PAPER

Efficient Update Activation for Virtual Machines in IaaS Cloud Computing Environments

Hiroshi YAMADA[†a)], **Shuntaro TONOSAKI**[††], ***Nonmembers***, ***and*** **Kenji KONO**[††], ***Member***

SUMMARY Infrastructure as a Service (IaaS), a form of cloud computing, is gaining attention for its ability to enable efficient server administration in dynamic workload environments. In such environments, however, updating the software stack or content files of virtual machines (VMs) is a time-consuming task, discouraging administrators from frequently enhancing their services and fixing security holes. This is because the administrator has to upload the *whole* new disk image to the cloud platform via the Internet, which is not yet fast enough that large amounts of data can be transferred smoothly. Although the administrator can apply incremental updates directly to the running VMs, he or she has to carefully consider the type of update and perform operations on all running VMs, such as application restarts. This is a tedious and error-prone task. This paper presents a technique for synchronizing VMs with less time and lower administrative burden. We introduce the *Virtual Disk Image Repository*, which runs on the cloud platform and automatically updates the virtual disk image and the running VMs with only the incremental update information. We also show a mechanism that performs necessary operations on the running VM such as restarting server processes, based on the types of files that are updated. We implement a prototype on Linux 2.6.31.14 and Amazon Elastic Compute Cloud. An experiment shows that our technique can synchronize VMs in an order-of-magnitude shorter time than the conventional disk-image-based VM method. Also, we discuss limitations of our technique and some directions for more efficient VM updates.

key words: *Cloud computing, IaaS, virtual machines, software updates*

1. Introduction

Infrastructure as a Service (IaaS), a form of cloud computing, is gaining attention for its ability to enable cost-effective server administration. Its pay-as-you-go pricing model allows companies of any size or individuals to easily start new services and to scale up and down the services as needed. At any given time, a cloud user may choose to run only enough servers to handle the load that the service is facing. Amazon Web Services (AWS) [1], a major IaaS provider, is an example. Anyone can start using Amazon's infrastructure immediately after registration, with no initial cost or monthly minimum fees.

The trend of cloud computing is derived from the progress of virtualization technology. Virtualization technology enables multiple virtual machines (VMs) to run concurrently on the same physical computer. Each VM is isolated from other VMs, enabling the user to customize the entire software stack within the VM. Users may install operating systems (OSes) and applications of their choice. Cloud computing especially utilizes *virtual disks*. A snapshot of a virtual disk, called a *virtual disk image*, can be easily created and transferred over the network. By transferring a virtual disk image of a VM's root disk, replicas of the VM can be easily launched at other parts of the cloud, notably for load balancing. Encapsulating the entire software stack as a virtual disk means the developer does not need to consider dependencies between software components.

Although virtualization provides a way to easily make replicas of VMs, updating the VMs is a time-consuming operation, discouraging administrators from frequently enhancing their services and fixing security holes. A typical method to update VMs is to upload the new virtual disk image to the cloud platform, and then launch new VMs from the uploaded disk image and terminate the old ones. When updating the VMs, the administrator has to upload the *whole* new virtual disk image to the cloud platform even if the update is quite small, such as one modified line in a configuration file. This is a time-consuming task since virtual disk images are quite large and the upload is performed over the Internet, which is not currently fast enough to smoothly transfer large amounts of data. Launching VMs also takes a considerably long time. In fact, instantiating a VM in Amazon Elastic Computing Cloud (EC2) typically takes minutes [2]. Although the administrator can apply incremental updates directly to the running VMs, this is a tedious and error-prone task since he or she has to consider the type of update and perform operations on all running VMs, such as restarting applications. These facts can lead to the delay of deploying new contents and, even worse, the unfixed vulnerabilities being exploited by network attackers.

This paper presents a technique for updating VMs in shorter time and lower administrative burden. The key idea is that we manage incremental update information, inspired by commodity version control systems. To efficiently apply update information to VMs, we introduce the *Virtual Disk Image Repository*, which updates automatically the virtual disk image and the running VMs with only the incremental update information. The Virtual Disk Image Repository runs on the cloud platform and receives the incremental updates from the administrator. The administrator needs to send only the updated files via the Internet since the Virtual Disk Image Repository creates a new virtual disk image from the sent files and registers it for subsequent launches of additional VMs. The Virtual Disk Image Repository trans-

Manuscript received June 12, 2013.
Manuscript revised October 13, 2013.

[†]The author is with Tokyo University of Agriculture and Technology, Koganei-shi, 184-8588 Japan.
[††]The authors are with Keio University, Yokohama-shi, 223–8522 Japan.
a) E-mail: hiroshiy@cc.tuat.ac.jp

DOI: 10.1587/transinf.E97.D.469

fers the updated files to the serving VM, and our mechanism automatically performs necessary procedures, such as restarting server processes, on the running VM, based on the types of files that are updated.

We implement a prototype on Linux 2.6.31.14 and Amazon EC2. An experiment shows that our technique can synchronize VMs in an order-of-magnitude shorter time than the conventional disk-image-based VM cloning. Moreover, our overhead measurements show that installing our system will not affect service quality. Although our system imposes about 30% overhead on the developer's environment, it imposes no observable overhead on public servers.

The rest of the paper is organized as follows. Section 2 gives a brief explanation of VM update methods in IaaS environments. Section 3 and 4 describe the design and implementation respectively of our VM synchronization technique. Section 5 shows our experimental results, comparing our technique with the conventional one, and also describing the overhead of our prototype. Section 6 discusses some limitations and future directions of our technique. Section 7 describes related work, and Sect. 8 concludes this paper.

2. Conventional Update Methods

A conventional method for updating VMs is to create a new disk image having the new desired software stack and then launch as many new VMs as necessary to replace the existing VMs. Launching new VMs to replace old VMs is an expensive but simple way of updating VMs. These procedures are shown in Fig. 1. The administrator first launches a new VM to use as a development environment. He or she makes necessary changes to the development VM then stops the VM and bundles a new disk image from it. Using the new disk image, the administrator launches new VMs that are used as serving VMs. The number of VMs being launched at this point is typically the same as the number of existing serving VMs running at that time. After all the new VMs have been launched, the configuration of the load balancer is changed so that incoming traffic is directed to the new VMs, not the old ones. After all these steps, the old VMs are terminated.

Although this method enables us to handle any kind of update in a uniform way, it takes a long and variable time. In updating a disk image, we have to send the *whole* new disk image to the cloud platform even if the file updates are quite small, such as one modified line in a configuration file or an updated binary file of a utility program. The upload of the new disk image is performed via the Internet, which is not yet fast enough that large amounts of data can be smoothly transferred. Some researchers report that shipping of an actual updated disk sometimes takes less time than disk image uploads via the Internet [3], [4].

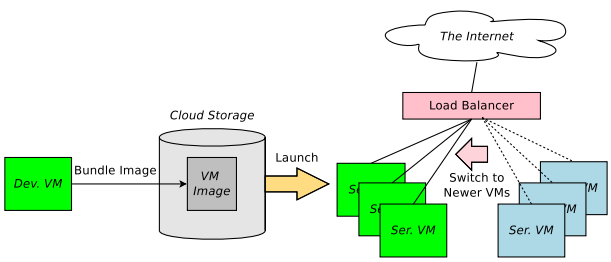

Fig. 1 Manual update.

In addition, launching new VMs takes a considerably long time. In the conventional method, we have to instantiate new VMs from the new disk image from scratch, even if the only thing needed to update the VM states is only to restart the application or store a new file in the disk. Typically, instantiating new VMs in Amazon EC2 takes minutes [2]. Although we could apply only incremental updates directly to the existing serving VMs by using `scp` and/or `ssh` instead of launching new VMs, this task is tedious and error-prone since we must carefully consider the type of update and perform operations on all the serving VMs to update their states, such as restarting applications and doing nothing.

3. Design

3.1 Overview

We present a technique to efficiently synchronize changes on a development VM to serving VMs. Our technique has three main characteristics. First, we send only the files that were updated on the development VM since the last synchronization, to minimize the network bandwidth usage and the time taken to transfer data. Second, we apply changes directly to the serving VMs instead of launching new VMs. This eliminates the time for launching new VMs. Last, we reduce the administrative burden of applying updates to all the serving VMs by preparing a kernel-level mechanism that supports automatic determination of what kinds of process operations are necessary regarding the updated files.

We introduce the *Virtual Disk Image Repository*, which aggregates the incremental update information from the developers and applies it to the disk image and the serving VMs. An overview of our technique is shown in Fig. 2. Our approach is inspired by recent versioning systems including pull and push models. However, simply applying these concepts to IaaS environments is difficult because the pull/push model propagates updates only to disks. In IaaS environments, we need to apply the changes to 1) running processes and 2) virtual disks to be used by subsequently launching VMs. When the administrator indicates that synchronization should start, newly created files, modified files, and the paths of deleted files are aggregated as delta information and then sent to the Virtual Disk Image Repository, a management node running on the cloud platform. From the delta information, the Virtual Disk Image Repository automatically creates a new disk image and registers it for subsequent launches of additional VMs.

The Virtual Disk Image Repository distributes the delta to all serving VMs to update their states. After a serving VM receives the delta, the delta is applied to its local virtual disk.

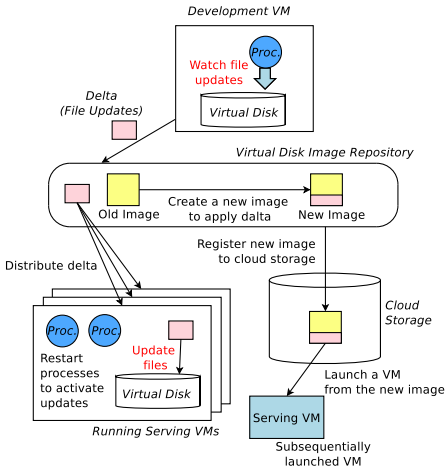

Fig. 2 Overview of our technique.

Files that were created or modified on the development VM are saved to the local virtual disk, and files that were deleted on the development VM are deleted from the local virtual disk. Depending on the types of files that are updated, our kernel-level mechanism automatically performs proper process operations in order to safely update the files, and to put the changes into effect. For example, if the configuration file of the Apache web server is changed, then the Apache processes will be restarted after the new configuration file is saved on the local virtual disk.

3.2 Delta Transfer

We transfer only the files that were updated on the development VM. On the development VM, we run a process that watches the file system for all changes made. Whenever a file is modified in any way, this watching process records the path of the file and the type of modification, which is either "create", "delete", or "update".

To know the file system events performed on the development VM, we make a checkpoint at the last record of the file system event log right after the administrator indicates the start of synchronization. We gather all file system event records from the last checkpoint to the current checkpoint. For files that have multiple pieces of update information within the timespan, all but the newest record are omitted. For files that were either created or updated, the paths of the files along with their contents are packed into the delta. For files that were deleted, only the paths are recorded in the delta. When all necessary data are gathered into the delta, it is transferred to the Virtual Disk Image Repository.

As soon as the Virtual Disk Image Repository receives the delta, it distributes the delta to all serving VMs. The Virtual Disk Image Repository is also responsible for merging the delta information with the old disk image and creating a new image. The new image is registered for the automatic scaling feature of the cloud service.

3.3 Process Operations in Serving VM

When a serving VM receives a delta, it applies the delta to its local virtual disk. Before and after the delta is applied, necessary process operations are performed. We have classified the procedures needed within serving VMs into two types. The first type is to stop a process before updating the files and then start the processes again after update. This is necessary when updating the executable file of a running process, a dynamic link library file being used by a running process, or a file that has been opened (and not closed yet) by a running process.

The second type of procedure is restarting a process after updating the files. This is necessary when updating a file that was previously opened, and closed, by a running process. Restarting such processes is necessary since the content of the file may somehow affect the behavior of the running process. We also restart any process that has opened the directory containing an updated file. This is necessary since the content of the directory may affect behavior of the process. An example would be a *.d directory, which a process reads all files contained within the directory and regards them all as configuration files.

In some cases, rules stated above are too strict. For example, in general we do not need to restart Apache after updating an HTML file in a public web directory, even if it had been opened by the Apache process in the past. We provide configuration options to ignore such files (e.g., all files in a public web directory) when determining process operations. Note that correctly setting up the configuration file require deeply administrative knowledge. Even if we do not configure the file, we can successfully update our application with our mechanisms. This is because the effects of the configuration files is to skip redundant software restarts.

4. Implementation

We have implemented our technique on 32-bit VMs running within Amazon EC2. We modified Linux 2.6.31.14, and used MySQL version 5.0.45 for recording metadata and Apache 2.2.17 for implementing HTTP messaging features between VMs. We implemented most features in Perl, and we used Perl interpreter version 5.8.8. Note that our kernel modification is small, and thus we believe that our mechanism is not difficult to port to other OS kernels.

4.1 Development VM

4.1.1 Recording File Updates

We use our desktop PC or laptop as the development VM. The delta is sent from them through the CGI provided by Virtual Disk Image Repository, as described later. To watch and record updates on the file system, we run the *File Update Watch process* on the development VM. The File Update Watch process uses *inotify*, a feature of the Linux kernel

capable of notifying a user-mode program of changes to the file system. By using inotify, an application can watch file system events without polling the entire file system.

The File Update Watch process uses a modified version of *inotify-tools* 3.14 [5], a series of command line programs and libraries. Whenever the File Update Watch process fetches event information via inotify-tools, it registers the event information into the `fs_events` table in the local MySQL database. The structure of the `fs_events` table is shown in Table 1. `Fs_event` basically includes information on file system events. It has the record id (ID), date time (time), file path name (path), performed file operations (op), and a bit (isdir) showing whether the target is a directory or a file.

4.1.2 Delta Transfer

We developed the Delta Uploader that transfers the file update information to the Virtual Disk Image Repository. When the developer decides to synchronize the changes made on the development VM to the serving VMs, the Delta Uploader first creates a checkpoint in the database and gathers the file update information since the last update from the database. Newly created files and updated files are collected into a single tar file (updates.tar), with the full file path and file attributes preserved. The full paths of the deleted files are recorded into a text file (removals.txt), with one file path per line. The tar file having the created and updated files (updates.tar) and the text file having the paths of deleted files (removals.txt) are compressed into one gzip-compressed tar file with the checkpoint number as its base name (e.g., 123.tar.gz). We refer to this compressed tar file as a *delta*.

As soon as the delta is created, the Delta Uploader places it in a public web directory of the development VM and notifies the Virtual Disk Image Repository that a new delta is available. The notification is sent as an HTTP request, which includes the URL of the delta, the checkpoint number, and the MD5 checksum of the delta.

4.2 Virtual Disk Image Repository

The Virtual Disk Image Repository consists of three modules: the *Update Receiver*, *Disk Image Manager*, and *Delta Distributor*. The Update Receiver receives a delta from the delta transmission program described above. The Disk Image Manager creates a new disk image. The Delta Distributor distributes the delta to all running serving VMs.

Table 1 Structure of `fs_events` database table in development VM.

Field	Type
id	bigint(20)
time	datetime
path	varchar(256)
op	varchar(16)
isdir	tinyint(1)

4.2.1 Update Receiver

The Update Receiver is implemented as a Common Gateway Interface (CGI) script. When the Update Receiver receives notification of a new delta from the development machine, it first downloads the delta from the development VM, and then calculates the MD5 checksum and verifies that it matches the MD5 checksum value that the development VM passed. Then, it launches the Disk Image Manager and Delta Distributor to put the update into effect.

4.2.2 Disk Image Manager

The Disk Image Manager is in charge of creating a new disk image. The Disk Image Manager makes use of Amazon Elastic Block Storage (EBS), a volume of which can be mounted in VMs. Aside from the EBS volume that is mounted as its root disk, the Virtual Disk Image Repository mounts an extra EBS volume that has the contents of the previous disk image of the serving VMs. The Disk Image Manager first applies the new delta to the virtual disk; it creates, updates, and deletes files on the virtual disk based on the information included in the delta. Then, the virtual disk is unmounted and a new EBS snapshot is created using an AWS API call. After the EBS snapshot is created, the Disk Image Manager registers it as an Amazon Machine Instance (AMI), which is used to create a VM. The new AMI is also registered to the Auto Scale feature, so that VMs that are subsequently launched by Auto Scale can be launched from the new AMI.

4.2.3 Delta Distributor

The Delta Distributor is in charge of transferring the delta to all serving VMs. To hold the list of currently running VMs, we prepare the "pubvms" table in the local MySQL database. The structure of the `pubvms` table is shown in Table 2. `Pubvms` includes the record id (id) and information on running VM; hostname of the running VM inside and outside Amazon EC2 (internal_host and external_host), the ID of the running VM (instance_id), the VM image version used by the VM (image_ver), and the time when the VM is launched (launch_time). Delta distribution is done in basically the same way as how the development VM transfers a delta to the Virtual Disk Image Repository. The Delta Distributor notifies the availability of a new delta to all serv-

Table 2 Structure of `pubvms` database table on virtual disk image Repository.

Field	Type
id	bigint(20)
internal_host	varchar(128)
external_host	varchar(128)
instance_id	varchar(16)
image_ver	bigint(20)
launch_time	datetime

Table 3 File types and necessary process operations.

File Type	procfs File used for Determination	Before File Update	After File Update
Executable file of running process	/proc/[PID]/exe	Stop process	Start Process
DLL used by running process	/proc/[PID]/maps	Stop process	Start Process
File currently opened by running process	/proc/[PID]/fd	Stop process	Start Process
File previously opened by running process	/proc/[PID]/openlog (newly implemented)	-	Restart Process

ing VMs, using an HTTP request with the URL, checkpoint number, and MD5 checksum value of the delta. We currently place the delta in a public web directory on the Virtual Disk Image Repository, but it is possible to upload the delta to a different server and give the URL for that server to the serving VMs instead.

To know the currently running serving VMs, AWS provides users with an API call that returns a list of live instances. However, we cannot use this information to simply determine which VM is ready to provide service and is capable of receiving and handling deltas. This is because an instance may still be in the middle of the boot procedure, even if the API response shows the instance as a "running" instance. In that phase, the VM fails to receive the delta.

To successfully send the VMs the delta, we also implemented the VM Manager, which runs on the Virtual Disk Image Repository. The VM Manager communicates with running serving VMs to know that the VMs are ready to provide service. When a new serving VM is launched, it sends a *LAUNCH message* to the VM Manager as an HTTP request. This is invoked by the Linux init script at the end of the launch procedure. The LAUNCH message contains the DNS host name of the VM, and the checkpoint number of the disk image used to launch the new serving VM.

When receiving a LAUNCH message, the VM Manager first checks if the disk image used for the launch is the newest available one. If the disk image is outdated and a new delta is already available, then the Virtual Disk Image Repository notifies the serving VM of the delta, in the same manner as that of the Delta Distributor. After the new delta has been successfully applied to the new VM, a new record is created in the `pubvms` table, and the new VM is registered to the load balancer.

In reverse, when a serving VM is terminated, it sends a *STOP message* to the VM Manager at the beginning of the shutdown procedure. The request contains the host name of the VM. The VM Manager deregisters the VM from the load balancer and deletes the record in the `pubvms` table that corresponds to the VM being shut down. The VM Manager also has a "livecheck" feature, which periodically checks if each serving VM is alive. This simply sends an HTTP request to each of the serving VMs to see if they return valid responses. If a serving VM fails to return a valid response within a certain amount of time, it is deregistered from the load balancer, removed from the `pubvms` table, and terminated via an API call. Auto Scale will launch a new VM if necessary.

4.3 Serving VM

On each serving VM, the *Process Manager* receives an update notification from the Virtual Disk Image Repository. The Process Manager downloads the delta from the URL specified in the notification, calculates the MD5 checksum and verifies that it matches the MD5 checksum value that was passed as a parameter of the notification. Then, for each of the files that will be affected by applying the delta, either as a created, updated, or deleted file, the Update Program checks if any process operation is necessary.

To safely perform updates, a process must be stopped before updating any of the following:

- an executable file of a running process,
- a file that is currently being opened by a running process, or
- a file that is mapped to the address space of a running process.

The Process Manager determines if the above files are created, updated or deleted by traversing the process file system (procfs). Table 3 lists files the Process Manager reads. To check whether updated files are an executable file of the running processes, we read a /proc/[PID]/exe file. We also read /proc/[PID]/maps and /proc/[PID]/fd to check whether updated files are being opened or mapped into the address space by the process. In addition, we read /proc/[PID]/openlog, which is newly implemented as described later, to check whether the updated files are previously opened by the processes.

The Process Manager restores the process status across update propagation. Before actually stopping the process, the Process Manager collects metadata of the process, including the working directory, the command line, the process authority, the environment variables, and its root directory, to launch the process. After that, it stops the process by sending a SIGTERM signal. The Process Manager restores these metadata when it later restarts the process. The metadata, along with their sources within the procfs, are listed in Table 4. Our approach conservatively stops and restarts processes when its binary files are updated. This is because the updated files might be used in the processes whose binary files are older. In that case, the program could be crashed due to the conflict of the updates. By using our approach, we can easily avoid this situation without complicated mechanisms.

Table 4 Process metadata obtained and restored.

Metadata	procfs Source	Restore Method
Working Directory	/proc/[PID]/cwd	chdir()
Command Line	/proc/[PID]/cmdline	execve()
User / Group	/proc/[PID]/status	setresuid(), setresgid()
Environment Variables	/proc/[PID]/environ	execve()
Root Directory	/proc/[PID]/root	chroot()

Note that an openlog file becomes quite large if a process frequently opens and closes files. We can refresh the length of the file by restarting the processes if the users want to make the openlog size smaller.

The update program also checks if any process must be restarted after updating the files. When updating a file that a process has opened in the past, the process must be restarted after the file is updated or deleted. Moreover, when creating, updating, or deleting a direct child of a directory that was opened by a process in the past, that process must also be restarted.

To determine which process has opened which files in the past, we have modified the Linux kernel so that each process keeps a list of files that it opened. We have added a new `struct list_head openlog` member to the `struct task_struct` structure. Each element of the `openlog` contains the device ID and the inode number of a file that was opened by the process. Elements are added to the list when the `open()` system call is called. When the `fork()` system call is called, the entire list is copied from the parent to the child, since at this point the context of the child is simply a clone of the parent. When the `execve()` system call is called, we clear the `openlog` list, except for the elements that indicate the files that are still opened at that point. The Process Manager can access the content of the `openlog` of each process via procfs using the path "/proc/[PID]/openlog", where [PID] represents the process ID.

After successfully restarting all the processes, the Process Manager sends an "UPDATE SUCCESS" message to the Virtual Disk Image Repository. This enables the VM Manager to manage the checkpoint number of each serving VM and the Delta Distributor to know that the serving VM is ready to receive the next delta file.

A configuration file, written in the YML format, is read prior to determining the necessary process operations. A list of regular expressions, such as "^/var/www/html/.+$", is found in this configuration file. Out of the files that will be updated, all files that have a path matching one or more of the listed regular expressions would be regarded as updatable without any process operation, and therefore they will be ignored in the process operation determination steps stated above.

4.4 Limitations

The target of our technique is on applications' updates in IaaS environments, which means that it does not manage any kinds of software updates. We need to extend the prototype to conduct system configuration updates, such as adding a user, and OS kernel updates. It is also difficult for the prototype to handle updates that involves system configuration changes, and the fix of security vulnerabilities involving special administrative operations such as operations in the single user-mode. Applying these updates is out of the scope of this work.

We also note that the current prototype does not handle any restarts of applications. For ease implementation, our prototype sends signals to the target processes to restart the processes. This means that the prototype does not take into consideration the order of application restarting. Since it restarts applications without the dependency between updating applications, the applications which closely coordinate with each other fail to restart. To successfully restart them, we need to extend our prototype by specifying the order of restarts of applications. For example, we associate restart operations with process names in advance, and perform a restart operation when the process associated with the operation is restarted for its update.

5. Experiments

We conducted all of our experiments in the US-East region of Amazon EC2. Unless otherwise stated, the VMs we launched were *m1.small instances*, each having 1.7 GB RAM, one 32-bit EC2 Compute Unit (equivalent to 1.7 GHz Xeon). A 15 GiB EBS volume was attached to each of the instances. We used Linux kernel 2.6.31.14, with a patch applied to run user-provided Linux kernels on Xen, as specified by Amazon's official documentation [6].

5.1 Synchronization Time

The goal of this experiment is to show that our method can synchronize VMs in a shorter and more stable length of time compared to the conventional method. We first compared the time needed to synchronize VMs, using the conventional disk-image-based method and our technique.

The scenario of the experiment is as follows. There are nine serving VMs running in three availability zones: three VMs on each of "us-east-1b", "us-east-1c", and "us-east-1d". We consider six different cases where we need to update the software stack of the VMs. We summarize our experimental details in Table 5. First, we ran Apache web server. For the *Web Content Update (Small)* case, we update the top page of the public web directory to the top page of Wikipedia. This includes an HTML file, along with the CSS files and image files referenced by the HTML file. This is a simple update of a static web page, so no process operation is necessary. *Web Content Upate (Large)* is basically the same as the previous case, except for the size. This case transfers the full set of Apache documentation. *Apache Config Change* is a case that changes the 404 error page of Apache. This involves updating the main configuration file of Apache and creating a new HTML file of the 404 error page. On the serving VM, Apache needs to be restarted

Table 5 Update cases for synchronization time experiment.

Case	Update Size	# of Update files	Process Operations	AMI size [KB] (before updates)	AMI size [KB] (after updates)
Web Content Update (Small) update top page (HTML / CSS / images)	136 KB	15	none	425608	425712
Web Content Update (Large) update entire website (HTML / CSS / images)	3.1 MB	27	none	425608	427830
Apache Config Change update Apache confing file and 404 page	34.7 KB	2	Restart Apache	425608	425668
Servlet App. Update update Java class File	1.7 KB	1	Restart Tomcat	503268	503278
Servlet App. Install add new war file	9.9 KB	2	Restart Apache and Tomcat	503268	503282
Apache Update update Apache binary and related files	1.5 MB	94	Stop Apache during file update	503268	505696

after the file update, to enable the new configuration. In the latter cases, we started Tomcat servlet engine. *Servlet Application Update* is a case that updates a simple Servlet application running on Tomcat. This updates a single Java class file. Tomcat needs to be restarted after the file update. *Servlet Application Install* is a case that adds a Servlet application. This adds a single web application archive (WAR) file. Apache configuration (mod_jk configuration) will also be changed so that HTTP requests to the new application can be handed over from Apache to Tomcat. Both Apache and Tomcat need to be restarted after updating the files. Finally, *Apache Update* will update the executable file and related files of Apache. We recompile Apache on the development VM and transfer it to the serving VMs. On the serving VMs, Apache must be stopped while files are being updated.

For the conventional method, we followed the below steps.

1. Launch a development VM with the old software stack.
2. Update the development VM.
3. Stop the development VM.
4. Bundle an AMI from the development VM.
5. Launch nine serving VMs from the new AMI.
6. Terminate the old VMs.

Although the Virtual Disk Image Repository is not necessary for the conventional method, we ran it to execute an experiment script that goes through the above steps. Furthermore, for time measurement, the serving VMs were configured to send LAUNCH messages to the Virtual Disk Image Repository, as in the implementation of our system. We started measuring time when making the API call for stopping the development VM, and ended the measurement when all nine serving VMs had been launched and had sent LAUNCH messages to the Virtual Disk Image Repository. Although AMI bundling can be conducted for running VM instances, we have confirmed that this takes substantially longer compared to stopping the VM first and then bundling. We therefore stopped the development VM before bundling an AMI.

For our proposed technique, we followed the below steps.

1. Launch a development VM, with the old software stack.
2. Update the development VM.
3. Start synchronization by invoking the Delta Uploader on the development VM.

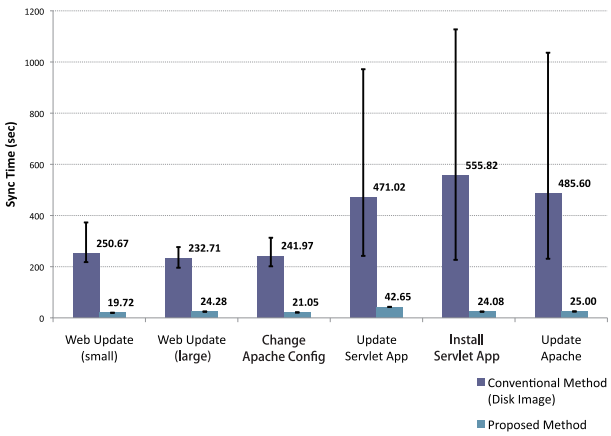

Fig. 3 Average time consumption for synchronization: Error bars indicate range of individual results.

We started measuring time when invoking the Delta Uploader. We ended the measurement when all nine serving VMs had finished the update procedures and had sent UPDATE SUCCESS messages to the Virtual Disk Image Repository. In both techniques, we repeated the measurements 10 times from the top to the bottom in Table 5

The time measurement results are shown in Fig. 3. They show that not only can our technique synchronize VMs in an order-of-magnitude shorter time than the conventional method, but the time it takes is very stable. The time the conventional method takes can vary significantly, even for the same workload. For example, the minimum time for *Servlet Application Install* was 227 seconds, and the maximum time for the same workload was 1,127 seconds. We also confirmed that necessary enhancement procedures were adequately taken on the serving VMs.

Figure 4 and 5 exhibit breakdown of the time consumption for the conventional and our method. As shown in Fig. 4, the time consumption for the conventional method consists of 3 factors; bundling an AMI from the development VM, launching nine public VM, and terminating old VMs. The time for terminating old VMs is similar in all cases. Bundling time is longer and longer as the update size is larger and larger. This is because the size of an AMI be-

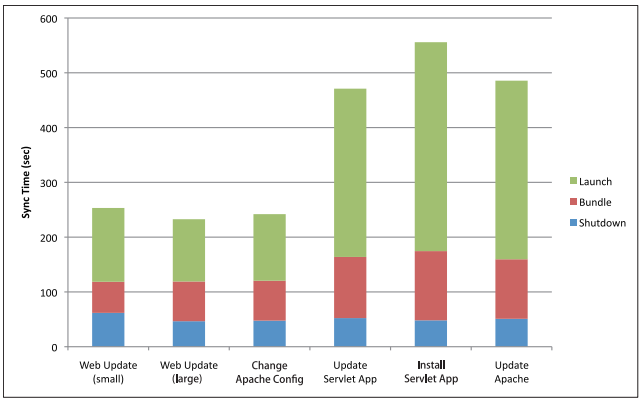

Fig. 4 Time consumption breakdown for the conventional method.

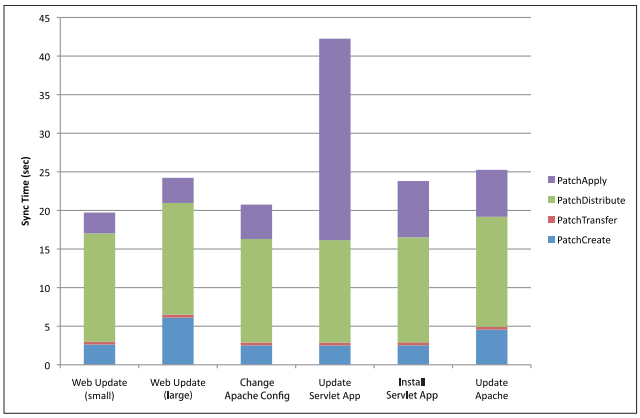

Fig. 5 Time consumption breakdown for the proposed technique.

comes larger when more files are stored during the updates. For example, bundling time in Web Update (small) is faster than the other update cases since it just modifies one html file. On the other hand, bundling an AMI in Apache Update takes longer time because they create more new files by Tomcat's war file decompression and so on. The average time for launching VMs is longer because the launching time fluctuates largely when the size of AMIs is larger. Since we can not analyze the internal behavior of the Amazon EC2 cloud environment, we do not have any clue so far why the launching time fluctuates so largely.

Figure 5 shows that the phases of our technique that take longer time are distributing patches and applying them to the serving VMs. The time required for our technique consists of four factors; creating patches from the updates conducted in the development VM, transferring the patches to Virtual Disk Image Repository, distributing them from the Virtual Disk Image Repository to the nine serving VMs, and applying the patches to the VMs. Creating patches depends on the number of file system operations. The longest time required in creating patches is Web Update (large) case in which file operations including deleting old apache's files and adding its newer files are performed. Transferring patches takes shorter time in all cases since each patch is not large. Time required for Distributing patches is similar in all cases. The time for applying patches depends on the type of updates. For example, both the Web Update cases are shorter than the other cases since any operation is not necessary to activate them. Servlet Application Install and Apache Update take longer time since the former needs to restart Apache and Tomcat, and the letter stops and resumes Apache before/after applying updates. Unexpectedly, the time in Servlet Application Install is much longer. Our expectation was that time for applying patches in Servlet Application Install is similar to Servlet Application Update since the necessary operations to activate the update in both cases are to restarting Tomcat. This may be mysterious behavior in Amazon EC2 which takes longer time to assign CPU time to some serving VMs.

5.2 Overhead

5.2.1 Serving VM

We made amendments to the Linux kernel that runs on serving VMs so that it maintains a list of the files that each process opened in the past, as explained in Sect. 4.3. Since our modified kernel needs to maintain extra process information compared to the vanilla kernel, the overhead imposed by our modification may affect the overall performance of the serving VMs. Serving VMs provide service to end users, so any observable overhead suggests degradation of the service and therefore is not favorable.

We set up two server VMs, one with a vanilla kernel and the other with our modified kernel. Note that even for the vanilla kernel, we applied a patch to run a user-provided kernel on Xen, as specified by Amazon's official documentation [6]. On each of the VMs, we set up a pseudo auction site named RUBiS [7]. RUBiS comes with a client benchmarking tool that is capable of evaluating performance scalability. We set up RUBiS 1.4.3 on Tomcat 5.5.31, and used Apache 2.2.17 as the HTTP server. We conducted a benchmark for 900 seconds and calculated the average throughput.

To avoid the unlikely chance of a client VM and server VM residing on the same physical machine, in which the performance of the client and the server may affect each other [8], we placed the server VM and client VM in different availability zones. We launched the server VM in the "us-east-1c" availability zone and the client VM in the "us-east-1d" one. In addition, to ensure that any bottleneck in the throughput measurement was not due to the performance of the client VM, we used the *c1.medium* VM instance, which has 5 EC2 Compute Units, as the client VM.

The results are shown in Fig. 6. The average throughput using our modified version of the Linux kernel was 40.04 requests per second, while the average throughput using the vanilla kernel was 39.67 requests per second. From these results, we conclude that our modification to the kernel does not impose observable overhead and therefore does not degrade server performance.

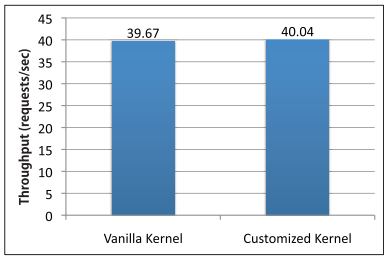

Fig. 6 Results for serving VM overhead experiment.

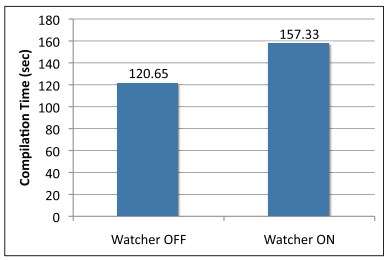

Fig. 7 Results for development VM overhead experiment.

5.2.2 Development VM

The File Update Watch process runs on the development VM to watch and record all the events on the file system, as described in Sect. 4.1.1. This can impose performance overhead on the development VM. Although the development VM itself does not provide service to end users and therefore its performance is not as important as that of the serving VM, we conducted an experiment to ensure that the overhead is at an acceptable level for a development environment.

To measure the overhead, we measured the time it takes for the development VM to compile the Apache web server, while both running and not running the File Update Watch process. We compiled Apache version 2.2.17 with default configurations. The time taken to complete the `make` command 10 times was measured for both cases. We believe this time measurement to be a strict scenario for our system, considering all the file creation and deletion that occurs during the compilation process.

The average time consumption for one `make` is shown in Fig. 7. When the File Update Watch process is running, compilation takes 157.33 seconds, which is approximately 30.4% more than the compilation time of 120.65 seconds when the File Update Watch process is not running. We believe this overhead to be acceptable for a development environment.

If the developer needs to handle a heavy workload in minimal time, he or she may choose to use a high-end instance with higher instance-hour price. This would only slightly affect the overall cost because, unlike serving VMs, only one VM needs to be launched as a development VM. In addition, it only has to be running during the actual development.

6. Discussion

In our current implementation, we transfer the entire content for files that were updated. This may be inefficient when a large file was partially updated. To solve this problem, we transfer sub-file incremental data for file updates, using a delta encoding algorithm as `rsync` does. The challenge here is how to correctly apply an encoded delta to serving VMs since the file may be amended differently on each serving VM. To deal with this problem, we plan to keep the original copy of the file whenever a file is first amended on a serving VM. In this case, the delta encoding data can be computed against the original copy, making it possible to distribute same data to all VMs. As this method would impose extra overhead on serving VMs, there needs to be a threshold based on the file size, to decide whether or not delta encoding is necessary.

Even if our technique is employed, we have to prepare newer VMs and redirect clients requests to them, i.e. rolling upgrade, in a situation that a service provider has to uphold the SLA guarantee. In our technique, some updates incur downtime caused by restarting applications, which can be critical in services requiring high availability. In such situations, we have to configure a load balancer to successfully perform a rolling upgrade in activating updates that need to restart processes.

The current prototype enforces all serving VMs to download the delta directly from the Virtual Disk Image Repository. This style is not be scalable for the number of serving VMs since access to the Virtual Disk Image Repository is increased and then it becomes a bottleneck. To address this issue, we can distribute the delta using cloud-based data storage, such as Amazon Simple Storage Service (S3). Amazon S3 could be sufficiently scalable for distributing deltas to hundreds or thousands of serving VMs, as Amazon claims that S3 is able to "provide the same highly scalable, reliable, secure, fast, inexpensive infrastructure that Amazon uses to run its own global network of web sites [9]".

7. Related Work

Various research has been conducted on storage systems specialized for virtualized environments. Ventana [10] is a virtualization-aware file system. It enables the administrator to update shared files, and the change will immediately become effective on other VMs that share the file. However, it does not help select administrative operations to propagate updates, such as application restarts and OS reboots.

Parallax [11] is a block-level virtualization of a storage system. It is a distributed file system constructed of an array of *storage VMs*, which reside on the same physical hosts as the VMs that they serve. It is capable of creating snapshots very frequently, e.g., every 10 ms. This quick snapshot-taking functionality may be used to shorten the time taken to create disk images in an IaaS environment. Thus, we

can complementarily use both Parallax and our technique to shorten update time.

Snowflock [12] provides VM fork, which is very similar in sprit to the UNIX process fork. VM fork creates child VMs, each having the full state of the parent VM. Snowflock is able to create replicas of a VM in subseconds, through lazy state replication and multicast distribution of states. Since Snowflock is a mechanism for efficiently creating VM replicas, it does not focus on VM updates.

A single system image (SSI) provides a single logical computing environment on an array of physical computers. With SSI, the administrator does not need to manage multiple serving VMs, which would eliminate the need for a universal way of synchronizing VMs. MOSIX [13]–[15] provides SSI at the OS level. It is implemented as a modification of the Linux kernel, and MOSIX transparently migrates processes from overloaded machines to underutilized machines. Factored operating system (fos) [16] is an OS that runs on multiple cores and VMs, capable of scaling up and down while providing SSI. Although these systems provide high manageability, we have to greatly modify the current cloud platforms that consist of system virtualization. This is quite difficult and sometimes impossible due to high engineering costs.

vNUMA [17] provides SSI at the VM level. vNUMA runs on multiple hypervisors running on separate physical machines. It also provides a virtual NUMA machine, so the OS that runs on it must be a NUMA-aware OS. It provides high portability, since any NUMA-aware OS such as Linux can run on it. Experiment results showed that it can scale out on various workloads. However, vNUMA is not capable of increasing and decreasing the number of physical nodes that compose its system. Therefore, using vNUMA in a cloud would sacrifice elasticity.

8. Conclusion

This paper presented a technique to synchronize VMs with less time and lower administrative burden. Our proposal, Virtual Disk Image Repository, automatically creates a new disk image from incremental update information, thus enabling us to send only the updated files via the Internet. It also distributes the updates to the serving VMs. Our technique reduces administrative burden of applying updates to all running serving VMs by running a kernel-level mechanism that performs necessary operations such as application restarts and OS reboots, based on the types of updated file. Experimental results for Amazon EC2 showed that our technique can synchronize VMs in an order-of-magnitude shorter time than the conventional disk-image-based VM cloning method. Our technique also correctly selects and performs necessary operations to activate updates.

References

[1] Amazon Web Services LLC, “Amazon Web Services,” http://aws.amazon.com/

[2] Amazon Web Services LLC, “Amazon Elastic Compute Cloud (Amazon EC2),” http://aws.amazon.com/ec2/

[3] B. Cho and I. Gupta, “New algorithms for planning bulk transfer via Internet and shipping networks,” Proceedings of the 30th IEEE International Conference on Distributed Computing Systems (ICDCS ’10), pp.305–314, June 2010.

[4] B. Cho and I. Gupta, “Budget-constrained bulk data transfer via Inernet and shipping networks,” Proceedings of the 8th ACM International Conference on Autonomic Computing (ICAC ’11), pp.71–80, June 2011.

[5] GitHub Inc., “Inotify Tools,” https://github.com/rvoicilas/inotify-tools/wiki/

[6] Amazon Web Services LLC, “Enabling User Provided Kernels in Amazon EC2,” http://ec2-downloads.s3.amazonaws.com/user_specified_kernels.pdf, Sept. 2010.

[7] Rice University, “RUBiS: Rice University Bidding System,” http://rubis.ow2.org/, Oct. 2004.

[8] T. Ristenpart, E. Tromer, H. Shacham, and S. Savage, “Hey, you, get Off my Cloud! Exploring information leakage in third-party compute clouds,” Proceedings of the 16th ACM conference on Computer and communications security (CCS ’09), pp.199–212, 2009.

[9] Amazon Web Services LLC, “Amazon S3,” http://aws.amazon.com/s3/

[10] B. Pfaff, T. Garfinkel, and M. Rosenblum, “Virtualization aware file systems: Getting beyond the limitations of virtual disks,” Proceedings of the 3rd USENIX Symposium on Networked Systems Design and Implementation (NSDI 2006), pp.353–366, May 2006.

[11] D.T. Meyer, G. Aggarwal, B. Cully, G. Lefebvre, M.J. Feeley, N.C. Hutchinson, and A. Warfield, “Parallax: Virtual disks for virtual machines,” Proceedings of the 3rd ACM European Conference on Computer Systems (EuroSys ’08), pp.41–54, April 2008.

[12] H.A. Lagar-Cavilla, J.A. Whitney, A. Scannell, P. Patchin, S.M. Rumble, E. de Lara, M. Brudno, and M. Satyanarayanan, “SnowFlock: Rapid virtual machine cloning for Cloud computing,” Proceedings of the 4th ACM European Conf. on Computer Systems (EuroSys ’09), April 2009.

[13] L.O. Barak A. and S.A., “Scalable Cluster Computing with MOSIX for Linux,” Proceedings of 5th Annual Linux Expo (Linux Expo ’99), pp.95–100, 1999.

[14] A. Barak and A. Shiloh, “The MOSIX Management System for Linux Clusters, Multi-Clusters, GPU Clusters and Clouds,” http://www.mosix.org/pub/MOSIX_wp.pdf, 2010.

[15] S. McClure and R. Wheeler, “Mosix: How linux clusters solve real world problems,” Proceedings on the 2000 USENIX Annual Technical Conference (ATC ’00), p.32, June 2000.

[16] D. Wentzlaff, C. Gruenwald, III, N. Beckmann, K. Modzelewski, A. Belay, L. Youseff, J. Miller, and A. Agarwal, “An operating system for multicore and clouds: Mechanisms and implementation,” Proceedings of the 1st ACM symposium on Cloud Computing (SOCC ’10), pp.3–14, June 2010.

[17] M. Chapman and G. Heiser, “vNUMA: A virtual shared-memory multiprocessor,” Proceedings of the 18th USENIX Security Symposium (USENIX ’09), pp.15–28, June 2009.

Hiroshi Yamada received his B.E. and M.E. degrees from the University of Electro-communications in 2004 and 2006, respectively. He received his Ph.D. degree from Keio University in 2009. He is currently an associate professor of the Division of Advanced Information Technology and Computer Science at Tokyo University of Agriculture and Technology. His research interests include operating systems, virtualization, dependable systems, and cloud computing. He is a member of ACM, USENIX and IEEE/CS.

Shuntaro Tonosaki received his B.E. and M.E. degrees from Keio University. His research interests include virtualization and cloud computing.

Kenji Kono received the BSc degree in 1993, MSc degree in 1995, and Ph.D. degree in 2000, all in computer science from the University of Tokyo. He is an associate professor of the Department of Information and Computer Science at Keio University. His research interests include operating systems, system software, and Internet security. He is a member of the IEEE/CS, ACM and USENIX.