A Tunable Version Control System for Virtual Machines in an Open-Source Cloud

Chung Pan Tang, Patrick P. C. Lee, Tsz Yeung Wong

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 design and implement *CloudVS*, a practical 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. We further exploit a higher degree of content similarity by applying RE to multiple VM images simultaneously, and support the copy-on-write image format. Using real-world VM snapshots, we experiment CloudVS in an open-source cloud testbed built on Eucalyptus. We demonstrate how CloudVS can be parameterized to balance the performance trade-offs between version control and normal VM operations.

Index Terms—VM image versioning, redundancy elimination, open-source cloud management, implementation, experimentation

1 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 [2]. Typically, cloud computing uses the virtualization technology to manage a shared pool of physical resources. On the other hand, commercial clouds may not fit the needs of some users. For example, there are security concerns of outsourcing computation to third-party commercial clouds [25]. Open-source cloud platforms, such as Eucalyptus [15] and OpenStack [16], implement the Infrastructure-as-a-Service (IaaS) model and 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.

• C. P. Tang, P. P. C. Lee, and T. Y. Wong are with the Dept of Computer Science and Engineering, The Chinese University of Hong Kong, Shatin, N.T., Hong Kong (emails: {tangcp, pclee, tywong}@cse.cuhk.edu.hk).

However, there are 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.

The key motivation of this work is to address the above challenges from an *applied* perspective. We aim to develop a system that enables version control for VMs and can be practically deployed in today's open-source cloud platforms. We also support our system design with extensive testbed experiments based on real-world VM image traces.

Our contributions are four-fold. First, we propose *CloudVS*, a 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. While RE is a well-proven technology, our goal is to demonstrate how RE can be deployed to minimize the overheads of transmission and storage in the version control for VMs.

Second, 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 versioning process. In short, CloudVS uses different performance adaptation strategies to easily make performance trade-offs between version control and normal VM operations. Furthermore, we propose a *multi-VM versioning mechanism*, such that CloudVS can apply RE to multiple VM images simultaneously, so as to exploit a higher degree of content similarity among multiple VM images and further reduce redundancy. We also extend CloudVS to support the copy-on-write image format, so as to reduce the provisioning overhead.

Third, we implement a prototype of CloudVS and integrate it into Eucalyptus [15] as an add-on system. The current open-source implementation of Eucalyptus 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.

Finally, we conduct extensive experiments for CloudVS on a Eucalyptus-based cloud testbed. We evaluate CloudVS using two datasets of VM images: (i) a 3-month span of snapshots of a regularly updated VM, and (ii) a 7-week span of snapshots of VMs from different users who are involved in a programming project in an undergraduate course. We show via both datasets 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.

We point out that commercial VM management systems (e.g., VMWare Workstation, VirtualBox) also provide version control for VMs. However, their implementation details remain private, so we cannot formally compare CloudVS with them. Nevertheless, based on our experiences, we address several issues that remain unexplored in such commercial systems: (i) we consider a tunable version control mechanism that is amenable to commodity hardware settings, (ii) we consider a mechanism that works seamlessly in an existing open-source cloud platform, and (iii) we conduct extensive testbed experiments to evaluate our design.

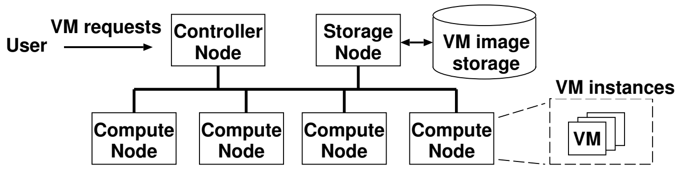


Fig. 1: A simplified cloud architecture being considered.

The rest of the paper proceeds as follows. In Section 2, we overview the cloud architecture considered in this paper. In Section 3, we explain the design of CloudVS and propose several practical optimization techniques. In Section 4, we experiment CloudVS in a Eucalyptus cloud testbed. In Section 5, we review related work, and finally, Section 6 concludes the paper.

2 Background

In this section, we present an overview of a cloud architecture. We also describe how today's open-source cloud platforms address version control of VMs.

Fig. 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 [15] and OpenStack [16] are designed based on the same layout as in Fig. 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 and OpenStack handle the lifecycle of a VM instance[1]. In the normal VM lifecycle, when a VM instance is stopped, all modifications made to the VM instance will be permanently purged. Both Eucalyptus and OpenStack provide a snapshot feature that supports regular backups of VM images [7], [17]. However, they only send 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.

3 CloudVS Design

In this section, we present the design of *CloudVS*, an add-on 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.

3.1 Definitions and Roadmap

We first provide the key definitions and a roadmap of the design of CloudVS. 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. For each user, CloudVS can generate different *VM versions*, each of which describes the user-specific state changes made to the base image. Instead of storing the full image of the user-modified VM, the versioning process of CloudVS is to incrementally build 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 Fig. 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 3.2.

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.

1. At the time of the writing, the latest releases of Eucalyptus and OpenStack are version 3.1, and version 2012.2, respectively.

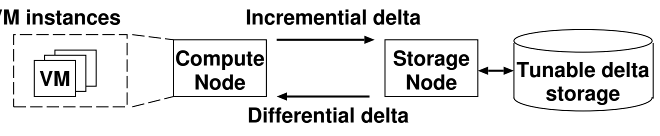


Fig. 2: CloudVS architectural overview.

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 [24]. 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.

CloudVS combines several mechanisms to make VM version control feasible. They include *tunable data storage*, which balances the trade-off of storage and fragmentation via a single tunable parameter (Section 3.3), *I/O optimizations*, which mitigates the interferences to other co-resident processes (Section 3.4), *multi-VM versioning*, which exploits content similarity among multiple VM images (Section 3.5), *copy-on-write support*, which reduces the cost of I/O-intensive operations such as VM provisioning and snapshotting (Section 3.6), and *in-place VM restore*, which exploits the content similarity of the working VM and the prior VM version to be restored (Section 3.7).

The CloudVS implementation is a fork of the original execution flow of existing cloud platforms. In Section 3.8, we demonstrate how CloudVS can be integrated into Eucalyptus as a proof of concept. We also discuss the limitations of this work in Section 3.9.

To summarize, Fig. 2 gives an overview of CloudVS. Each VM instance runs on the hypervisor of a compute node. Each VM version is sent as incremental delta to a centralized storage node, which then stores the VM version based on the tunable delta storage mechanism described above. Any VM version can be reconstructed with the local base image and the corresponding differential delta retrieved from the storage node.

In this work, we do not consider the security issues in version control. In the context of RE, CloudVS may provide confidentiality guarantees to different VM versions via *convergent encryption* [1], [5], in which data is encrypted/decrypted by a secret key that is derived from the cryptographic hash of the data content. This preserves content similarity and hence effectiveness of RE even after data encryption.

3.2 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. 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. By no means do we claim that RE is our contribution. Similar RE-based versioning approaches have been proposed, such as in cloud backup systems [22], [30], which target the storage of general data types. Here, our focus is to show the proof of applicability of RE, and demonstrate how RE is applied 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 [28] and Rabin fingerprinting [21]), but fixed-size chunking has been shown to be effective in RE for VM images [10].

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 [20].

There are two scenarios in which 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 VM version is created in the last VM shutdown, CloudVS generates hashes for the last VM 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 is created. In both cases, the hashes of the last VM version will be cached in the compute node where the VM is currently running, and later compared with the current VM version.

We emphasize that we only consider eliminating the redundancy between two successive VM versions. It is possible that this RE approach misses identifying certain redundant chunks across non-successive VM versions. For instance, a unique data chunk is found in the i-th VM version but not in the $(i+1)$-th one, yet it reappears again in the $(i+2)$-th VM version. In this case, the data chunk will be stored twice. However, we justify that it is uncommon to have redundant chunks in non-successive VM versions in practice. Based on our investigation on a real-usage dataset (DS-PA, see Section 4.1), less than 0.15% of such chunks are identified.

To create the current VM version, we compare by hashes the different chunks of the VM images of the previous 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 previous version. The resulting incremental delta, together with the references that refer to the already existing chunks in prior versions, will be sent to the storage node.

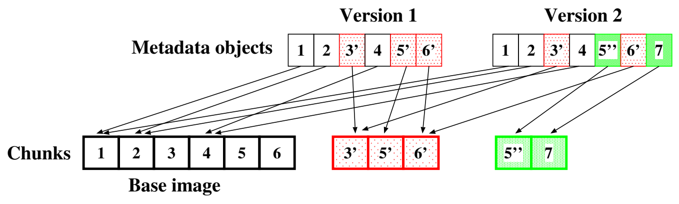


Fig. 3: Example of how CloudVS uses RE to correlate different VM versions. The compute node will send the incremental delta for each version to the storage node.

Fig. 3 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 Fig. 3). 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 2). 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 Fig. 3 is to be restored. Then the storage node will transmit only the 3rd, 5th, 6th, and 7th chunks. Note that we reconstruct the differential delta in the storage node rather than in the compute node, so that the compute node does not need to make a significant number of requests for individual chunks from the storage node. In Section 3.7, we further improve the restore efficiency, by exploiting the content similarity of the working VM image and the prior VM version to be restored.

3.3 Tunable Delta Storage

If the storage node directly stores the incremental delta (i.e., the new and modified chunks since the previous VM version) for each VM version, then the fragmentation overhead may exist during the restore of a VM version. Referring to Fig. 3 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 a 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 target 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. It determines the trade-off between the storage overhead and restore performance. Increasing α implies that the storage overhead increases, yet the restore performance improves due to less fragmentation. In the extremes, if $\alpha = 1$, then each VM version stores the differential delta; if $\alpha = 0$, then each VM version stores only the incremental delta. We observe that even with this simple heuristic, we can effectively make the trade-off (see Section 4). We point out that the trade-off result is content-specific. In the rare case, if all VM images are diverge and have no RE opportunity, then the effect of α is limited. However, in practice, VM images are known to have high redundancy [10]. Our goal is not to propose the best α value, but to give the flexibility to system administrators to choose the right α value according to the resource requirements.

To illustrate, we consider again how to restore Version 2 in Fig. 3. 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) the 3rd and 5th chunks, and (ii) the 6th and 7th chunks. Both chunk groups have one non-sequential read. If $\alpha = 0.5$, then we have the first chunk group (i.e., the 3rd and 5th chunks) store the differential delta, while the second chunk group (i.e., the 6th and 7th chunks) still stores the incremental delta. Fig. 4 shows the final result.

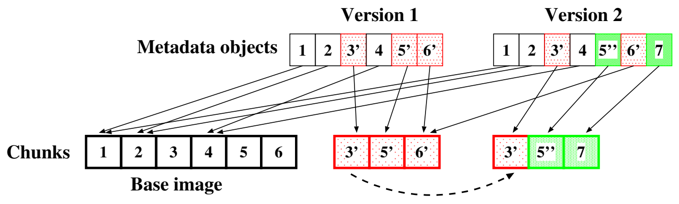


Fig. 4: Example of how CloudVS stores the incremental and differential deltas for different chunk groups. Here, the first chunk group (the 3rd and 5th chunks) is stored in Version 2 as the differential delta, assuming $\alpha = 0.5$.

3.4 I/O Optimization

When creating a VM version, CloudVS scans the 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 reduce the performance interferences due to versioning.

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)* [12]. 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.

3.5 Multi-VM Versioning

We thus far apply RE on a per-VM basis, such that we seek to eliminate redundant content of different VM versions evolved from a single VM. On the other hand, it is possible that multiple VMs (typically owned by different users) have identical content changes over time. One example scenario is discussed in [4], in which users of the same organization are assigned VMs that are initially installed with standardized operating systems. We also present a similar scenario in a real-life use case in Section 4.1. If all VMs enable regular system updates, then we expect that the underlying system files are modified or patched in the same way. The identical changes across VM images can be exploited by CloudVS to achieve better RE efficiency.

We extend CloudVS to support *multi-VM versioning*, in which we identify redundancy across multiple VMs running in the same compute node during the versioning process. Here, we introduce a *delay commit policy* on each compute node. The main idea is that instead of committing the incremental deltas of the individual VM versions in the same compute node directly to the storage node as in per-VM versioning, we now collectively identify the redundancy of these incremental deltas. Specifically, after generating the incremental delta for a VM image in the versioning process, we also generate the hashes of data chunks of the incremental delta. Both the incremental delta and the hashes are locally stored in the compute node. Suppose that multiple VM versions have been created and stored in the compute node. Then we compare the pre-generated hashes of those VM versions, and eliminate any redundant chunks in the incremental deltas. Finally, the updated incremental deltas are collectively sent to the storage node.

Fig. 5 shows the idea of multi-VM versioning. Suppose that there are three VM versions taken from three different VMs (denoted by VM A, VM B, and VM C) at the same time. Each VM version contains the modified chunks since its previous version, as indicated in the figure. In per-VM versioning, we note that there are a total of nine modified chunks. Now let us assume that the 3rd chunk of each VM version is modified to have the same content. In multi-VM versioning, only one copy of the 3rd chunk is kept, and the metadata objects of the VM versions will point to the copy. In this case, only a total of seven chunks are committed to the storage node.

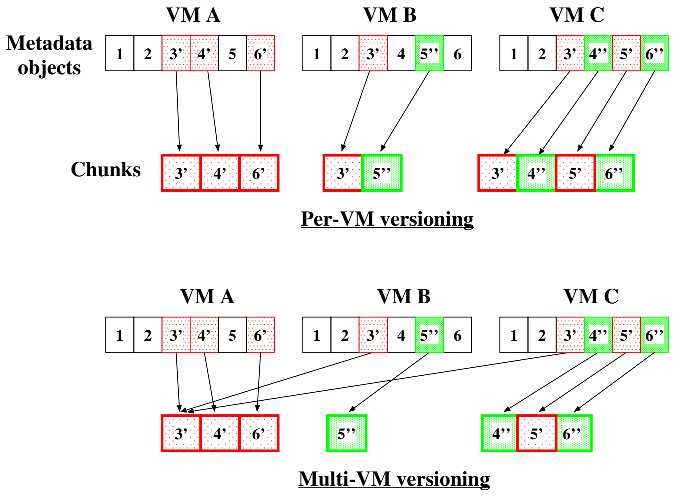


Fig. 5: Example of multi-VM versioning. Suppose that the contents of the 3rd chunks of VM A, VM B, and VM C are the same. In multi-VM versioning, only one chunk copy is kept.

Note that the tunable delta storage technique mentioned in Section 3.3 is also applicable for multi-VM versioning, and hence enables us to make a trade-off between the storage and disk seek overheads. In a nutshell, we first eliminate the redundant chunks across multiple VM versions as described above. Then for each VM version, we transform the chunk groups with the most non-sequential reads into differential deltas as described in the per-VM case (see Section 3.3). For example, let us consider the collective VM versions in Fig. 5 being received by the storage node. Suppose we set the chunk group size to two and $\alpha = 0.5$. For VM C, there are two chunk groups: (i) the 3rd and 4th chunks, and (ii) the 5th and 6th chunks. We note that the first and second chunk groups have one and zero non-sequential reads, respectively. Thus, the first chunk group (i.e. 3rd and 4th chunk) will be stored as the differential delta. Fig. 6 shows the final result.

We argue that multi-VM versioning introduces only a small metadata cost in terms of processing and storage. Let us consider 1GB of incremental deltas generated by different VM versions. Suppose that we use SHA-1 to generate 20-byte hashes for the chunks of size 4KB each. Then there are a total of 262,144 hashes, which account for 5MB only. Also, our analysis shows that modern CPUs (based on our testbed described in Section 4.2) can achieve a throughput of 400MB/s of

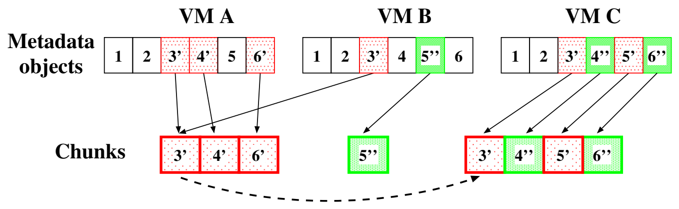


Fig. 6: Example of applying the tunable delta storage in multi-VM versioning. For VM C, the 3rd and 4th chunks are stored in the form of the differential delta.

SHA-1 hash computations, so the computational cost of hash computations remains low. In terms of metadata storage, each chunk is mapped to an 8-byte file offset and a 20-byte SHA-1 hash, so the total metadata space is 7MB. Compared to the size of increments, the metadata space overhead is also very low.

3.6 Copy-on-Write Extensions

To increase storage utilization and reduce the VM provision time, *thin provisioning* is often used to delay the storage allocation of VM images. Modern hypervisors (e.g., VMWare and KVM) apply this technique by supporting the *copy-on-write (CoW)* disk image format, which allocates dedicated storage space only when disk write requests are made. In the following, we show how CloudVS can be extended to support one of the CoW formats called QCoW2 [19], so as to take advantage of thin provisioning.

QCoW2 [19] is a VM sparse disk image format supported by the open-source hypervisor KVM. Unlike the raw file format, QCoW2 supports various advanced features such as CoW, snapshotting, and compression. It introduces a thin low-overhead layer of metadata (e.g., logical block mappings and reference counts) atop the base VM image, which is called the *backing file*. CoW is supported as follows. If a write request is issued to a block in the backing file, a new physical block is allocated and maintained in a separate QCoW2 image file. In addition, if a physical block in the QCoW2 is shared by multiple logical blocks and is modified, then a new physical block is allocated as well. The new block allocation ensures that the original block is not overwritten.

QCoW2 supports both internal snapshotting and external snapshotting. In internal snapshotting, all snapshots (or VM versions) will be appended to the same QCoW2 image file; in external snapshotting, a series of snapshots will be kept such that each snapshot will store the incremental deltas of the previous snapshot. Thus, external snapshotting suffers the fragmentation problem, since the VM content is scattered across all previous snapshots. There is no tunable feature to choose between incremental and differential deltas as in CloudVS.

Instead of directly adopting QCoW2 as the versioning mechanism, we modify the execution flows of CloudVS to support QCoW2 to take advantage on the CoW feature. Instead of operating on the raw VM image as described in the previous subsections, we now have CloudVS operate on the QCoW2 image file, which stores the changes of the backing file (base VM image).

Specifically, in the original versioning flow, CloudVS scans for the changes of the entire VM image. In the new versioning flow, CloudVS builds on internal snapshotting of QCoW2. When a VM starts, CloudVS takes an internal snapshot to record the original state; when versioning is performed, CloudVS takes another internal snapshots. It then compares the block mapping tables of the two internal snapshots and identifies the changes to construct the incremental delta. Note that it does not need to scan the whole VM image.

To restore a VM version, the original flow of CloudVS merges the differential delta with the base image in the compute node. In the new restore flow, CloudVS skips the merging process, but instead reconstructs a QCoW2 image file by rebuilding the metadata (including block mappings and reference counts) and the data blocks from the differential delta. Then we link this QCoW2 image to the backing file.

3.7 In-place VM Restore

One use case of VM version control is to revert any unwanted changes made on the currently working VM, for example, due to accidental corruption of system configurations and virus infection. In this case, we may either restore a prior VM version that has been saved, or rebuild the base VM from sketch.

Instead of disposing the working VM image, we implement an *in-place restore* scheme that exploits the content similarity of the working VM and the prior VM version to be restored to speed up the restore process. Suppose that the working VM is labeled as version N and we want to restore to the prior version $N-i$ for some $i \in \{1, 2, \cdots, N-1\}$. We assume that the working VM and the prior version are both hosted in the same compute node. First, we make a snapshot of the working VM disk image and generate the metadata of an incremental delta in the compute node. This is similar to performing versioning of the working VM, but here we only record the offsets of the modified chunks without storing the actual modified data. Then the compute node sends the metadata to the storage node, which then generates a *reverse delta* from version N to a prior version $N-i$. The reverse delta can be viewed as an incremental delta from version N if we treat the prior version $N-i$ as the latest version $N+1$. The reverse delta is sent back to the compute node, which merges the reverse delta with the working VM image to form the new VM image. We expect that the size of the reverse delta is smaller than that of the differential delta built from the base VM described in Section 3.2. How to dynamically choose between in-place restore and the baseline restore approach will be posed as future work.

3.8 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.

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.

Compute node. CloudVS is composed of two modules: the *Snapshot* and *Restore* modules. The Snapshot module generates deltas for different VM versions. It also handles multi-VM versioning and it supports both raw and QCoW2 image formats. The Restore module is responsible for restoring a VM version. We integrate both modules into the operations of each compute node. In our current prototype, we mainly consider shutdown-based versioning (see Section 3.2). We insert the Snapshot module when the VM is shut down (inside the function `scCleanupInstanceImage()`), and insert the Restore module right before the VM is started (inside the function `get_cached_file()`). We add both modules in ∼`eucalyptus/storage/storage.c`. The integration involves no more than 30 lines of code changes. We refer readers to the project website **http://ansrlab.cse.cuhk.edu.hk/software/cloudvs** for the integration details.

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. The daemon retrieves and saves the specified VM version delta, and manipulates the tunable delta storage (see Section 3.3).

3.9 Discussion

We point out several open issues of the existing CloudVS design. In particular, we focus on the scalability aspect of CloudVS. We pose these issues as future work.

Global RE. Our current multi-VM versioning design is limited to a single compute node, which may host multiple VMs. We may extend it to multiple compute nodes to exploit a better RE opportunity, and index all chunks and identify duplicates across different compute nodes using the distributed hash table (DHT) [6], [31]. A challenge is to limit the message exchange overhead due to DHT, particularly in a commodity setting where network resources may be scarce.

Distributed storage. We currently assume a centralized storage node is used for simplicity. To provide scalability and fault tolerance, we can deploy the storage node using a standard distributed file system (e.g., HDFS [26]) to keep the data chunks and metadata files of CloudVS. Using a distributed file system, one may parallelize the retrieval of data chunks so as to improve restore performance. We plan to study the potential of parallelism in future work.

Metadata management. We now assume that our metadata management overhead is limited (see Section 3.5). However, this assumption no longer holds if the number of VMs significantly grows. We may exploit version segment trees [14] for scalable metadata management for a large number of VMs. The current focus of CloudVS is to address the I/O overhead in restoring data chunks. Integrating CloudVS with scalable metadata management remains an open issue to address.

4 Experiments

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

4.1 Datasets

In our experiments, we consider two datasets of VM image snapshots namely *DS-SU* and *DS-PA*, which are collected from different deployment scenarios. We briefly describe how the datasets are collected, and we refer readers to the digital supplementary file for the detailed analysis of the datasets.

DS-SU (Dataset taken from system updates). The dataset DS-SU is used to address the case where a VM image is being modified over time. 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. 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.

DS-PA (Dataset taken from programming assignments). The dataset DS-PA describes the user activities of VMs in real usage. We consider a university course on computer programming that is offered to undergraduate students in the Fall 2011 semester. In around November 2011, each student group is offered an identical VM image (i.e., the base image) that is installed with Ubuntu 10.04 and allocated with a 10GB disk space. Students are then asked to do their programming assignments on their assigned VMs. We collect the snapshots of 21 VM images weekly over a span of seven weeks (from

November 23, 2011 to January 4, 2012). The incremental delta sizes range from 18.4MB to 8.9GB. Our analysis ignores four VMs that generate unexpectedly large files (e.g., video files) and focuses on the remaining 17 VMs.

4.2 Testbed Setup and Default Settings

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.

By default, we deploy CloudVS with the following configurations. The base image is cached locally in each compute node (see Section 2). Our RE approach uses fixed-size chunking with chunk size 4KB. To minimize the impact of fragmentation, we set the fragmentation parameter α to 1, meaning that only the differential deltas are stored in the storage node (see Section 3.3). We enable the LVM snapshot feature and use `posix_fadvice` to pre-declare the access patterns (see Section 3.4), but disable read limiting so as to obtain the best possible versioning performance.

4.3 Performance of VM Operations

We first analyze the performance of different basic VM operations when CloudVS is used. We focus on per-VM versioning based on the dataset DS-SU.

Experiment 1 (VM startup time). We first consider the startup time required for CloudVS to start a VM version in 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 3.8), 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.

Fig. 7 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.

Fig. 8 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.

We note that the network transfer time of CloudVS only contributes no more than 3s, since we only send incremental deltas in versioning. If we send a full 5GB image without RE, the network transfer time increases to 70s (measured in our testbed). While sending a full image eliminates the overhead of delta generation, its network transfer overhead can become dominant depending on the network condition. For example, in a 100Mb/s network, the transfer time can increase to over 400s, which is significantly greater than the current versioning time of CloudVS.

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.

Fig. 9 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.

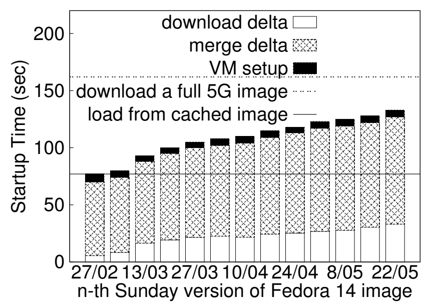


Fig. 7: Experiment 1: VM startup time.

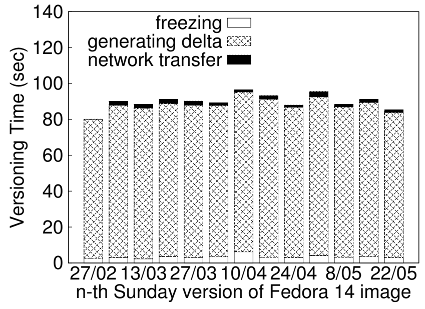


Fig. 8: Experiment 2: VM versioning time.

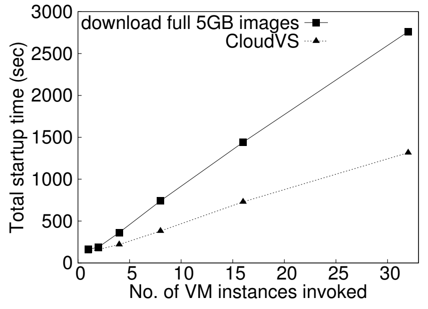


Fig. 9: Experiment 3: Total VM startup time for starting N VM instances.

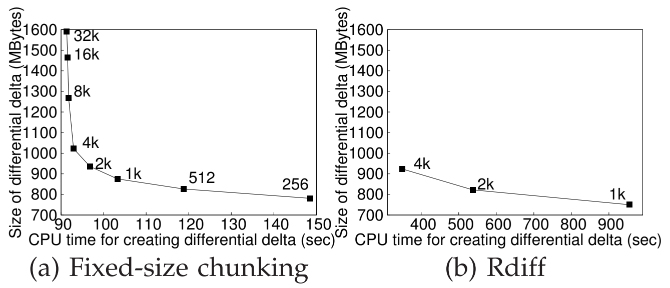


Fig. 10: Experiment 4: Performance of RE.

4.4 Trade-Off Study

We analyze how CloudVS addresses the performance trade-offs via different parameters. In the following, we focus on the performance of per-VM versioning with only a single compute node, using the dataset DS-SU.

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 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.

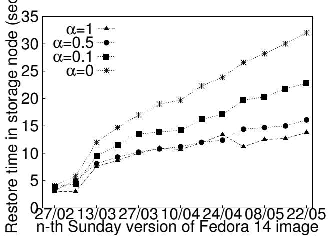
Fig. 10(a) plots the trade-off curve between the storage cost and the versioning time. We observe that with a larger chunk size, the size of the differential delta will be larger since it is less likely to have identical chunks, but less time is spent on delta creation due to fewer hash computations. 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 [28] to generate delta files. Fig. 10(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.

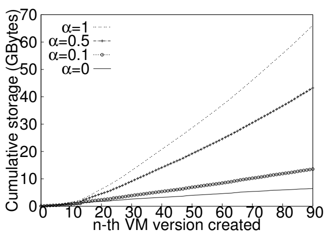
Experiment 5 (Impact of α on storage and fragmentation). We evaluate how different values of the fragmentation parameter α trade between the storage and fragmentation overheads. Here, we set the chunk group size to be 500 chunks of size 4KB each. For a given α, we measure the cumulative storage of the 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 execution time for each restore process.

Figs. 11(a) and 11(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 $\alpha = 0$ and $\alpha = 1$ 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. We note that our current implementation is I/O-intensive in differential delta reconstruction, in which multiple disk seeks to the data chunks are required. Thus, increasing α can mitigate fragmentation and hence the disk seek overhead. 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 $\alpha = 0.1$ and $\alpha = 0.5$. For example, with $\alpha = 0.5$, the restore time is 3s more than the extreme point $\alpha = 1$ (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 3.4) 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

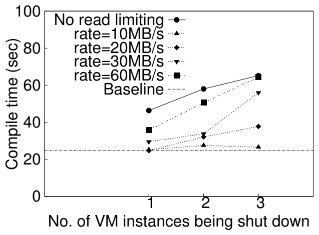


(a) Restore time in the storage node

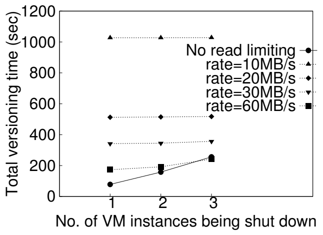


(b) Cumulative storage

Fig. 11: Experiment 5: Impact of α on storage and fragmentation.



(a) HTTP server compile time



(b) Versioning time

Fig. 12: Experiment 6: Impact of read limiting.

the versioning process.

Fig. 12(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. Fig. 12(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.

4.5 Analysis of Multi-VM Versioning

We now evaluate the performance of CloudVS using the real-world dataset DS-PA. We first study the performance of VM operations of CloudVS based on DS-PA. We then analyze how multi-VM versioning can further eliminate redundancy compared to per-VM versioning.

Experiment 7 (VM startup time with DS-PA). We evaluate the startup time required for CloudVS to start a

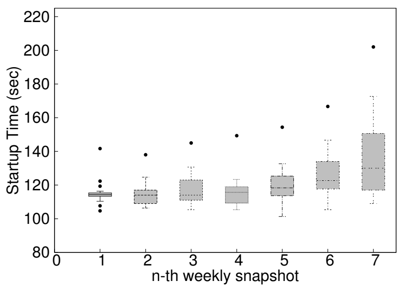


Fig. 13: Experiment 7: Box plots of VM startup time in each weekly snapshot of the dataset DS-PA.

VM version in a single compute node using dataset DS-PA. Here, we only focus on the overall startup time of each VM version over the 7-week span. We measure the startup time for each VM in each weekly snapshot. We also measure the time for starting the base image (of size 10GB) that is retrieved from the storage node. We again present the results in box plots to show the distributions of VM startup time in each weekly snapshot.

Fig. 13 shows the box plots of distribution of VM startup time of VM in each weekly snapshot. Overall we observe that the startup time ranges from 101s to 202s. On the other hand, if we simply download a full VM image without RE, then we find that the total startup time can reach 400s (not shown in the figure). We observe the significant startup time reduction because the original Eucalyptus design does not eliminate redundant blocks in download.

Experiment 8 (Multi-VM versioning with tunable delta storage). We evaluate how the multi-VM versioning works with different values of the fragmentation parameter α in the tunable delta storage. The goal of this experiment is to show that the tunable delta storage mechanism remains applicable in multi-VM versioning. In this experiment, we enable multi-VM versioning and keep other parameters the same as in Experiment 5 (which only considers per-VM versioning). For a given α, we restore VM versions of the snapshot of the 7th weekly snapshot of DS-PA locally within the storage node. The restore process incurs disk seeks due to reading the deltas associated with the ascendant versions. We measure the execution time for the restore process of each of the 17 VMs. Before each run, we flush the cache in memory by executing the commands `sync` and `echo 3 >/proc/sys/vm/drop_caches`. The results are averaged over four runs.

Fig. 14 plots the restore time for all VM versions from DS-PA under several values of α. Recall that the two extreme points $\alpha = 0$ and $\alpha = 1$ correspond to storing incremental and differential deltas, respectively. Despite the variance of the experimental results, we can see roughly the larger α leads to less restore time, similar to the observations in per-VM versioning (see Experiment 5).

Experiment 9 (Storage and bandwidth transfer savings with multi-VM versioning). We evaluate how

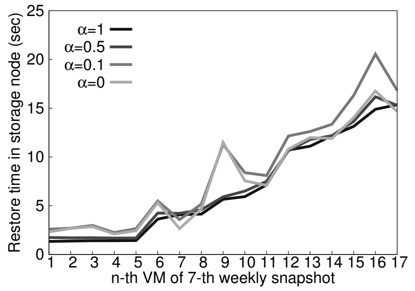
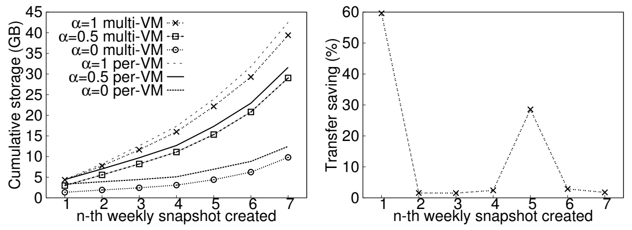


Fig. 14: Experiment 8: Restore time in the storage node with multi-VM versioning.



(a) Cumulative storage (b) Transfer saving

Fig. 15: Experiment 9: Storage and bandwidth transfer savings with multi-VM versioning.

multi-VM versioning saves storage and the bandwidth of network transfer. Here, we consider a compute node in which we apply multi-VM versioning to the 17 VM versions over the seven weekly snapshots.

We measure the cumulative storage of the deltas across all VM versions over the 7-week span. Fig. 15(a) plots the results for both per-VM versioning and multi-VM versioning for different values of α. We can see that multi-VM versioning further saves the storage of per-VM versioning, by 21-59% for $\alpha = 0$ (incremental delta) and 1.5-7.9% for $\alpha = 1$ (differential delta).

We also measure the bandwidth transfer (recall that the data is always sent as incremental delta from the compute node to the storage node over the network). Fig. 15(b) plots the results. It is worth nothing that multi-VM versioning specifically saves 59% of bandwidth in the first week of snapshot and 28% of bandwidth in the fifth week of snapshot. After investigation of the results, we find that users have installed or upgraded the common software package needed for their course assignments in these two weeks. Thus, multi-VM versioning takes advantage of the similar disk modifications made to all VMs, and saves the transfer bandwidth.

4.6 Analysis of Copy-On-Write (CoW) Format

We analyze the performance of different basic VM operations when CloudVS is applied to the QCoW2 image format. We focus on per-VM versioning based on the dataset DS-SU. Our goal is to show that CloudVS can take advantage of the CoW feature and reduce the startup/restore overhead when compared to the original CloudVS that operates on the raw image format. We also compare the restore performance of CloudVS and the plain QCoW2. The latter is chosen as the default QCoW2 installation with external snapshotting, as described in Section 3.6.

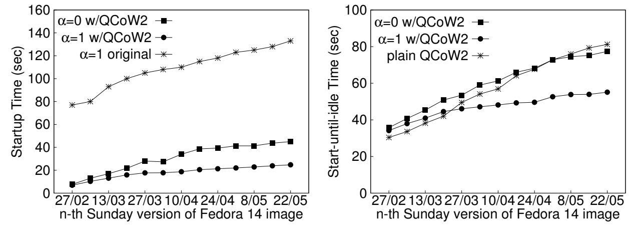
Experiment 10 (VM startup performance with QCoW2). We re-examine the startup time required for CloudVS to start a VM version in a single compute node using the dataset DS-SU. The experiment setting is the same as Experiment 1, except that CloudVS is now used with QCoW2. We choose two special cases for the tunable delta storage: $\alpha = 1$ (differential delta) and $\alpha = 0$ (incremental delta). For comparison, we also plot the results of Experiment 1 that represent the original CloudVS that operates on the raw image format.

Fig. 16(a) shows the startup time results. When QCoW2 is used, the VM startup time of CloudVS ranges from 8s to 47s for $\alpha = 0$ (incremental delta) and from 7s to 25s for $\alpha = 1$ (differential delta). It shows a similar increasing trend of startup time as in Experiment 1, but the startup time is significantly reduced compared to the original CloudVS. For example, for $\alpha = 1$, the support of QCoW2 saves 81-91% of startup time by mitigating the overhead of whole VM image reconstruction.

We further compare the startup performance of CloudVS with QCoW2 support and the plain QCoW2. Our goal is to show how CloudVS mitigates fragmentation in the storage node by storing differential delta (i.e., $\alpha = 1$). To test the plain QCoW2, we set up a remote NFS service in the storage node to store a series of external snapshots of a QCoW2 image. We first download the latest QCoW2 external snapshot, and take the remaining snapshot files at the remote NFS as backing files to start the VM. The data chunks store at the remote NFS will be retrieved on demand during startup. We start the VM versions through both CloudVS and the plain QCoW2. To make a fair comparison on the actual startup performance, we measure the start-until-idle time, which is defined as the startup time (shown in the previous experiments) plus the time required to boot the guest OS until the system is ready to use.

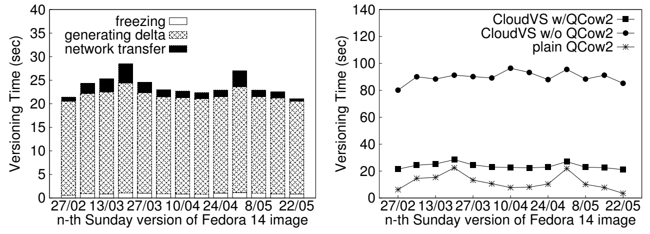
Fig. 16(b) shows the results. The start-until-idle time for the plain QCoW2 ranges from 30s to 81s, and that of CloudVS ranges from 34s to 55s for $\alpha = 1$ and from 35s to 77s for $\alpha = 0$. The plain QCoW2 outperforms CloudVS in restoring the earlier versions, since only few chunks are downloaded on demand when the guest OS is booted with the plain QCoW2. However, we note that when restoring later versions (after 27th March), the start-until-idle time of CloudVS for $\alpha = 1$ is less than that of plain QCoW2, as the versioning chain of QCoW2 image is getting longer and the fragmentation plays a major role in causing significant overhead. For the VMs of May, the start-until-idle time of the plain QCoW2 closely matches CloudVS with $\alpha = 0$. On the other hand, CloudVS can mitigate the fragmentation problem by reconstructing differential delta in the storage node for $\alpha = 1$.

Experiment 11 (VM versioning performance with QCoW2). We re-evaluate the versioning time of CloudVS



(a) versus baseline (b) versus plain QCoW2

Fig. 16: Experiment 10: VM startup performance of CloudVS with QCoW2 enhancement compared with (a) the baseline CloudVS and (b) the plain QCoW2.



(a): breakdown (b): versus plain QCoW2

Fig. 17: Experiment 11: VM versioning for CloudVS when QCoW2 is used.

with QCoW2, as in Experiment 2. Again, we provide a breakdown for the times for creating a QCoW2 snapshot, generating the incremental delta from the QCoW2 snapshot, and uploading the incremental delta to the storage node. We also compare the snapshot result with (i) the baseline version of CloudVS, which works on RAW image and (ii) the plain QCoW2 that stores the external snapshots on a remote NFS backend as in the previous experiment.

Fig. 17(a) shows the breakdown results. With QCoW2, the versioning time of CloudVS ranges from 21s to 29s. As in Experiment 2, the time of generating delta is much less than the baseline by around 75%. The reason is that the CoW feature of QCoW2 separates the write requests from the previous version, and therefore the delta generation only requires a rapid comparison of items in the block mapping table in the QCoW2 image. It eliminates the need to scan the whole VM image. Fig. 17(b) show the versioning time results compared with the plain QCoW2. CloudVS is slower (by 5-20s) in the versioning process because there is a format conversion that requires extra logic and computation. We regard the overhead as a trade-off as CloudVS provides extra features such as tunable delta storage.

In the digital supplemenary file, we analyze the restore performance of CloudVS when in-place VM restore is applied (see Section 3.7).

5 Related Work

VM snapshotting and versioning. Several studies focus on creating memory or disk snapshots for VMs. Park *et al.* [18] 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.* [33] propose to estimate the working set of VM memory so that VM snapshots can be efficiently restored. While [18], [33] focus on saving memory states, some studies [8], [29] also consider saving the VM disk states. Goiri *et al.* [8] 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. Mirage [23] proposes a new VM image format that includes semantic information, and addresses version control of VM images as in our work. It requires the knowledge of the underlying file system semantics in the VM image to construct VM versions, while CloudVS directly scans for the changes in the VM image file using RE and is generic to any file systems inside the VM. Foundation [24] is an archiving system for VM disk snapshots. To save storage space, it leverages deduplication to eliminate the storage of redundant data blocks. Nicolae *et al.* [14] propose a distributed versioning storage service to store VM snapshots. On the other hand, CloudVS focuses on the performance issues when RE is applied in versioning under commodity settings. Note that CloudVS supports tunable storage of VM versions, which is not addressed in prior work.

VM migration. VM migration [3], [13] is to move a running VM across different physical hosts over the network. Both studies [3], [13] 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.* [9] use a post-copy approach to speed up the migration. CloudNet [32] can migrate both memory and disk states over the Internet, using content-based RE to minimize the migration bandwidth.

RE techniques. RE is used in many applications in minimizing redundant data, such as data forwarding (e.g., [32]) and data storage (e.g., [11], [22], [24], [30]). 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.

6 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 leverages 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 further propose a multi-VM versioning mechanism and extend

CloudVS to support the QCoW2 image format. We evaluate the performance of CloudVS via real-world VM image traces and conduct extensive testbed experiments to validate the effectiveness of CloudVS. Our work provides an applied study on how to build 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**.

Acknowledgments

An earlier 8-page conference version of this paper appeared in IEEE/IFIP NOMS'12 [27]. This work was supported in part by grant GRF CUHK 413813 from the Research Grant Council of Hong Kong.

References

[1] P. Anderson and L. Zhang. Fast and Secure Laptop Backups with Encrypted Deduplication. In *Proc. of USENIX LISA*, 2010.

[2] 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.

[3] 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.

[4] A. Clements, I. Ahmad, M. Vilayannur, and J. Li. Decentralized Deduplication in SAN Cluster File Systems. In *Proc. of USENIX ATC*, 2009.

[5] J. R. Douceur, A. Adya, W. J. Bolosky, D. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proc. of IEEE ICDCS*, 2002.

[6] C. Dubnicki, L. Gryz, L. Heldt, M. Kaczmarczyk, W. Kilian, P. Strzelczak, J. Szczepkowski, C. Ungureanu, and M. Welnicki. HYDRAstor: A scalable secondary storage. In *Proc. USENIX FAST*, Feb 2009.

[7] Eucalyptus. Eucalyptus Command Line Interface Reference Guide. http://www.eucalyptus.com/docs/3.1/cli/euca-bundle-instance.html.

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

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

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

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

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

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

[14] 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.

[15] 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.

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

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

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

[19] QEMU. Qcow2 specification. https://github.com/qemu/QEMU/blob/master/docs/specs/qcow2.txt.

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

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

[22] 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.

[23] 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.

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

[25] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage. Hey, You, Get Off of My Cloud: Exploring Information Leakage in Third-Party Compute Clouds. In *Proc. of ACM CCS*, 2009.

[26] K. Shvachko, H. Kuang, S. Radia, and R. Chansler. The Hadoop Distributed File System. In *Proc. of IEEE MSST*, May 2010.

[27] C. P. Tang, T. Y. Wong, and P. P. C. Lee. CloudVS: Enabling Version Control for Virtual Machines in an Open-Source Cloud under Commodity Settings. In *Proc. of IEEE/IFIP NOMS*, 2012.

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

[29] 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.

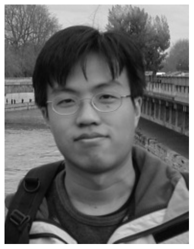
[30] M. Vrable, S. Savage, and G. M. Voelker. Cumulus: Filesystem backup to the cloud. *ACM Trans. on Storage (ToS)*, 5(4), Dec 2009.

[31] J. Wei, H. Jiang, K. Zhou, and D. Feng. MAD2: A scalable high-throughput exact deduplication approach for network backup services. In *Proc. of IEEE MSST*, 2010.

[32] 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.

[33] 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.

Chung Pan Tang received the BSc. degree in Physics from the Chinese University of Hong Kong in 2011. Currently he is a M.Phil. student of the Department of Computer Science and Engineering at the same school. His research interests include cloud computing and data deduplication.



Patrick P. C. Lee received the B.Eng. degree (first-class honors) in Information Engineering from the Chinese University of Hong Kong in 2001, the M.Phil. degree in Computer Science and Engineering from the Chinese University of Hong Kong in 2003, and the Ph.D. degree in Computer Science from Columbia University in 2008. He is now an assistant professor of the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include distributed systems, computer networks, cloud computing, and data storage.

Tsz Yeung Wong received his Ph.D., M.Phil., and B.Sc. degrees all from the Department of Computer Science and Engineering at the Chinese University of Hong Kong in 2007, 2002, and 2000, respectively. He is now a lecturer of the Department of Computer Science and Engineering at the Chinese University of Hong Kong. His research interests include computer and network security and operating systems.