情報処理学会研究報告
IPSJ SIG Technical Report

Vol.2011-OS-118 No.4
2011/7/27

起動フェーズの再現性に着目した OS 再起動高速化手法

山木田　和哉[†1]　山田浩史[†1,†2]　河野健二[†1,†2]

可用性の高いサービスを実現するために，オペレーティングシステム（OS）には高い信頼性が求められる．しかし，現在の OS 内部には多くのバグが含まれており，カーネルクラッシュの原因となっている．古くからカーネルクラッシュに対し広く用いられているリカバリ手法として，OS の再起動が挙げられる．OS 再起動はシンプルかつ強力なリカバリ手法であり，カーネルクラッシュの原因を特定せずとも，システムを復旧することができる．しかし，OS の再起動には時間を要するという問題点も存在する．これは，OS 再起動がハードウェアやカーネルの初期化など，多くの煩雑な手順を踏まなければならないためである．そこで本研究では，システム起動フェーズの再現性に着目した OS の再起動高速化手法 *Phase-based Reboot* を提案する．Phase-based Reboot では，OS 起動時の動作を実行フェーズ毎に分割して，システムの状態を保存する．そして，OS 再起動時に，過去の OS 起動時と同じ動作をする場合は，過去の実行フェーズを再利用することで，迅速に再起動と同等の効果を得る．また，Phase-based Reboot を Xen 3.4.1 上で稼働する Linux 2.6.18 内に実装し，評価実験を行った．実験では，既存の OS 再起動に要する時間を約 34 % から 94 % 削減できることを確認した．

Exploiting Boot Sequence Similarity for Quick Reboot-based Recovery

Kazuya Yamakita,[†1] Hiroshi Yamada[†1,†2] and Kenji Kono [†1,†2]

Although operating systems (OSes) are crucial to achieving high availability of computer systems, modern OSes are far from bug-free. Rebooting the OS is simple, powerful, and sometimes the only remedy for kernel failures. Once we accept reboot-based recovery as a fact of life, we should try to ensure that the downtime caused by reboots is as short as possible. Unfortunately, OS reboots involve significant downtime, which is unacceptable in commercial services. This paper presents "phase-based" reboots that shorten the downtime caused by reboot-based recovery. The key idea is to divide a boot sequence into *phases*. The phase-based reboot reuses a system state in the previous boot if the next boot reproduces the same state. A prototype of the phase-based reboot was implemented on Xen 3.4.1 running para-virtualized Linux 2.6.18. Experiments with the prototype show that it successfully recovered from kernel transient failures inserted by a fault injector, and its downtime was 34.3 to 93.6% shorter than that of the normal reboot-based recovery.

1. Introduction

High availability is important for all ranges of computer systems from high-end enterprise systems to low-end consumer devices. High-end enterprise systems lose millions of dollars if their services are unavailable. Low-end device vendors would lose their customers if their products such as smart phones and HDD recorders were not very stable or sometimes got hung up. Upgrading iPhoneOS 3.x to iOS 4.0 on iPhone 3G causes severe performance degradation and makes iPhone 3G service nearly unavailable. Apple was criticized for delivering an inferior operating system and finally took action to investigate the series of complaints related to performance.

Operating systems are crucial for achieving high availability of computer systems. Compared with application-level failures, kernel-level failures are known to occur less frequently, but they have a considerable impact on the overall availability of software systems. Even if the applications running on the operating system are highly available, a bug inside the kernel may result in a failure of the entire software stack; no application can continue to run on the crashed kernel.

Modern operating systems are far from bug-free. Palix et al. [1] report that the rate of introduction of bugs continues to rise even in Linux 2.6. In addition, the average time between when a bug is introduced and when a fix is released is 1.8 years for Linux kernels. Our investigation of the change logs of Linux 2.6.24 and 2.6.25 also revealed that there are critical bugs inside the kernel core components. Kernel bugs are not the

†1 慶應義塾大学
Keio University
†2 科学技術振興機構 戦略的創造研究推進事業
JST CREST

sole reason for kernel failures. Soft errors in high-density semiconductors are increasing [2], and they cause incorrect values to be read from memory or incorrect instruction results to be produced.

For end users of computer systems, sometimes the only remedy for kernel failures is to reboot the operating system (and thus the entire software stack). For example, if a smart phone freezes due to a kernel failure, the end user reboots it in the expectation that the reboot will recover the smart phone; she does not have any skill or tools to diagnose and recover from the failure. Aside from low-end consumer devices, skillful administrators for high-end enterprise systems sometimes reboot the system to avoid or recover from failures. A Cisco Security Advisory [3] reported that their network products had a bug involving a memory leak, and the reboots were necessary to recover from it until a bug fix was released. IBM Director, a cluster management system for xSeries servers, periodically reboots (i.e., rejuvenates) the system to counteract software aging [4].

Once we accept reboot-based recovery as a fact of life, we need to try to reduce the downtime caused by reboots as much as possible. This paper proposes "phase-based" reboots that shorten the downtime caused by reboot-based recovery. In a phase-based reboot, a boot sequence is divided into three *phases*: 1) the hardware-initialization phase, 2) the kernel-boot phase, and 3) the daemon-startup phase. The key idea behind phase-based reboot is that a reboot repeats the same procedure as in the previous boot and sometimes reproduces the system state identical to the previous one; we can *reuse* a system state in the previous boot if the next boot reproduces the same state. In the phase-based reboot, a system state is saved after each boot-phase is finished. When a reboot is done for recovery, our system restores the saved state to skip the boot-phases that reproduce the same states as in the previous boot.

To save and restore a system state, the phase-based reboot uses the *snapshot* functionality of virtual machines (VMs). At first glance, saving and restoring a system state is straightforward; the entire memory image of the VM is saved to and restored from a disk. However, this is time-consuming, especially when the memory size assigned to a VM is large. In the worst case, the phase-based reboot takes a longer time to reboot than a normal reboot. To avoid this situation, our mechanism avoids saving unnecessary memory pages that can be reconstructed after the memory image is restored.

Restoring a system state is much more complicated. The memory image saved to a disk contains a disk cache that may be updated after the snapshot is taken. In other words, the disk cache in the saved image may be out of date. If the saved image is simply restored, the out-of-date disk cache is also restored and regarded as up-to-date. To solve this problem, our mechanism refreshes in-memory file objects with the corresponding disk blocks after it restores the saved image.

A prototype of the phase-based reboot was implemented on Xen 3.4.1 running para-virtualized Linux 2.6.18. Experiments with the prototype showed that the phase-based reboot successfully recovered from kernel transient failures inserted by the kernel fault injector, and its downtime was 34.3 to 93.6% shorter than that of the normal reboot-based recovery.

2. Phase-based Reboot

2.1 Key Idea

A boot sequence can be divided into three *phases*: 1) hardware initialization, 2) kernel boot, and 3) daemon startup. Normal reboot-based recovery executes all the boot phases in order to reconstruct a consistent system state and restart services. As illustrated in Fig. 1, the normal reboot-based recovery repeats the same boot sequence as in the previous boot. Most parts of the sequence are similar to the previous boot because the system configuration is not changed in the context of reboot-based recovery.

The key idea behind the phased-based reboot is to save and reuse consistent system states during the reboots. If the next reboot always creates the same state as in the previous one, we can simply save and restore the previous state for reboot-based recovery instead of rebooting the entire system. Unfortunately, this is an oversimplification. During service operations, an administrator may change the configuration of some daemons. In this case, we cannot reuse the system state saved in the previous boot because the different configuration may result in a different system state.

To address this reusability problem, the phase-based reboot saves system states at several points, called *restartable points*, during the boot sequence. System states saved at restartable points are called *restartable candidates*, from which the user can select

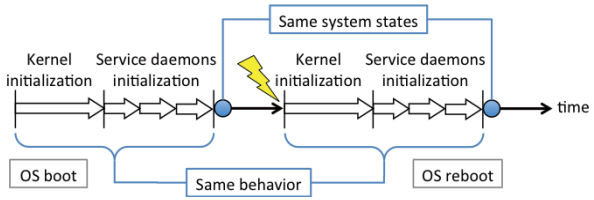


Fig.1 Key idea behind phase-based reboot. In most cases, the reboot process produces the same system state as in the previous boot.

the appropriate point to start the system reboot from. By default, the phase-based reboot saves the candidates every time each boot-phase is finished. Administrators can add more restartable points based on their intimate knowledge of the target system. If no configuration is changed, we can use the most recent candidate from which the system reboot starts. Note that this is the most common case in reboot-based recovery. If a configuration is changed as in the above example, the user can select an appropriate candidate from which to restart the system from. In the above example, the user restarts the system just after the kernel initialization is finished. To help the user select an appropriate restartable point, the phase-based reboot can determine which restartable point can be used for recovery.

2.2 Recovery Semantics

Fig. 2 shows the failure coverage for OS reboots and the phase-based reboot. The phase-based reboot handles kernel transient failures in a way similar to a normal OS reboot. Kernel transient failures include memory leaks and non-deterministic kernel panics. Rebooting an OS eliminates a corrupted memory state and returns the system state back to its initial state, which is known to be consistent, making it possible to safely restart services. The phase-based reboot inherits this advantage from the OS reboot, and it restores the system state to a clean and consistent one at a restartable point.

Unlike normal OS reboots, the phase-based reboot cannot recover from a failures caused by inconsistent hardware states. To recover from inconsistency in hardware devices, the devices must be re-initialized; the faulting machine must be reset physically. Since the phase-based reboot skips the hardware-initialization phase, this type of reboot cannot recover from hardware inconsistency. This is not a serious shortcoming of the

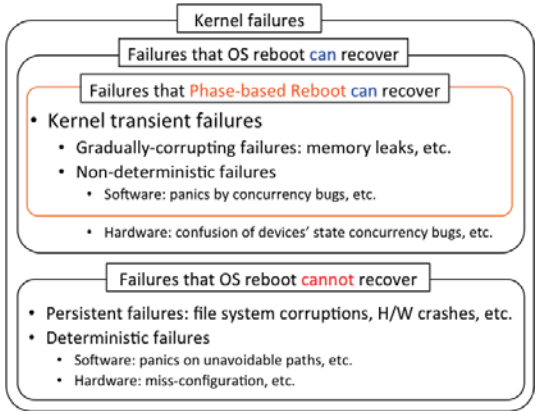


Fig.2 Recovery semantics of OS reboot and Phase-based Reboot

phase-based reboot. When a failure occurs, the user tries the phase-based reboot first. If the failure cannot be recovered, the user physically resets the entire machine. From our experience in investigating Linux change logs, most of the bugs in Linux corrupt in-memory kernel states, which can be recovered from the phase-based reboot. There are only a few bugs that make hardware devices inconsistent.

As in the normal reboot-based recovery, the phase-based reboot cannot handle all types of failures. First, the failures that persist across reboots cannot be recovered. For example, if a hardware device is corrupted physically, reboot-based recovery is useless. If the persistent data in file systems are corrupted, we have to run `fsck` to repair the corruption. Second, the reboot-based recovery cannot handle deterministic failures that can be reproduced by executing the same path. Finally, the reboot-based recovery sometimes fails to restart user-level applications that save their states to non-volatile devices. If a kernel failure prevents an application from saving its state, the application may be confused after the OS reboot. To correctly restart a service, the application must perform the recovery operation. For example, an application using a database server must roll back the SQL transactions that were processed when the kernel crashed.

3. Overview

The phase-based reboot leverages the snapshot function provided by system virtu-

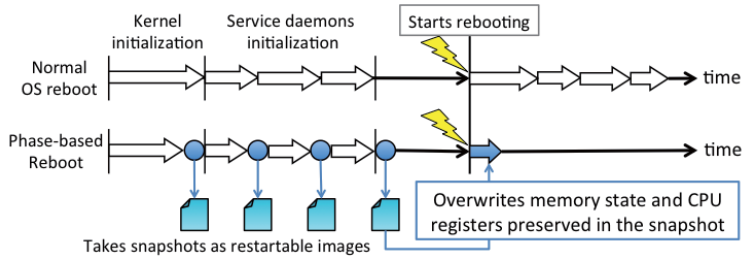


Fig.3 Comparison of Normal reboot and Phase-based Reboot

alization to restart the system at a restartable point. System virtualization is becoming commonplace in a computing environments. The snapshot function enables us to save/restore a virtual machine (VM) state including CPU registers, memory, and disks at an arbitrary point. The phase-based reboot uses a snapshot taken during an OS boot to restart the VM at a restartable point. We refer to the snapshot as a *restartable image*. The phase-based reboot overwrites CPU registers and memory states preserved in the restartable image to the running VM; the phase-based reboot never rolls back the disk state to save updates of disks in the service operation.

In the phase-based reboot, we treat snapshots as restartable candidates that can be used as a restartable image. In the phase-based reboot, we appropriately select a snapshot from the restartable candidates and restores it. To prepare restartable candidates, we take snapshots at many points during an OS boot. Fig. 3 shows a typical example of how restartable candidates are prepared. We can collect them by taking a snapshot when the kernel boot is complete, every daemon has been launched, and after a log-in prompt appears. When a phase-based reboot is conducted, we pick up a snapshot from the restartable candidates that has the same state as after the normal reboot-based recovery.

However, the phase-based reboot does not come without effort. It poses several design challenges. First, the existing snapshot restoration takes a long time if the memory size assigned to the VM is large. Next, when a snapshot is restored, the file system objects are restored as well, which leads to eliminating the disk update in the service operation. Lastly, we need a way to determine which restartable candidate is a restartable image in order to help select a proper restartable image.

4. Design

For the first challenge, we avoid saving pages that are unnecessary for the VM to work correctly after the snapshot is restored. For the second challenge, we design a kernel module that updates the file system objects. To address the last challenge, we prepare a tool that infers application states that will be built by the normal reboot. To do so, the tool checks whether files accessed during the guest OS boot are updated in the service phase.

4.1 Snapshot Optimization

The conventional VM snapshot function saves and restores all the memory pages of a VM even if the pages are not used for the kernel and user processes. As the assigned memory size is larger, the restoration of the snapshot takes longer, causing many disk I/O requests.

To shorten the time for restoring a restartable image, we *shrink* the size of VM memory checkpoints. Our technique reduces disk I/O involved in saving and restoring the memory checkpoints. The phase-based reboot avoids saving pages that are not necessary for the system to work correctly after the restore operation. Such pages include a free page and file cache pages. We believe that our technique is effective because memory usage is not heavily utilized during the boot phase where restartable candidates are taken.

In this work, we focused on a free page and a page containing soft-state kernel objects. Even if a VMM discards the contents of free pages, the guest works correctly because free pages are initialized when the kernel uses them. Soft-state kernel objects include caches for disk blocks and caches for kernel resource managers. A file cache is a typical example of soft-state kernel objects. Because a file cache contains the data on disk, the guest can reproduce it by reading the data from the disk.

To avoid saving these pages, we modify the guest kernel to explicitly inform the VMM which pages are unnecessary. When a snapshot is taken, the guest kernel examines its memory objects and sends the VMM the guest physical address of the unnecessary pages. The VMM does not store them in a snapshot, based on the given addresses. When the snapshot is restored, the VMM compensates for the lost pages by allocating

new pages to the guest. After that, the restored VM starts to run.

4.2 Update of File System Objects

We need to take into account the memory objects of file systems after restoration from a restartable image. File systems are OS core components that manage disk caches including the cache of data blocks, metadata, and file system metadata. Because file systems manage such memory objects, a restartable image naturally contains them, but the file systems fail to keep the disk updates in the service phase when a restartable image is restored. For example, the filesystems' metadata such as the super block cause this problem. The kernel only updates these metadata in the memory, and writes them back to the disks; the metadata are never read from the disks after the partition has been mounted. When a restartable image is restored, the older file system metadata are overwritten on the current metadata. This causes the file system to inconsistently manage disk blocks such as free blocks and data blocks.

Although remounting the disk volumes is a simple solution for this inconsistency problem, this solution is not suitable for phase-based reboots. Specifically, we take a snapshot after unmounting the disk volumes. When the snapshot is restored, we mount them. This way naturally refreshes file system objects, thus avoiding the inconsistency problem described above. However, we have to close all the files in the disk volumes to safely unmount the disk volumes. This constraint is critical for the phase-based reboot because some applications keep files open while running. For example, `syslog_d` keeps its log file open with `O_APPEND`, and `crond` keeps its pid file open. Therefore, we cannot put these applications into restartable candidates. To put such applications on a restartable image, we explore an alternative to solve the inconsistency of file system objects.

To solve the inconsistency, our kernel module forces the kernel file system component to read such metadata again just after a restartable image has been restored. When the restoration of a snapshot is completed, our kernel module forces the file system component to read the file system metadata and i-node from the disk and it updates them. The consistency of file cache pages is ensured with the snapshot optimization because the phase-based reboot does not save them in a restartable image.

4.3 Finding Restartable Images

We use as a restartable image a restartable candidate, where the application states are the same as after a normal OS reboot. Because the memory contents saved in a restartable image are overwritten to the target VM, the restartable image needs to contain applications' memory contents that will be built by the normal OS reboot. If an application memory image in a restartable image is different from that after the normal OS reboot, the wrong memory image will be built on the VM. This means we cannot produce the effect of the reboot-based recovery.

Suppose that a restartable candidate contains a running application that reads a configuration file in its boot phase. If the file is updated, the application should be launched with the new configuration after reboot-based recovery. However, restoring the restartable candidate builds an application image that is based on the older configuration. Another example is that a restartable candidate contains a running application that keeps a file open to log its state. Restoring this image may cause a log corruption if the application logs its state in the service phase. Because the file offset of the application is also restored, the application may overwrite the log contents that were logged before restoring. Although one way to solve this problem is to redesign applications to force them to reconstruct their states after a snapshot is restored, modifying all of the applications is unreasonable.

To find an appropriate restartable image, our checker infers the application states that will be built by the normal reboot. To do so, it checks whether files accessed until a restartable candidate is taken are updated in the service operation. If these files are not updated, we evaluate whether the selected restartable candidate can be used as a restartable image. We assume that applications launch in the same way if files accessed by them are not updated. For example, some applications start to run based on their configuration files. If the configuration files are not modified, the applications start in the same way at the next OS boot. Even if an application reads files and caches their contents in memory, it builds the cache again when the file contents have not been modified. When a log file is not updated, the application does not overwrite old log contents since the file offset has not changed.

We note that our checker does not cover all types of applications. For example, it

does not manage applications whose behavior is defined by network conditions and time. If such applications are put into a restartable image, the phase-based reboot may build the wrong application states after network conditions and times are changed. To manage the applications, we need to extend our checker to determine whether applications' states in the restartable candidates are the same as those built by the normal OS reboot.

We prepare a mechanism for files opened with the append mode such as `O_APPEND` to aggressively omit the boot phase. When files are opened with the append mode, the kernel sets files' offset of the application to the end of the files in `write()`. This means the file offset is automatically set to the end of the file by `write()` even after the snapshot is restored. If the file contents are not updated, except for the appended region, the checker determines whether the application that opens the file with the append mode can consistently run after a restartable image is restored. In this situation, it does not issue a warning that a restartable candidate is not a restartable image.

5. Implementation

We implemented the phase-based reboot in Linux 2.6.18 running on Xen 3.4.1. Our core implementation consists of three modules; *a file access monitor*, *kernel object manager*, and *file update checker*. Both the file access monitor and kernel object manager are running inside the guest kernel in a domain U. The file update checker is running inside domain 0. The file access monitor logs the name and last modification time of files accessed until a restartable candidate is taken. To do so, it records information on files accessed by the processes. The kernel object manager appropriately handles disk cache being managed by the ext3 file system. In addition, it frees slab cache and tells the addresses of free pages to the underlying hypervisor in order to remove the entries of the P2M table when a restartable candidate is taken. The file update checker inspects virtual disk images mounted by the target VM and checks file updates by referring to a log produced by the file access monitor.

Our implementation is overviewed in Fig. 4. Fig. 4 (a) depicts the execution of saving restartable candidates. For ease of implementation, we run a daemon server that triggers our guest kernel-level mechanism. Since we can take a snapshot only in domain 0, the client running in domain 0 communicates with the daemon server. To take a restartable candidate, the client asks the server to run the kernel-level mechanism. The client starts taking a snapshot when it is notified of the completion of the module tasks by the daemon server. Note that there is a race condition where a process can modify files until the client starts taking a snapshot after the completion of the module tasks. To avoid this situation, we need to implement a mechanism that enables domain U to take its snapshot.

First, the file access monitor logs information on accessed files for the file update checker. Next, the kernel object manager flushes the dirty buffer and releases the disk cache and slab cache. Then, the kernel object manager tells the underlying hypervisor to remove entries of free pages in the P2M table with the balloon driver to shrink the memory checkpoint. Finally, we save the shrunken memory image as a restartable candidate.

Figure 4 (b) shows the execution flow when the phase-based reboot is triggered. Xen restores the selected restartable candidate, and the kernel object manager reallocates free pages because the VM snapshot has been shrunk. Finally, the kernel object manager updates i-nodes, dentries, and super block data in the memory. If necessary, the file update checker checks whether the VM has updated the files in the file access logs and determines which restartable candidate is restartable.

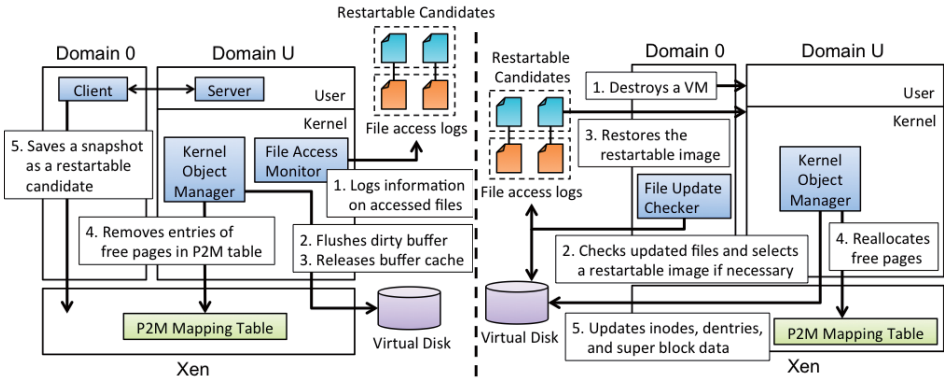
5.1 File Access Monitor

The file access monitor logs information on accessed files for the file update checker. Specifically, it memorizes the absolute path of the accessed file, its i-node number, and its last modification time. The file access monitor also memorizes whether a file has been opened with `O_APPEND`. It saves the memorized information as a file on the guest file system when the kernel object manager triggers it. The log is used by the file update checker, as will be described later.

The file access monitor monitors `sys_open`, `sys_stat`, and `sys_exec` to find out which files have been accessed. The monitoring is stopped when our system call, `pbr_ready()`, is issued in order to avoid overhead of file access monitor activities in the service phase.

5.2 Kernel Object Manager

The kernel object manager manipulates kernel objects being managed by ext3 and



(a) Execution of saving a restartable candidate (b) Execution of phase-based reboot

Fig.4 An Overview of Phase-based Reboot

the slab allocator. The manipulation is carried out when restartable candidates are taken and when they are restored. When a restartable candidate is taken, the kernel object manager flushes the dirty buffer and releases the page cache corresponding to the buffer cache, i-nodes, and dentries. This is done to shrink the memory checkpoint.

The kernel object manager also unregisters free pages from the P2M table for the Xen hypervisor to shrink the memory checkpoint of the domain. To remove the entries of free pages in the P2M table, the kernel object manager leverages a *balloon driver* [5]. When the balloon is inflated, it pins down free pages and tells the Xen hypervisor to remove their entries in the P2M table. When the balloon is deflated, it releases the pinned pages and registers their entries to the P2M table again.

The kernel object manager controls the balloon when a restartable candidate is taken and restored. After releasing the page cache and slab cache, it inflates the balloon to remove the entries of free pages in the P2M table. When the restartable candidate is restored, the kernel object manager deflates the balloon to register the free pages in the P2M table again.

5.3 File Update Checker

The file update checker helps to determine which restartable candidate is restartable. We can run the file update checker just after a restartable candidate is taken. To detect file updates, it pairs the log produced by the file access monitor with the restartable candidate. The file update checker mounts the virtual disk used by the target domain U after the restartable candidate is taken to salvage the produced log.

To determine which restartable candidate is restartable, the file update checker detects file updates by referring to the log paired with the restartable candidate. The file update checker obtains the current state of the logged files by mounting the used virtual disks. It compares the obtained i-node number and the last modification time with the corresponding values in the log. If the current values are different from the logged ones, the file update checker judges the files to be updated, and the restartable candidate cannot be used as a restartable image.

To successfully deal with the files opened with `O_APPEND`, the file update checker calculates and logs the hash value of their contents just after a restartable candidate is taken. The file update checker gets the file contents by mounting the virtual disks. When we check whether a restartable candidate is restartable, the file update checker calculates the file contents except for the appended region again, and compares it with the logged value. If the values are the same, the file update checker does not issue a warning that the restartable candidate is not restartable.

6. Experiments

We conducted experiments to evaluate the effectiveness of the phase-based reboot. We used a machine equipped with a 3 GHz quad-core Xeon processor, 16 GB of memory, and a 73-GB SAS NHS 10,000 rpm hard disk. On this machine, we ran Xen 3.4.1 and the Linux 2.6.18 kernel in domain 0. We also ran the modified Linux 2.6.18 para-virtualized for Xen on guest domains connected to a 10-GB virtual disk. We installed Fedora Core 8 on each domain, and turned off unnecessary service daemons.

6.1 Downtime

We measured the downtime of the phase-based reboot to determine how the phase-based reboot shortens the downtime of reboot-based recovery. To execute the phase-based reboot, we prepared two restartable images. The first was a restartable image that was taken before the guest kernel mounted the virtual disk. We refer to the phase-based reboot that is restoring this restartable image as *pr-naive*. The other was a

restartable image that was taken when a log-in prompt appeared after the kernel and all the daemons were ready. We simply refer to the phase-based reboot that is restoring this restartable image as *pr-opt*. For comparison, we also measured the downtime of a normal boot and a normal reboot on the guest domain (*guest boot* and *guest reboot*). To clarify how effective our optimization described in Sec. 4.1 was, we executed the phase-based reboot without our snapshot optimization and measured its downtime (*pr without snapshot opt*). We started measuring when each operation was triggered, and stopped when all the daemons registered in run level 3 were ready on the domain. We measured the downtime of each reboot-based recovery, varying the memory size of the guest domain.

Table 1 lists the average downtime of each reboot-based recovery. Table 1 indicates that the downtime of the phase-based reboot was shorter than that of the guest reboot in many cases. In pr-opt, the downtime was 86.1% to 93.6% shorter than the guest reboot. The downtime of pr-naive was 60.1% to 77.6% shorter than that of the guest reboot. In pr without snapshot opt, its downtime is shorter than the guest boot when the domain memory size was smaller than 2 GB.

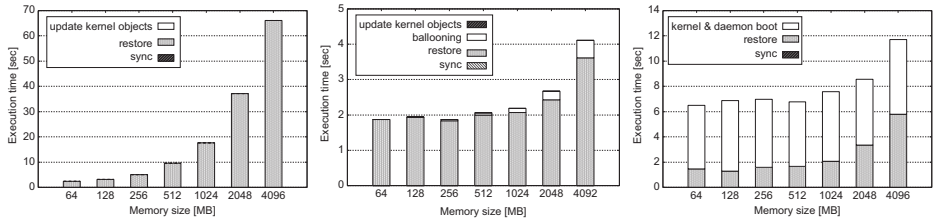
To analyze the downtime caused by the phase-based reboot, we show the breakdown of the downtime of pr without snapshot opt, pr-naive and pr-opt in Fig. 5. Fig. 5 (a) and (b) reveal that our snapshot optimization significantly contributed to shortening the downtime of reboot-based recovery. In pr without snapshot opt, the restore time was much longer than the other configurations, pr-naive and pr-opt. This is because Xen's snapshot function saves and restores all the memory pages assigned to the guest domain even if the pages are not used by the kernel and user processes.

In addition, our snapshot optimization successfully shrank the restartable images. For example, when a VM was assigned 1024 MB of memory, the optimized snapshot function saved only 99 MB as a restartable image, but Xen's snapshot function saved 1050 MB. Since we can prepare RAM disks or solid-state drives where these memory checkpoints are placed, we can shorten the downtime of the phase-based reboot.

The figure also shows that omitting the kernel and daemon boot phase is effective to shorten the downtime of reboot-based recovery (Fig. 5 (b) and (c)). In pr-naive, booting the kernel and daemons is the main part of its downtime since the impact of

Table 1 Average downtime of guest reboot, pr without snapshot opt, phase-based reboot, pr-naive

Memory size [MB]	Guest reboot [sec]	pr w/o opt. [sec]	pr-opt [sec]	pr-naive [sec]
64	29.01	2.42	1.87	6.51
128	28.58	3.18	1.93	6.87
256	28.27	5.01	1.84	6.98
512	28.42	9.57	2.02	6.78
1024	28.83	17.65	2.15	7.57
2048	28.92	37.17	2.63	8.56
4096	29.38	66.15	4.08	11.72



(a) pr without snapshot opt (b) pr-opt (c) pr-naive

Fig.5 Breakdown of pr without snapshot opt, pr-opt, and pr-naive downtime.

the restore operation and ballooning is relatively smaller. Phase-based reboot in pr-opt effectively shortens its downtime by omitting the launch phase.

6.2 Finding Restartable Images

To determine which restartable candidate is a restartable image under a complicated workload, we check which restartable candidate can be a restartable image after running a benchmark that models a real web site. We used RUBiS [6] on the Java EE platform, which is a three-tailored auction site prototype modeled after eBay.com [7]. We prepared additional physical machines for this experiment. The specifications of these machines were the same as the machine described previously. These machines were connected via Gigabit Ethernet. We ran the RUBiS client emulator on one machine while Xen 3.4.1 was running on another machine. The Xen machine was used as a server where three guest domains were running, a web server domain (*FrontVM*), application server domain (*AppVM*), and database server domain (*DBVM*). Apache 2.2.9, Tomcat 5.5.28, and MySQL 5.0.45 were running on FrontVM, AppVM, and DBVM, respectively. We emulated 500 clients and checked whether or not all the restartable

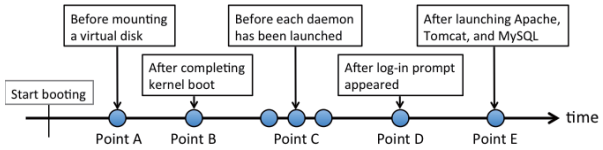


Fig.6 Restartable points in the second experiment.

candidates were restartable images. Our check was carried out two ways. One is that we conducted the phase-based reboot when the emulation had finished. The other is that we conducted the phase-based reboot while the client emulator was running. We assigned 1.7 GB of memory to each guest domain. This memory size comes from the small VM configuration in Amazon Elastic Compute Cloud [8].

We prepared restartable candidates in the following way, which is shown in Figure 6. We took a snapshot before mounting the virtual disk (Point A), when the kernel boot was completed (Point B), when all the configured daemons were launched (Point C), when a log-in prompt was displayed (Point D), and after Apache, Tomcat, and MySQL were launched on each VM (Point E).

The results are exhibited in Tables 2 and 3. Table 2 indicates which restartable candidate is restartable or not when the emulation has finished. In this situation, the phase-based reboot does not issue warnings in FrontVM and AppVM. This means that we can use Point E to restart the VMs. On the other hand, the phase-based reboot judges that Point E is not restartable since some files have been updated. Specifically, `/var/log/mysqld.log` and `/var/lib/mysql/ib_logfile` were opened without `O_APPEND` and were updated during the RUBiS operation. `/var/log/mysqld.log` is used for MySQL to log its execution state. `/var/lib/mysql/ib_logfile` is a log file where MySQL records transactions states.

Table 3 indicates which restartable candidate is restartable or not when we conduct the phase-based reboot while the client emulator is running. DBVM is sent the same warnings as when we conduct the phase-based reboot after the client emulator is completed. The phase-based reboot sometimes judges that Point E is not restartable in FrontVM and AppVM since `/etc/httpd/logs/error_log` has been updated. When the workload of RUBiS is interrupted, Apache logs the error condition into `/etc/httpd/logs/error_log`. We cannot restart candidates at Point E because this file is opened by Apache without `O_APPEND`.

Table 2 Restartable points when executing phase-based reboot after completing the emulation

VM	Point A	Point B	Point C	Point D	Point E
FrontVM	OK	OK	OK	OK	OK
AppVM	OK	OK	OK	OK	OK
DBVM	OK	OK	OK	OK	`/var/log/mysqld.log` `/var/lib/mysql/ib_logfile`

Table 3 Restartable points when executing phase-based reboot during the emulation

VM	Point A	Point B	Point C	Point D	Point E
FrontVM	OK	OK	OK	OK	Depends (`/etc/httpd/logs/error_log`)
AppVM	OK	OK	OK	OK	Depends (`/etc/httpd/logs/error_log`)
DBVM	OK	OK	OK	OK	`/var/log/mysqld.log` `/var/lib/mysql/ib_logfile`

We found that some logs were updated frequently. For example, `auditd` records system call names issued by specified processes, opening the log file without `O_APPEND`. If this daemon is put into restartable candidates, we cannot use them for a restartable image. To consistently run such a daemon after a phase-based reboot, we carefully avoid putting it in restartable candidates. We need to configure the daemon to start after the phase-based reboot. If a user wants to skip the boot phase of such a daemon, he or she has to redesign the daemon to be phase-based reboot-aware. In this case, we add `O_APPEND` to the open argument.

7. Related Works

Various approaches have been proposed to reduce the downtime stemming from a whole program restart. Microreboot [9] achieves fine-grained software reboots. To enable a microreboot, the target application is divided into small independent software components which become units for a reboot. If rebooting a small component cannot recover from a failure, a bigger component will be rebooted. The work aims at application-level failures, and thus, it cannot shorten the reboot time for recovery from kernel failures. Also, the microreboot is complementary notion to the phase-based re-

boot. We can say that the microreboot focuses on "components" of software systems. We benefit from this when a reboot of small components recovers from failures. On the other hand, the phase-based reboot focuses on "phases" of software systems. We benefit from this in cases where we have to reboot larger components such as OS kernels, which take a long time to restart.

Kexec [10] and Fast Reboot [11] allow us to quickly start up a kernel. When they are invoked on a running kernel, another kernel boots without any hardware reset. Since these mechanisms require kernel support, they cannot be used when the kernel is stopped due to kernel failures. The phase-based reboot can work even when the kernel has crashed.

Different approaches have been proposed to recover from kernel failures. Otherworld [12] reboots the kernel without clobbering the state of the running applications. After the kernel crashes and is rebooted, Otherworld restores the application memory spaces, open files, and other resources. However, the downtime of Otherworld is reported to be about 1 minute. To restart the service quickly, we use both Otherworld and our method as the situation demands.

Akeso [13] is a kernel-level mechanism that is request-oriented in the sense that it handles the recovery at the request level such as system calls or interrupts. When a failure occurs in the kernel, Akeso rolls back the kernel state to the beginning of the function and makes the function return an error. However, it requires a complicated annotation in various places within the kernel code along with the context. Writing a correct annotation requires accurate knowledge of the kernel and is a laborious and error-prone task.

Previous studies have focused on a kernel component. Nooks [14, 15] pushes a device driver into a lightweight protection domain and transparently recovers device drivers when they fail. LeVasseur et al. proposed an approach to isolating device drivers using dedicated VMs [16] to limit the drivers' crash influence. Membrane [17] is a kernel-level mechanism to make file systems restartable. It periodically saves checkpoints of file system states. If the file system fails, Membrane restores the file system state from the recent checkpoint and consistently and transparently updates the stateful information to applications. These approaches focus on certain kernel components' failures, but the phase-based reboot focuses on failures in any kernel component.

Some previous studies have made better use of virtualization to improve the reliability of the system. Bresoud and Schneider proposed a hypervisor-based approach to implementing a fault-tolerant system [18]. It replicates the state of a system remotely and recovers from failures in a failover manner. Remus [19] is a failover mechanism that uses VMM. Remus replicates snapshots of an entire running OS instance between a pair of physical machines. These failover approaches basically focus on hardware failures, while the phase-based reboot focuses on software failures in the kernel.

8. Conclusion

We proposed a "phase-based" reboot that shortens the downtime of reboot-based recovery. The key idea is to divide a boot sequence into phases. The phase-based reboot reuses a system state in the previous boot if the next boot reproduces the same state. By doing so, it skips some phases of a time-consuming boot sequence that reproduces the same states as in the previous boot. A prototype of the phase-based reboot was implemented on Xen 3.4.1 running para-virtualized Linux 2.6.18. Experimental results showed that the prototype successfully recovered from kernel failures inserted by a kernel fault injector, and its downtime was 34.3 to 93.6 % shorter than that of the normal reboot-based recovery.

Reference

1) Palix, N., Thomas, G., Saha, S., Calvés, C., Lawall, J. and Muller, G.: Faults in Linux: Ten Years Later, *Proceedings of the ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '11)*, pp.305–318 (2011).
2) Baumann, R. C.: Soft errors in commercial semiconductor technology: overview and scaling trends (2002).
3) Advisory, C.S.: Cisco catalyst memory leak vulnerability. ID:13618 (2001).
4) Castelli, V., Harper, R. E., Heidelberger, P., Hunter, S. W., Trivedi, K. S., Vaidyanathan, K. and Zeggert, W.P.: Proactive Management of Software Aging, *IBM Journal of Research and Development*, Vol.45, No.2, pp.311–332 (2001).
5) Waldspurger, C.A.: Memory Resource Management in VMware ESX Server, *Proceedings of the 5th USENIX Symposium on Operating System Design and Imple-*

mentation (OSDI '02), pp.181–194 (2002).
6) RUBiS: . http://rubis.objectweb.org/.
7) eBay.com: . http://www.ebay.com/.
8) Amazon.com: Amazon Elastic Compute Cloud (Amazon EC2). http://aws.amazon.com/ec2/.
9) Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G. and Fox, A.: Microreboot - A Technique for Cheap Recovery, *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pp.31–44 (2004).
10) Nellitheertha, H.: Reboot Linux faster using kexec. http://www.ibm.com/developerworks/linux/library/l-kexec.html.
11) Sun Microsystems: Using Fast Reboot on the x86 Platform (2008). `http://dlc.sun.com/osol/docs/content/SYSADV1/ghsut.html/`.
12) Depoutovitch, A. and Stumm, M.: Otherworld - Giving Applications a Change to Servive OS Kernel Crashes, *Proceedings of the 5th European Conference on Computer Systems (EuroSys '10)*, pp.181–194 (2010).
13) Lenharth, A., Adve, V. and King, S.T.: Recovery Domains: An Organizing Principle for Recoverable Operating Systems, *Proceedings of the 14th ACM International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS '09)*, pp.49–60 (2009).
14) Swift, M.M., Bershad, B.N. and Levy, H.M.: Improving the Reliability of Commodity Operating Systems, *Proceedings of the 19th ACM Symposium on Operating Systems Principles (SOSP '03)*, pp.207–222 (2003).
15) Swift, M.M., Annamalai, M., Bershad, B.N. and Levy, H.M.: Recoverying Device Drivers, *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pp.1–16 (2004).
16) LeVasseur, J., Uhlig, V., Stoess, J. and Gätz, S.: Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines, *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI '04)*, pp.17–30 (2004).
17) Sundararaman, S., Subramanian, S., Rajimwale, A., Arpaci-Dusseau, A. C., Arpaci-Dusseau, R. H. and Swift, M. M.: Membrane: Operating System Support for Restartable File Systems, *Proceedings of the 8th USENIX Conference on File and Storage Technologies (FAST '10)*, pp.281–294 (2010).
18) Bressoud, T.C. and Schneider, F.B.: Hypervisor-based Fault-tolerance, *Proceedings of the 15th ACM Symposium on Operating Systems Principles (SOSP '95)*, pp.1–11 (1995).
19) Cully, B., Lefebvre, G., Meyer, D., Feeley, M., Hutchinson, N. and Warfield, A.: Remus: High Availability via Asynchronous Virtual Machine Replication, *Proceedings of the 5th USENIX Symposium on Networked Systems Design and implementation (NSDI '08)*, pp.161–174 (2008).