Proceedings of the International MultiConference of Engineers and Computer Scientists 2012 Vol I,
IMECS 2012, March 14 - 16, 2012, Hong Kong

Faster Recovery from Operating System Failure and File Cache Missing

Yudai Kato, Shoichi Saito, Koichi Mouri, and Hiroshi Matsuo

Abstract—**Rebooting a computer is one method to fix operating system (OS) failures. However this method causes two problems. First, the computer will not be available until the rebooting is finished. Second, the user data on the volatile main memory will be erased. To address these problems, we propose a system that can fix failures within one second without losing file caches by replacing the OS encountering a fatal error by another OS that is running on the same machine. Our proposed system is constructed on one machine in which two OSes are simultaneously operating. One OS is dedicated to the backup of the other OS. Using the backup OS, we can instantly conduct a failover. During the failover, the backup OS migrates the remaining file caches in the main memory to protect the changes in them before the OS failure.**

Our proposed system was implemented on a Linux kernel and can run on commodity x86 machines without any modifications to the applications. Therefore, it can be adapted for a variety of applications and platforms.

Index Terms—**Reliability, Fault-tolerance, File-cache, NFS, LPAR**

I. Introduction

The demand for reliable computer systems continues to increase. One of the factors that threatens their reliability is operating system (OS) failures caused by bugs in the OS kernel. However, completely removing bugs from the kernel is difficult, because the amount of OS codes is growing every year, which increases the possibility of introducing bugs [1]. We have to construct systems that accept the existence of bugs.

OS failure causes various problems, including loss of temporary data on the main memory. As a result, files are broken or the data of applications are lost. The general method for restoring an OS facing failure is rebooting the computer. However, the computer is not available for users for tens of seconds during the rebooting process. In this paper, we resolve these problems by making the restoration process faster and protecting file caches from OS failure.

There is an existing solution for the resilience of file systems for high availability (HA) systems. Distributed Replicated Block Device (DRBD) is a distributed storage system for clusters of HA systems. DRBD synchronizes disks over the network to protect their contents from failure. Using cluster management frameworks with DRBD, server failover can be carried out and the cluster can transparently continue service to clients. However, HA systems need such expensive apparatus as a load balancer for failover and load balancing, and these apparatuses also need to be duplicated for redundancy. Therefore, the cost of their introduction and their managing is expensive.

The following are the goals of this paper.

- Making OSes robust against bugs
- Constructing a high availability system in a single machine
- Using a machine with minimum hardware
- Quickly making failover
- Minimizing overhead

For these goals, we simultaneously execute two OSes on a single machine with active/backup configurations. When the active OS encounters a failure, the backup OS can take over the active OS by migrating the devices and file cashes from the active OS to itself. Our proposed system, which does not need special hardware, is constructed on a single machine, and thus the introduction cost is less than HA clusters. In addition, our proposed system is lightweight, because it rarely performs special processing. Instead of restoring the hardware problems, our proposal addresses the OS failures caused by bugs in the kernels and reduces the amount of time for failovers. Our proposed system reduces the downtime caused by OS failure and increases computer availability.

Various approaches have been researched to protect an OS from bugs. For example, one work focuses on the codes of device drivers, which often have bugs, and proposed a mechanism that protects an OS from device drivers [2]. Although this research only targets the device driver, our proposal has a wide scope that targets the whole kernel.

Otherworld [3] views an OS as software and proposes a technique that replaces the crashed OS when it fails, and applications can continue their execution after the replacement. OS replacement is performed by booting a new OS using warm-boot [4] and migrating the necessary kernel data from the crashed OS to the new OS. However, the technique greatly depends on the kernel data structures, which may be destroyed when the OS crashes. No guarantee exists that the technique will operate properly after the replacement. For this reason, our proposed system depends less on such data structures for robustness.

Methods also exist for the redundancy of file systems. Fsck [5] is a basic tool for restoring broken file systems. It scans a whole disk and can resolve the inconsistencies of the file system. However, Fsck has a problem: its restoration time is in proportion to the disk size. For this problem, several techniques have been proposed, including Journaling [6], SoftUpdates [7] and Log-structured File System [8]. Using these techniques, we can quickly recover the file systems after failures. Ext3 and Ext4, which are the default file systems of Linux, have three modes: journal, ordered, and write-back. The most redundant mode is journal, which can protect both meta and user data from failures. However, it significantly decreases the performance of the file system

Manuscript received December 27, 2011; revised January 16, 2012.
Kato Yudai yudai@matlab.nitech.ac.jp, Shoichi Saito shoichi@nitech.ac.jp and Hiroshi Matsuo matsuo@nitech.ac.jp are with Nagoya Institute of Technology Gokiso-cho, Showa-ku, Nagoya 466-8555 Japan.
Koichi Mouri mouri@cs.ritsumei.ac.jp is with Ritsumeikan University.

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

because it writes the same blocks twice to disks. Therefore, the default mode is ordered, which only protects the meta data. As a result, journaling cannot protect the user data from failure. To resolve this problem, we protect user data by protecting the file caches from OS failure by migrating them from the crashed OS to the backup OS. In addition, the migration needs no special processing under normal operating conditions to avoid increasing the overhead for the file system.

This paper is organized as follows. In the next section we outline our proposal. In Section III, we describe its design. In Section IV, we discuss how it can be applied to existing applications using an example of NFS servers. In Section V, we describe the implementation of our proposed system. In Section VI, we show the evaluation results of the amount of time for performing a failover. After that, we discuss related work in Section VII and conclude in Section VIII.

II. Proposal

The amount of time for the failover of crashed OSes is the total suspension time until the administrator detects the failure and restarts the OSes. The computer is not available at least while it is rebooting even if the administrator can reboot the computer immediately after the failure. To reduce the computer downtime, we automatically detect OS failures and start a quick failover instead of restarting the computer. In this section, we describe the outline of our proposal and the behavior of our proposed system.

A. Outline

High availability (HA) clusters automatically detect defunct components using health check mechanisms [9] and inspect the states of applications and networks. Our proposed system provides a method to check the health of an OS kernel. If the kernel is defunct, we simply conduct a failover. To reduce the rebooting time, we simultaneously run two OSes on the machine: active and backup. The backup OS can quickly take over functions of the active OS when it fails to operate properly. This shortens the failover time more than rebooting. Therefore we can reduce the downtime caused by OS failure. For failovers, the backup OS migrates devices from the active OS to itself. During normal times, the backup OS operates with minimal devices. Therefore, the backup OS has to migrate the devices to take over as the active OS. This migration is a particular feature of the proposed system that is not necessary for rebooting. After the migration of the devices, we migrate the file caches from the active to the backup OS. Using the migration, we can protect the file caches that remain in the volatile main memory from OS failure.

B. Behavior

During failover, we migrate devices and file caches and launch applications. Fig 1 shows a concept image of our proposed system. The left shows the system during its normal time, and the right shows the system after the active OS encountered a failure. The machine has two devices: network interface card (NIC) and hard disk drive (HDD). During normal times, the devices belong to the active OS. After failure, they are migrated from the active to the backup OS.

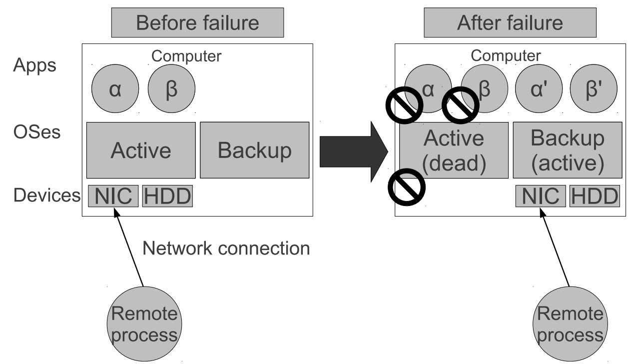

Fig. 1. Proposed system before encountering a failure and after failover

After migrations, the backup OS can communicate with the remote processes with the same IP and MAC addresses used by the active OS. Similarly, the backup OS can handle the same files as the active OS. Next, we migrate the file caches remaining in the volatile main memory from the active to the backup OS so that the backup OS can properly reflect the file operations performed at the active OS to the HDD. Finally, we restore applications simply by launching the same applications the active OS was executing before the failure.

III. Design

We designed four components that have the functions discussed in Section II.

- Multiple-OS execution platform
- Alive monitoring
- Device migration
- File cache migration

First, we describe the design of each component and then discuss the coverage of the proposed system.

A. Multiple-OS execution platform

On the same machine, we execute two OSes: active and backup. For such simultaneously running of the active and backup OSes, we use Logical PARtition (LPAR) [10], which is a virtualization technique that emulates multiple machines on a physical machine. LPAR divides hardware into several partitions that work as a virtual machine. Our multiple-OS execution platform is software implementation of LPAR [11] to reduce the dependency on hardware platforms.

Each OS is booted on a partition specified by kernel parameters that are given by a bootloader. For example, a machine with four CPU cores, 8 GB bytes of RAM, and two hard disks may be divided into two separate virtual machines with two CPU cores, 4 GB bytes of RAM, and a hard disk.

We employ software implementation for LPAR for our proposed system because (1) the virtualization overhead is very small, and (2) the partition layout can be dynamically changed. The second characteristic is necessary for migrating devices between OSes.

B. Alive monitoring

Next we describe how the backup OS confirms whether the active OS is dead or alive. There are two situations when an OS encounters a fault. First, the crashed OS itself

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

realizes the fault. Second, the OS cannot realize the fault. The confirmation of the death of the active OS for the first situation is easy because the active OS can send a dying message to the backup OS. The second situation, which is caused by serious bugs, immediately stops the OSes after they encounter the bugs or the bugs don't release the acquired locks [12]. In this situation, since the crashed OS cannot send a dying message, we use heartbeat messages that represent that the active OS is alive. If the backup OS does not receive any heartbeat messages at regular intervals, it concludes that the active OS is dead. After confirmation of the death in either way, the backup OS starts failover.

C. Device migration

The device migration mechanism is intended for migrating environments from the active to the backup OS when the former dies. Here we use a term environment to refer to the configuration files, the libraries, the IP addresses and the function of the devices. Environments are necessary for applications on the backup OS to perform the same way as the applications on the active OS. Because the active and backup OSes run on other LPAR partitions, the devices of both OSes are different when both are alive. Therefore, we conduct device migration to move the environments from the active OS to the backup OS. For example, migrating NIC between the OSes can pass IP and MAC network addresses. After the migration, remote processes that communicate with the active OS can communicate transparently with the backup OS after failures.

Virtual IP address (VIP), which is another method to migrate IP addresses among multiple machines, shares IP addresses among multiple hosts and can be migrated using a gratuitous ARP. However we do not use this method, because it requires two NICs for the proposal system to share an IP address between the active and the backup OS. However the backup OS does not work during normal times. Thus, the NIC of the backup OS will be wasted if we use VIP. On the other hand, device migration allows us to have only one NIC for sharing IP and MAC addresses. In this way, device migration can reduce the devices of the machine on which our proposed system operates.

We can discuss the storage in the same way as for the NICs. As mentioned in Section I, since DRBD syncs the storages of separate machines, we can use DRBD for sharing storage contents between the active and backup OSes instead of migrating the storage devices. However if we employ this method, one device will be wasted and syncing storages needs communication between the OSes. Thus, we migrate the storage devices instead of using a method to reduce devices. However, a method that syncs the contents of disks has an advantage if the storage device is broken physically. But our proposal is not intended for physical problems of machines but for OS bugs. In addition, we can make storage redundant by other methods, e.g., RAID.

The migration of devices is carried out after the backup OS confirms the death of the active OS. Migration is accomplished by modifying the layout of the LPAR partition of the backup OS to include the devices of the active OS.

D. File cache migration

File cache reduces I/O processing by caching the storage contents in the memory and accessing the cached contents instead of fetching them from storage. There are two ways for writing the cached contents. One is synchronously writing both the cache and the storage: write-through. The other is writing only to the cached contents, and reflecting to the storage is delayed for a certain amount of time to reduce I/O for writing the same storage blocks: write-back. Write-back is usually more efficient than write-through for CPUs latency and throughput. However, file updates might be lost when the OS is disrupted because some caches haven't been written to the storage yet.

The loss of file caches is a major problem occurred by OS failures. Our file cache migration mechanism can resolve this problem to enhance the credibility of the file system. Using a file cache migration mechanism, we can migrate the file cache that remains on the volatile main memory from the active to the backup OS. As a result, the file cache can be written properly to the storage device.

The following steps are performed during migration. First, the backup OS obtains the data structures related to the file caches from the memory region of the active OS. Second, it reconstructs file caches from the data structures. Finally, it writes the file caches to the storage.

E. Coverage

Here we discuss the coverage of our proposed system by clarifying failures that our system cannot fix. First, our proposed system is intended for OS bugs. Therefore hardware malfunction cannot be covered. If a storage device (e.g., hard disks or solid state drives) is broken, we cannot properly migrate file caches. As a result, the storage contents may be lost forever. However, this isn't a fatal problem, because we can overcome it by combining our system with other methods, e.g., RAID. A more fatal problem is caused by the interruption of the power supply, because both OSes on the proposed system must be stopped at the same time. To cope with a power outage, we can use an uninterruptible power source (UPS) to supply power so that the OS can shutdown properly. Thus, our proposed method can overcome the hardware problem with other techniques. Second, since our proposed method does not migrate process states, we cannot restore applications that depend on them. For migrating process states between OSes, Otherworld [3] was proposed. However, we believe the application states are not important because many applications prepare for urgent stops by storing important data in storage devices and can restore the state using the data. Therefore, we only migrate devices and file caches.

Last of all, our proposed system cannot handle bugs that damage data structures related to file caches. If such damages occur, we cannot migrate file caches properly. However, few bugs damage the data, because kernel functions carefully check the incorrect values of variables to stop the spread of errors among kernel data structures. File system bugs are exceptions, because the bugs directly damage the file caches. For this reason, we cover the OS bugs, except file system bugs.

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

IV. Application

In this section, we explain how our proposed system enhances the availability of a operating system on a machine using an example of an application to a Network File System (NFS) server.

A. Influence of OS failure for NFS

NFS is a kind of server-client application for sharing files. The NFS server exports its local directory to the network. The NFS client mounts the directory to its own file system. Since NFS is a stateless application, the NFS server can handle client requests even if the server is rebooted after a sudden interruption. However, only the case file operations are properly reflected to storage. If the NFS server stops because of OS failure while the NFS client was writing a file, the NFS client retries the operation that was being processed. Thus the NFS server will start processing the operation request after the NFS server returns. However, the previous operations for file-write are lost because the file cache was lost. So the operation will be incomplete. Therefore file-write operations will fail if the operations were being processed. We verified this by crashing OS while writing big files on NFS.

B. NFS server on proposed system

An OS failure suspends NFS until rebooting was finished and damages the NFS files. Our proposed system resolves these problems. In it, the NFS server will return soon after the backup OS takes over the active OS, and all file operations are reflected properly to the storage device by file cache migration. Here, we discuss our system's details.

For the initial state, we create an NFS server process on both the active and the backup OS. A NFS client is running on a remote machine and communicating with the NFS server on the active OS through the network. The NFS server on the active OS exports a directory /export, which is a partition on the storage device. When the active OS crashed because of bugs, alive monitoring detects it, and the backup OS starts the failover to restore the NFS server. First, the backup OS migrates the storage device and the NIC from the active OS to it. The storage device contains the partition for the NFS server. Then the backup OS mounts the partition on the same directory /export. After that, the backup OS migrates the dirty file cache that remains on the volatile main memory to it. Next, attaching the same IP address to the NIC used by the active OS, the backup OS can communicate with remote NFS clients. In this stage, the backup OS transparently finishes the failover for the client. After that, the server begins processing the suspended operations, which are successfully processed for the client because the previous operations are never lost. Hence, our proposed system can protect files even if the OS crashed when the file was being written by the NFS clients.

In this way, we migrate environments from the active to the backup OS, which transparently takes over for the crashed OS to the NFS clients. We only describe the NFS example, which is convenient for our system because of the stateless and continuous reconnections. However other applications (e.g., httpd) are also adaptable. We do not modify the applications so that existing applications can be easily executed on our system.

V. Implementation

In this section, we discuss the implementation of our proposed system, which we implemented on Linux (2.6.38, processor type x86_64). Both the active and backup OSes use this kernel. We describe the implementation of a multiple-OS execution platform, the failover mechanism, and alive monitoring.

A. Implementation of multiple-OS execution platform

We implemented multiple-OSes execution platform by referring to SHIMOS [11], which is a software implementation of LPAR. For creating a LPAR partition that represents a virtual machine, every OS on the machine must exclusively access the hardware of its partition. For this, SHIMOS uses the kernel parameters given by the bootloader and specifies the hardware at the initializing stage. An OS will not initialize all the hardware but only specified hardware to avoid accessing out of its partition. First, the OS is booted from the normal bootloader, and subsequent OSes are booted by a special bootloader based on kexec [4].

The CPU cores for a partition are specified by the first core id and the number of cores; e.g., the first core id is two, and number of cores is four, and then the partition has cores from #2 to #5. The main memory region for a partition is specified by the minimum and maximum addresses in bytes. The partition devices are specified by the list of devices represented by their bus and device numbers.

B. Failover

The failover of our proposed system is composed of device and file cache migrations. First, we conduct device migration. In the normal state, both OSes exclusively access their devices by detaching other's devices without initializing them. Devices are migrated by reinitializing them by the backup OS to which the devices are migrated. After reinitialization, we conduct device specific initialization. For example, we mount a partition of a storage device to the appropriate directory and attach the same IP address used by the active OS to the NIC.

After that, we conduct file cache migration with Ext3 for the target file system. For the migration, the backup OS reconstructs inodes from the dirty inodes that remain in the memory region of the active OS. The backup OS needs a way to the access memory of the active OS for retrieving dirty file caches. Despite exclusive access to the partition, it is not difficult to access the memory region of other OSes because exclusive access is not based on a mediation mechanism but mutual understanding of their partitions. In other words, an OS can read/write all of the main memory if needed. The backup OS can access out of the partition without errors or exceptions. However, we must be cautious about the virtual memory mechanism of Linux. Data structures related to the file caches have pointers to other data structures, and the address of the point is finally translated to a physical address by a virtual memory mechanism. This translation depends on the page table of the OS. If the address mapping differs between the active and backup OSes, the pointers cannot be translated properly. Fortunately, this is not a problem for our implementation because Linux uses a straight mapping region for most of the pointers. The straight mapping region does not differ between

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

TABLE I
MACHINE SPECIFICATIONS

CPU	Intel (R) Core (TM) i5 760 @ 2. 80 GHz
Memory	8GB
Devices	SSD (Crucial m4) x2, NIC (Realtek 8111)

the active and backup OSes. The backup OS can retrieve each data structure by simply accessing the pointers. Migrations are performed for each disk partition. Ext3 manages disk partitions as a structure named super block. A super block has a pointer to the thread structure for syncing the disk, and the thread has a pointer to the list of dirty inodes, which are migration targets. After retrieving the inodes, we reconstruct them for integration to the file system of the backup OS. Finally, the inodes are inserted to the list of dirty inodes. After migration, we can write or read the inodes in the usual way, and file caches will be written to the disk by the dedicated thread.

C. Implementation of alive monitoring

For alive monitoring, we use Inter Processor Interrupt (IPI) with which a core can communicate with other cores through the processor's interrupt controller. As we discussed in III-B, there are two messages, heartbeat and dying, for detecting the failure of the active OS. Both messages are sent by IPI. Their sources are the processor core of the backup OS, and their destinations are the first core of the active OS. The dying message is sent in the function panic(), which is called when the kernel encounters fatal errors. After receiving this message, the backup OS notices the death of the active OS and immediately starts failover. During the normal time, the active OS sends heartbeat messages at regular intervals using interval timers. If any message cannot be received for several seconds, the backup OS assumes that the active OS is dead. With these two methods, alive monitoring detects the failure of the active OS.

VI. EVALUATION

In this section we measure the time required to perform failover. The machine specifications for the evaluation are shown in Table I. We assigned an SSD for each OS and used two SSDs. However, in the future, we plan to remove one SSD by making the backup OS disk-less. For the measurement, we deliberately crashed the active OS and measured the length of the failover. In this evaluation, for simplicity, the backup OS immediately detected the failure of the active OS by receiving a dying message from the active OS. While the backup OS was taking over the active OS, the backup OS performed the following: (1) device migration, (2) mounting storage, (3) file cache migration, and (4) assignment of IP addresses to the NIC. We measured the amount of time for (3) with several different amounts of caches. The times for (1)(2)(4) did not differ by conditions, and thus we measured the total time for (1)-(4) instead of measuring them individually.

The measurement results of the amount of time for cache migration are shown in Fig. 2. About 90 MBytes of the file cache can be migrated within 70 milliseconds. We omitted the results less than 16 MBytes because the amount of time was too short. We tried to obtain a larger file cache by writing

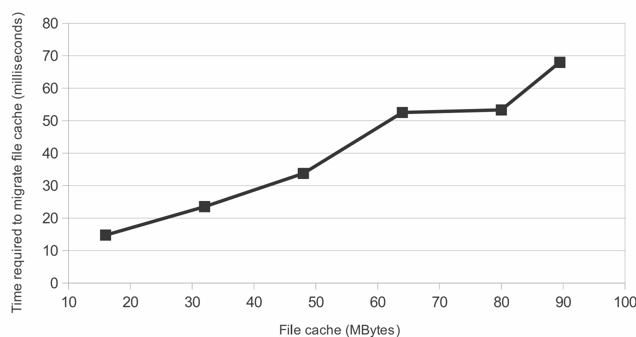

Fig. 2. Time required to migrate file cache

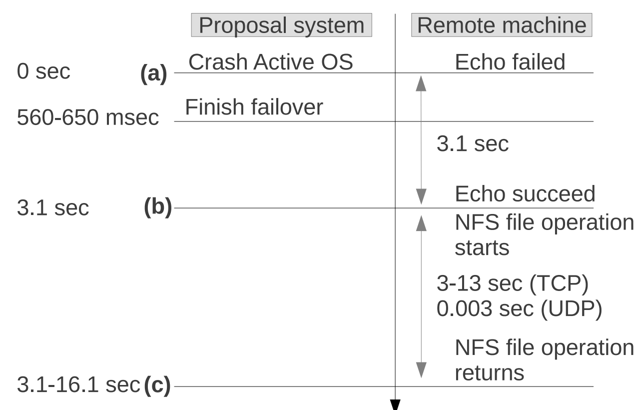

Fig. 3. Flowchart for evaluations

128 MBytes file just before the failure of the active OS. However, the amount of the retrieved file cache was less than 90 MBytes. Part of the file was already written by the thread dedicated to syncing the file cache, because the OS kernel tends to aggressively write-back file caches when there are so much in the memory. Therefore, the file cache we retrieved was less than 90 MBytes in our environment. After that, we measured the time for (1)-(4). The results were from 560 to 650 milliseconds. The margin of 90 milliseconds reflects the amount of time for file-cache migration. Our proposed system can finish a failover process within one second.

Next, we measured the amount of downtime for the network and the NFS on the proposed system using a remote machine connected to a switching hub to which our proposed system was also connected. We measured the time (a)(b)(c) shown in Fig. 3. First, we measured the network downtime. We continuously sent echo requests using Internet control message protocol (ICMP) before the active OS crashed. Next we measured the downtime using the amount of time between (a) the beginning of the first echo request that failed to come back and (b) the time of the echo request that first came back after the failover. The result was about 3.1 seconds, which is slightly longer than the one second of the failover. Thus we should identify a cause. We also measured the downtime for the NFS service by mounting a NFS directory exported by the active OS to the remote host's directory /mnt. We measured the time a remote host's request was suspended (requests for NFS server are suspended until the server comes back). The start time of the suspended operation was the

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)

same as (b), and its end is denoted by (c). The results ranged from about three to 13 seconds. They were much longer than the network downtime. We conducted the same measure using a UDP protocol, and the time was about 3 milliseconds. Therefore, if NFS uses a UDP protocol, it can communicate immediately after the network comes back.

VII. Related work

Otherworld [3] uses Microreboots [13] technique and migrates processes from the crashed kernel to the new booted kernel to maintain application executions. A new kernel is booted by kexec [4], which is a method for warm-booting. The data structures of the processes are retrieved from the main memory to reconstruct and migrate them to the new kernel. In this way, Otherworld protects the application executions from OS failure. However, Otherworld greatly depends on the kernel data structures that might be damaged by failures. As we mentioned in Section I, we don't think the application state is important. For safety, we only migrate file caches because the dependency of the file caches on the kernel data structures are much smaller than the process states.

RIO File Cache [14] proposes a method to write-back file caches to the disk after OS failure. For this, a function sync() is called after the failure. For ensuring the call of sync(), a dedicated interrupt handler is called after the failure. Sync() is modified so that it doesn't depend on the crashed kernel data structures, except the file caches. The RIO File can write-back the dirty file caches more safely than write-through. Our proposed system wrote-back dirty file caches using the backup OS instead of sync(). Therefore, it is less dependent on the kernel data of the active OS and can write-back the file caches as safely as Rio File Cache.

Microkernel is as an approach to reduce kernel bugs [15]. Pure microkernels have minimum functions in their kernels, and other functions are implemented as user processes. Therefore, microkernels can eliminate the possibility of inserting bugs into their kernels, and the other bugs can be treated as process errors, which are well isolated from other processes. Our proposed system improves the availability of the OS with monolithic kernels. However, our migration mechanism may be useful for microkernels.

The codes of device drivers often have bugs in their monolithic kernels [16]. The problems with driver codes are not only their size but also their complexity, which reflects device handling. Nooks [2] proposed a method that allows applications to continue after driver failure. However, other larger parts contain bugs (e.g., specific architecture codes) [17]. Therefore, methods for device drivers are not sufficient. Our proposed system can handle a large variety of bugs in the kernel.

VIII. Conclusion

In this paper, we proposed a system with a failover mechanism for operating systems. Our proposed system's failover takes less than one second. The user data in the file cache are never lost because they are migrated by a file cache migration mechanism. Consequently, our proposed system can resolve problems caused by rebooting after OS failure. It runs on commodity x86 machines and needs no modifications for applications. During normal times, since there is no special processing except for heartbeat messages, our proposed system is lightweight.

Future work will randomly insert bugs into kernels to measure the durability of our proposed system.

References

[1] A. S. Tanenbaum, J. N. Herder, and H. Bos, "Can we make operating systems reliable and secure," *Computer*, vol. 39, pp. 44–51, 2006.
[2] M. M. Swift, M. Annamalai, B. N. Bershad, and H. M. Levy, "Recovering device drivers," *ACM Transactions on Computer Systems*, vol. 24, pp. 333–360, 2006.
[3] A. Depoutovitch and M. Stumm, ""otherworld": Giving applications a chance to survive os kernel crashes," in *EuroSys*, 2008, pp. 181–194.
[4] A. Pfiffer, "Reducing system reboot time with kexec." http://www.osdl.org/.
[5] M. K. Mckusick and T. J. Kowalski, "Fsck - the unix file system check program," 1994.
[6] A. Sweeney, A. Sweeney, D. Doucette, W. Hu, C. Anderson, M. Nishimoto, and G. Peck, "Scalability in the xfs file system," in *In Proceedings of the 1996 USENIX Annual Technical Conference*, 1996, pp. 1–14.
[7] M. K. Mckusick, M. K. Mckusick, G. R. Ganger, and G. R. Ganger, "Soft updates: A technique for eliminating most synchronous writes in the fast filesystem," in *In Proceedings of the Freenix Track: 1999 USENIX Annual Technical Conference*, 1999, pp. 1–17.
[8] M. Rosenblum and J. K. Ousterhout, "The design and implementation of a log-structured file system," *ACM Transactions on Computer Systems*, vol. 10, pp. 1–15, 1991.
[9] "Keepalived for linux." [Online]. Available: http://www.keepalived.org/
[10] T. L. Borden, J. P. Hennessy, and J. W. Rymarczyk, "Multiple operating systems on one processor complex," *IBM Systems Journal*, vol. 28, no. 1, pp. 104 –123, 1989.
[11] T. Shimosawa, H. Matsuba, and Y. Ishikawa, "Logical partitioning without architectural supports," in *International Computer Software and Applications Conference*, 2008, pp. 355–364.
[12] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Symposium on Operating Systems Principles*, 2001, pp. 73–88.
[13] G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot - a technique for cheap recovery," in *Operating Systems Design and Implementation*, 2004, pp. 31–44.
[14] P. M. Chen, W. T. Ng, S. Chandra, C. M. Aycock, G. Rajamani, and D. E. Lowell, "The rio file cache: Surviving operating system crashes," in *Architectural Support for Programming Languages and Operating Systems*, 1996, pp. 74–83.
[15] C. Giuffrida, L. Cavallaro, and A. S. Tanenbaum, "We crashed, now what?" in *Proceedings of the Sixth international conference on Hot topics in system dependability*, ser. HotDep'10. Berkeley, CA, USA: USENIX Association, 2010, pp. 1–8.
[16] M. M. Swift, B. N. Bershad, and H. M. Levy, "Improving the reliability of commodity operating systems," in *Symposium on Operating Systems Principles*, 2003, pp. 207–222.
[17] N. Palix, G. Thomas, S. Saha, C. Calvès, J. Lawall, and G. Muller, "Faults in linux: ten years later," in *Architectural Support for Programming Languages and Operating Systems*, 2011, pp. 305–318.

ISBN: 978-988-19251-1-4
ISSN: 2078-0958 (Print); ISSN: 2078-0966 (Online)