, when a VM instance is stopped, all modifications made to the VM instance will be permanently purged. OpenStack provides a snapshot feature that supports regular backups of VM images [13]. However, it only sends the full snapshot of the entire VM image to the storage backend, and this introduces a large transmission overhead. We believe that an efficient version control mechanism will be a desirable add-on feature for existing open-source cloud platforms.

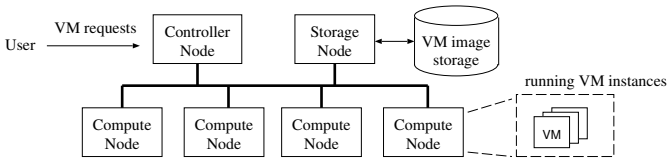


Fig. 1. A simplified cloud architecture considered in this paper.

III. CloudVS Design

In this section, we present the design of *CloudVS*, a practical system that enables version control for VMs in mainstream open-source cloud platforms. We focus on the scenario where the cloud platform is deployed atop low-cost commodity hardware and operating systems that have limited available resources. Thus, the design of CloudVS aims to mitigate the overheads in storage, computation, and transmission.

A. Overview

The original cloud platform (e.g., Eucalyptus and OpenStack) only launches a VM instance from a VM image that contains only basic configurations without user-specific states. This VM image, which we call the *base image*, is accessible by all users. With CloudVS, we keep different versions of a user-modified VM for each user. Instead of storing the full image of the user-modified VM, CloudVS incrementally builds each VM version from the prior versions, such that the current VM version only keeps the *delta*, defined as the new or changed content of a VM image. For the unchanged content, the current VM version keeps references that refer to the prior versions. The delta will be stored in the persistent storage managed by the storage node (see Figure 1). Here, we consider two types of a delta: (i) the *incremental delta*, which holds only the new or modified content since the last version, and (ii) the *differential delta*, which holds all the new and modified content with respect to the base image. Keeping either incremental or differential deltas for different VM versions minimizes the redundant content being stored. We call this approach *redundancy elimination (RE)*. We elaborate how we construct deltas for different VM versions in Section III-B.

We can easily see that the incremental delta stores the minimum redundant content, with the trade-off that a VM version needs to be restored by accessing the content of multiple prior versions. This introduces *fragmentation*, meaning that the content of a VM is scattered in different VM versions instead of being stored sequentially. It results in more disk seeks, thereby increasing the restore time of a VM. Fragmentation is known to be a fundamental problem in RE-based storage

systems [19]. On the other hand, the differential delta mitigates the fragmentation problem since it can be directly merged with the base image to have the VM image reconstructed. However, this requires the storage of the redundant content that appears in the prior VM versions. In CloudVS, we balance the trade-off of storage and fragmentation via a single tunable parameter. See Section III-C for details.

During versioning, CloudVS needs to scan the entire VM image for hash computation. This involves substantial disk processing and disrupts other co-resident processes in the same compute node. To minimize the interference, we propose several I/O optimization techniques, as detailed in Section III-D.

The CloudVS implementation is a fork of the original execution flow of existing cloud platforms, so CloudVS can be freely enabled or disabled without interfering in the original execution logic. In Section III-E, we illustrate how CloudVS can be integrated into Eucalyptus as a proof of concept.

B. Versioning with Redundancy Elimination

CloudVS creates the delta for a VM version based on *redundancy elimination (RE)*, which aims to minimize the network transfer bandwidth and the storage overhead. Similar RE-based versioning approaches have been proposed, such as in cloud backup systems [22], [17], which target the storage of general data types. On the other hand, since we only apply RE for VM images, we can use a more lightweight RE algorithm. Here, we show how RE is used as a building block in CloudVS. In later subsections, we will further optimize our RE approach.

For simplicity and efficiency, we choose *fixed-size chunking* as our RE algorithm, whose main idea is to divide a VM image into fixed-size chunks (e.g., of size 4KB) and only keep the new and modified chunks in the current VM version. Note that the RE approach used in CloudVS can also be implemented with more robust RE algorithms (e.g., rsync [20] and Rabin fingerprinting [16]), but fixed-size chunking has been shown to be effective in RE for VM images [5].

We now elaborate how CloudVS uses RE to perform versioning for VMs in detail. We apply cryptographic hashing (e.g., SHA-1) to the content of each fixed-size chunk, such that two chunks with the same hash are considered to have identical content. It is shown that if cryptographic hashing is used, then the probability of having hash collisions is negligible in practice [15].

There are two scenarios where CloudVS can trigger the versioning process: (i) *shutdown-based*, in which CloudVS creates a new VM version when the VM is about to shut down and release its resources, and (ii) *time-based*, in which CloudVS performs periodic versioning on a running VM. In both scenarios, we need to first identify the hashes of the VM image of the last version so as to compute the delta. In shutdown-based versioning, since the last version is created in the last VM shutdown, CloudVS generates hashes for the last version when the VM is launched again in a newly assigned compute node. On the other hand, in time-based versioning, CloudVS generates hashes each time when the VM version

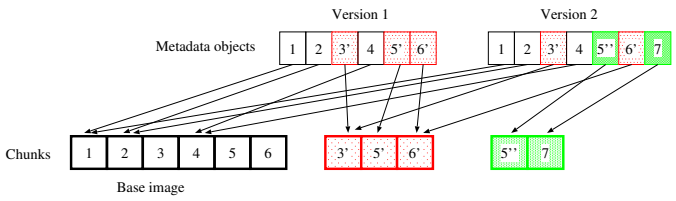


Fig. 2. Example of how CloudVS uses RE to correlate different VM versions. The compute node will send the incremental delta for each version to the storage node for persistent storage.

is created. In both cases, the hashes of the last version will be cached in the compute node where the VM is currently running, and later compared with the current VM version.

To create the current VM version, we compare by hashes the different chunks of the VM images of the last and current versions. The comparison is done in the compute node that runs the VM. We generate the incremental delta, which contains the new and modified chunks since the last version. The incremental delta, together with the references that refer to the already existing chunks in prior versions, will be sent to the storage node and stored in the persistent storage backend.

Figure 2 illustrates how different VM versions are correlated using RE. Each VM version has a metadata object that keeps the references for all the chunks that appear in the current or prior VM versions. To illustrate, suppose that the original base image contains six chunks (see Figure 2). In Version 1, if the 3rd, 5th, and 6th chunks have been modified, then Version 1 will allocate space for holding such modified chunks, while keeping references that point to the 1st, 2nd, and 4th chunks in the base image. Now, in Version 2, if the 5th chunk is modified again and the 7th chunk is created, then Version 2 will allocate space for holding the 5th and 7th chunks and have the references to refer to other chunks appearing in the base image or Version 1. In general, the metadata object and the new/modified chunks of each VM version altogether have a much smaller size than the full VM image, thereby minimizing the overhead of maintaining various VM versions.

To restore a particular VM version, the storage node looks up the metadata object and fetches the corresponding chunks. Suppose that the base image is already cached in the compute node (see Section II). Then the storage node will construct the differential delta (i.e., the new and modified chunks with respect to the base image) for the VM version and send it to the compute node, which will then merge it with the cached base image. This introduces less transmission overhead than sending the full VM image. To illustrate, suppose that Version 2 in Figure 2 is to be restored. Then the storage node will transmit only the 3rd, 5th, 6th, and 7th chunks.

Typically, a VM image contains many system files of the guest operating system that rarely change. Thus, we expect that RE can effectively reduce the storage of such redundant content. We validate this in Section IV.

C. Tunable Delta Storage

If the storage node directly stores the incremental delta (i.e., the new and modified chunks since the last version) for each

version, then the fragmentation overhead may exist during the restore of a VM version. Referring to Figure 2 again, suppose that Version 2 is to be restored. In this case, the storage node needs to retrieve the 3rd, 5th, 6th, and 7th chunks. If these chunks are returned in a sequential order, then the storage node needs to access Version 1 and Version 2 alternately. Let us define a *non-sequential read* if the next chunk to be read appears in a different version from the current chunk being read. Then in the above example, we have a total of three non-sequential reads (i.e., for the 5th, 6th, and 7th chunks).

In another extreme, the storage node can simply store the differential delta (i.e., all the new and modified chunks with respect to the base image). For example, Version 2 may store the 3rd, 5th, 6th, and 7th chunks. Then all reads become sequential, but this introduces a high storage overhead.

We emphasize that the versioning process always transmits incremental deltas from a compute node to the storage node, so as to minimize the transmission overhead. However, we must address how the storage node should store the deltas for different versions that can balance the costs of storage and fragmentation.

Here, we consider one heuristic design that uses a single *fragmentation parameter* α to trade between the fragmentation and storage overhead by exploring the intermediates between the extremes of storing incremental and differential deltas. The design is composed of four steps.

1) We divide all chunks in the differential delta into chunk groups, each of which has the same number of chunks (except the last chunk group).
2) For each chunk group, we count the number of non-sequential reads as defined above.
3) We sort the chunk groups by the number of non-sequential reads in descending order.
4) The top proportion α ($0 \leq \alpha \leq 1$) of the chunk groups will store the differential deltas, while the remaining chunk groups will store the incremental deltas.

The parameter α is tunable according to different application needs. In the extremes, if $\alpha = 1$, then each version stores the differential delta; if $\alpha = 0$, then each version stores only the incremental delta. We observe that even with this simple heuristic, we can effectively make the trade-off (see Section IV).

To illustrate, we consider again how to restore Version 2 in Figure 2. Suppose that we set the chunk group size to be two and $\alpha = 0.5$. In Version 2, the differential delta consists of the 3rd, 5th, 6th, and 7th chunks. Thus, there are two chunk groups, i.e., the 3rd and 5th chunks, as well as the 6th and 7th chunks. Both chunk groups have one non-sequential read. If $\alpha = 0.5$, then we have the first chunk group store the differential delta, while the second chunk group still stores the incremental delta. Figure 3 shows the final result.

D. I/O Optimization

When creating a version for a VM, CloudVS needs to scan the entire VM image for hash computation in the compute node. The scanning process may degrade the performance of normal operations of the scanned VM as well as other co-resident processes in the same compute node. We propose several I/O optimization techniques that minimize the performance interferences due to versioning.

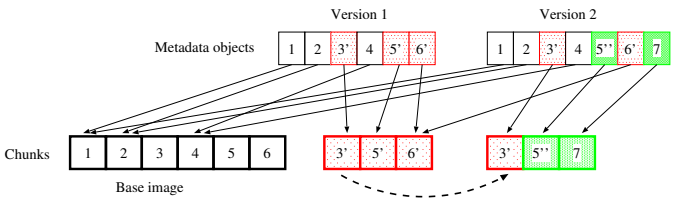


Fig. 3. Example of how CloudVS stores the incremental and differential deltas in different chunk groups. Here, the 3rd and 5th chunks are stored in Version 2 as the differential delta for the first chunk group (assuming α = 0.5).

LVM Snapshot. CloudVS needs to avoid content changes during versioning so that the hashes are correctly computed. One simple approach is to apply an exclusive lock to the entire VM image, but this also makes the VM unavailable (or "frozen") in that period. Here, we have CloudVS work on a mirror snapshot by leveraging the file system snapshot feature of the *logical volume manager (LVM)* [7]. Each compute node hosts VM images with the LVM. To create a version for a VM, we then apply the snapshot feature of LVM to create a snapshot of the VM image volume. Once a snapshot is created, the VM returns to its normal operations, while in the background, CloudVS computes the hashes on the created snapshot rather than on the VM. The snapshot will be destroyed when the versioning process is finished. With LVM snapshot, we now only lock the VM image during the snapshot creation, instead of locking the VM image throughout the versioning period.

Pre-declaring access patterns. When CloudVS scans the entire VM image, the image data will be read from disk and saved in the system cache, thereby flushing the existing cached data. Since the image data is read once only, we should avoid disrupting the accesses to existing cached data for other co-resident processes. Here, we pre-declare the access patterns of the VM image using the POSIX system call `posix_fadvise`. After computing the hash of a specific data chunk, we invoke `posix_fadvise` with the parameter `POSIX_FADV_DONTNEED` to notify the kernel that the data chunk will no longer be accessed in the near future. This keeps the kernel from caching the entire VM image during versioning.

Rate limiting of disk reads. The scanning of a VM image can invoke a large burst of disk reads that will disturb other co-resident processes that share the same physical disk. CloudVS implements a rate throttling mechanism that limits the rate of disk read accesses. This mechanism is coupled with the LVM snapshot function (see above), i.e., after a snapshot is created, we monitor the read throughput of scanning the snapshot. If the read throughput is higher than the specified rate, we invoke the POSIX call `nanosleep` to put the read operation on hold. The throttling rate can be parameterized in advance. Although this increases the versioning time, since the versioning process is done on the mirror snapshot in the background, the extended

versioning time has minimal impact.

E. Implementation Details

We implement a prototype of CloudVS in C. We integrate it into a cloud platform based on the Eucalyptus open-source edition 2.0. As shown below, the integration only involves slight modifications in the source code.

Deployment in the controller node. A user can specify a specific VM version by providing `versionID` as an input, where `versionID` is a global identifier that uniquely identifies different VM versions. To launch a VM, the user needs to first prepare his legitimate access key and secret key, both of which are required by the original Eucalyptus implementation. The user may store the keys as environment variables. Then the user can issue the command `euca-run-instance --user-data versionID` to start the specific VM version, where the command `euca-run-instances` comes with the command-line management tool `euca2tools` of Eucalyptus.

Deployment in each compute node. CloudVS is composed of two modules: the *Snapshot* and *Restore* modules, which are responsible for generating deltas for different VM versions and restoring a VM version, respectively. We integrate both modules into the operations of each compute node. In our current prototype, we mainly consider shutdown-based versioning (see Section III-B). We insert the Snapshot and Restore modules right before the VM is started and after the VM is shut down, respectively. We add both modules in `~eucalyptus/storage/storage.c`. The integration involves no more than 20 lines of code changes.

Deployment in the storage node. We add a new daemon in the storage backend that listens to the requests from the Snapshot and Restore modules in the compute nodes, and retrieves and saves the specified version, respectively.

IV. Experiments

We conduct testbed experiments on our CloudVS prototype on a Eucalyptus-based cloud platform that is running atop commodity hardware and operating systems.

A. Dataset

We drive our experiments using a 3-month span of VM snapshots that are generated as follow. We prepare a VM that is installed with Fedora 14 and configured with 5GB harddisk space. We deploy the VM with the Internet connectivity, and leave it in the "always-on" state for a 90-day period from February 23, 2011 to May 24, 2011. We schedule a daily cron job `yum -y update` to make the VM regularly download and install any latest updates from the Internet. The installed updates will modify various system files, causing changes to the disk content of the VM. Note that the VM also runs various background jobs that constantly change its disk content. We then take a full snapshot for the VM image daily.

To understand how different VM snapshots evolve, we compute the sizes of incremental and differential deltas using fixed-size chunking with chunk size 4KB. Figure 4 shows the changes of the VM image on individual days, including the sizes of the differential deltas with respect to the base image on the first day (i.e., February 23, 2011), the sizes of the incremental deltas with respect to the image of the previous day, as well as the numbers of updates downloaded. In general, the sizes of the differential deltas have an increasing trend, except that we see two "dips" on March 15-17 and April 4-5. We do not know the real reason, but we conjecture that the dips are related to how the VM kernel reclaims unused blocks that are associated with deleted files. Nevertheless, we verify that each differential delta can be used to correctly recover the VM image for the particular day.

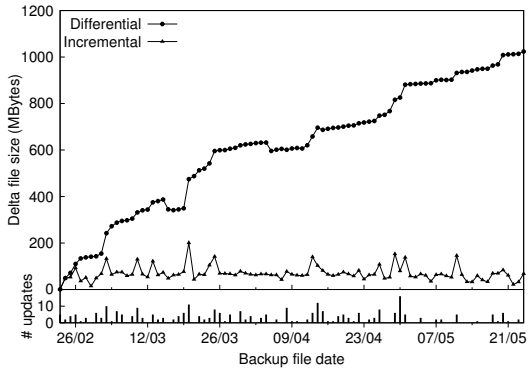


Fig. 4. The evolution of our dataset. The top graph shows the differential and incremental delta sizes of our Fedora VM, while the bottom graph shows the number of downloaded updates on individual days.

Overall, we observe that the differential delta size is at most 1GB, and the incremental delta size is within 100MB throughout the 3-month span. Note that the size of a delta (regardless of incremental or differential) remains to be much smaller than that of the entire VM image, which we configure to be 5GB. Thus, by using RE to store incremental/differential deltas, we can significantly save the storage cost of keeping the full VM images for different versions.

Our dataset is used to address the case where a VM image is being modified over time. By no means do we claim this dataset is representative in real usage. The goal of our evaluation is to demonstrate the feasibility of deploying CloudVS in an open-source cloud environment. Actually, CloudVS works for a general set of VM images that have changes in content.

B. Setup

We set up a Eucalyptus-based cloud testbed with the following servers: (i) one controller node, which is equipped with a 2.8GHz Intel Core 2 Duo E7400 CPU, 4GB of RAM, and 250GB of harddisk, (ii) one storage node, which is equipped with a 2.66HGz Intel Xeon W3520 quad-core CPU, 16GB of RAM, and 1TB of harddisk, and (iii) four compute nodes, each of which is equipped with a 2.66GHz Intel Core i5 760 CPU, 8GB of RAM, and 1TB of harddisk. All six nodes are installed with CentOS 5 and Eucalyptus 2.0.2, and are connected via a Gigabit Ethernet switch.

We deploy CloudVS with the following default configurations. The base image (configured with size 5GB) is cached

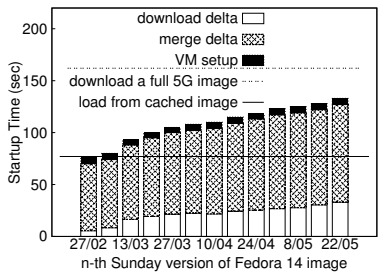


Fig. 5. Experiment 1: VM startup time.

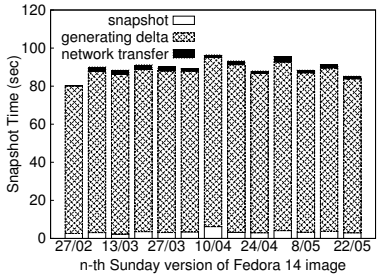


Fig. 6. Experiment 2: VM versioning time.

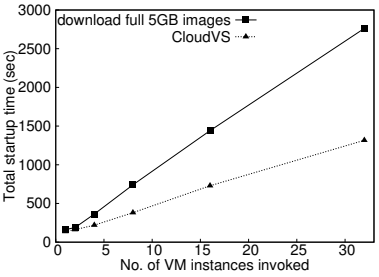


Fig. 7. Experiment 3: Total VM startup time for starting N VM instances.

locally in each compute node. For redundancy elimination, we use fixed-size chunking with chunk size 4KB. To minimize the impact of fragmentation, we set the fragmentation parameter α to 1, meaning that the differential deltas are stored in the storage node (see Section III-C). We enable the LVM snapshot feature and use `posix_fadvice` to pre-declare the access patterns (see Section III-D), but disable read limiting so as to obtain the best possible versioning performance.

C. Performance of VM Operations

We first analyze the performance of basic VM operations when CloudVS is used.

Experiment 1 (VM startup time). We first consider the startup time required for CloudVS to start a VM version on a single compute node. The startup time is measured from the time when the controller node issues the VM startup request with the command `euca-run-instances` (see Section III-E), until the time when the compute node turns the resulting VM instance into the power-on state (i.e., right before the guest VM is booted). Here, we only focus on the VM versions that are created on Sundays. We also measure the time for starting the times for starting the base image (of size 5GB) that is retrieved from the cache and from the storage node.

Figure 5 shows the results. It provides a performance breakdown when CloudVS is used to start a VM version, including (i) downloading the delta, (ii) merging the delta with the cached base image, and (iii) launching the VM instance from the merged image. We observe that the startup time ranges from 79s to 122s, and is mainly attributed to downloading and merging the delta. Note that during the process of merging the delta, CloudVS also loads the cached base image, which is the necessary step even without CloudVS. If we examine the time of starting the cached base image, then we observe that it takes about 77s. That is, the additional startup time introduced by CloudVS ranges from 2s to 45s. On the other hand, if we simply download a full VM image without RE, then the total startup time is about 162s. Thus, the VM startup time still benefits from RE by retrieving less data than the full VM image.

Experiment 2 (VM versioning time). We now evaluate the versioning time of CloudVS, which we define as the time required to create a VM version. This includes the time for creating an LVM snapshot, computing hashes, generating the incremental delta, and uploading the incremental delta to the storage node. Here, we focus on the creation of the Sunday versions as in Experiment 1.

Figure 6 shows the results. We observe that the versioning time ranges from 80s to 100s. In addition, we note that the LVM snapshot time (i.e., the time when the VM is locked or "frozen") can be done within 5s. After creating an LVM snapshot, the versioning process will be done in the background. Thus, the versioning process has limited negative impact from the user perspective.

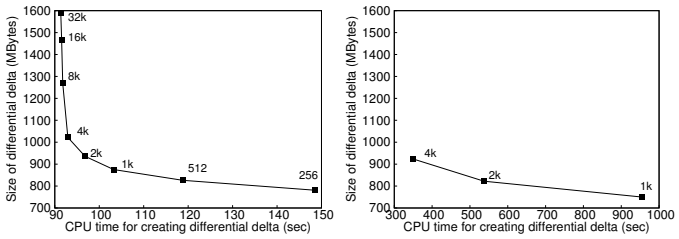
Experiment 3 (Starting multiple VM instances). We further evaluate CloudVS when we start multiple VMs simultaneously, using all four compute nodes in our testbed. In the controller node, we issue the command `euca-run-instances -n N`, where N = 1, 2, 4, 8, 16, and 32, so that Eucalyptus starts a total of N VM instances and allocates them among the four nodes in our testbed. Based on our study, Eucalyptus picks nodes to start VM instances in a round-robin manner, so that the compute nodes receive about the same number of VM instances. For instance, if N = 32, then each of our four compute nodes will be allocated 8 VM instances. Here, we choose to start N instances of the VM version on May 24, which has the largest differential delta size among all versions in the dataset. We then measure the total startup time (as defined in Experiment 1) to start all VM instances. We also show the baseline case to start multiple VM instances with the 5GB base image retrieved from the storage node, as in Experiment 1.

Figure 7 shows the results of the total startup time for starting N VM instances. We observe that CloudVS reduces the startup time when compared to downloading the same number of full base images, for example, by 50% when N = 32. The observations are consistent with those in Experiment 1.

D. Trade-Off Study

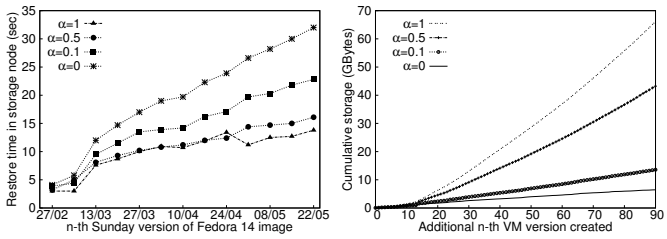
We now analyze how CloudVS addresses the performance trade-offs via different parameter settings. In the following experiments, we focus on the performance of VM versioning with only a single compute node.

Experiment 4 (Performance of RE). Recall that CloudVS uses fixed-size chunking as its RE approach. We now evaluate how the chunk size affects the storage and time performance. For the storage performance, we consider the size of the



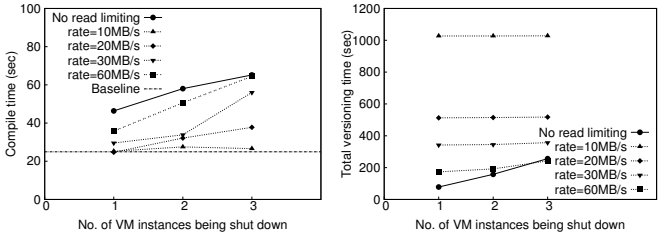
(a) Fixed-size chunking (b) Rdiff

Fig. 8. Experiment 4: Performance of RE.



(a) Restore time in storage node (b) Cumulative storage

Fig. 9. Experiment 5: Impact of α on storage and fragmentation.



(a) HTTP server compile time (b) Versioning time

Fig. 10. Experiment 6: Impact of read limiting.

resulting differential delta; for the time performance, we measure the CPU time needed to create the differential delta in the compute node (without sending the delta to the storage node). As in Experiment 3, we focus on the VM version on May 24.

Figure 8(a) plots the trade-off curve between the storage cost and the versioning time. We observe that with a larger chunk size, the size of the differential delta will be larger since it is less likely to have identical chunks, but less time is spent on delta creation due to fewer hash computations. We observe that our default chunk size 4KB strikes a good balance between the storage and time performance in general.

Note that CloudVS can also apply other RE techniques. To illustrate, we use the utility `rdiff`, which implements the rsync algorithm [20] to generate delta files. Figure 8(b) plots the trade-off curve. Note that it generates a smaller delta size than fixed-size chunking with the same chunk size (e.g., 10% less for chunk size 4KB), but it needs significantly more CPU time for delta creation (e.g., 4× more for chunk size 4KB). We do not dig into the detailed analysis of different RE techniques, as it is beyond the scope of this paper. Our goal is merely to show that CloudVS can apply different RE techniques to trade between the storage and time performance.

Experiment 5 (Impact of α on storage and fragmentation). We evaluate how different values of the fragmentation parameter α trade between the storage and fragmentation overheads. Here, we set the chunk group size to be 500 chunks of size 4KB each. For a given α, we measure the cumulative storage of the deltas across all VM versions. Also, we restore each VM version locally within the storage node, which involves the disk seeks of reading the delta associated with each version. We measure the CPU time for each restore process.

Figures 9(a) and 9(b) plot the restore times for the Sunday versions and cumulative storage consumption for all 90 days of versions, respectively. Note that the two extreme points α = 0 and α = 1 correspond to storing incremental and differential deltas, respectively. As expected, we observe that the larger α leads to less restore time but more storage space, and vice versa. Nevertheless, storing differential deltas still significantly saves the storage space compared to simply keeping full images without using RE, as the latter approach consumes a total of 450GB of space (recall that each VM is configured with 5GB of space). We also choose two intermediate values α = 0.1 and α = 0.5. For example, with α = 0.5, the restore time is 3 seconds more than the extreme point α = 1 (with purely differential deltas), while consuming 35% less storage.

Experiment 6 (Impact of read limiting). Finally, we evaluate how the read limiting feature (see Section III-D) minimizes the interferences to other processes or VM instances in the same compute node during versioning. We start four VM instances on a single compute node, and then terminate N of the instances, where N = 1, 2, and 3. Once we start terminating the VM instances, we start compiling the source code of Apache HTTP Server 2.2.19 on one of the remaining active VM instances. The compile process includes both CPU and I/O intensive jobs, so we expect that the compile time is affected by the versioning process.

Figure 10(a) shows the time required to compile Apache with different read limiting rates. We also plot the baseline compile time when no versioning is performed. Without read limiting, the compile time takes up to 2× more than the baseline case during versioning. Read limiting mitigates the interference. For example, when the read limiting rate is set to 20MB/s and only one VM is shut down, the compile time is reduced to almost the same as in the baseline case.

The trade-off of read limiting is the increase in the versioning time. Figure 10(b) shows the total time of creating all versions for the shutdown VMs when read limiting is used. For example, when the read limiting rate is 20MB/s, the total versioning time is about 520s. Note that the versioning process is done in the background, so the longer versioning time has minimal impact. In addition, read limiting smoothes the burst of creating multiple VM versions simultaneously. Even with more VM instances being shut down at the same time, there is still enough processing power to handle multiple simultaneous versioning processes. As a result, the total versioning time remains constant.

V. Related Work

VM checkpointing. Several studies focus on creating memory or disk snapshots (or checkpoints) for VMs. Park *et al.* [14] propose to avoid storing redundant memory pages of a VM to reduce the time and space of saving the VM memory states. Zhang *et al.* [24] propose to estimate the working set of VM memory so that VM snapshots can be efficiently restored. While [14], [24] focus on saving memory states, some studies [21], [3] also consider saving the VM disk states. Goiri *et al.* [3] differentiate the read-only and read-write parts of a VM disk, and each checkpoint only stores the modifications of the read-write points. CloudVS does not make such differentiation, but instead it directly identifies content-based similarities by scanning the whole VM disk image. Nicolae *et al.* [10] propose a distributed versioning storage service to store VM snapshots. On the other hand, CloudVS focuses on the performance issues when redundancy elimination (RE) is applied in versioning under commodity settings.

VM migration. VM migration [2], [8] is to move a running VM across different physical hosts over the network. Both studies [2], [8] focus on migration of memory snapshots. To minimize the bandwidth of migration, they use the pre-copy approach, in which the first step copies the entire memory snapshot, and the subsequent steps only copy the modified memory pages. Hines *et al.* [4] use a post-copy approach to speed up the migration. CloudNet [23] can migrate both memory and disk states over the Internet, using content-based RE to minimize the migration bandwidth.

VM image storage. Some studies address the management of VM image storage. Foundation [19] is an archiving system for VM disk snapshots. To save storage space, it leverages deduplication to eliminate the storage of redundant data blocks. Mirage [18] proposes a new VM image format that includes semantic information, and addresses version control of VM images as in our work. In contrast to Mirage, our work focuses on the deployment of VM version control in existing open-source cloud platforms. Lithium [4] is a storage layer that provides fault tolerance for VM storage in a cloud. LiveDFS [9] applies deduplication for storing different versions of base VM images of different Linux distributions. It is interesting to further study how to seamlessly integrate deduplication and fault tolerance into CloudVS in future work.

RE techniques. RE is used in many applications in minimizing redundant data, such as data forwarding (e.g., [23]) and data storage (e.g., [6], [22], [17]). In this work, we focus on managing VM images using RE, and specifically consider different tunable mechanisms based on our RE approach to make CloudVS amenable to different commodity platforms.

VI. Conclusions

We propose CloudVS, an add-on system that provides version control for VMs in an open-source cloud that is deployed with commodity hardware and operating systems. CloudVS is built on redundancy elimination to build different VM versions, such that each VM version only keeps the new and modified data chunks since the prior versions. We also propose a simple tunable heuristic and several optimization techniques to allow CloudVS to address different performance trade-offs for different deployment scenarios. We evaluate the performance of CloudVS via a 3-month span of VM snapshot traces. Our work proposes a practical system that facilitates the operational management of an open-source private cloud. The source code of CloudVS is published for academic use at **http://ansrlab.cse.cuhk.edu.hk/software/cloudvs**.

References

[1] M. Armbrust, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica, and M. Zaharia. A View of Cloud Computing. *Comm. of the ACM*, 53(4):50–58, Apr 2010.

[2] C. Clark, K. Fraser, S. Hand, J. G. Hansen, E. Jul, C. Limpach, I. Pratt, and A. Warfield. Live migration of virtual machines. In *Proc. of USENIX NSDI*, 2005.

[3] I. Goiri, F. Juliá, J. Guitart, and J. Torres. Checkpoint-based Fault-tolerant Infrastructure for Virtualized Service Providers. In *Proc. of IEEE NOMS*, 2010.

[4] J. G. Hansen and E. Jul. Lithium: Virtual Machine Storage for the Cloud. In *Proc. of ACM SOCC*, 2010.

[5] K. Jin and E. L. Miller. The effectiveness of deduplication on virtual machine disk images. In *Proc. ACM SYSTOR*, 2009.

[6] P. Kulkarni, F. Douglis, J. LaVoie, and J. M. Tracey. Redundancy elimination within large collections of files. In *Proc. of USENIX ATC*, 2004.

[7] A. Lewis. LVM Howto. http://tldp.org/HOWTO/LVM-HOWTO/index.html, 2006.

[8] M. Nelson, B.-H. Lim, and G. Hutchins. Fast Transparent Migration for Virtual Machines. In *Proc. of USENIX ATC*, 2005.

[9] C.-H. Ng, M. Ma, T.-Y. Wong, P. P. C. Lee, and J. C. S. Lui. Live Deduplication Storage of Virtual Machine Images in an Open-Source Cloud. In *ACM/IFIP/USENIX Middleware*, 2011.

[10] B. Nicolae, J. Bresnahan, K. Keahey, and G. Antoniu. Going back and forth: efficient multideployment and multisnapshotting on clouds. In *Proc. of ACM HPDC*, 2011.

[11] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The Eucalyptus Open-source Cloud Computing System. In *Proc. of IEEE CCGrid*, 2009.

[12] OpenStack. http://www.openstack.org.

[13] OpenStack. XenServer Snapshot Blueprint. http://wiki.openstack.org/XenServerSnapshotBlueprint.

[14] E. Park, B. Egger, and J. Lee. Fast and space efficient virtual machine checkpointing. In *Proc. of ACM VEE*, 2011.

[15] S. Quinlan and S. Dorward. Venti: a new approach to archival storage. In *Proc. USENIX FAST*, 2002.

[16] M. O. Rabin. Fingerprinting by random polynomials. Technical Report Tech. Report TR-CSE-03-01, Center for Research in Computing Technology, Harvard University, 1981.

[17] A. Rahumed, H. C. H. Chen, Y. Tang, P. P. C. Lee, and J. C. S. Lui. A secure cloud backup system with assured deletion and version control. In *International Workshop on Security in Cloud Computing*, 2011.

[18] D. Reimer, A. Thomas, G. Ammons, T. Mummert, B. Alpern, and V. Bala. Opening Black Boxes: Using Semantic Information to Combat Virtual Machine Image Sprawl. In *Proc. of ACM VEE*, 2008.

[19] S. Rhea, R. Cox, and A. Pesterev. Fast, inexpensive content-addressed storage in foundation. In *Proc. USENIX ATC*, 2008.

[20] A. Tridgell and P. Mackerras. The rsync algorithm. Technical Report TR-CS-96-05, The Australian National University, 1996.

[21] G. Vallée, T. Naughton, H. Ong, and S. L. Scott. Checkpoint/restart of virtual machines based on xen. In *High Availability and Performance Computing Workshop (HAPCW)*, 2006.

[22] M. Vrable, S. Savage, and G. M. Voelker. Cumulus: Filesystem backup to the cloud. *ACM Trans. on Storage (ToS)*, 5(4), Dec 2009.

[23] T. Wood, K. K. Ramakrishnan, P. Shenoy, and J. van der Merwe. CloudNet: Dynamic Pooling of Cloud Resources by Live WAN Migration of Virtual Machines. In *Proc. of ACM VEE*, 2011.

[24] I. Zhang, Y. Baskakov, A. Garthwaite, and K. C. Barr. Fast Restore of Checkpointed Memory using Working Set Estimation. In *Proc. of ACM VEE*, 2011.