PAPER

A VMM-Level Approach to Shortening Downtime of Operating Systems Reboots in Software Updates

Hiroshi YAMADA[†a)], ***Nonmember and*** **Kenji KONO**[††], ***Member***

SUMMARY Operating system (OS) reboots are an essential part of updating kernels and applications on laptops and desktop PCs. Long downtime during OS reboots severely disrupts users' computational activities. This long disruption discourages the users from conducting OS reboots, failing to enforce them to conduct software updates. Although the dynamic updatable techniques have been widely studied, making the system "reboot-free" is still difficult due to their several limitations. As a result, users cannot benefit from new functionality or better performance, and even worse, unfixed vulnerabilities can be exploited by attackers. This paper presents *ShadowReboot*, a virtual machine monitor (VMM)-based approach that shortens downtime of OS reboots in software updates. ShadowReboot conceals OS reboot activities from user's applications by spawning a VM dedicated to an OS reboot and systematically producing the rebooted state where the updated kernel and applications are ready for use. ShadowReboot provides an illusion to the users that the guest OS travels *forward* in time to the rebooted state. ShadowReboot offers the following advantages. It can be used to apply patches to the kernels and even system configuration updates. Next, it does not require any special patch requiring detailed knowledge about the target kernels. Lastly, it does not require any target kernel modification. We implemented a prototype in VirtualBox 4.0.10 OSE. Our experimental results show that ShadowReboot successfully updated software on unmodified commodity OS kernels and shortened the downtime of commodity OS reboots on five Linux distributions (Fedora, Ubuntu, Gentoo, Cent, and SUSE) by 91 to 98%.

key words: *virtual machines, software updates*

1. Introduction

Operating system (OS) reboots are an essential part of updating contemporary kernels and applications on our laptops and desktop PCs. In updating a kernel, an OS reboot is typically invoked to terminate the older kernel and start the newer one after the update patches are applied. OS kernels are still being developed to improve their performance, add new functionality, and repair security vulnerabilities. The Linux Foundation reports that on average 3.83 patches are applied to the Linux kernel tree every hour between the 2.6.11 and 2.6.30 kernel [1].

An OS reboot is also required for updating applications. Application updates sometimes involve system configuration changes such as Windows Registry keys and shared component updates such as glibc and WebKit. To activate updates, we commonly conduct an OS reboot, which is the easiest way to restart all applications. For example, an OS reboot is often needed for updates of Internet Explorer and Safari because they involve changing Windows Registry keys and updating WebKit.

The downtime during OS reboots, however, severely disrupts users' computational activities. While an OS is rebooting, the user cannot perform his or her computational tasks. This disruptive downtime is becoming longer and more costly since there are more and more software updates. Although announced updates should be applied as soon as possible since they often fix critical vulnerabilities, the long disruption caused by the OS reboots discourages users from rebooting OSes, so they often fail to update software. As a result, users cannot benefit from new functionality or better performance, and even worse, unfixed vulnerabilities can be exploited by attackers. A research literature [2] notes that "many desktop machines are not rebooted to apply kernel patches because of the burden imposed by rebooting".

To eliminate the need for an OS reboot with software updates, dynamic updatable kernels are effective for applying patches to the kernels at runtime. However, making the systems "reboot-free" is still difficult even when using dynamic updatable kernels for the following reasons. First, some of these kernels are designed for fixing bugs in the kernel code region, such as condition misses [2]. Therefore, it is difficult to manage the semantic changes to memory objects, such as when adding a new field to a data structure. They also cannot manage system configuration updates because a restart of all the processes is not involved. In these cases, we have no choice but to conduct an OS reboot. Second, some dynamic updatable kernels require intimate knowledge about the target kernels [3], [4]. To use them, we have to develop special patches from the original ones. This task is non-trivial because it requires knowledge about the internal structures of the target kernels at the source code level. Lastly, we have to pay a high engineering cost for redesigning and modifying a large part of the target kernel [5]–[7]. This is difficult and often impossible because recent kernels are more complex and some are closed-source and/or proprietary. We believe that there is room for improvement in managing OS reboots in software updates.

Our goal is to mitigate the disruption to users' computational tasks caused by OS reboots to encourage users to conduct software updates as soon as possible. *ShadowReboot*, presented in this paper, is a virtual machine monitor (VMM)-based approach to shortening downtime of OS reboots in software updates. ShadowReboot conceals OS reboot activities by spawning a VM dedicated to an OS

Manuscript received January 27, 2014.
Manuscript revised June 1, 2014.

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

DOI: 10.1587/transinf.2014EDP7031

reboot and systematically producing a rebooted state. In ShadowReboot, users can run their applications while simultaneously rebooting the OS. ShadowReboot engenders an OS reboot effect by restoring the produced rebooted state where the updated kernel and applications are ready for use. It provides an illusion to users that an OS travels *forward* in time to the rebooted state.

The main contributions of this paper are as follows:

- We propose ShadowReboot, which offers the following advantages: (1) It shortens the downtime during software updates, thus mitigating the disruption to users' application activities. (2) ShadowReboot can be used to apply any patch to the kernels and even to update system configuration. (3) It does not require intimate knowledge about the target kernels at the source code level; we do not have to develop kernel modules or special patches. (4) ShadowReboot requires no modification of the target kernels. These features make ShadowReboot complementary to existing systems and dynamic updatable kernels.
- We clarify mechanisms required to achieve ShadowReboot, and show it is applicable to five real Linux distributions (Fedora, Ubuntu, Gentoo, Cent, and SUSE). In Sect. 7, we describe how the Linux distributions have been configured to be suitable for ShadowReboot.
- We implement ShadowReboot and evaluate a prototype on VirtualBox 4.0.10 OSE with the five real Linux distributions. The experiments also show that ShadowReboot successfully updates software on unmodified Linux kernels and the downtime caused by ShadowReboot is 91 to 98% shorter than that of commodity OS reboots on the five Linux distributions.

Note that ShadowReboot does not keep the memory states of user's running applications through the restoration of the rebooted state. Although process migration approaches can be used to move the running applications to the rebooted state, selecting processes to be migrated is a challenge because the components linked to the migrated processes would conflict with the newer ones in the rebooted state. The challenge of keeping running applications' states across an OS reboot in software updates is out of the scope of this paper. Fortunately, we can quickly restore the applications' states after shadow-rebooting by taking advantage of the applications' support that saves their states to disks, such as a FireFox extension of restoring the contents in each tab.

The rest of this paper is organized as follows. Section 2 summarizes previous approaches, and Sect. 3 presents key observations and an overview of ShadowReboot. Sections 4 and 5 describe the design and implementation of ShadowReboot, respectively. Section 6 discusses limitations and a use case of ShadowReboot. Section 7 details our experimental results and Sect. 8 describes work related to ours. Finally, Sect. 9 concludes this paper.

2. Previous Approaches

In this section, we discuss previous approaches that have a similar goal of managing OS reboots for software updates before presenting ShadowReboot.

2.1 Dynamic Update Techniques

Dynamic updatable kernels are effective for applying patches to the kernels at runtime so that we do not need to conduct an OS reboot. Exploring dynamic updatable techniques for kernels is a hot research topic. Ksplice [2] fixes kernel vulnerabilities at runtime by safely translating the binaries in the kernel memory. DynAMOS [4] and LUCOS [3] make use of special patches to dynamically update the target kernel. K42 [5]–[7] is an architecture of dynamic updatable kernels.

Although dynamic updatable kernels have been extensively studied, making the system "reboot-free" is still difficult for the following three reasons. First, the applicability of existing dynamic updatable kernels is often limited. Some of them fix bugs in the kernel code region such as condition misses and array bounds-checking errors. Ksplice [2] dynamically translates the function code at a safe time when no thread's instruction pointer falls within that function's text and when no thread's kernel stack contains a return address within that text. Such approaches are designed to manipulate the text region, not to handle memory objects in the kernel heap region. Therefore, it is difficult to manage the semantic changes to memory objects such as adding a new field to a data structure. Additionally, these approaches are not inherently suitable for updating *non-quiescent* kernel functions that are always on the call stack of kernel threads. These updates still require rebooting the OS. Also, these approaches do not manage system configuration changes and shared component updates since the running processes are not restarted.

Second, some approaches need intimate knowledge about the target kernel. To benefit from such approaches, we have to develop special patches from the original ones. This task is non-trivial because it requires detailed knowledge on both the target kernel and the original patches at the source code level. The special patches include a procedure that checks whether the current kernel state is safe to dynamically translate code and memory objects. LUCOS [3] forces users to implement new functions that can handle kernel memory objects to keep them consistent before and after the translation. In DynAMOS [4], users have to investigate how the target functions are called by kernel threads and implement a routine that consistently updates the target functions. As pointed out in [3], developing special patches is a tedious and error-prone engineering task.

Lastly, we sometimes have to pay high engineering costs of redesigning and modifying a large part of the kernels. To incorporate K42's dynamic updatable functions [5]–[7] into commodity OS kernels, we have to

redesign the target kernels in an object-oriented manner. Modifying commodity OS kernels is difficult and often impossible because recent kernels are complex and some are closed-source and/or proprietary.

To eliminate the need for restarting applications in their updates, application-level dynamic update techniques have been widely explored [8]–[11]. These techniques dynamically update applications at runtime, but have limitations similar to the dynamic updatable kernels described above; some of them cannot handle semantical updates and others require the source code of the target applications. In addition, their applicability to kernel updates is obscure because kernels are much more complex than applications.

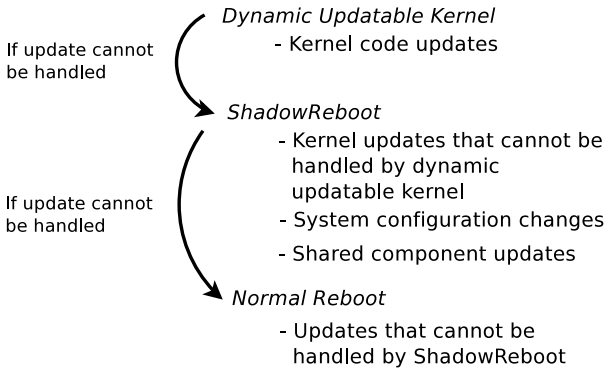


Fig. 1 ShadowReboot and current approaches.

2.2 Cluster Environments

Techniques of managing OS reboots have been traditionally explored in cluster computing environments. A typical example is a cluster rolling upgrade. This technique prepares a spare node that is used by applications when the OS is rebooted on the original node. An administrator shifts the applications from the original node to the spare node, shuts down the original node for software updates, then shifts the applications back to the original.

However, it is not reasonable to apply this model to a single machine such as a desktop PC or laptop. This is because rolling upgrades require a spare machine. To use the rolling upgrade model, a desktop PC or laptop user has to prepare a spare machine only for software updates.

To help cluster rolling upgrades, some approaches perform process migration. Autopod [12] migrates processes from the original node to the spare one to mitigate the impact of OS reboots on the services. MicroVisor [13] conducts process migration between two virtual machines (VMs) connected to a shared network storage such as an NFS server and a SAN. An administrator runs applications in one VM and maintains the kernel in the other VM. When maintenance finishes, the applications running on the older kernel in the first VM are migrated to the newer kernel in the second VM. Finally, the first VM is discarded.

Although these approaches successfully hide the downtime of kernel maintenance, process migration is not suitable for system configuration changes and shared component updates. Since migrated processes run with the configuration of the older OS, their states remain older on the newer OS. If the libraries linked with the migrated processes conflict with ones in the newer OS, the processes may not run correctly, or even worse, crash. An administrator has to determine which configuration or component is updated, and carefully choose the processes that can be migrated to avoid configuration mismatch of processes between the older and newer OS. This is quite difficult particularly for proprietary OSes such as Windows since update behavior such as with Windows Updates is unknown even to the administrators.

2.3 Summary

From the above discussion, we believe that there is an important room for managing an OS reboot in software updates. In this work, we explore a way to manage downtime of OS reboots, overcoming the limitations of current approaches. Figure 1 summarizes the relationship between ShadowReboot and current approaches. ShadowReboot is complementary to these approaches. Suppose that a software update is announced. If a dynamic updatable kernel is running, we first try to dynamically apply the update. If the update cannot be applied or the running kernel does not support dynamic updatable features, we invoke ShadowReboot. If the update still cannot be applied, we finally conduct a normal OS reboot.

3. Key Observations and ShadowReboot

ShadowReboot allows users to keep their applications running while the OS is rebooting. To produce an OS reboot effect successfully, ShadowReboot introduces a directory view and constraints based on file accesses patterns of users' applications and OS reboots. In this section, we describe key observations behind ShadowReboot, its overview, and its semantics.

3.1 Key Observations

In ShadowReboot, we exploit the file access patterns of commodity OSes during their reboots in software updates. We checked the directories and files accessed during the OS reboots after software updates. To obtain the names, we started monitoring the file accesses when an OS shutdown operation is triggered after a software update is completed. We continued to monitor the file accesses until the OS displays a login prompt. We ran five Linux distributions: Fedora Core 10 (`fedora`), Ubuntu 9.04 (`ubuntu`), Gentoo Linux 2007.0 (`gentoo`), CentOS 5.3 (`cent`), and OpenSUSE (`suse`). Their configurations are in default. For the five Linux distributions, we applied a kernel patch to each kernel and updated each `glibc` library.

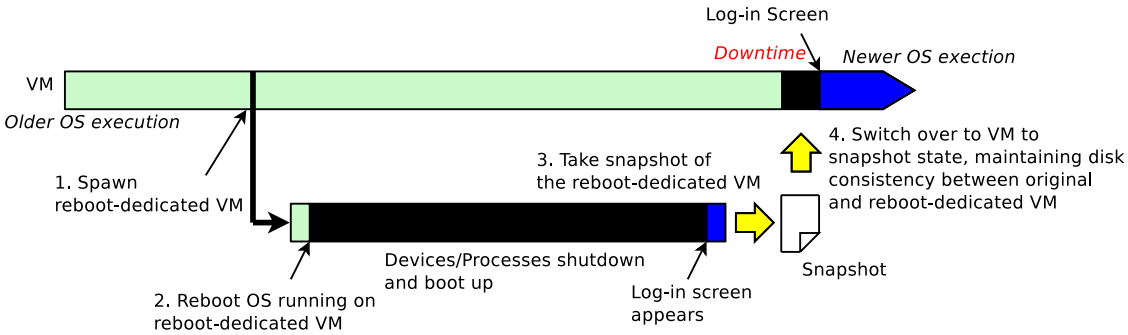


Fig. 2 An overview of ShadowReboot.

The results from the five Linux distributions show that during each reboot they all access the administrative files but never the user files in /home. All the Linux distributions frequently access files in /lib in their boot phase because almost all the daemon processes are linked to the glibc shared library whose files are stored in /lib/. In addtion, the files in /etc are often accessed because the configuration files are conventionally in /etc. Each Linux distribution performs slightly different file accesses due to the configuration difference. For example, `fedora` accesses /lib/libselinux.so, while `gentoo` does not. This is because `gentoo` does not support the selinux service that `fedora` does.

These results indicate that files accessed during OS reboots tend to be in administrative directories that cannot be modified without an administrative privilege. In other words, almost no files in user directories, such as files in /home directories, are accessed. The characteristics of the file access patterns give us the following points. Even if we heavily modify files in the user directories while the OS is simultaneously rebooting, the modification does not interfere with the reboot activity. Moreover, the modification of files in administrative directories by service processes such as Linux daemons and Windows services does not disturb common users' applications. This motivates us to execute the users' tasks and an OS reboot in parallel.

3.2 ShadowReboot

ShadowReboot leverages system virtualization. Virtual machine monitors (VMMs) are a software layer on which existing OS kernels can be executed without any modification. A new feature implemented inside a VMM becomes available to all the guest OSes running on it. System virtualization is commonplace in desktop PCs and laptops as well as data centers.

An overview of ShadowReboot is shown in Fig. 2. ShadowReboot executes users' tasks in parallel with the OS reboot. To create the context of an OS reboot, ShadowReboot spawns a VM dedicated to an OS reboot, called a *reboot-dedicated VM*. It has a copy of the virtual disks of the original VM and identical resources such as memory and registers. In ShadowReboot, an OS is rebooted on the reboot-dedicated VM after software updates are applied, while the user is executing applications on the original VM. After the OS reboot is completed, ShadowReboot takes a snapshot of the reboot-dedicated VM. ShadowReboot enables the user to restore the snapshot state at a convenient time, providing the directories in a manner described in Sect. 3.3. This design also allows us to treat a given update as a *blackbox* since the concealed OS reboot refreshes the software component in the same way as the normal OS reboot. Due to this advantage, ShadowReboot does not require analysis of the update patches and is applicable to existing updates such as Windows Updates.

We note that applications that save some files during their shutdown phase need to be terminated before ShadowReboot execution in order to keep their behavior consistent across ShadowReboot. ShadowReboot can be performed in a conservative way if the users do not know if such applications create and/or write some files in their shutdown phase. In this case, we restore the snapshot generated from the reboot-dedicated VM after the original VM shutdown has been complete since all applications are terminated during the shutdown. Although this conservative ShadowReboot guarantees consistency of such applications, downtime gets longer. If an end user know the running applications can behave correctly without their termination phases, the user does not need to wait for the OS shutdown, performs ShadowReboot if he or she wants to do.

3.3 ShadowReboot Semantics

3.3.1 Challenge

The parallel execution of the applications and OS reboot poses a challenge: how can we to maintain disk consistency in a rebooted state? Since files can be modified simultaneously by both activities of the applications and OS reboot, we may fail to produce a reboot state users expect. For example, when the service processes have been launched in the concealed OS reboot before the user's task modifies the configuration files on the original VM, their running states in the produced rebooted image are based on the files before modification. Another problem is that computational results saved by the applications are overwritten when the

concealed OS reboot activity modifies the files. As a result, some data that the user does not expect would be saved in the application's files. Although we can solve this problem by carefully tracking service process and users' task behavior, semantically maintaining consistency is complicated and may require users' interaction like `fsck`, which would discourage users from employing ShadowReboot.

3.3.2 ShadowReboot Directory View

To cope with this problem, ShadowReboot builds directories in the restored VM by appropriately selecting directories from the original and reboot-dedicated VM. Specifically, ShadowReboot selects the user-specified working directories in the original VM to preserve the user's computational results during the OS reboot. The other directories including administrative directories come from the reboot-dedicated VM. This directory view is based on the fact that user's non-administrative applications tend to modify only their own working directories, which are typically stored in the user directory, while reading the shared libraries and system configuration files stored in administrative directories. Also, the view allows us to maintain the consistency between the service processes' states and the accessed files such as configuration files since the service processes access the administrative directories in the reboot-dedicated VM.

For example, suppose a user is executing a video player or a word processor during shadow-rebooting. He or she can use these applications as usual even under the ShadowReboot constraints since they edit files in user's directories and typically read administrative files such as shared libraries. At this time, the user cannot conduct administrative tasks such as system configuration changes due to the constraint in order to maintain the consistency between the configuration and the service processes states saved in the rebooted image. In the context of the OS reboot, the kernel and service processes are ready for use. After the OS reboot, ShadowReboot returns the consistent rebooted state with user's expectations, providing its directory view; it preserves user's computational results in the working directories and administrative files whose contents are consistent with the running service processes.

3.3.3 Constraints

To successfully produce a consistent rebooted state with users' operations, ShadowReboot enforces constraints on the original and reboot-dedicated VM. One constraint forces applications running on the original VM to modify only the working directories specified in advance, which is used after the restoration of the produced snapshot. This means that the constraint prevents the applications from updating files in the other directories that are discarded after the restoration. Although the user cannot perform administrative tasks such as system configuration changes during this time, ShadowReboot allows them to execute non-administrative tasks such as web browsing and text editing. To avoid conflicting with the applications running on the original VM, ShadowReboot also imposes a constraint that forbids the concealed OS reboot to modify the files in the applications' working directories. When file operations performed in the original VM or reboot-dedicated VM violate the constraints, ShadowReboot notifies the user of the violation and destroys the reboot-dedicated VM. After that, he or she can try ShadowReboot again or perform a normal OS reboot.

We can mitigate the constraints with knowledge of the applications. Specifically, we can do this by ignoring updates of directories defined in advance. Even if some files are modified simultaneously by both VMs, file contents are consistent with user's file operations and service processes' states in the restored VM. For example, service processes sometimes cache their data into a directory such as /var/cache. Another example is that some processes temporally save their results in some directories such as /tmp. The updates in these directories can be ignored during ShadowReboot.

Another way to mitigate the ShadowReboot constraints is to configure applications to redirect their file writes to user directories. By doing so, we can bring the modified files to the restore VM. For example, if application's log files in the system directory are frequently updated, we can keep the updates across ShadowReboot by changing the applications' configuration to write their log files in user directories.

Specifically, we found three types of files for which the constraints can be mitigated: temporary files, state files, and log files. Temporary files include cache files such as files in /var/cache and temporal results such as files in /tmp. Even if these files are updated in the original VM and discarded through the restoration of ShadowReboot, the restored VM can consistently start, as described above. State files include pid files such as /var/run/yum.pid and /var/run/anacron.pid, and lock files such as /var/lock/makewhatis.lock and /var/lock/subsys/vsftpd. Since these files depend on the states of running service processes, we do not track updates to them in the original VM since they are appropriately generated in the reboot-dedicated VM. Log files include application's logs such as /var/log/messages. The update of log files is a main cause of violations of constraints. To solve this problem, we reconfigure logger services to switch their log file to files in a working directory just after spawning a reboot-dedicated VM. By doing so, we can avoid the constraints violation and preserve logs containing events happened in the original VM. On the basis of these points, we can configure the five Linux distributions to be shadow-rebootable, as described in Sect. 7.

Note that configuration changes for end users to make their system shadow-rebootable can be avoided completely in some scenarios. One compelling use case of ShadowReboot is in a cloud computing platform. The cloud provider can provide a selection of operating system images with the ShadowReboot configuration changes already applied. Such a scenario completely eliminates the need for the user to do any OS configuration themselves. This means

that the user can benefit from ShadowReboot even if he or she is a novice.

4. Design Details

Designing ShadowReboot poses several questions: (1) how can we efficiently spawn a reboot-dedicated VM? (2) how can we appropriately restore directories from the original and reboot-dedicated VM? (3) how does ShadowReboot check whether the applications and the concealed OS reboot follow the constraints? To solve the first question, we design a *VM fork*, which is a technique for forking a running VM, that is semantically based on the familiar process fork. Against the second question, we introduce *unrollback virtual disks* whose contents are never rolled back by snapshot restores. Our solution for the last question is to monitoring file accesses in the original and reboot-dedicated VM to check whether each VM violates ShadowReboot constraints. In this section, we describe our solutions in details.

While none of the above techniques is new, the novelty of our work is that it encompasses all of them in achieving its goals; mitigating the disruption to users' computational activities caused by OS reboots in software updates by shortening their downtime.

Note that this paper mainly focuses on basic mechanisms required for ShadowReboot, as described Sect. 1. Automatic execution of VM fork and snapshot saving/restoring is attractive to encourage users to use ShadowReboot. Although we manually invoke VM fork and snapshot operations in the current prototype, we believe that this can be executed automatically by preparing processes issuing hypercalls. For example, we achieve automatic VM fork invocation by modifying an updater to issue a hypercall to fork the VM after completing the update. On the other hand, we achieve automatic snapshot saving by preparing a last launching service process during OS boots that issues a hypercall to save snapshot. Also, we do automatic snapshot restoration by preparing a process issuing a hypercall for snapshot restoration. This process is executed by users or after all other processes are stopped.

4.1 VM Fork

We need an efficient approach to creating a reboot-dedicated VM. A naive approach is to run a new VM instance with the same configuration as the original VM. However, at every announcement of software updates, we have to create a new VM instance, copy the image of the VM, boot the OS, update the software, and conduct an OS reboot. However, this is tedious and may discourage users from updating software. Although these procedure can be automated with an engineering cost to avoid the tedious task, time during which the original and new VMs run in parallel is long; the new VM is active while booted, updated and rebooted. This long time provides more opportunities for applications to violate ShadowReboot constraints. We should make this time as short as possible.

To create a reboot-dedicated VM efficiently and make its active time as short as possible, we introduce a *VM fork* that forks a running VM, borrowing an idea from previous research [14], [15]. The semantics of the VM fork are similar to those of the familiar process fork; users issue a fork call to the VMM that creates a child VM. The child VM inherits the runtime state of the parent VM such as memory and registers. It then proceeds with an identical view of the system. The child VM has its own independent copy of the OS, virtual disk, network interface card (NIC), and snapshot. The state updates of the child VM are not propagated to the parent.

To avoid network configuration conflicts between parent and child VMs, we employ a way to drop in/out packets of child VM until its shutdown is completed. And the shutdown completion, new MAC addresses are assigned to the child VM. This way is simple but imposes limitation that the child VM cannot inherit network services. For example, we cannot use networked storage services such as NFS; the child VM cannot detach these services in a correct manner.

Our VM fork is semantically different from the other ones [14], [15]. Flash cloning [15] swiftly clones the VM from the reference image using a copy-on-write technique. However, this cannot provide stateful runtime cloning; all new VMs are copies of a frozen template. SnowFlock [14] is designed to clone VMs to use them temporally to handle sudden and huge workloads. It discards the state changes of child VMs when they stop. Our VM fork preserves the disk changes conducted by a child VM to restore the directories of the reboot-dedicated VM.

Note that we can use a page reclaiming mechanism running inside the VMM to efficiently utilize memory in forking a VM. We run it if there is not enough memory space to execute a child VM, though the resources in desktop environments are basically idle [16], [17]. We can make use of a page sharing technique like memory ballooning [18]. It allows one physical page to be shared with several virtual pages whose contents are the same. The page sharing mechanism reclaims the memory pages of the process, which means the number of free memory pages increases. Therefore, we can reclaim the memory pages for a child VM without needing kernel modules such as a balloon driver. If we need to share pages more aggressively, novel sharing techniques [19], [20] can be employed.

4.2 Use of Unrollback Virtual Disks

To build working directories in the restored VM, we leverage an unrollback virtual disk that is independent of snapshot restoration. Unlike normal virtual disks, unrollback virtual disks do not roll back even if the VM is restored to a snapshot. By saving files into an unrollback virtual disk in the original VM, the files are accessible in the VM restored from a snapshot of the reboot-dedicated VM. We prepare two policies for the use of unrollback virtual disks: *all-copy* and *partial-copy*. Users can choose the one more suitable for their environments.

- **All-copy policy:** The all-copy policy records update operations that occur in the working directories of the original VM during shadow-rebooting, and replays them on the restored VM. We start monitoring file update operations (including write, remove, and rename) to the working directories just after a reboot-dedicated VM is spawned. When the rebooted state is about to be restored, we mount an unrollback virtual disk on the original VM, shut down applications, and save the file update operations into the log in the mounted directory. After the logging completes and the rebooted snapshot has been restored, we mount the unrollback virtual disk again and replay the recorded operations in the restored VM. We can execute the all-copy policy without any modification in the existing directory layout. However, saving and replaying file update operations take longer as more files are updated.
- **Partial-copy policy:** The partial-copy policy *partially* saves and replays file update operations at the expense of the use of the existing directory layout. Under this policy, we assign some working directories to a partition of the unrollback virtual disk. Since an unrollback virtual disk keeps its contents through snapshot restoration, updated files and directories in the partition remain after the restoration of the rebooted state, thus accessing them without replaying any file operation. We monitor file update operations to the other working directories and log and replay them like the all-copy policy. A typical partial-copy configuration of Linux systems is that the mount point of the working directory (/home/users/work) is assigned to the unrollback virtual disk and the other directories' mount points are assigned to standard virtual disks. When we restore the snapshot of the reboot-dedicated VM, the /home/users/work directory is not restored because its mount point is assigned to the unrollback virtual disk.

Similar to unrollback virtual disks, some approaches can protect the files and directories from snapshot restoration. We can protect them by using an additional VM on which an NFS server is running. The files and directories that are stored in the NFS server are not affected by snapshot restoration. However, this approach requires setting up a VM and incurs network virtualization overhead that tends to cause a large performance penalty. We can also protect the files and directories by sharing them with the host OS. Although they are not rolled back by snapshot restoration, users sometimes want isolation between the VMs and the host to protect the host against VMs compromised by viruses or malicious attacks.

We need to carefully configure a guest OS so that the OS does not mount the partitions of the unrollback virtual disks in its boot phase. Common file systems read their metadata, such as super blocks, from disks only once when the partitions are mounted. After a file system has been mounted, it manages its metadata in memory and only writes updates to the disks. When we take a snapshot of the reboot-dedicated VM after the unrollback virtual disks' partitions have been mounted, the restored file system states are inconsistent with the disk contents. Since the restored file system objects are not reflected on the disk updates conducted on the original VM after the OS mounts the partitions on the reboot-dedicated VM, the user cannot access the updated contents.

4.3 File Access Monitor

To notify a user of the violation of the ShadowReboot constraints, we have developed a mechanism that checks whether or not user's tasks and the OS reboot follow the constraints. The mechanism consists of two processes. One monitor runs on the reboot-dedicated VM and monitors file accesses to the working directories to detect file updates violating ShadowReboot constraints. The other monitor runs on the original VM. It monitors file operations to user-specified working directories to log them in order to realize the all-copy and partial-copy policies. It also monitors file accesses to the other directories to detect the violations of the constraints. When a violation is detected on a VM, the process notifies the user and requests the VMM to destroy the reboot-dedicated VM.

The file access monitor also helps us set up our system configuration to be shadow-rebootable. Since the file access monitor checks whether or not the system follows the constraints through shadow-rebooting, we can configure the service processes and our applications to be suitable for the constraints, based on the checks. We believe that the file access monitor makes it easier to configure the VM to be shadow-rebootable.

5. Implementation

We implemented a prototype of ShadowReboot on VirtualBox 4.0.10_OSE [21] and Linux. VirtualBox is an open-source VMM for x86 hardware, where we can execute OSes such as Windows and Linux without any modification to them. Our prototype consists of three modules: *vmm-module*, *guest-module* and *host-module*. The vmm-module is a part of VirtualBox. It provides and performs VM fork and unrollback virtual disks. The guest-module is a process running on the guest OS. It executes the file access monitor. The host-module runs on the host Linux executing the VirtualBox. The host-module requests the VMM to fork a VM and take/restore a snapshot. It exchanges network messages with the guest-module, which requests a VM fork. We describe implementation issues of the VMM-module and guest module in this section.

5.1 VMM Module

To implement the VM fork, we use online snapshot functionality. Online snapshot functionality enables us to take a snapshot without the downtime of the VM. When a snapshot of a VM is taken, VirtualBox produces two files. One

contains the current memory image of the VM. The other is a delta disk file for preserving the current state of the virtual disk. When a VM fork is requested, the vmm-module first registers a new VM instance for a child VM and sets the same hardware configuration as the target VM such as memory size and virtual disks. Next, it takes a snapshot of the target VM and sets the child VM's memory state to the memory image and creates delta disk files for each virtual disk. After that, the new VM is launched as the child VM.

To implement unrollback virtual disks, we extended a type of virtual disk named write-through. Write-through disks fully support read and write operations like normal disks. The difference from normal disks is that the state of write-through disks is not saved when a snapshot is taken and not restored when a VM's state is reverted. We extended the write-through disk so that multiple VMs can connect to them. When the VM fork is invoked, it creates two delta files to save the disk updates committed by each VM. When we take a snapshot of the reboot-dedicated VM and terminate it, the snapshot is connected to the delta file for the original VM and the other one is discarded.

5.2 Guest Module

We implemented a file access monitor on Linux with the i-notify function. The i-notify function allows us to monitor file operations for specified files and directories. When ShadowReboot is invoked, our file access monitor starts to record file writes, deletions, and creations to administrative directories on the original VM. It continues to run until we restore the snapshot of the reboot-dedicated VM. On the other hand, it monitors the user-specified working directories on the reboot-dedicated VM. After the shutdown phase, it does not monitor files since the the partitions containing working directories are not mounted. To lower the monitoring cost, we do not monitor pseudo file systems such as /proc and /sys, and device file directories such as /dev.

Also, the file access monitor logs file operations to user-specified directories on the original VM and memorizes the operated file names. The file access monitor mounts an unrollback virtual disk just before the produced snapshot is restored. For simplicity of the implementation, the current prototype copies the updated files to a partition in the unrollback virtual disk. It copies them to the restored VM by mounting the unrollback virtual disk in it just after the restoration. To enhance our implementation, the use of some tools such as versioning systems is attractive for effectively managing file update operations.

6. Discussion

Although the prototype is runnable only on Linux platforms, we believe that ShadowReboot can be applied to other OS platforms. Our vmm-module can be reused for other OSes running on the VirtualBox since it is a part of the VMM. The file access monitor can be implemented with file I/O monitoring mechanisms supported by the target OS. For example, Windows supports the filter driver mechanism that allows us to monitor file system events. One of our future directions is to implement a file access monitor on Windows, configure Windows to be shadow-rebootable, and confirm the effectiveness of ShadowReboot.

We note that ShadowReboot cannot handle all types of software updates: it fails to manage software updates that involve accessing the directories on unrollback virtual disks. This behavior violates the ShadowReboot constraints, which means that our file access monitor detects this violation and stops shadow-rebooting. In this case, the user needs to conduct a normal OS reboot.

Controlling the resource usage of a reboot-dedicated VM is another challenge. If the reboot-dedicated VM obtrusively utilizes computational resources, it interferes with the original VM so severely that we cannot adequately do our tasks. To minimize the interference of the reboot-dedicated VM, we schedule it as a background task and the original VM as a foreground task. Many schemes for properly scheduling foreground and background processes have been proposed [16], [17], [22]. We can employ these novel schemes to mitigate the interference of the reboot-dedicated VM.

We pay attention to a case where the user wants to assign a fixed IP address to the target VM. When the user does not use a fixed IP address, he or she can enjoy the network without any consideration since our VM fork provides the child VM virtual NICs whose mac addresses are different from the parent ones. When the user wants to assign the VM a fixed IP address through the OS configuration or a DHCP server, we need to extend the current ShadowReboot. To assign a fixed IP address after shadow-rebooting, ShadowReboot provides the child VM virtual NICs whose mac addresses are the same as those of the parent. When a reboot-dedicated VM is rebooted and the guest OS starts to boot, we take a snapshot before the guest OS turns on its NICs. Although it takes a longer time until we can perform our tasks since some services may not finish launching, the user can obtain a fixed IP address.

7. Experiments

We conducted experiments to examine the effectiveness of ShadowReboot. In this paper, we investigate the following fundamental questions. The first is how ShadowReboot shortens downtime of OS reboots. The second is whether our page sharing mechanism shares pages. The third is how much overhead the file access monitor incurs. The fourth is how long the disk managements of all-copy and partial-copy policies take. The fifth is whether ShadowReboot can successfully produce a rebooted state under real software updates. The last is how applications behave through ShadowReboot.

The experiments described in this section were conducted on a DELL OptiPlex 780DT with a 3.0 GHz Core 2 Duo processor, 4 GB of memory and a 160 GB SATA disk. Our prototype runs on this machine on which Linux

2.6.34 runs. To confirm the applicability of ShadowReboot, we used five Linux distributions, Fedora Core 10 (`fedora`), Ubuntu 9.04 (`ubuntu`), Gentoo Linux 2007.0 (`gentoo`), CentOS 5.3 (`cent`), and OpenSUSE (`suse`). These OSes were installed on VMs provided by VirtualBox. Each VM is assigned one VCPU and connected to a 20 GB normal virtual disk and a 10 GB unrollback virtual disk as a primary master and slave, respectively. Its memory size was changed in the experiments. We installed the OSes with their desktop configurations. The normal virtual disk was partitioned by the default install instruction, and the unrollback virtual disk was formatted manually. In the experiments, the default system configurations were used.

7.1 Experimental Setup

We configured the five Linux distributions to be shadow-rebootable with our file access monitor. In addition, we checked whether installed applications adhere to the ShadowReboot constraints. These configurations were carried out manually. Developing an automatic configuration scheme to set the system suitable for ShadowReboot is out of the scope of this paper.

The configuration details are as follows.

- `Fedora`: We change `rsyslog`, `auditd`, and `sendmail` configurations to switch their log files on the original VM just after a VM fork is invoked. To do so, when the file access monitor launches on the original VM, it modifies /etc/rsyslog.conf, /etc/audit/auditd.conf, and /etc/mail-/sendmain.cf so that their files can be stored into a file in the working directories to preserve the contents through the restoration. We also configure the file access monitor to avoid monitoring temporary and state files. Specifically, it does not monitor directories including /tmp, /usr/tmp, /var/tmp, /var/lock, /var/cache, or /var/run.
- `Ubuntu`: Similarly to `fedora`, we modify the configuration file of a `syslog` daemon. Specifically, the file access monitor modifies /etc/syslog.conf to switch log files into files in the working directories on the original VM just after a VM fork is invoked. Moreover, the file access monitor running in the original VM does not monitor files in /tmp, /var/tmp, /var/lock, /var/cache, /usr/tmp, /var/run, or /run. In addition, the file access monitor ignores file updates to /var/lib/apt-xapian-index/update-lock and /var/lib/apt-xapian-index/update-socket, which are state files of the Ubuntu update system.
- `Gentoo`: We change a `syslog-ng` configuration to switch a log file on the original VM when the VM fork is invoked. The file access monitor modifies /etc/syslog-ng.conf for `syslog-ng` to store log events into a file in the working directories. Also, the file access monitor running in the original VM does not monitor files in /tmp, /var/tmp, /var/lock, /var/cache, /usr/tmp, or /var/run.
- `Cent`: We reconfigure `sendmail` and `syslog` to switch the directories where their produced files are stored. We modify /etc/syslog.conf and /etc/mail-/sendmail.cf on the original VM. The file access monitor does not log file operations to /tmp, /usr/tmp, /var/tmp, /var/lock, /var/cache, or /var/run.
- `Suse`: Similarly to the other four Linux distributions, we change `rsyslog` and `postfix` configuration to change log files. The file access monitor modifies /etc/rsyslog.conf and /etc/postfix/main.cf to write events in files in the unrollback virtual disk. The file access monitor does not log file operations to /tmp, /usr/tmp, /var/tmp, /var/lock, /var/cache, or /var/run.

Some applications log their states to files when they finish. For example, `firefox` saves its configurations into the user's directory. If such applications are running when a VM fork is invoked, the files are saved in the disks in the reboot-dedicated VM, which violates a ShadowReboot constraint. To avoid this problem, we configured the VM in such a way that these files on the original VM are used on the restored VM. Specifically, the file access monitor brings the files from the original VM with the unrollback virtual disk. This is because such state files are the latest in the original VM.

In our investigation, one application (`terminal`) violates ShadowReboot constraints by logging a login event to /var/log/wtmp when it launches. To use this applications through shadow-rebooting, we need to switch the log files to a file in the unrollback virtual disk.

7.2 Downtime

To demonstrate that ShadowReboot shortens downtime of OS reboots, we compared the downtime of ShadowReboot and normal OS reboots. Our prototype causes downtime when a snapshot of a rebooted state is restored. We measured downtime caused by the snapshot restores. We varied the VM memory size: 256 MB, 512 MB, 1024 MB, 2048 MB, and 2560 MB. The maximum memory size the VirtualBox can assign in our environment was 2560 MB. We here define downtime as time during which users cannot operate the system (i.e. time from when snapshot restoration or shutdown is started to when a log-in screen is displayed.

Table 1 lists the downtimes of ShadowReboot and normal OS reboots. The results show that the downtime of ShadowReboot is shorter than that of normal OS reboots. For example, with 256 MB, the downtime of ShadowReboot is 98.3% shorter than that of the normal OS reboot in `cent`. Even in `ubuntu`, the downtime of ShadowReboot is 91% shorter than that of the normal OS reboot. When we assigned 2560 MB of memory, downtime of ShadowReboot is 1.42 seconds in `gentoo`, while that of the normal OS reboot is 58.21 seconds. ShadowReboot downtime is 2.49 seconds in `ubuntu`, which means ShadowReboot is 94% shorter than the normal OS reboot.

Also, the downtime of restoring a rebooted state tends

Table 1 Downtime of ShadowReboot and Normal OS Reboot.

	fedora [sec]		ubuntu [sec]		gentoo [sec]		cent [sec]		suse [sec]	
VM Memory Size	Shadow Reboot	Normal Reboot	Shadow Reboot	Normal Reboot	Shadow Reboot	Normal Reboot	Shadow Reboot	Normal Reboot	Shadow Reboot	Normal Reboot
256 MB	2.49	42.23	2.43	27.47	1.38	55.39	2.42	141.11	3.16	30.95
512 MB	2.56	42.80	2.41	27.63	1.43	54.89	3.32	154.09	4.86	30.71
1024 MB	2.38	44.55	2.46	39.61	1.38	57.82	3.20	132.84	5.12	43.29
2048 MB	2.37	45.03	2.53	43.64	1.43	58.17	3.15	142.83	5.29	56.24
2560 MB	2.39	45.04	2.49	45.49	1.42	58.21	3.35	132.05	5.22	55.99

Table 2 Downtime of ShadowReboot used in a conservative way.

VM Memory	fedora [sec]	ubuntu [sec]	gentoo [sec]	cent [sec]	suse [sec]
256 MB	9.56	15.82	14.18	25.72	12.36
512 MB	10.55	15.98	14.73	26.32	13.43
1024 MB	10.85	20.57	17.08	32.03	21.85
2048 MB	10.09	19.67	17.93	30.68	23.95
2560 MB	10.94	19.61	17.32	30.97	27.84

Table 3 Overhead of the file system monitor.

	W/o File System Monitor [sec]	W/- File System Monitor [sec]	Overhead [%]
grep	24.37	24.52	0.61
make	174.35	176.18	1.05

to be stable even if the memory size is varied, except for cent and suse. In VirtualBox, the downtime of restoring a snapshot depends on how much memory a guest OS utilized. In cent and suse, their daemons utilize the memory in their boot phase, depending on the memory size of the machine. For example, readahead_early warms the file cache by accessing files that are frequently used.

Table 2 shows downtime of ShadowReboot used in a conservative way that restores snapshots after shutdown of the original VM is completed. The downtime is longer than one of ShadowReboot shown in Table 1 since we have to wait for completion of shutdown of the original VM. Compared with normal reboots, downtime of the conservative ShadowReboot is shorter in all the case.

7.3 Overhead for File Monitoring

To measure overhead incurred by the file system monitor, we compared execution times of the applications with and without the file access monitor. We prepared two applications: grep and make. Grep searches for lines containing 'shadowreboot' in the source code and documents of Linux 2.6.29. Make compiles Apache 2.0.64 [23]. We ran these applications on fedora with 1600 MB of memory, which is the size recommended by the VirtualBox.

The result is shown in Table 3. From the result, we can say that the overhead of the file system monitor is very small. The i-notify is a lightweight monitoring mechanism where a monitoring process can run asynchronously with processes accessing the monitored files. This feature seems to contribute to this low overhead. The overhead in grep is 0.61%, while that in make is 1.05%. The reason make's overhead is larger than that of grep is that make reads more library object files in administrative directories that are monitored by the file access monitor.

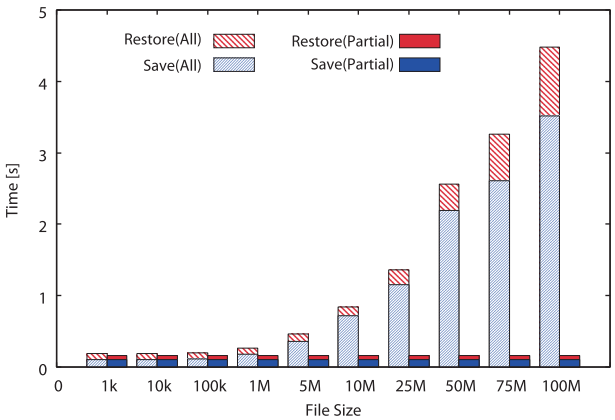


Fig. 3 Time for saving/restoring files under all- and partial-copy policies.

7.4 Overhead for File Saving/Restoring

To measure overhead for file saving to and restoring from unrollback virtual disks, we measured time for saving and restoring files under all-copy and partial-copy policies. We used fedora with 1600 MB of memory. We generated files of various sizes during an OS reboot in the reboot-dedicated VM. We generated the files in a partition in the unrollback virtual disk under the partial-copy policy.

The results are shown in Fig. 3. The results reveal that the all-copy policy takes longer as the file size becomes bigger. This is because the all-copy policy copies all the updated files in the working directories to a partition of the unrollback virtual disk. On the other hand, the partial-copy policy does not copy the files in the unrollback virtual disk. When we create 1 KB, 10KB, and 100KB of files, required times in the all-copy policy are about 200 msec, which are similar to ones of the partial-copy policy. When the file is more than 25MB, the required time is longer than one second. The time of the partial-copy policy is constant regardless of file sizes since the files are created in the unrollback virtual disk and thus need not be copied.

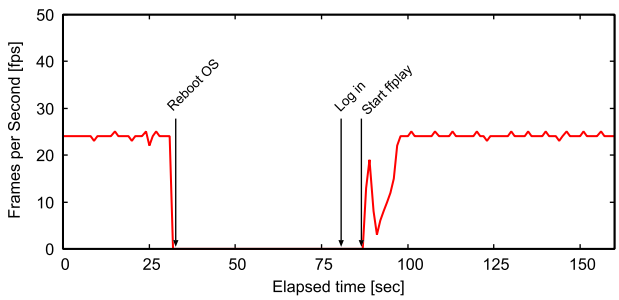
7.5 Software Updates

To confirm that ShadowReboot successfully performs given software updates, we conducted real software updates on the five distribution. We chose updates including kernel updates, library updates, and window system updates by using the package system each distribution employs. We conducted 43 updates on `fedora`, eight on `ubuntu`, four on `gentoo`, 19 on `cent`, and 24 on `suse`.

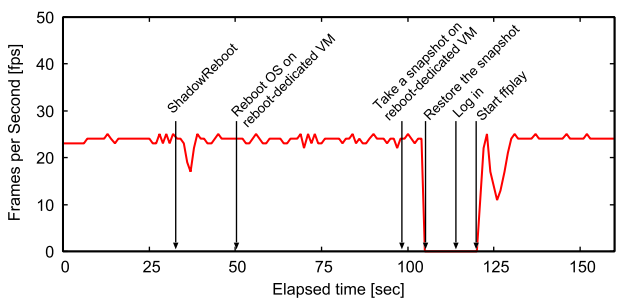
The result is that all of the software was successfully updated through ShadowReboot. During the OS reboots in the reboot-dedicated VM, our file system monitor does not warn of any violations in the updates. It detects a violation when we update software in `cent`. This is because `logrotate` is triggered by `cron` on the original VM and then starts compressing the log files in /var/log. We successfully performed the update by invoking ShadowReboot after the `logrotate`'s task completes.

7.6 Case Study

To demonstrate how a user's application behaves through ShadowReboot, we observed application behavior through the normal OS reboot and ShadowReboot. We ran a video player named `ffplay`, which is bundled in `ffmpeg` [24]. It plays a MPEG-4 format video at 24 frames per second (fps) with resolution 854 x 480 on `fedora` with 1600 MB of memory. We started `ffplay` and recorded its fps. After 30 seconds had been passed, we performed each reboot. When the log in prompt appears, we logged in `fedora` and restarted `ffplay`.



(a) OS Reboot



(b) ShadowReboot

Fig. 4 Frames per seconds in `ffplay` during normal OS reboot and ShadowReboot.

The results are shown in Fig. 4. From Fig. 4, we can see that ShadowReboot downtime is shorter than that of the normal OS reboot. In the normal OS reboot (Fig. 4 (a)), `ffplay` stopped its activity at 30 seconds when we conducted the OS reboot. It could not proceed until the OS reboot completed. After we logged in `fedora` and executed `ffplay`, it restarted playing the video. In ShadowReboot (Fig. 4 (b)), `ffplay` could proceed when ShadowReboot was performed. We rebooted the OS on the reboot-dedicated VM at the 50 seconds when the VM fork finished and the reboot-dedicated VM started to run. During the OS reboot, `ffplay` continued to decode the video as usual. We took a snapshot on the reboot-dedicated VM at 99 seconds when a log in prompt appeared on the reboot-dedicated VM. When we restored the snapshot, `ffplay` stopped (at 105 seconds). It was executed again after we finished logging in. In both cases, `ffplay`'s performance was degraded just after we restarted it. This is because other processes were running to set up the user's desktop environment and thus resource contention occurred.

8. Related Work

Some studies have explored ways to manage downtime of OS reboots. Phase-based Reboot [25] shortens the downtime of reboot-based recovery. It takes snapshots every boot phase such as OS kernel boot and service process boot phases, and reuses them if the next boot is the same execution as the previous boot. Since Phase-based Reboot focuses on reboot-based recovery and reuse of the previous states, it is not applicable to software updates.

Otherworld [26] hides the kernel termination from the user-level applications. When a kernel failure occurs, Otherworld restarts only the OS kernel, keeping the user-level memory states of the processes. After the OS kernel has been rebooted, the processes are resumed. However, Otherworld still has a longer downtime than ShadowReboot. Furthermore, Otherworld is not applicable to software updates since an OS kernel, which is loaded when the main kernel is stopped, needs to be set up when it launches.

The shadow driver technique [27] conceals device driver crashes from user's applications. When a device driver crashes, the shadow driver hooks the communications between the kernel and devices, restarts the crashed driver, and queues the messages until its restart completes. The shadow driver transmits the messages to the restarted driver. This technique also allows us to efficiently update device drivers [28]. While the shadow driver technique focuses on device driver restarts, our focus is on OS restarts.

9. Conclusion

This paper presented ShadowReboot, a VMM-based approach that shortens downtime of OS reboots in software updates. ShadowReboot provides an illusion that a guest OS travels *forward* in time to the rebooted state where the updated kernel and applications are ready for use. Specif-

ically, ShadowReboot conceals OS reboot activities from user's applications by spawning a VM dedicated to an OS reboot and systematically producing the rebooted state. Our experimental results show that ShadowReboot succeeded in software updates on unmodified commodity OSes and has 91 to 98% shorter downtime than commodity OS reboots on the five Linux distributions.

One future direction is to exploit cloud environments for rebooting an OS by combining cloud-aware clone techniques with ShadowReboot. Recent studies [29], [30] have shown techniques that clone the system states and send the cloned ones to the cloud platform. By spawning reboot-dedicated VMs in the cloud, we can perform ShadowReboot with less local resource contention with the original VM. In other words, we will build a "Reboot as a Service" platform that reboots an OS transparently for users and sends the rebooted image back to them, which will be able to encourage users to perform software updates more often.

References

[1] G. Kroah-Hartman, J. Corbet, and A. McPherson, "Linux kernel development," the Linux foundation, 2009.
[2] J. Arnold and M.F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," Proc. 4th ACM European Conference on Computer Systems (EuroSys '09), pp.187–198, April 2009.
[3] H. Chen, R. Chen, F. Zhang, B. Zang, and P.C. Yew, "Live updating operating systems using virtualization," Proc. 2nd ACM International Conference on Virtual Execution Environments (VEE '06), pp.35–44, June 2006.
[4] K. Makris and K.D. Ryu, "Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels," Proc. 2nd ACM European Conference on Computer Systems (EuroSys '07), pp.327–340, March 2007.
[5] O. Krieger, M. Auslander, B. Rosenburg, R.W. Wisniewski, J. Xenidis, D.D. Silva, M. Ostrowski, J. Appavoo, M. Butrico, M. Mergen, A. Waterland, and V. Uhlig, "K42: Building a complete operating system," Proc. 1st ACM European Conference on Computer Systems (EuroSys '06), pp.133–145, April 2006.
[6] A. Baumann, J. Appavoo, R.W. Wisniewski, D.D. Silva, O. Krieger, and G. Heiser, "Reboots are for hardware: Challenges and solutions to updating an operating system on the fly," Proc. USENIX Annual Technical Conference (ATC '07), pp.337–350, June 2007.
[7] C.A.N. Soules, J. Appavoo, K. Hui, R.W. Wisniewski, D.D. Silva, G.R. Ganger, O. Krieger, M. Stumm, M. Auslander, M. Ostrowski, B. Rosenburg, and J. Xenidis, "System support for online reconfiguration," Proc. USENIX Annual Technical Conference (ATC '03), pp.141–154, June 2003.
[8] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "OPUS: Online patches and updates for security," Proc. 14th USENIX Security Symposium (Security '05), pp.287–302, Aug. 2005.
[9] H. Chen, J. Yu, R. Chen, B. Zang, and P.C. Yew, "POLUS: A powerful live updating systems," Proc. 29th International Conference on Software Engineering (ICSE '07), 2007.
[10] K. Makris and R.A. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction," Proc. USENIX Annual Technical Conference (USENIX '09), pp.397–410, June 2009.
[11] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for C," Proc. 2006 ACM Conference on Programming Language Design and Implementation (PLDI '06), pp.72–83, June 2006.
[12] S. Potter and J. Nieh, "Reducing downtime due to system maintenance and upgrades," Proc. 19th USENIX Large Installation System Administration Conference (LISA '05), pp.47–62, Dec. 2005.
[13] D.E. Lowell, Y. Saito, and E.J. Samberg, "Devirtualizable virtual machines enabling general, single-node, online maintenance," Proc. 11th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '04), pp.211–223, Oct. 2004.
[14] 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 could computing," Proc. 4th ACM European Conf. on Computer Systems (EuroSys '09), pp.1–12, April 2009.
[15] M. Vrable, J. Ma, J. Chen, D. Moore, E. Vandekieft, A. Snoeren, G. Voelker, and S. Savage, "Scalability, fidelity, and containment in the potemkin virtual honeyfarm," Proc. 20th ACM Symposium on Operating Systems Principles (SOSP '05), pp.148–162, Oct. 2005.
[16] L. Eggert and J.D. Touch, "Idletime scheduling with preemption intervals," Proc. 20th ACM Symposium on Operating Systems Principles (SOSP '05), pp.249–262, Oct. 2005.
[17] Y. Abe, H. Yamada, and K. Kono, "Enforcing appropriate process execution for exploiting idle resources from outside operating systems," Proc. 3rd ACM European Conference on Computer Systems (EuroSys '08), pp.27–40, April 2008.
[18] C.A. Waldspurger, "Memory resource management in VMware ESX server," Proc. 5th USENIX Symposium on Operating System Design and Implementation (OSDI '02), pp.181–194, Dec. 2002.
[19] D. Gupta, S. Lee, M. Vrable, S. Savage, A.C. Snoeren, G. Varghese, G.M. Voelker, and A. Vahdat, "Difference engine: Harnessing memory redundancy in virtual machines," Proc. 8th USENIX Symposium on Operating Systems Design and Implementation (OSDI '08), pp.309–322, Dec. 2008.
[20] G. Milos, D.G. Murray, S. Hand, and M.A. Fetterman, "Satori: Enlightened page sharing," Proc. USENIX Annual Technical Conference (USENIX '09), pp.1–14, June 2009.
[21] ORACLE, "Virtualbox," 2007. http://www.virtualbox.org
[22] C.R. Lumb, J. Schindler, and G.R. Ganger, "Freeblock scheduling outside of disk firmware," Proc. 1st USENIX Symposium on File and Storage Technologies (FAST '02), pp.10–22, Jan. 2002.
[23] The Apache Software Foundation, "Apache HTTP server," 1995. http://www.apache.org/
[24] FFmpeg Project, "Ffmpeg." http://ffmpeg.org/
[25] K. Yamakita, H. Yamada, and K. Kono, "Phase-based reboot: Reusing operating system execution phases for cheap reboot-based recovery," Proc. 41st Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN '11), pp.169–180, June 2011.
[26] A. Depoutovitch and M. Stumm, "Otherworld - Giving applications a change to servive OS kernel crashes," Proc. 5th ACM European Conference on Computer Systems (EuroSys '10), pp.181–194, April 2010.
[27] M.M. Swift, M. Annamalai, B.N. Bershad, and H.M. Levy, "Recovering device drivers," Proc. 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04), pp.1–16, Dec. 2004.
[28] M.M. Swift, D. Martin-Guillerez, M. Annamalai, B.N. Bershad, and H.M. Levy, "Live update for device drivers," Tech. Rep. CS-TR-2008-1634, University of Winsconsin Computer Sciences, March 2008.
[29] N. Bila, E. de Lara, K. Joshi, H.A. Lagar-Cavilla, M. Hiltunen, and M. Satyanarayanan, "Jettison: Efficient idle desktop consolidation with partial migration," Proc. 7th ACM European Conference on Computer Systems (EuroSys '12), pp.211–224, April 2012.
[30] B.G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic execution between mobile device and cloud," Proc. 6th ACM European Conference on Computer Systems (EuroSys '11), pp.301–314, April 2011.

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.

Kenji Kono received the BSc degree in 1993, MSc degree in 1995, and PhD 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.