September 2, 2014

RT0963
Computer Science 11 pages

Research Report

Clonable and space-efficient images of running VMs for fast provisioning on IaaS cloud platforms

Yohei Ueda, Toshio Nakatani

IBM Research - Tokyo
IBM Japan, Ltd.
NBF Toyosu Canal Front Building
6-52, Toyosu 5-chome, Koto-ku
Tokyo 135-8511, Japan

Improving the Time for Provisioning using Snapshots and Hibernation

Yohei Ueda
IBM Research - Tokyo
yohei@jp.ibm.com

Toshio Nakatani
IBM Research - Tokyo
nakatani@jp.ibm.com

ABSTRACT

It is critical to minimize the time for provisioning the virtual machines (VMs) in the cloud environment, particularly because more dynamic deployment of cloud workloads, such as Hadoop jobs and auto scaling of Web servers, is becoming popular. Provisioning consists of transferring a disk image of the operating system and the target application from the image repository to a disk of the host machine and booting the VM from the disk to the memory of the host machine. If we can preload (cache) the disk images of popular (frequently-used) operating systems and applications as the base images at each host machine in advance and maintain the delta (only the updated part of the selected base image) as we go, we can greatly reduce the time for transferring and the disk space. To that end, we propose to use the copy-on-write snapshot mechanism of logical volume manager of Linux. Furthermore, if we can save the memory image of the VM from the host machine to the image repository when we shutdown the VM, we can greatly reduce its booting time when we resume it on a different host machine for the next time. To that end, we propose to use the hibernation mechanism of Xen hypervisor. With these two proposed techniques, we have successfully achieved 28X and 8X speedup in provisioning a single VM and 64 VMs in our experiment, respectively.

1. INTRODUCTION

The success of the Amazon Elastic Compute Cloud (EC2) shows the growing demand for cloud computing. There are also various open-source cloud platforms such as OpenNebula [11], Eucalyptus [8], and Nimbus [6], which are available for creating one's own cloud environment. It is critical to reduce the time for provisioning the virtual machines (VMs) in these cloud environments, particularly because more dynamic deployment of cloud workloads, such as Hadoop jobs and auto scaling of Web servers, is becoming popular.

For example, Hadoop is commonly used in cloud environments to speed up various computations by parallelizing them on many compute nodes. Even though using hundreds of virtual machines cut the execution time to several minutes, this would be pointless if provisioning the virtual machines takes several hours.

Another example is auto-scaling of Web servers. Cloud providers such as Amazon EC2 offer a mechanism to increase or decrease the number of virtual machines automatically depending on certain metrics such as CPU utilization. Using this mechanism, a cloud user can build a website that scales to the volume of its incoming Web requests. However, if provisioning virtual machines takes a long time, auto-scaling cannot react quickly enough to handle sudden increases in the traffic.

Provisioning a VM generally consists of (1) selecting a host machine, (2) transferring a VM disk image from an image repository to a host machine, (3) customizing the VM disk image, and (4) booting a VM from the VM disk image. To reduce overall VM provisioning time, we need to improve the performance of these steps.

Step 2, which is the most time-consuming one of the four steps, can be skipped by preloading the VM disk image at the host machine before user's provisioning request. Thus, image preloading greatly reduces time to provision VMs, but it is usually impossible to preload all of the VM disk images in advance due to a limited storage space at each host machine. Only a subset of the images can be preloaded, so when preloaded images do not match user's request, the cloud system cannot skip Step 2, and needs to fetch a large VM disk image from the image repository. Therefore, the key to reduce the time of Step 2 is to increase the hit ratio of preloaded images.

To increase the hit ratio, we eliminate redundancy among VM disk images. Cloud users usually create their own custom VM images based on some popular basic VM images such as Red Hat Enterprise Linux 5.5 or Windows XP. This means that there are similarities among VM images whose origin is the same base image. We only preload such base VM disk images, and do not preload individual custom VM images. The delta of a custom image to its base image is stored as a *diff* disk image at the image repository. When there are many custom images based on a single base image, we only need to preload the single base image, so we can reduce disk space for image preloading. Better disk utilization increases the number of images that can be preloaded at each host, and improves hit ratio of image preloading, so redundancy elimination of images will improve provisioning performance.

We implemented this redundancy elimination by extending copy-on-write (COW) snapshots of the Linux logical volume manager (LVM). In our system, a VM disk consists of a *base* and *diff* disk images. The diff disk image is actually a COW data of a LVM snapshot, and contains the delta to the base disk image. Both base and diff images are stored on the image repository, only the base images are preloaded at each host machine. The diff images are usually small, so copying a diff image from the repository to a host machine does not take long time.

To provision a VM from a custom VM image, our system fetches a diff disk image from the repository to a host machine where its base disk image is preloaded. Then, the custom image needs to be reconstructed from the base and diff disk images in Step 3. A naive implementation of image reconstruction may cause heavy disk I/O and degrade the performance of provisioning. In our system, an LVM snapshot volume is set up using the base and diff images, so explicit disk image reconstruction is not necessary. Setting up a snapshot volume takes very small amount of time, and does not degrade the performance of Step 3.

After optimizing Steps 2 and 3 by image preloading and redundancy elimination, Step 4 becomes a bottleneck in provisioning. Booting a VM usually takes several minutes, and requires extensive disk I/Os, straining the I/O capacity of the host machines and affecting the performance of other VMs on the same host machines.

Most modern hypervisor systems such as Xen[1] provide a mechanism to hibernate a running VM into a VM state file and to resume from the file, so a natural approach is to utilize this mechanism to skip the time-consuming boot process. Our method to clone VM state files eliminates the time needed for booting VMs.

Note that cloning VM disk images and cloning VM state files are different concepts. Cloning VM disk images only require copying of disk images of VMs. Cloning VM state files requires copying of VM states such as memory contents and processor states of running VMs. To clone a booted VM, we need cloning of the VM state file.

It is difficult to create clonable VM state files since the VM includes stateful data such as IP address, which means we cannot resume many distinct VM from a single VM state file. We developed a mechanism to eliminate such stateful data from the VM state file, and this allow us to skip the boot process in VM provisioning. This provides faster provisioning and decreases the interference with other VMs running on the same host machine.

In usual cloud system, a VM image consists of a VM disk image and some metadata about the VM. In our proposed system, a VM image contains a VM state file in addition to a VM disk and metadata. This VM state file is used to skip booting from the VM disk image.

Table 1: Comparison of VM provisioning systems

	Preloading	Diff	Cloning
Eucalyptus	Yes (caching)	No	No
OpenNebula	No	No	No
Nimbus	No	No	No
Ours	Yes	Yes	Yes

Preloading: VM image preloading, Diff: Image deduplication using diffs, Cloning: VM state cloning

In summary, our contributions for fast VM provisioning are (1) fast transfer of VM disk images utilizing COW data of LVM snapshots, and (2) fast booting of VMs by cloning VM state files. With these techniques, we achieved 28 times faster provisioning of a VM, and 8 times faster provisioning of 64 VMs than the base implementation.

2. RELATED WORK

Table 1 shows the comparison of OpenNebula [11], Eucalyptus [8], Nimbus [6], and our system in terms of fast provisioning. Eucalyptus has caching mechanism of VM disk images, and uses cached VM disk images without fetching them from the image repository when provisioning VMs. This reduces provisioning time when cached images are available, but Eucalyptus does not care about redundancy of disk images. When a user requests a new VM of an image that are a little different from another image that is already cached, the entire image must be fetched from the image repository even thought most of the contents of the cached and newly fetched images are common. So the image store may contain redundant data, and such redundant data degrade the cache hit ratio. None of Eucalyptus, OpenNebula and Nimbus have a mechanism for boot time reduction using VM state cloning. Thus, every time a user request a new VM, the VM must be booted from its virtual boot disk.

Mirage[9] provides a space efficient image repository where redundant data among images are stored only once. Mirage examines every file of each disk image, and checks whether the content of each file is already stored in the image repository. If a file is already stored, the file is not stored again in order to save storage space. When a user requests a new VM, a VM disk image is constructed by gathering all necessary files in the repository. This mechanism reduces necessary storage space for virtual disk images, but this image construction consumes disk I/O capacity of the repository server, and takes non-negligible time.

In our system, such construction time is not necessary since diff disk images are COW data that can directly work with the LVM system, and the resulting logical volumes already contain usable file systems. Mirage achieves more space efficiency than our system, but we did not employ Mirage's approaches since we place more importance on provisioning speed than space efficiency. Space efficiency of the same level to Mirage is our future work.

Hypervisors such as Xen, KVM, and VMware support various types of virtual disks including raw disk devices, regular files, QCOW2, and VMware disk images. QCOW2 and VMware are virtual disks developed for virtual machines, and have enhanced features such as lazy allocation and copy-on-write snapshots.

Even with raw disk devices, we can use copy-on-write and lazy-allocation through the Linux Logical Volume Manager (LVM). LVM provides copy-on-write snapshots of logical volumes. Therefore, we can achieve the same level of functionality among LVM, QCOW2, and VMware disk images.

We used raw disk devices with LVM, because there are no large difference among the three formats, and LVM is available for all Linux host machines. Since we used block-based VM images, the contents of the LVM volume is stored into a VM image as it is.

SnowFlock[7] provides a VM fork mechanism, which clones a virtual machine into multiple replicas running on different host machines. Like the fork() system call in UNIX, all replicas share the same initial state, and then run independently. VM fork enables parallel programming in a cloud environment in a way similar to UNIX mulitprocess programming through the SnowFlock VM API. The implementation of SnowFlock is efficient, so it would beneficial to integrate some of their techniques into our system, but this would not be trivial since they resolve the problem of conflicts in the duplicated unique data of VMs (such as IP addresses) by using network isolation. Forked virtual machines on SnowFlock can communicate with one another only via the SnowFlock API. This restriction is not compatible with our purposes.

Sun et al[12] devised fast live cloning of virtual machines using copy-on-write techniques for memory and disks. They resolved the problems of the I/O interfaces of the cloned virtual machines by relying on the OS-level recovery process for broken devices. The operating system of a cloned virtual machine is notified of the loss of its interface devices when the VM is ready to restart. The recovery process is expected to reinitialize the interface devices of the cloned VM so as to avoid conflicts with its parent VM. However, this reinitialization sometimes leads to inconsistent states. For example, if an application stores an IP address information and internally opens files on the data disk , then it may become inconsistent after reinitialization. Our approach is cleaner, since there is no network interface and no data disk when the VM is cloned, so which avoids inconsistent states among such devices.

There have been various proposals for optimal VM placement on cloud computing environments [3, 4, 2, 5]. These systems manage VM placement so that some resources are optimally utilized such as hardware resources or power consumption. Our system only uses a simple round-robbin algorithm for VM placement, and lacks a smart placement mechanism. We could refine our system along with a smarter placement mechanisms.

3. SYSTEM OVERVIEW

There are various trade-offs among provisioning performance and application performance. As can naturally be anticipated, disk and network I/O are the most important factors that affect provisioning time, so we can obviously reduce the time to provision virtual machines by hiding I/O latencies or by reducing the amount of I/O. However, some of the techniques for fast provisioning degrade the performance of the applications running on the provisioned VMs. We will focus primarily on this aspect for our system design.

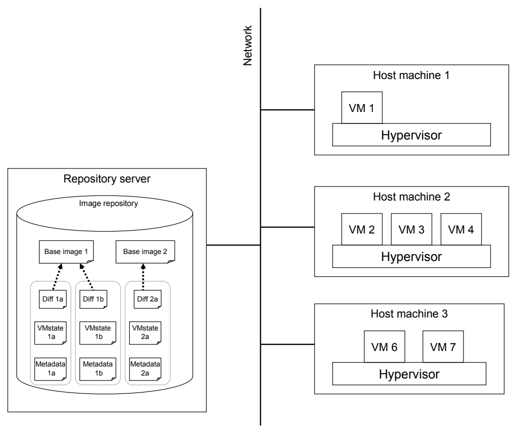

Figure 1: System overview

For example, if we share VM disks on a central image repository server via NFS, we can eliminate image transfer between the image repository and host machines. This NFS configuration degrades the disk I/O performance of provisioned VMs, so we did not use this approach.

Provisioning activities also interfere with applications in the other virtual machines running on the same host machine due to I/O and CPU contention, so minimizing the interference is a secondary focus of our system design.

Figure 1 depicts an overview of our system. There is a central VM image repository, and multiple host machine that run hypervisors to host VMs. In our system, a VM image consists of a VM state file, a diff image of the root disk, the name of base disk image, and metadata. When a user requests a new VM from a VM image, the VM image is transferred from the repository to a host machine, and then a new VM is created from the image at the host machine.

3.1 Virtual machines

Figure 2 shows the logical view of a VM in our system. From cloud users' perspective, each VM in our system has one or more virtual processor cores (VCPU), main memory, one virtual network interface, and two virtual disk interfaces. The network interface is connected to an external Ethernet network via Ethernet bridge in the host machine. The first disk interface is connected to a *root disk*, and the second one is connected to a *data disk*.

A root disk is a virtual disk that contains the files for a bootable operating system. For example, for the Windows operating system, the entire C drive is a root disk. So a root disk contains OS's kernel, device drivers, default application programs, optimal application programs, and user data.

A data disk is prepared for the user's additional data and software. Typical sizes of data disks are 100GB or larger

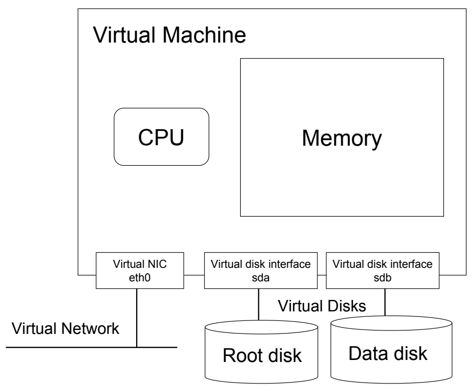

Figure 2: Logical view of a virtual machine

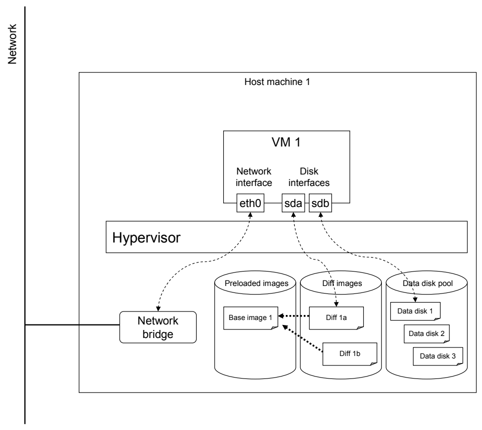

Figure 3: Host machine

whereas a typical root disk can be around 10GB. Initially, a data disk is formatted as a empty filesystem for use as needed. Even though a data disk is formatted for one filesystem, users may destroy for a database partition, swap partition, or any purpose that doesn't require its original filesystem.

3.2 Image repository

In the central image repository server, base disk images and diff disk images are stored as normal files in the filesystem on its local disk. These images can be fetched from host machines via HTTP. An HTTP server runs on the repository server, and a host machine can retrieve an image using HTTP.

Each base image is stored as a single file, and its metadata is stored in a separate file. Each diff disk image is also stored as a single file, and its metadata file contains the name of its base disk image. VM state files are also stored in the repository server. The details of VM state files are described later.

3.3 Host machines

Xen hypervisor runs on each host machine to host VMs. An HTTP server also runs on each host machine to exchange base disk images, diff disk images, and VM state files. Each host machine has storage space for preloaded base disk images, provisioned diff disk images, and a data disk pool.

Each diff image is paired with its base disk image, and a LVM snapshot volume is set up for each diff image. The snapshot volume is attached to a VM as a root disk. Since a root disk mainly contains operating system files and optionally user applications, the root disk is primarily read-only and rarely updated. Thus, we can use lazy-allocation of LVM snapshots for root disks without sacrificing VM performance.

Each data disk is a regular LVM volume. It is empty at first and is formatted as a normal filesystem. It is not a part of any VM disk image, so we do not need to transfer the data disk content from the image repository to the host machines.

Splitting virtual disks as root and data disks has another benefit. Cloud providers typically offer several types of VMs with difference disk sizes. If we use block-based VM disk images with single virtual disk, then we need to resize a filesystem in the VM disk image to support multiple disk sizes. Resizing a filesystem is time-consuming and should be avoided.

The data disks are empty at first, so we can create and pool the data disks before provisioning, and attach one from the pool to each new VM. Pooling data disks hides the time to format data disks from the VM provisioning. To support multiple data disk sizes, we can prepare a data disk pool per disk size.

4. IMAGE TRANSFER TIME REDUCTION

Since we do not use shared storage such as NFS, we use image preloading to accelerate the provisioning without shared storage. One problem with image preloading is the disk consumption. There may be many VM images of various kinds, and it is difficult to preload all of the VM disk images on a host machine at one time. The solution to this problem is deduplication of the similar images. We can reduce the size of the image store if we can eliminate the duplicates of the same content in multiple images in the image store.

4.1 Image redundancy elimination

In most cases, cloud users create their own custom images based om some VM images such as Red Hat or Windows, so if we provide these basic VM disk images, the difference of a base image and a customized image based on it is generally relatively small compared to a full base image.

If we preload base disk images, and transfer only the difference between the base and customized images, we can reduce the disk space necessary for custom disk images, and also reduce the time for copying images from the central repository to a target host machine as long as the base disk images are already preloaded on the target host machine. We call the difference of base and custom disk images *diff*, and we use diff images for VM disk images. We use copy-on-write (COW) data of LVM snapshot volumes for creating diff images.

When the system receives a capture request, it creates a new root disk from the root disk of the base image. The new root disk is an LVM snapshot volume, so there are an original volume and COW data. Figure 4 shows the structure of a Linux LVM snapshot.

4.2 LVM snapshots

LVM provides abstraction of logical volumes over physical disks. LVM groups multiple physical disks, and provide a logical volume with an arbitrary size from a portion of the grouped disks logical volumes. LVM reduces burden of complicated disk management.

LVM also provides a mechanism for volume snapshots, which can be used for consistent backups. When a user plans to back up a large logical volume on line, the consistency of the backup is usually required. Backing up a large volume takes long time, and the files on the volume may be altered during the backup. To avoid this inconsistency, LVM provides a snapshot of a logical volume using copy-on-write (COW) mechanism.

In each host machine, LVM is installed with free space in a volume group that is sufficient to hold the root and data disk images necessary for the incoming requests for VMs.

4.3 Root disks

Each root disk is an LVM snapshot volume, which consists of an original volume and a COW volume. First, a new original volume, which is a empty logical volume, is created with the same size as the a base disk image, and then the content of the base disk image is copied into that volume. Second, if a diff disk image is being used, then another logical volume is created as a COW volume and the content of the diff disk image is copied into it. If no diff disk image is needed, the COW is created but left empty. Finally the system uses the dm_setup command (of the Linux Device Mapper) to tell Linux that the original and COW volumes are used as an LVM snapshot volume. The structure of each diff disk image is the same as a COW volume of LVM, so we can simply copy the content of a diff disk image into a COW volume without any conversion.

The structure of a COW volume is shown in Figure 4. The original volume is split into fixed-sized chunks, each of which is typically 4,096 bytes. When a user tries to write data on a chunk, the operating system copies the content of the chunk to the COW volume, and the data is written into the copied chunk on the COW volume. The chunk of the original volume remains intact. The COW volume has data structures that manage the relationships between a chunks on the original volume and the corresponding chunks in the COW volume. The amount of COW data grows as users write data to the snapshot, and the total COW volume size will be the original volume size plus the sizes of chunks that are needed to manage new COW chunks.

We use COW data for diff images. This is called block-based management of contents of VM disk images. Another approach is file-based management. In block-based management, the contents of VM disk images are partitioned in

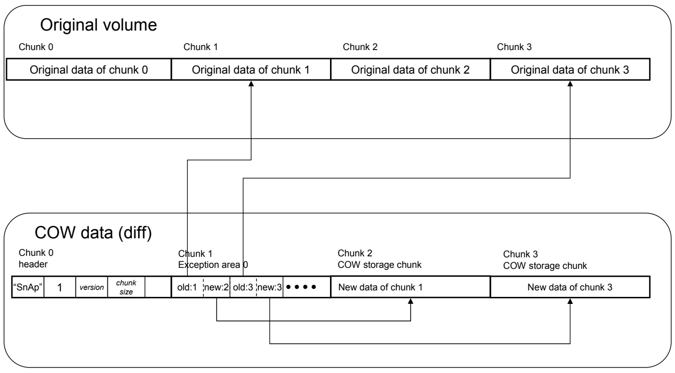

Figure 4: LVM snapshot volume

fixed-sized chunks, and the management of copy-on-write is done at the level of these chunk units. Thus, no knowledge of filesystem is required to create diff images.

In file-based management, all files in file systems of virtual disk images are recognized, and the files that are identical between multiple VM disk images are stored only once in the image repository. Even thought this approach dramatically reduces necessary storage space for virtual disk images, it has a drawback in terms of fast provisioning. Files are stored as independently in image repository of file-based management, and a VM disk image is constructed by gathering all necessary files in the repository. This image construction consumes disk I/O capacity of the repository server, takes non-negligible time.

In block-based management, such construction time is not necessary since diff disk images are COW data that can directly works with LVM systems, and the resulting logical volumes already contain usable file systems. We give more importance to fast provisioning than storage space efficiency of the image repository, so we chose the block-based approach.

4.4 Image preloading

Image preloading is done at installation of a new host machine, and image eviction of preloaded images is done when storage of a host machine becomes full.

During the initialization of image store on a new host machine, the base images are selected for preloading so that the expected time to transfer an image is minimized. We can calculate the expected transfer time from the storage size of the host machine, image sizes, number of running VMs, and other factors. Images to be evicted are also selected in a similar way. We describe these algorithms in Appendix A.

5. BOOT TIME REDUCTION

Booting a VM from a disk image takes a certain amount of time and consumes disk I/O bandwidth. We avoid the booting process by cloning the VM state of a running VM. Every VM image in our system contains a VM state file as well as a disk image, and provisioning a VM is resuming a VM from the VM state file.

5.1 VM state files

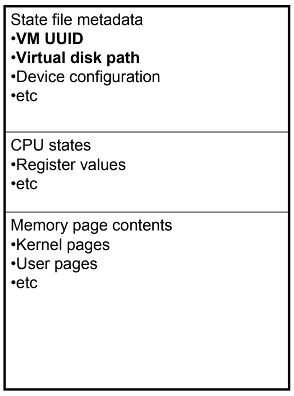

Figure 5: A VM state file

In cloud services such as Amazon EC2, VMs are provisioned from VM disk images. VM disk images are virutal disks including bootable OS files, and does not contain memory and processor state, which can be captured as VM state files with hibernation mechanism of hypervisors.

Figure 5 shows the contents of a VM state file. There are state file metadata, CPU states, and memory page contents. The state file metadata contain an identity information such as VM ID and VM name. The metadata also contains configuration information such as virtual disk path and device configuration. The contents of the virtual disk are not in the VM state file, and stored in a external file, so the metadata only contains the path of the external VM disk file on the host machine.

To clone a VM state file, we need to care about these identity information. In our system, the identity information is regenerated and updated to avoid conflicts with the original VM.

A VM state file contains CPU states and memory page contents. These areas also contain unique data of the original VM, and we need to care about such unique data to avoid conflicts, but it is almost impossible to directly update these areas in the same way for the metadata. CPU states and memory contents look like a black box from outside of the VM, and it is impossible to manipulate them without internal knowledge of the OS and applications running on the VM, so there are no general way to update the CPU and memory contents while keeping the consistency of the behaviors of the OS and applications.

We employed a different approach to eliminate unique information in the CPU state and memory contents of a VM state file.

5.2 Elimination of unique data

To clone a VM using a VM state file, we need to handle the unique data inside and outside of the VM state file such as assigned IP addresses, VM names and VM ID. We also need to handle the internal state of applications running on the VM, but handling such data is difficult since the application states are stored in the memory dump of the VM state file. It is usually very hard to automatically alter the state of an application in the memory dump.

Our proposed approach is to capture VM state just after the VM is booted. The initial state of a VM is identical when the VM is booted multiple times. The VM state should be captured at the end of boot process to make it reusable for cloning.

In our approach, the network interfaces are not attached to the VM while booting, so the captured VM state file does not contain any IP address information. This means it is possible to reuse the VM state file without conflicting IP addresses among the cloned VMs. We can also copy the VM state files for multiple distributed host machines via a network to reduce the time to provision many VMs in parallel.

The disk interface of the data disk is also detached while booting. Even though the data disk is initially empty, it should not be attached at first. If the data disk is attached to a suspended VM, the resulting VM state file contains filesystem data and the file cache of the data disk. To avoid dealing with that data, the data disk is attached only after resuming from the initialized VM state file.

One important point in capturing a VM state is how to determine when the booting process is done. There is no obvious point for the end of a boot process observable from outside of the VM. Our approach is to monitor the console output until a login prompt appears. The login prompt usually appears when the boot process finishes, so we can use this as the effective end of boot process.

Some network daemons will be started during the boot process. They are invoked without any network interfaces, but they can usually initialize their network services without problems. They do this by initializing their network sockets not for specific network interfaces or IP addresses, but for a wildcard interface, which is not related to any specific network interfaces or IP addresses. Some network daemons may try to obtain an IP address from the machine they are running on, but there is no standard way to do this, so such daemons are considered noncompliant.

Other network daemons may accept fixed IP addresses specification in their configurations. But using fixed IP addresses in their configuration files will not work in general, because IP addresses are normally assigned dynamically (via DHCP) in cloud environments unless static IP addresses are reserved in advance. Thus, configurations that do not rely on fixed IP addresses are preferred for reusable VM images in cloud environments. Linux's udev automatically detects newly attached devices, and configures them. If the DHCP setting is enabled, then a new IP address will be assigned to a newly attached network interface.

We confirmed that our method to clone VM state files works for various applications including Hadoop, Apache, MySQL, Samba, IBM WebSphere Application Server, and IBM DB2 database. Some of them did not work with their default configuration, but we only needed to apply some small configuration changes. This kind of configuration techniques are also common for other cloud computing environments such as Amazon EC2.

The details of how to clone VM state files are described in the next section.

6. VM PROVISIONING AND CAPTURING

In this section, we describe our VM image management processes in detail.

6.1 Provisioning process

Figure 7 shows four possible scenarios for provisioning a VM. The case 1 is the most simple case where a VM is provisioned as a base image without preloading. In this case, the entire base image is fetch from the repository server to the host machine.

In case 2, a VM is provisioned from a custom image and its base image is preloaded at the host machine. In this case, only the diff image is fetched from the repository server to the host machine.

In case 3, a VM is provisioned from a base image and the image itself is preloaded at the host machine. Additionally, a VM state file of the image is available. In this case, only the VM state file is fetched from the repository server to the host machine.

In case 4, a VM is provisioned from a custom image and is base image is preloaded at the host machine. Additionally, a VM state file of the image is available. In this case, both the diff image and the VM state file are fetched from the repository server to the host machine.

In our proposed system, the case 4 is the most common case, and we describe the provisioning process of this scenario below.

1. The system selects a host machine.
2. The system generates a new VM ID.
3. The system generates a new VM name.
4. The system creates a new directory for the new VM to store the virtual disks and other data.
5. The system fetches the base disk image from the repository (if it is not already available on the host machine).
6. The system fetches the diff disk image (COW data and VM state file) from the repository
7. The system modifies the VM state file with the VM ID, VM name, and root disk path.
8. The system resumes the VM from the modified VM state file.
9. The system attaches a network interface (NIC) to the running VM, and the operating system in the running VM automatically recognizes the newly attached NIC and configures it.

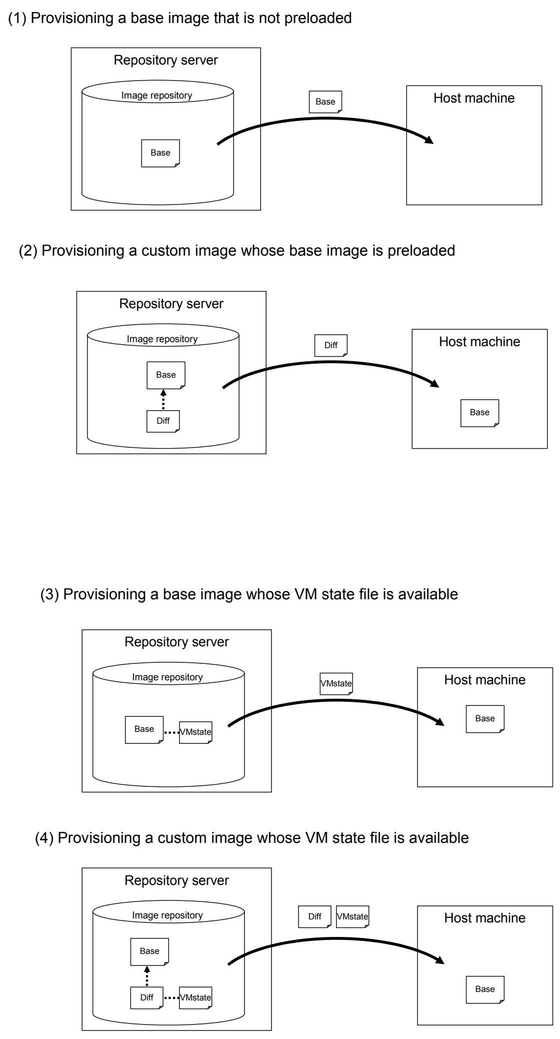

Figure 6: Provisioning scenarios

10. The system attaches a data disk to the running VM, and then the operating system in the running VM automatically recognizes the newly attached data disk, and configures it.
11. The system allocates the defined amount of memory to the running VM, and then the operating system automatically recognizes the new memory.

Note that the step 5 is not necessary if the base disk image is already preloaded at the host machine. In step 7, only the metadata of the VM state file is modified, and its memory content is not modified.

Steps 9, 10, and 11 are usually OS-independent since every modern OS has a mechanism to detect dynamically attached devices.

6.2 Capture process

Figure 7 shows four possible scenarios for capturing a VM. Case 1 is the most simple case where a VM is captured as a base image without preloading. In this case, the entire base image is stored from the host machine to the repository server.

In Case 2, a VM is captured as a custom image. In this case, only the diff image is stored from the host machine to the repository server.

In Case 3, a VM is captured as a base image with its VM state file. In this case, only the VM state file is stored from the host machine to the repository server.

In Case 4, a VM is captured as a custom image with its VM state file. In this case, both the diff image and the VM state file are stored from the host machine to the repository server.

In our proposed system, the case 4 is the most common case, and we describe this capturing process of this scenario below.

1. The system copies the VM definition file, and modifies the copied file so that there the VM has no network interfaces or data disk, and only have a minimum memory size.
2. The system boots a new VM using the modified VM definition file.
3. After the booting process is finished. the system hibernates the VM to create its VM state file.
4. The system copies the VM state file and the COW data to the central repository.

This capture process contains the key technology of our system. In usual cloud systems, capturing VM only stores the contents of VM disk images. In our system, the memory content is also captured to reduce boot time.

Capturing VM memory content requires fixing of the unique data of each VM instance. A long-running VM tents to have

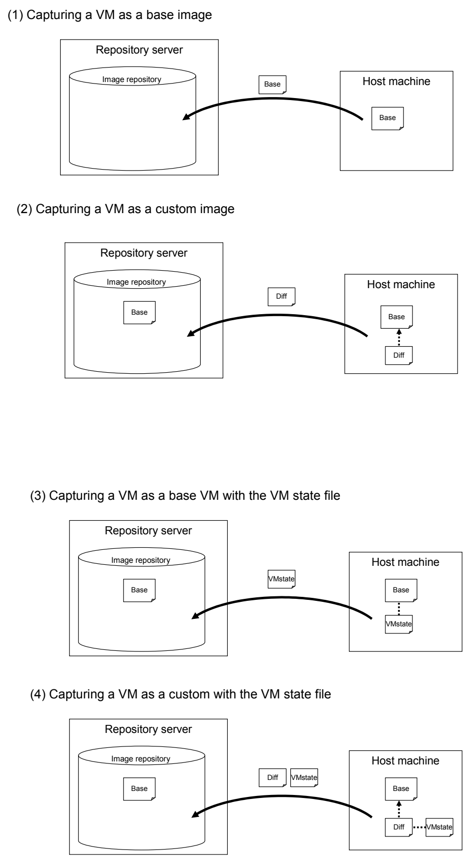

Figure 7: Capturing scenarios

various unique data, which make it difficult to capture the VM as a reusable VM image. Instead of capturing long-running VMs, we capture VMs just after booting finishes. We also detach network interfaces before booting to remove unique data such as IP addresses.

7. EXPERIMENTS

We implemented our provisioning mechanism with base and diff images, and evaluated its performance.

7.1 Experimental setup

We used 5 identical physical servers. One was for the repository server, and the other four were used as host machines. Our servers were IBM BladeCenter HS22, with dual sockets and Intel Xeon E5570 2.93GHz processors. A single Xeon processor has 4 cores, and each core has 2 hardware threads. Therefore, a total of 16 hardware threads are available in a single HS22. Each server had 24 GB of memory and a 10-Gb Ethernet card. For storage, each server had a 32-GB SATA SDD and a 500-GB SATA HDD. Red Hat Enterprise Linux 5.4 was installed on the SDD on each server. Cloud platform software was also installed on the SDD, but the image data for the VMs were stored in HDD. These blade servers were installed in a single IBM BladeCenter H chassis and connected via a 10-Gb Ethernet Switch.

We installed Xen 3.4.2 as the hypervisor on the host machines. Our software is mainly as Python scripts, and uses standard Linux software including httpd, ssh, wget, and the LVM tools. The program that updates the VM state files is written in C since it manipulates the binary data structures of the VM state metadata.

A single image was provisioned among the 4 host machines in a round-robbin. The base image was a standard installation of CentOS 5.4. The size of the base image was 10 GB. The diff image was created by installing an Apache HTTP Server and a PHP runtime. The diff image was 20 MB. Each VM had 1 VCPU, 2GB memory, a 10GB root disk, and a 2GB data disk. A VM state file was 512MB.

7.2 Experimental results

Figure 8 shows elapsed times for VM provisioning with the base implementation and with the improved implementation using our techniques. The number of provisioned VMs was 1, 2, 4, 8, 16, 32, and 64.

"Baseline" is the results without image preloading and VM cloning. That is, both the entire base and diff disk images were transferred from the central repository to each host machine, and the complete boot process was executed for each VM invocation. In this case, a data disk was also created for each VM invocation. This required 6 minutes for a single VM, and 11 minutes for 64 VMs. We had a single repository server and only 4 host machines, disk contention occurred in both the repository and host machines.

"Preloading" is the results with image preloading, but without VM cloning. That is, the base images were cached, so only the diff images was copied from the repository to each host machine. The data disks were pooled in advance. The complete boot process was executed for each VM invocation.

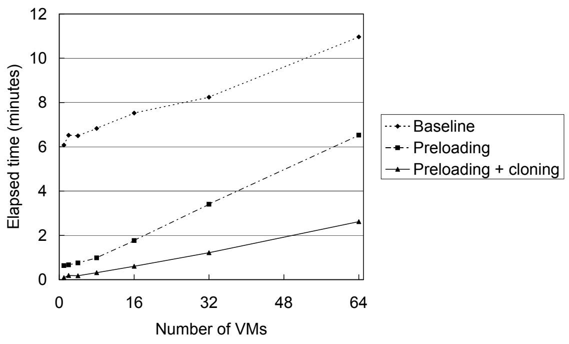

Figure 8: Provisioning time

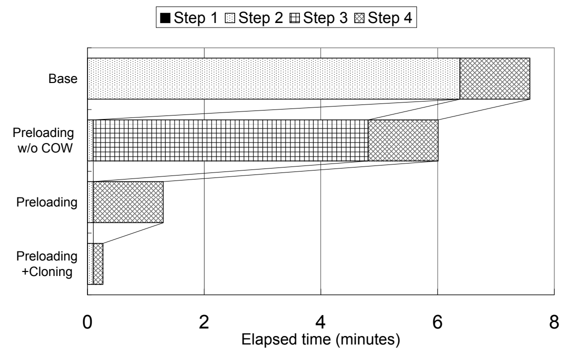

Figure 9: Breakdown of single-VM provisioning time

In this case, provisioning 1 VM took 38 seconds, since the time-consuming image copying was avoided by preloading the base image. However, the elapsed time became worse as the number of VMs increased. This was because we only had 4 host machines and boot process is disk-intensive, and thus disk contention occurred at each host machine.

"Preloading + cloning" is the results with both image preloading and VM cloning. In other words, the base disk images were preloaded, so the diff disk image and the VM state file were copied from the repository to each host machine. Boot processing was skipped by using the VM state file for each VM invocation. The data disks were pooled in advance. In this case, provisioning 1 VM took only 5.5 seconds. Most of the disk contention was eliminated by the image caching and VM cloning. However, we still saw performance degradation due to CPU limits as as we increased the number of VMs.

Figure 9 shows elapsed times for provisioning a single VM on the base implementation and on the improved implementations using our techniques. Step 1 selects a target host machine, and do some initialization works. Step 2 is the time to copy the image from the central repository to the target host machine. Step 3 is the time to customize the VM image. Step 4 is the time to boot each VM.

In the "Base" case, which was the result without our optimization techniques, the most time-consuming step was Step 2, which copies data from the repository server to the host machine. This means that the image preloading will be the most effective optimization among our techniques. Steps 3 and 4 also used siginificant amounts of times.

In the "Preloading w/o COW" case, an image diff is used, but not used as LVM COW data. This means we need to create a disk image by merging base and diff images. In this case, the most time-consuming step was Step 3, which merges the two images to a single usable VM disk image.

In the "Preloading" case, an image diff is used, and used as LVM COW data. This result shows that creating a disk image becomes very quick with COW. In this case, the most time-consuming step is now Step 4, which boots the VM from the VM disk.

In the "Preloading + Cloning" case, which was the result with our optimization techniques, Steps 2, 3, and 4 became very small, and the total elapsed time was 5.5 seconds.

8. CONCLUSION AND FUTURE WORK

In this paper, we proposed a new strategy for rapid provisioning of VMs in a cloud computing environment, and implemented a provisioning system based on it, and evaluated the system.

The key to fast provisioning is space-efficient and clonable images of running VMs. Space-efficient VM images using image diffs reduce the amount of data transferred from a VM image repository to the host machine to minimize the disk and network I/O bottlenecks. Clonable VM images skip the time for booting. Clonable VM images are implemented by eliminating stateful data on running VMs. With these techniques, we achieved 28 times faster provisioning of one VM, and 8 times faster provisioning of 64 VMs than the base implementation.

For future work, we will extend the level of our deduplication of images into that of file-based image repository such as Mirage[9]. Another future work item is evaluation with large-scale deployment. Our experimental setup was relatively small, so we will prepare larger environment to evaluate effectiveness of our image preloading mechanism under the situation where various VM images exist.

9. REFERENCES

[1] P. Barham, B. Dragovic, K. Fraser, S. Hand, T. Harris, A. Ho, R. Neugebauer, I. Pratt, and A. Warfield. Xen and the art of virtualization. In *SOSP '03: Proceedings of the nineteenth ACM symposium on Operating systems principles*, pages 164–177, New York, NY, USA, 2003. ACM.

[2] M. Cardosa, M. R. Korupolu, and A. Singh. Shares and utilities based power consolidation in virtualized server environments. In *IM'09: Proceedings of the 11th IFIP/IEEE international conference on Symposium on Integrated Network Management*, pages 327–334, Piscataway, NJ, USA, 2009. IEEE Press.

[3] B. Chen, N. Xiao, Z. Cai, Z. Wang, and J. Wang. Fast, on-demand software deployment with lightweight, independent virtual disk images. In *GCC '09: Proceedings of the 2009 Eighth International Conference on Grid and Cooperative Computing*, pages 16–23, Washington, DC, USA, 2009. IEEE Computer Society.

[4] L. Grit, D. Irwin, A. Yumerefendi, and J. Chase. Virtual machine hosting for networked clusters: Building the foundations for "autonomic" orchestration. In *VTDC '06: Proceedings of the 2nd International Workshop on Virtualization Technology in Distributed Computing*, page 7, Washington, DC, USA, 2006. IEEE Computer Society.

[5] F. Hermenier, X. Lorca, J.-M. Menaud, G. Muller, and J. Lawall. Entropy: a consolidation manager for clusters. In *VEE '09: Proceedings of the 2009 ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pages 41–50, New York, NY, USA, 2009. ACM.

[6] K. Keahey, M. Tsugawa, A. Matsunaga, and J. Fortes. Sky computing. *IEEE Internet Computing*, 13(5):43–51, 2009.

[7] H. A. Lagar-Cavilla, J. A. Whitney, A. M. Scannell, P. Patchin, S. M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan. Snowflock: rapid virtual machine cloning for cloud computing. In *EuroSys '09: Proceedings of the 4th ACM European conference on Computer systems*, pages 1–12, New York, NY, USA, 2009. ACM.

[8] D. Nurmi, R. Wolski, C. Grzegorczyk, G. Obertelli, S. Soman, L. Youseff, and D. Zagorodnov. The eucalyptus open-source cloud-computing system. In *CCGRID '09: Proceedings of the 2009 9th IEEE/ACM International Symposium on Cluster Computing and the Grid*, pages 124–131, Washington, DC, USA, 2009. IEEE Computer Society.

[9] 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 *Proceedings of the fourth ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, VEE '08, pages 111–120, New York, NY, USA, 2008. ACM.

[10] T. J. Rolfe. An alternative dynamic programming solution for the 0/1 knapsack. *SIGCSE Bull.*, 39(4):54–56, 2007.

[11] B. Sotomayor, R. S. Montero, I. M. Llorente, and I. Foster. Virtual infrastructure management in private and hybrid clouds. *IEEE Internet Computing*, 13(5):14–22, 2009.

[12] Y. Sun, Y. Luo, X. Wang, Z. Wang, B. Zhang, H. Chen, and X. Li. Fast live cloning of virtual machine based on xen. In *HPCC '09: Proceedings of the 2009 11th IEEE International Conference on High Performance Computing and Communications*, pages 392–399, Washington, DC, USA, 2009. IEEE Computer Society.

APPENDIX

A. IMAGE PRELOADING

To reduce the time to provision virtual machines, we need to preload the base images on each host. Since we use base and diff images, we need a preloading mechanism suitable for both of them. In this section, we describe our proposed preloading mechanism in detail.

When the image store of a host machine is initialized, popular base images are prefetched from the image repository. Popular base images are selected based on *popularity* of each base image. The popularity of a base image is the number of running virtual machines that were created from one of the diff images based on the base image. We assume the probability that a image will be used for future provisioning is proportional to the popularity of that image, and an image with a higher popularity is more likely to be preloaded

A.1 Selecting images to be preloaded

During the image store initialization, the base images are selected for prefetching so that the expected time to copy an image is minimized. We can calculate the expected copy time from the image store size, image sizes, number of running VMs, and other factors. We can efficiently solve this problem with an approximation algorithm for 0-1 knapsack problem [10].

We assume that the average throughput of image transfer is constant, so the average time of the transfer time of a VM image is proportional to its size. Transfer time of a VM image is different depending whether its base disk image is cached or not. If the base disk image is cached, only its diff disk image is transferred. Otherwise, both base and diff disks images are transferred. If the information about which images are cached is available, we can calculate the expected transfer time of a VM image using its popularity described above.

We assume the probability that a VM image will be used for future provisioning is proportional to the popularity of that image, so we calculate the expected transfer time of the VM image using the probability for cached and uncached cases.

Since we can calculate the image transfer time of a VM image for cached and uncached cases, we can examine all of combination of cached and uncached cases of every VM images in order to find which combination minimizes the average expected transfer time of a VM image for a newly created VM.

To find optimal solution by check all combination of cache and uncached cases is not tractable. However, we can use an approximation algorithm for 0-1 knapsack problem as described in detail below.

Let N be the number of images, and T be the average throughput for images. Then the expected time to transfer an image is

$$E = \sum_i \sum_{j \in D_i} \frac{n_j}{N} \frac{(1 - x_i)S_i + \Delta_j}{T}, \qquad (1)$$

where S_i is the size of image i, Δ_j is the size of diff image j, D_i is the set of the diff images based on base image i, n_j is the number of running VMs created from diff image j, and x_i is a Boolean variable indicating whether or not the base image i is cached. E can be transformed to

$$E = \frac{1}{NT}\left(\sum_i p_i S_j + \sum_i \sum_{j \in D_i} \Delta_j - \sum_i p_i S_i x_i\right), \qquad (2)$$

where the popularity $p_i = \sum_{j \in D_i} n_j$.

The first and second terms are constant because they do not contain x_i, so we need to minimize the last term. The last term has a minus sign, so only we need to maximize $\sum_i p_i S_i x_i$. The 0-1 knapsack problem to be solved is maximizing $\sum_i p_i S_i x_i$ subject to $\sum_i S_i x_i \leq W, x_i \in \{0, 1\}$, where W is the size of the image store.

A.2 Selecting images to be evicted

When a VM is provisioned, any uncached images are fetched from the image repository. If the image store is full, one or more images are selected for eviction so that the expected time to copy the image again is minimized. We cannot evict cached images that are being used by VMs running on the host machine, so victims are selected from the currently unused images.

We can also solve this problem by an approximation algorithm for 0-1 knapsack problems.

The expected time to transfer image i is

$$E = \sum_{i \in C-R} \sum_{j \in D_i} \frac{n_j}{N} \frac{(1 - x_i)S_i + \Delta_j}{T}, \qquad (3)$$

where C is the set of images cached on the target host machine, and R is the set of images from which the running VMs were created on the target host machine.

Therefore, the 0-1 knapsack problem to be solved maximizing is $\sum_{i \in C-R} p_i S_i x_i$ subject to $\sum_{i \in C-R} S_i x_i \leq W - S_k - \sum_{i \in R} S_i, x_i \in \{0, 1\}$ where k is the image to be provisioned.