Traveling Forward in Time to Newer Operating Systems using ShadowReboot

Hiroshi Yamada and Kenji Kono
Keio University, JST CREST
3-14-1, Hiyoshi, Kohoku-ku, Yokohama, Japan
yamada@sslab.ics.keio.ac.jp, kono@ics.keio.ac.jp

ABSTRACT

This paper presents *ShadowReboot*, a virtual machine monitor (VMM)-based approach that shortens the downtime for software updates during an OS reboot. ShadowReboot reboots the guest OS in the background by spawning a VM dedicated to an OS reboot and enables the user to switch over to 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 where the updated kernel and applications are ready for use. ShadowReboot offers the following advantages. It can be applied to any patch to the kernels and even system configuration updates. Second, it does not need any special patch requiring intimate knowledge about the target kernels. Third, it does not require any target kernel modification. We implemented a prototype in VirtualBox 3.0.8 OSE. Our preliminary experimental results show that ShadowReboot shortened the downtime of commodity OS reboots on Windows XP and five Linux distributions (Gentoo, Fedora, Cent, Ubuntu, and SUSE) by 43 to 96%.

1. INTRODUCTION

Operating system (OS) reboots are an essential part of updating contemporary kernels and applications on laptops and desktop PCs. The downtime during OS reboots severely disrupts the users' computational activities. While the OS is rebooting, the user cannot use his or her PC. This disruptive downtime is getting longer and more costly since there are more and more software updates. This long disruption caused by these OS reboots discourages users from conducting them, failing to enforce them to conduct software updates. Although announced updates should be applied as soon as possible because they tend to include fixing critical vulnerabilities, the resultant downtime may force users to delay updating their software. As a result, users cannot enjoy the new functionality or a better performance, and even worse, the unfixed vulnerabilities can be exploited by attackers. Research literature [1] notes "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 a powerful way to apply 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, existing dynamic updatable kernels are often designed for fixing bugs in the kernel code region, such as condition misses [1]. 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 need intimate knowledge about the target kernels [3, 7]. 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 kernels [4, 2, 8]. This is not easy because recent kernels are more complex, and some of them are closed-source and/or proprietary.

This paper presents *ShadowReboot*, which shortens the downtime of OS reboots during software updates. ShadowReboot is designed to avoid the weak points of dynamic updatable kernels. First, ShadowReboot can be used to apply any patch to the kernels and can even be used for system configuration updates. Second, ShadowReboot does not need intimate knowledge about the target kernels at the source code level; we do not have to develop kernel modules or special patches. Finally, ShadowReboot requires no modification of the target kernels.

In ShadowReboot, we exploit the file access patterns of commodity OSes during their reboots in software updates. There are two key observations behind ShadowReboot. One is that commodity OSes during reboots tend to access files in administrative directories in which the system configuration files and shared components files are stored; these files are basically unmodified by the non-administrative tasks, such as web browsing and e-mailing. The other is that almost all the files in the user directories are not accessed during the OS reboots. These observations bring us to the following point; even if we heavily modify the files in the user directories while the OS is simultaneously rebooting, the modification does not interfere with the reboot activity and vice versa. This motivates us to parallelly run the users' non-administrative tasks and an OS reboot.

To execute the users' applications and an OS reboot in parallel, ShadowReboot spawns a VM dedicated to an OS reboot, which is called a *reboot-dedicated VM*. Since the OS is rebooted on the reboot-dedicated VM, the user can continue to execute applications on the original VM. ShadowReboot restores a snapshot of the reboot-dedicated VM where the reboot is completed, keeping the disk consistency between the original and reboot-dedicated VM. Although the user still has to deal with the restoration downtime, it is shorter than that of directly rebooting an OS on the original VM. Through these operations, ShadowReboot provides an illusion to users that a guest OS travels *forward* in time to the rebooted state where the updated kernel and applications are ready for use.

To create a rebooted state that is consistent with users' operations, we introduce the notion of *reboot-terms*, where users can modify their working directories specified in advance while not modifying their administrative directories. We need to pay close attention to restoring the created rebooted state. Since we restore the snapshot of the reboot-dedicated VM, the restored VM naturally provides users with only the disk states of the reboot-dedicated VM. This means that the users' activities saved in the original VM are discarded. To solve this problem, ShadowReboot performs the following operation. It starts a reboot-term on the original VM when the reboot-dedicated VM is spawned. Next, when the rebooted state is restored, ShadowReboot maintains the disk states of user directories on the original VM by using an unrollback virtual disk whose state is not affected by the restore operation. By doing so, we can retain the saved users activities and make the states of the running processes, such as the daemons, consistent with the administrative files.

We implemented a prototype in VirtualBox 3.0.8 OSE. Our preliminary experimental results show that ShadowReboot shortened the downtime of commodity OS reboots on Windows XP and five Linux distributions (Gentoo, Fedora, Cent, Ubuntu, and SUSE) by 43 to 96%.

2. 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 Windows XP professional edition (`winxp`) and 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. The updates conducted on `winxp` include all the Windows updates for the service pack 3 that need reboots, which were announced before October 2010, and an Internet Explorer upgrade to version 8. For the five Linux distributions, we applied a kernel patch to each kernel.

The results on `winxp` show that it basically accesses the same files and directories during the reboots. It frequently accesses \WINDOWS\system32\ and \WINDOWS\ Fonts\ for restarting services. In the shutdown phases, `winxp` accesses \Program Files\ to stop applications. Since the Windows Updates request an OS reboot during logging on, the shutdown phases involve accessing the user setting files such as \Document and Settings\username\NTLOGIN.DAT, and \Document and Settings\username\NTLOGIN.LOG. `Winxp` also stores the volume states in \System Volume Information\ for recovery. In the boot phases, winlogon.exe accesses \Documents and Settings, \Documents and Settings\NetworkService and \Documents and Settings\LocalServices for a logon. `Winxp` also sometimes accesses \WINDOWS\SoftwareDistribution\ and \WINDOWS\LastGood.Tmp for unknown reasons.

The results from the five Linux show that during each reboot all of them access the administrative files, but never access 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 found in /lib/. In addtion, the files in /etc are often accessed because the configuration files are conventionally stored in /etc. Each Linux distribution conducts slightly different file accesses due to the difference in configurations of the daemon processes. 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 the commodity OSes during their reboots access specific files and directories. They tend to access files in administrative directories that cannot be modified without administrative privileges. On the other hand, almost all the files in user directories, such as \Documents and Settings\username\My Documents and /home/users/ Desktop, are not accessed. The characteristics of the file access patterns bring us to the following point; even if we heavily modify the files in the user directories while the OS is simultaneously rebooting, the modification does not interfere with the reboot activity and vice versa. For example, even if we run non-administrative tasks and an OS reboot in parallel with a shared disk, the tasks' activities do not interfere with the OS reboot activity. This motivates us to execute the users' tasks and an OS reboot in parallel.

3. SHADOWREBOOT

ShadowReboot makes use of a system virtualization to parallelly execute the users' applications and an OS reboot. ShadowReboot conducts an OS reboot in the background by spawning a VM dedicated to an OS reboot, which is appropriately called a *reboot-dedicated VM*. Since the OS is rebooted on the reboot-dedicated VM, the user can continue to execute applications on the original VM. After the OS reboot is complete, ShadowReboot takes a snapshot of the reboot-dedicated VM. It enables the user to restore the snapshot states at their convenience. Although the user experiences some downtime during the restoration, it is shorter than that of directly rebooting an OS on the original VM.

We also have to focus on restoring a snapshot from the reboot-dedicated VM. ShadowReboot runs a pair of VMs: the original and reboot-dedicated one. Since each VM has its own file system and individual disk states, the file updates of the two VMs are reflected on each virtual disk. During shadow rebooting, the users' applications may issue file writes to store their data on the virtual disk of the original VM, while the files may be modified in the reboot-dedicated VM. Since the state of the reboot-dedicated VM is restored, the files updates issued on the original VM are discarded. As a result, the user's tasks may roll back to the point when

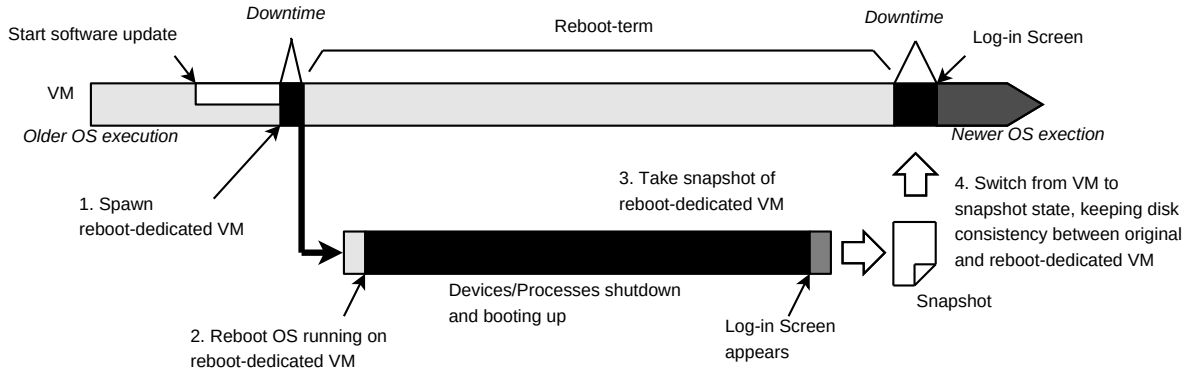

Figure 1: Overview of ShadowReboot.

the reboot-dedicated VM was spawned. If the VM continues to use the disks from the original VM after the restoration, the disk contents updated on the reboot-dedicated VM are discarded.

To successfully build a rebooted state that is consistent with the users' operations, we introduce the notion of *reboot-terms* during which the users can modify their working directories specified in advance while not modifying the administrative directories. ShadowReboot performs the following operation, exploiting the reboot-term. ShadowReboot starts a reboot-term on the original VM when the reboot-dedicated VM is spawned, and finishes it when the snapshot restoration is complete. Although the users activities on the original VM is limited since they cannot modify the administrative directories, they can do non-administrative tasks such as web browsing and e-mailing. Next, when we restore the rebooted state, ShadowReboot keeps the directories specified as working directories on the original VM and restores the other directories on the reboot-dedicated VM. By doing so, ShadowReboot allows users to access the files that the running processes, such as daemons, are based on. This means that the constructed processes' states are consistent with the files on the disks.

An overview of ShadowReboot is shown in Figure 1. An OS is rebooted on the reboot-dedicated VM after the software updates are applied. When the reboot-dedicated VM is spawned, ShadowReboot starts a reboot-term. After that, we restore the directories specified as working directories on the original VM and the other directories on the reboot-dedicated VM. Through these operations, ShadowReboot provides the users the illusion that a guest OS travels *forward* in time to the rebooted state where the updated kernel and applications are ready for use.

4. DESIGN

Several questions are posed while designing ShadowReboot, such as (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 do we check whether or not ShadowReboot successfully create the rebooted state. We answer them in this section.

4.1 VM Fork

We need an efficient way to create a reboot-dedicated VM. A naive approach to creating a reboot-dedicated VM is to run a new VM instance with the same configuration as the original VM. However, at every announcement of a software update, we have to create a new VM instance, copy the image of the VM, boot an OS, perform the software update, and conduct an OS reboot. This is tedious, and thus, may fail to encourage users to update their software.

To efficiently create a reboot-dedicated VM, we introduce a *VM fork* that forks a running VM, borrowing an idea from the existing literature [5, 9]. 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. In addition, it 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 reclaim the memory pages for a child VM execution, we make use of a page sharing mechanism running inside the VMM to produce behavior like memory ballooning. This page sharing mechanism allows one physical page to be shared with several virtual pages whose contents are the same. If there are not enough memory pages to run a reboot-dedicated VM, we run a process on the original VM that fills its memory region with the same data. The page sharing mechanism reclaims the memory pages of the process, which means the number of free memory pages increased. By doing so, we can reclaim the memory pages for a child VM without needing kernel modules such as a balloon driver.

4.2 Unrollback Virtual Disk

To keep the disk updates on the original VM in restoring the rebooted state of reboot-dedicated VM, we make use of an unrollback virtual disk that is independent of a snapshot restoration function. Unlike normal virtual disks, unrollback virtual disks do not roll back even if the VM is restored to a snapshot. By leveraging the unrollback virtual disks, we can keep the files and directories on the original VM after the restoration. While the guest OS is rebooting on the reboot-dedicated VM, we save the computational activities into the file system on the unrollback virtual disks. After the original VM has been restored to a snapshot of the reboot-dedicated VM, we can access the saved contents by mounting the target partitions in the unrollback virtual disks connected to the original VM since the state of the unrollback virtual disk is not affected by the restore operation.

A typical system 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. We

shadow-reboot the VM after updating the software. Daemon processes become available for use based on their configuration files put on the administrative directories such as /etc on the reboot-dedicated VM, while we execute our computational tasks on the original VM, such as web browsing, e-mailing, and word processing. When we restore the rebooted state of the reboot-dedicated VM, the /home/users/work directory is not restored because its mount point is assigned to the unrollback virtual disk. As a result, the built VM provides the /home/users/work directory of the original VM and the other directories of the reboot-dedicated VM. This indicates that we can successfully preserve the disk updates in the user directories on the original VM and keep the daemons states that are consistent with the files in the administrative directories.

In a way that is similar to that in unrollback virtual disks, some approaches can protect the files and directories from snapshot restoring. We can protect them by using an additional VM on which an NFS server is running. The files and directories put on the NFS server are not affected by the snapshot restoration. However, in this approach, we have to set up a VM and experience 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 the snapshot restoration, the users sometimes want isolation between the VMs and the host to protect the host against VMs compromised by viruses or attackers.

4.3 File Access Monitor

It is helpful to prepare a mechanism that checks whether ShadowReboot successfully creates a rebooted state. Since administrative directories are restored from the reboot-dedicated VM, ShadowReboot naturally cancels any updates to the directories on the original VM during the OS reboots on the reboot-dedicated VM. On the other hand, ShadowReboot also discards the updates to working directories on the reboot-dedicated VM because they are restored from the original VM. If we do not detect any updates that violate the constraints of the reboot-terms, we fail to systematically create a consistent rebooted state; on the restored VM, the running processes' states are inconsistent with the files in the disk and/or the updates to the working directories are discarded.

To check whether ShadowReboot successfully creates a rebooted state, we prepare two processes. One monitors the access to the working directories on the reboot-dedicated VM. The other monitors the access to the administrative directories on the original VM. We can implement such processes by using a file monitoring mechanism such as a filter driver or i-notify. When the update is detected on either VM, the process tells it the user and recommends to conduct a normal OS reboot. We are now implementing this feature on Linux and Windows.

5. PRELIMINARY EXPERIMENTS

We are implementing a prototype in VirtualBox 3.0.8 OSE. We performed a preliminary experiment to examine the basic performance of ShadowReboot. The experiments described in this section are conducted on a DELL OptiPlex 780DT with a 3 GHz Core 2 Duo processor with 4 G of memory and a 160 GB SATA disk. Our prototype is running on this machine, where Linux 2.6.34 is also running.

To confirm ShadowReboot successfully manages the downtime of OS reboot, we compared the downtime of ShadowReboot and normal OS reboots. Our prototype causes downtime at two points. One point is when a VM fork is invoked and the other is when a snapshot of a rebooted state is restored. We measured the downtime caused by VM forks and snapshot restorations. We regard the sum of the two downtime as the ShadowReboot downtime. We used six commodity OSes (`fedora`, `ubuntu`, `gentoo`, `cent`, `suse`, and `winxp`) as our guest OSes, which were described in Section 2. Each VM is assigned one VCPU and is connected to a 20 GB normal virtual disk and a 10 GB unrollback virtual disk as a primary master and slave respectively. We varied the VM memory size to 256, 512, 1024, 2048 and 2560 MB. The maximum memory size the VirtualBox can assign on our environment is 2560 MB. The measurement was performed using the five Linux distributions and `winxp`.

Table 1 lists the downtime of ShadowReboot (SR) and normal OS reboots (NR). The results show that the downtime of ShadowReboot is shorter than that of normal OS reboots. For example, the downtime of ShadowReboot at 256 MB is 96.6% shorter than that of the normal OS reboot in `cent`. Even in `winxp`, the downtime of ShadowReboot is 71.9% shorter than that of the normal OS reboot. When we used 2560 MB of memory, the downtime of ShadowReboot is 1.97 seconds in `gentoo`, while that of the normal OS reboot is 58.21 seconds. Although the ShadowReboot downtime is about 10 seconds in `winxp`, it is 43.4% shorter than that of the normal OS reboot.

Table 1 also lists the downtime of VM forks and the restoring snapshots of the reboot-dedicated VMs. The downtime of VM forks is different in these cases. Since our prototype simply uses the snapshot functionality to fork a VM, the downtime is equal to the downtime of taking snapshots. In VirtualBox, longer downtime when taking snapshots is incurred since it saves all the physical pages allocated by the VMM. For example, the VM fork in `gentoo` stops the VM for 0.55 seconds even when the VM is assigned 2560 MB. This is because `gentoo` does not aggressively utilize the memory, just after a log-in. On the other hand, the VM fork downtime in `winxp` is 8.10 seconds at 2560 MB. `Winxp` accesses all the pages due to its mysterious behavior, which forces the VirtualBox to assign the VM memory pages.

The downtime of restoring a rebooted state also tends 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 uses. In `cent` and `suse`, their daemons use the memory in their boot phase, taking the memory size of the machine into consideration. For example, `readahead_early` warms the file cache by accessing the files that are frequently used. Although Windows is equipped with such a feature, we were able to take a snapshot before it runs.

6. RELATED WORK

Using dynamic updatable kernels is an effective way to apply patches to the kernels at runtime so that we do not need to conduct an OS reboot [1, 7, 3, 4, 2, 8]. Ksplice [1] 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 function's text. Ksplice is designed to manipulate the text region, not to handle the memory objects in the

Table 1: Downtime of ShadowReboot and normal OS reboot.

Memory size	fedora (second)				ubuntu (second)				gentoo (second)			
	SR			NR	SR			NR	SR			NR
	VM fork	Restore	Total		VM fork	Restore	Total		VM fork	Restore	Total	
256 MB	2.19	2.49	4.68	42.23	2.16	2.43	4.59	27.47	0.42	1.38	1.90	55.39
512 MB	2.52	2.56	5.08	42.80	2.38	2.41	4.79	27.63	0.47	1.43	1.90	54.89
1024 MB	2.61	2.38	4.99	44.55	2.36	2.46	4.82	39.61	0.48	1.38	1.86	57.82
2048 MB	2.73	2.37	5.10	45.03	2.71	2.53	5.12	43.64	0.53	1.43	1.96	58.17
2560 MB	2.74	2.39	5.03	45.04	5.27	2.78	2.49	45.49	0.55	1.42	1.97	58.21

cent (second)				suse (second)				winxp (second)			
SR			NR	SR			NR	SR			NR
VM fork	Restore	Total		VM fork	Restore	Total		VM fork	Restore	Total	
2.32	2.42	4.74	141.11	2.26	3.16	5.42	30.95	1.60	2.30	3.90	13.88
3.72	3.32	7.04	154.09	3.71	4.86	8.57	30.71	2.39	2.38	4.77	13.73
3.75	3.20	6.95	132.84	4.11	5.12	9.23	43.29	6.28	3.83	2.45	16.67
3.79	3.15	6.94	142.83	4.00	5.29	9.29	56.24	6.67	2.32	8.99	18.42
3.86	3.35	7.21	132.05	4.46	5.22	9.68	55.99	10.41	8.10	2.31	18.38

kernel heap region. Additionally, this approach cannot update the *non-quiescent* kernel functions that are always on the call stack of some kernel threads. These approaches also do not manage the system configuration changes and shared component updates because the running processes are not restarted. We can complementarily use ShadowReboot to handle such updates with shorter downtime.

Some approaches require the development of special patches from the original ones. LUCOS [3] forces users to implement new functions that can handle the kernel memory objects to keep them consistent before and after the translation. In DynAMOS [7], users have to investigate how the target functions are used by the kernel threads and implement a routine that consistently updates them. ShadowReboot does not need to perform such tedious tasks.

There are approaches that involve paying the high engineering cost of redesigning and modifying a large part of the kernels. To use K42's techniques [4, 2, 8] on commodity OS kernels, we have to redesign the target kernels in an object-oriented manner. Modifying commodity OS kernels is often difficult because recent kernels are complex and some of them are closed-source and/or proprietary. ShadowReboot does not require any modification of the OS kernels.

MicroVisor [6] conducts a process migration between two VMs connected to a shared network storage, such as an NFS server and a SAN. An administrator runs the applications in one VM and maintains the kernel in the other. When the maintenance has finished, 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 the kernel maintenance, the process migration is unsuitable for system configuration changes and shared components updates. Since migrated processes are running with the configuration of the older OS, their states remain older on the newer OS. An administrator has to carefully choose the processes that can be migrated to avoid a configuration mismatch of the processes between the older and newer OSes, based on which configuration or component is updated. ShadowReboot systematically provides users a consistent system state by introducing the reboot-terms.

7. CONCLUSION AND FUTURE WORK

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

We need to implement a file access monitor to guarantee that ShadowReboot successfully creates a rebooted state. After that, we conduct experiments to confirm that ShadowReboot can successfully update the commodity OSes with shorter downtime. We also explore ways to schedule a reboot-dedicated VM to prevent it from severely interfering with the users computational tasks on the original VM.

8. REFERENCES

[1] J. Arnold and M. F. Kaashoek. Ksplice: Automatic rebootless kernel updates. In *Proc. of the 4th ACM European Conf. on Computer Systems*, Apr. 2009.

[2] 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. In *Proc. of the USENIX Annual Technical Conf.*, Jun. 2007.

[3] H. Chen, R. Chen, F. Zhang, B. Zang, and P.-C. Yew. Live Updating Operating Systems Using Virtualization. In *Proc. of the 2nd ACM Int'l Conf. on Virtual Execution Environments*, Jun. 2006.

[4] 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. In *Proc. of the 1st ACM European Conf. on Computer Systems*, Apr. 2006.

[5] 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. In *Proc. of the 4th ACM European Conf. on Computer Systems*, Apr. 2009.

[6] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. In *Proc. of the 11th ACM Int'l Conf. on Architectural Support for Programming Languages and Operating Systems*, Oct. 2004.

[7] K. Makris and K. D. Ryu. Dynamic and Adaptive Updates of Non-Quiescent Subsystems in Commodity Operating System Kernels. In *Proc. of the 2nd ACM European Conf. on Computer Systems*, Mar. 2007.

[8] 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. In *Proc. of the USENIX Annual Technical Conf.*, Jun. 2003.

[9] 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. In *Proc. of the 20th ACM Symp. on Operating Systems Principles*, Oct. 2005.