Orthros: A High-Reliability Operating System with Transmigration of Processes

Kenji Yoshida*, Shoichi Saito*, Koichi Mouri†, and Hiroshi Matsuo*

* Nagoya Institute of Technology, Japan
Email: orthros@mail.ssn.nitech.ac.jp

† Ritsumeikan University, Japan
Email: mouri@cs.ritsumei.ac.jp

***Abstract*—We propose a method to solve problems that accompany recovering from operating system (OS) failures. First, to reduce recovery time, we make two OSes run simultaneously and configure them as an active-backup structure in one computer. This structure can provide a fast recovery from failures by a failover. Recovery time when using the proposed method is about 0.4 seconds at a minimum and up to about 10 seconds even if 2 GB memory is restored. Next, for smooth continuation of services after recovery, the proposed method preserves processes, their network connections, and file caches, and does not have runtime overhead to obtain a process execution status from the running active OS before a crash. In addition, the resources consumed to build the active-backup structure are only one CPU core and a small amount of memory. The hardware required to implement the proposed method is a multi-core processor and one disk for each OS; consequently, introduction of the proposed method incurs low cost. In the evaluation, we confirmed that the downtime was up to about 1.5 seconds when the active OS of the proposed system crashed while running a text editor, an NFS server, and a database server.**

***Keywords*—recovery, operating system, process migration, file cache migration.**

I. INTRODUCTION

Stable operations of OSes are very important because they are responsible for successful execution of all processes on a computer. However, bugs always exist in them and increase with every instance of adding or modifying features [7][21]. These bugs cannot be erased and can cause errors. Therefore, a fault-tolerant OS that is premised on faults is inevitable.

If an OS fails and stops, a recovery is generally performed by rebooting the computer. However, it is a problem that the reboot renders the computer unusable for a long time and destroys the execution status of the computer, which has running processes, unsaved file caches, and network connections held by the processes. If the network connections are halted, the recovery requires cooperation with communication partners to reconnect to the network. For this reason, the damage caused by the reboot spreads confusion to the partners. In this paper, we propose a method to improve the fault tolerance of a computer by reducing recovery time and preserving the execution status from the irregular stop caused by OS bugs without cooperation with other computers.

An existing approach to fault tolerance is use of High-Availability (HA) Clusters. However, HA Clusters have higher cost of deploying multiple computers. Furthermore, runtime overhead is high for synchronizing data between the cluster nodes every time data is updated. Another method to preserve the execution status of a computer is Checkpoint/Restart (C/R) [11][18], which method has much runtime overhead for checkpointing. If a checkpointing interval is extended in order to reduce this overhead, the success rate of restarting decreases.

From the above, we set the following goals for our proposed method:

- Conservation of processes and file caches,
- Fast recovery,
- Minimum hardware requirements,
- Minimum performance degradation.

There are other existing methods created with these goals in mind that save and utilize a memory image of a crashed OS [9], and run multiple OSes on one computer [19]. Otherworld [9] launches a new OS using a warm-boot at failures, and the new OS saves a memory image of the crashed OS to its own memory. That enables it to restore the execution status of processes by scanning the memory image. However, the warm-boot is not usually faster because it includes the boot operation of the new OS. Also, because Otherworld does not account for corruption of file caches, files that the processes handle may lose reliability after the recovery. Moreover, network reconnection is required after the recovery because Otherworld does not preserve the network connections. Next, Shimos2 [19], based on Software Logical Partitioning (Software LPAR) [22], provides an OS redundancy by running OSes on each CPU core and retaining process execution statuses by C/R. However, C/R has problems in that its runtime overhead will cause performance degradation, and it cannot necessarily preserve the latest execution statuses.

Considering the advantages and disadvantages of existing methods, we focus on achieving a complementary relationship to them and propose a new fault-tolerant method. We solve the slowness problem of Otherworld's recovery by running multiple OSes using Software LPAR in a similar way to Shimos2. On the other hand, we solve the overhead problem of Shimos2's C/R by getting an execution status of processes from the memory image of a crashed OS in a similar way to Otherworld. That is to say, the proposed method is a failover mechanism that uses warm-standby by multiple-OS execution using Software LPAR and gets the execution status without C/R by scanning the memory image of the crashed OS.

The proposed method has some advantages. There is no virtualization overhead because Software LPAR can directly execute instructions on a real computer without a virtualization mechanism. Furthermore, Software LPAR does not need additional special hardware. Because of this, Software LPAR does not interfere with achieving our goals of minimum hardware requirements and minimum performance degradation. Moreover, in obtaining the execution status of the processes, the proposed method removes effects of an abnormal status on the crashed OS by restricting kernel data obtention to a minimum. Here, the proposed method cannot repair a broken process. If it gets an execution status of a bugged process, the status includes the results of the bugs exactly.

From the above, contributions of this paper are establishment of each following method:

- Fast recovery method from OS failures,
- A method to reduce runtime overhead for recovery,
- A process migration method by scanning the memory image of the crashed OS.

By using the proposed method, you can achieve both the fast failover and the preservation of processes and file caches in curbing the hardware requirements and the performance degradation. Although the proposed method is inferior to specialized approaches for each of our goals, it is the first one to achieve high levels of performance for all goals at the same time. Concretely speaking, we succeeded in reducing downtime to about 0.4 seconds at a minimum, and required resources are only appropriative consumptions of one CPU core and 512 MB on physical memory. In addition, the proposed method requires little hardware: only a multi-core processor and a disk for each OS.

This paper is organized as follows. In Section II, we describe related works in order to clarify existing problems. In Section III, we outline the proposal and its advantages. We describe the system design and its implementation in Section IV. In Section V, we show the evaluation results. In Section VI, we discuss how the proposed method is applied to existing applications. Finally, we present our conclusions and touch on future works in Section VII.

II. Related Work

In this section, we describe related works on fault tolerance against OS faults.

A. Data Preservation

Otherworld is a method using microreboot to preserve execution statuses of processes. When a failure occurs in an OS, it warm-boots a new OS using a kernel image that was loaded into memory in advance. Warm-boot is a way to boot the new OS without stopping the computer; therefore, it can save the memory image of the crashed OS. By scanning this memory image of the crashed OS, the new OS reads kernel data of processes designated by a user, restores the processes, and resumes execution of the processes. In this way, it is possible to preserve the processes and their data from the OS failure. However, Otherworld does not preserve file caches and network connections; therefore, the processes that depend on the file caches or connection to the network cannot work correctly after the recovery. Further, the warm-boot does not need to initialize a BIOS, so it is faster than a normal reboot; nevertheless, it is slow because it has to initialize an OS. Recovery time when Otherworld is applied to applications was reported as 63 seconds in a shell, 64 seconds in MySQL, and 68 seconds in Apache.

Our proposed failover method is based on warm standby and reduces initialization time. In addition, it has two methods to preserve processes: write-back of file caches and migration of network sockets with packet buffers. Thus, the processes that depend on the files or connection to the network can work correctly after the recovery. Our proposed method achieves the data acquisition of file caches and process status such as network sockets by obtaining the memory image of the crashed OS just like Otherworld. However, obtaining the memory image is performed not by the warm-booted new OS but by another OS running simultaneously. Another OS starting the recovery provides more reliability than the crashed OS starting the recovery.

Chen *et al.* proposed a method that guarantees a write-back operation of file caches to preserve their contents in a failure [6]. Therefore, they introduce a crash handler that performs *sync()* whenever the OS crashes. A write-back operation at that time relies as little as possible on data of the crashed OS and preserves integrity of the file caches with high reliability. Our proposed method also ensures high reliability of file caches because an independent OS that has not failed checks failures and preserves the file caches.

In terms of the preservation of an execution status, a method proposed by Le and Tamir is similar to our proposal [16]. This method preserves the execution status of Virtual Machines (VMs) when a Virtual Machine Monitor (VMM) fails. A newly booted VMM obtains the execution status of VMs from a memory image of the crashed VMM. This achieves a fast recovery and reduces overhead for the preservation of the execution status. Although this method absolutely preserves the status of VMs, it does not deal with failures of OSes in VMs.

B. Recovery Improvement

Shimos2 is a method for fast recovery by allocating redundancy for OSes and preserving processes by using a C/R. Shimos2 runs OSes on each core using Software LPAR, so other OSes can continue even if one fails. This enables a faster failover at an OS failure than using a warm-boot or a reboot. For the failover, it uses C/Rs to preserve the execution status of processes. However, a C/R has greater runtime overhead. It is reported that overhead penalty is about 45% at a checkpoint interval of 50 milliseconds, and about 10% even at an interval of 1 second. Moreover, a C/R cannot preserve an unsaved execution status after the latest checkpointing. In our proposed method, two OSes run simultaneously as with Shimos2 in order to perform a failover. However, we achieve the preservation of the execution status not by using C/R, but by scanning the memory image of the crashed OS. This enables us to preserve the execution status of the processes and the file caches without the runtime overhead.

Another method using multiple OSes is a self-healing system of an OS using a virtual environment [14]. In this method, a second OS is introduced not for a redundant configuration but for remedying inconsistencies in the main OS. Although it can shift reboot timing somewhat, if a reboot is finally required, it takes a long time.

As a method providing fast recovery using a single OS, the one proposed by Baker and Sullivan should be mentioned [2]. This method implements a fast recovery of an OS and applications by providing a framework for backup/restore of the execution status, which takes a long time to regenerate. Although it does not spawn overhead substantially, the recovery time is several tens of seconds according to the report. In addition, this framework can only handle data that can be serialized.

In order to provide a fast reboot when an OS failed, a method which Yamakita *et al.*. were proposed takes several snapshots of a VM during booting with C/R technique [24]. When a failure occurs, it uses the best suited snapshot that can skip parts of a reboot sequence to reboot in a short time. However, it requires a VM environment and does not preserve status of user processes.

C. Fault Localization

As another approach for improving the fault tolerance of an OS, localizing the faults is conceivable. This method enables separation of a malicious code from an OS and local recovery of a crashed part. The limited-range recovery is fast and makes it possible to protect the entire status.

For example, there is a microkernel OS. If device drivers run in user-space, the OS is protected from the malicious device drivers, and the device drivers become possible to microreboot [4][5][10]. In the proposal of Swift *et al.*, the crashed device drivers are restored to the pre-failure status after reloading the drives [23]. Further, Ishikawa *et al.* provide a framework for self-healing device drivers [12]. Beside this, there is a method that detects a lockup in an OS and discards an operation at that time [8], and another method that makes operations request-oriented to limit a fault to the request level [17].

These approaches limit a fault location to inside of a device driver or a specified function in advance. Our proposed method also mainly deals with bugs in the device driver but does not need to identify the point of a bug. Therefore, it is more general.

III. Orthros

This section provides a brief description of the proposed system and its purposes.

Downtime in an OS failure is the total time it takes for administrators to notice a failure, reboot a computer, and restart processes. The longer the downtime is, the bigger the damage that the administrators and users suffer. In addition, launching the new OS causes loss of temporary data. To solve these two problems, we propose *Orthros* (ORganized Transmigratory High-Reliability OS), which improves fault tolerance by duplication of the OS.

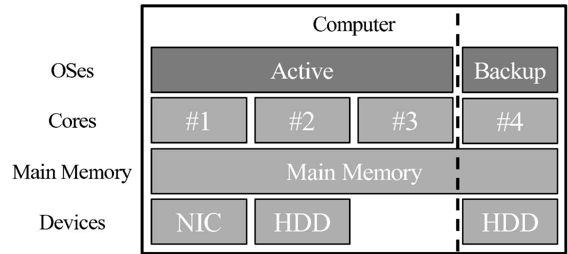

Fig. 1: Example of proposed system

A. Target and non-Target Faults

Target faults of Orthros are critical bugs that stop an OS, such as a null pointer dereference in a device driver and deadlock in a scheduler. In particular, most bugs are in device drivers [1]. Orthros preserves processes that an administrator specifies in advance of an OS stop and resumes their executions after recovery.

On the other hand, Orthros cannot handle faults that are in a user process or that destroy such kernel data as process tables and file caches in a kernel. Concerning the user process faults, Orthros does not touch any user processes and therefore cannot heal them. As for the kernel bugs, Orthros cannot deal with the above kernel bugs because it cannot save kernel data immediately before a breaking point. However, we suppose a possibility that such kernel fault occurrence is low. This is because kernel functions that handle kernel data always check the value of a variable in the kernel and prevent propagation of an incorrect value [25]. For this reason, kernel data on processes and file caches are rarely destroyed from unrelated faults. Naturally, if kernel data on processes and files were destroyed, Orthros could not recover any processes and file caches.

Further, Orthros cannot treat faults of hardware. However, disk faults are resolved by using a RAID configuration, and power outages are dealt with by using UPS. Combining these techniques, it is possible to improve the fault tolerance against hardware faults.

B. System Structure

Fig. 1 shows an example of an Orthros configuration. For fast recovery, Orthros prepares two OSes and configures them as an active-backup structure. An active OS is one that handles all tasks, and a backup OS is a spare that is used when a failure occurs in the active OS. These two OSes run simultaneously on one computer for minimum hardware requirements. In Orthros, resources for the backup OS are wasted because it does not perform any task before a failure occurs. Therefore, the active OS owns almost all of the resources, and the backup OS occupies only a minimum number of CPUs and amount of memory during normal execution.

To minimize performance degradation when both the backup and active OSes run, we use LPAR for simultaneous execution of the OSes. This is because there is hardly any virtualization overhead, unlike virtualization mechanisms such as Xen [3] and KVM [15], and increase of resource consumption for introducing the backup OS is limited to one CPU core

and a small amount of memory. We think that the consumed resources would not be a problem due to popularization of the many-core CPU and large amounts of memory. Finally, the hardware requirement for the backup OS is only one multi-core CPU because we implement LPAR by software. Thus, obtaining the required hardware for our proposal is easy and low cost.

C. Auto Detection and Migration

To reduce the time until an administrator notices a failure, the backup OS constantly monitors whether a failure has occurred on the active OS. If the backup OS detects a failure, it automatically starts the failover and preserves the processes and the file caches that have been designated by the administrator before the failure. Process preservation is implemented by a process migration, and file cache preservation is implemented by file cache migration.

If C/R is used to preserve processes, runtime overhead of the active OS becomes large. To avoid this overhead, Orthros does not employ C/R but obtains the memory image of the active OS, and reads the execution status from the memory image. The process migration is achieved by generating a new process with the same execution status on the backup OS. Preservation of file caches is implemented by reading and migrating them as well. As a result, the processes whose behavior depends on files can continue to work correctly. In addition, our method migrates a network environment into the backup OS. By migrating sockets the processes hold, Orthros enables a transparent recovery without disconnections.

IV. Design and Implementation

In this section, we give an overview of the failover and mechanisms of the following functions required for the failover:

- Multiple-OS execution platform,
- Alive monitoring,
- Resource migration,
- Process migration.

Finally, we summarize data and conditions required for the failover.

We implemented the proposed system on Linux (version 2.6.38, processor type x86_64). Also, we have always implemented a system that reduces downtime and preserves file caches at an OS failure [13]. The implementation from section IV-B to IV-D was achieved using this system.

A. Failover Overview

During a normal time execution, the active and backup OSes run in a computer on a multiple-OS execution platform. The backup OS continually monitors the active OS by alive monitoring, and waits until the active OS fails. The resource migration is an operation for the backup OS to use the active OS's resources. In particular, the backup OS migrates file caches and a network environment. The file cache migration is the center of the preservation of file caches. Not only that, it also intends to execute the processes in the same environment as the active OS. After the resource migration, the backup OS migrates the stopped processes into the backup OS. The failover is completed when the migrated processes resume correct operation.

After the failover, the migrated process continues execution on the backup OS, which then serves the subsequent processing. We will consider functions following the start of the backup OS in future work. The backup OS initializes and makes use of CPU cores and physical memory of the active OS because the backup OS has only minimum resources. Also, in order to use the backup OS as the main one, the backup OS newly boots an other OS that becomes the backup of the backup OS after the failover. Thus, two OSes run simultaneously in the same way as before the failure.

B. Multiple-OS Execution Platform

To execute the two OSes simultaneously on one computer, Orthros uses LPAR, a type of virtualization technology. LPAR divides hardware into logical partitions, and transforms each partition into a VM. Therefore, running OSes on each partition allows simultaneous operation of the OSes on a single computer. Each OS on LPAR uses only designated hardware and cannot use other hardware. This allows the OSes to share many devices on one computer without a centralized virtualization mechanism. For example, a computer with a four-core CPU, 8 GB of memory, and two SSDs can be divided into two partitions with two CPU cores, 4 GB of memory, and one SSD. The two OSes can run independently on each partition.

The reason for Orthros to use LPAR is that virtualization overhead is very small as stated previously. Moreover, Orthros uses Software LPAR, which is a type of LPAR implemented by software. The first reason for this is that Software LPAR requires only a multi-core processor in principle. Normal LPAR requires special hardware, and therefore interferes with our goal of minimum hardware requirements. The second reason is easy hardware migration between OSes in a Software LPAR environment by changing partition configuration dynamically. Using this feature, the backup OS can read the memory image of the active OS, and migrate devices such as a disk or NIC if necessary.

We referred to SHIMOS [22] and Mint [20] for implementation of Software LPAR. Each assignment of CPU cores, physical memory, and devices is designated using kernel parameters given by a boot loader. Each OS recognizes its own role and available hardware at an initializing stage, and then boots up with initializing its hardware. The two OSes boot in the following order: the active OS and the backup OS; the active OS boots with general steps but the backup OS is launched with an exclusive function of Orthros by the active OS. An initialization stage of the backup OS is modified in various hardware initialization codes affecting execution of the active OS. The backup OS does not use most devices that the active OS requires, such as NIC and keyboard, and interfere with the active OS. Assignments of drivers for devices exclusive to each OS are pending at these initialization stage. Because an OS recognizes the pending devices as unusable, each OS can exclusively use the devices. This is a mere impermanent reservation; therefore, the backup OS can get control of the devices after initializing them again when it

requires them. By the above method, the device migrations of disks and NICs in the resource migration are achieved.

The memory migration can be implemented easily by setting page tables after using a memory hot-plug function of Linux. However, when reading data in another OS memory area, it is necessary to pay attention to handling of virtual addresses. The same virtual address is mapped to different physical addresses in the active and backup OSes because their virtual memory managements are independent. Consequently, we use a straight mapping region in Linux x86_64, where all physical memory is mapped to fixed virtual addresses; thus, virtual addresses in the straight mapping region are the same in all OSes. By converting virtual addresses into straight mapped virtual addresses, there is no need for special implementation to access data that Orthros needs. We used *WANT_PAGE_VIRTUAL* macro and convert address: *__va(__pa(address))*. Although an access to a user space of process requires a mapping, a code of *fork()* can be used, as we will describe in IV-E.

We implemented the above functions that are used for resource and process migration. However, the current implementation of multiple-OS execution platform needs an extra dedicated disk for the backup OS because an extra disk for a file system of the backup OS is required. To eliminate this, we think it is good to provide a virtual disk such as an initial ramdisk for the backup OS.

C. Alive Monitoring

We describe an alive monitoring mechanism in which the backup OS monitors the active OS. When an OS stops by a fault, the OS may or may not be able to detect the failure. The OS can perform minimum fault handling such as outputting the cause of the failure if it detects the failure, but cannot perform any handling if it cannot detect the failure.

The backup OS uses Inter-Processor Interrupt (IPI) in order to detect abnormality of the active OS. We introduce two types of IPI: a heartbeat message that communicates proper working status and a dying message that informs of a failure. The backup OS recognizes the status of the active OS by these messages, and if it detects a failure, it migrates resources and processes by executing a failover script. With the above method, the active OS can begin the failover early at a mild failure, and the backup OS can perform the failover in fixed time even if a failure is serious.

If the active OS detects a failure in itself, it can start the failover by sending a message that indicates an abnormality to the backup OS. An IPI that represents the dying message is sent from the *panic()* function of the active OS. *panic()* is called when an unrecoverable fault occurs in the OS. The backup OS recognizes that the active OS has stopped abnormally by receiving the dying message.

The active OS cannot send the dying message when it is stopped by a serious error such as a deadlock. To settle this, the active OS sends the IPI that represents the heartbeat message to the backup OS periodically. The backup OS recognizes abnormality if it does not receive the heartbeat message over a certain period of time. The heartbeat is sent in turns by all cores, so this ensures that all cores are working properly. Although the detection becomes proportionally quicker as the heartbeat interval becomes shorter, it is usually undesirable to increase the number of interrupts. However, overhead per interrupt is not significant because sending and receiving IPIs are simple operations. We set the heartbeat interval for whole cores to 0.1 seconds currently, and believe overhead in this interval is sufficiently small. If the heartbeat message is lost for 1 second, the backup OS detects abnormality and starts the failover.

D. Resource Migration

Here, we describe a resource migration that migrates resources of the active OS to the backup OS when a failure occurs. The resource migration has two objectives: file cache preservation and process preservation. The file caches and the network environment are required for properly running processes dependent on files, and recovering connections transparently with a communication partner across the network. We describe and solve problems that occur when migrating the file caches and the network environment. How to migrate devices is mentioned in section IV-B, and thus we describe implementations of required operations after device migration. These operations are performed by a failover script that is started when the alive monitoring detects a failure. Then, these operations are performed only at a failure, and therefore it does not involve runtime overhead.

1) File Cache Migration: First, the backup OS migrates and gets control of the disk the active OS has been using in order to enable the processes to seamlessly use the file cache that they had been using until the failure. By mounting the disk in the same path as when it was used on the active OS, the migrated process can handle files in the same paths.

Here, in order to maintain consistency between files that are used by a migrated process before and after the migration, the backup OS must synchronize file caches in the active OS with files in a disk. There are two methods to reflect the file caches in the disk: a write-through method synchronously reflects the changes, and a write-back method asynchronously reflects them. The write-through method updates a file in a disk right after changing a file cache. In contrast, the write-back method only changes a file cache at first, and then updates a file in a disk with a delay. The write-back is faster than the write-through, and thus many OSes choose the write-back. However, it cannot maintain cache consistency when an OS fails.

There are two methods to maintain cache consistency, copying and remapping. Remapping continues to use physical memory pages in the active OS and is faster than copying. However, it interferes with reusing them for a new backup OS, which is a backup of the current backup OS after recovery in a future work. Therefore, Orthros copies the cache pages for maintaining cache consistency.

A file cache migration looks up dirty file caches from the memory of the active OS, and then copies their contents into the backup OS. The dirty file caches can be traced from *super_block* structures in the memory of the active OS corresponding to the file caches on the disks. A file cache preservation copies their contents and associates them with *inode* structures on the backup OS. We implemented the file cache migration for the ext3 file system.

2) Network Environment Migration: For processes to communicate in the same network environment pre-failure, the backup OS migrates NICs that the active OS has been using, gets control of them and assigns IP addresses of the active OS to them. Consequently, it can communicate with the same IP and MAC addresses.

To achieve a transparent recovery of communications over the network, errors must be in a range that can be automatically corrected by communication protocols. Inconsistency under the network layer does not arise because the backup OS can have the same IP address and a port number of the active OS by the network migration. Therefore, the communication partner does not need to detect occurrence of the migration. Moreover, errors over the session layer also do not occur if a process migration succeeds because session information is contained in an execution status of the process. Finally, we have to deal with TCP/UDP in regard to the transport layer. Control information of TCP/UDP communications is contained by kernel data of processes. We will describe how to correct TCP/UDP communications in section IV-E.

Another way for the backup OS to use the same IP address of the active OS is a virtual IP address. The backup OS can assign the same IP of the active OS to its own NIC by using Gratuitous ARP. However, if Orthros adopted Gratuitous ARP, the backup OS would require at least an extra NIC and would waste it because it does not do any work during normal execution. We reduce required hardware by migrating the NIC in Orthros.

In the network environment migration, the backup OS sets up a network configuration using *ifconfig* and *route* commands after the device migration of NICs. It sets an IP address, a routing table, a communication speed, and a communication mode.

E. Process Migration

A process migration mechanism restores temporary data of processes which are designated by administrators. If processes are executed when a failure occurs, the mechanism continues execution of the processes. Similar to the file cache migration, it is achieved by reading and reconstructing process management structures that exist in a memory image of the active OS. We used a code of *fork()* system call for the process migration. The code of *fork()* that copies execution statuses (contexts) of the current process is changed to copy contexts of a designated process that has stopped on the active OS. However, some data, such as a *task_struct* of the parent process, *preempt_count*, and *sched_rt_entity*, are not copied. This is because these data depend on a surrounding environment of the process and the environment changes before and after a recovery.

When Orthros preserves memory contents of the processes, it only migrates the memory contents in the user space because its targets are failures that occur in an OS. Failures in the OS are almost all due to an operation in the kernel space, and reoccur with high probability if much kernel data are taken over.

1) Context Migration: A *task_struct* of Linux kernel is a collection of complex data. Hence, it is difficult to copy all of its data between different running OSes from the viewpoint of consistency. We accordingly defined the minimum contexts required to keep the correct execution of processes as follows:

- register values in a user-space execution,
- memory contents in the user-space,
- file descriptors,
- thread group structures,
- communication statuses.

The backup OS copies the above contexts and reconstructs them on itself.

a) Register Values in a User-Space Execution: Location of the register values depends on status of a process at the moment of a failure. When the process is executing a system call or is waiting, the backup OS can get the values by reading the memory of the active OS because the process stores the register values in the memory. If the process is running in the user-space, the backup OS cannot directly read the values because the values were on a CPU. To get the values on the CPU, we use a Non-Maskable Interrupt (NMI), which forces the process to switch into a kernel-space and stores the register values in the memory. If the active OS can call the *panic()* function, the NMI is sent from the *panic()* function of the active OS to all cores of the active OS. If not, the backup OS sends the NMI after detecting a failure by alive monitoring.

b) User-Space Memory: Among the user-space memory, only dirty pages and virtual memory management structures are copied. A demand paging can regenerate non-dirty pages automatically even after a failover, and hence copy time of user-space pages can be shortened.

c) File Descriptors: Data structures that are directly linked to device drivers might be causes of failures. When making a copy of the file descriptors, the backup OS avoids migrating all of the *file* structures. Therefore, instead of copying them, it makes a new *file* structure by opening the same file and only migrating necessary data such as flags or a seek position.

d) Thread Group Structure: If a process has formed a thread group, a lightweight process that plays the role of a child thread has to be migrated with the process. At this time, pointers to management structures of memory contents, files, and signals are equalized through a whole thread group. Further, in order to maintain a parent-child relationship in the thread group, Orthros has to assign the same process ID of the active OS for a process of the backup OS. This is because a process may store its process ID in user-space memory and Orthros has to maintain consistency with its ID. In our implementation, in order to avoid an overlap of process IDs between the active OS and the backup OS, we prepare an exclusive *pid-namespace* for the process migration. The *pid-namespace* function makes it possible to manage separate process ID spaces in one OS. This enables process migration without any change of the process ID.

e) Network Communication Statuses: To maintain network connections, a network environment migration settles errors under the network layer, and a process migration settles errors over the session layer as well. Then, to settle errors

on the TCP/UDP of the transport layer, we have to preserve data of sockets linked from a *task_struct*. Simple TCP communication can be basically continued if only a network layer status, window control information, and packet queues are preserved. Specifically, the preserved packet queues are *recerve queue*, *send queue*, and *out_of_order queue*. This can maintain connection without inconsistency because the TCP has a retransmission control, and the UDP does not need reliability assurance.

We have implemented the above functions, and applications described in section VI are available for maintaining connection. However, the current implementation can handle the TCP and the UDP statuses, except for a TCP Timestamp option and TCP asynchronous wait. The TCP Timestamp option is difficult to migrate because the backup OS cannot completely synchronize a system time with that of the active OS. To control packet delay due to this, the backup OS sets its *jiffies* ahead in comparison to the active OS.

2) Error Handling: As already mentioned, the kernel memory contents are annulled at the process migration. Thus, if a system call has been in operation for the processes, it cannot continue. Because of this, we adapt the process status to deem the system call as failed.

If such an operation is performed on a program that is vulnerable to the system call failure, the migrated process may abort or get an incorrect return value of the system call. For these programs, we provide a library that allows flexible error handling after the process migration. The library adapts the process's status to automatically re-invoke the system call without any modification of the program if loaded at the start of the process by using *LD_PRELOAD*. In addition, a programmer can instruct the program to do a detailed error handling at a failure by changing its source code and linking the library.

The error-handling is achieved by rewriting the program counter and the stack. If Orthros detects this library at a process migration, it sets the program counter to a function in the library and pushes an original program counter and registers information required for the error handling onto the stack. Then, the migrated process automatically executes the library function. The function calls the interrupted system call or a user-defined function, and makes adjustment to registers as if nothing had happened, then returns to the original program.

A re-execution of the interrupted system call may cause an overlapping effect. However, local system calls such as *read()* and *write()* only write the same data doubly to the same place, and there is no problem. Also, an effect of network system calls such as *send()* and *recv()* is concealed by an error correction of TCP and a user code for UDP. Thus, we have to consider a few system calls such as *lseek()*. Their re-execution must be manually controlled in the error-handling library.

F. Required and Preserved Data

Based on the description in this section, we summarize the necessary data for recovery. To recover, the active OS hands the following data to the backup OS during a normal execution: *task_struct* addresses of specified processes, an address of *runqueues*, an address of *per_cpu_offset* region, and *super_block* addresses of the specified file system. Moreover, the recovery process supposes the following data are not corrupt: almost all data under *task_struct*, information of the current process on *runqueues*, register data on kernel interrupt stacks, links from *super_block* to dirty *inode*s, links from *super_block* to dirty file cache pages, memory contents of processes and file caches.

Orthros can preserve processes from the active OS crashes and resume them, but cannot keep some data that are stored in device drivers, for instance graphics and sound devices. They are initialized at the recovery process. In addition, a time synchronization between the active OS and the backup OS is difficult; therefore, process times before and after a recovery do not perfectly match. These problems are for further study.

TABLE I: Hardware allocations for each OS

	CPU	Memory	Device
ActiveOS	3 cores	7680 MB	SSD,NIC
BackupOS	1 core	512 MB	SSD

CPU : Core i5 760 2.80 GHz 4 cores.

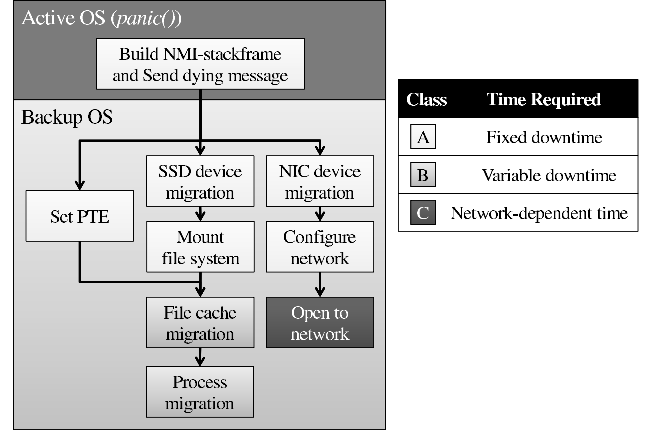

Fig. 2: Failover flowchart

V. Evaluation

In this section, we measure time required for a failover. A machine configuration for an evaluation is shown in TABLE I. This allocation of a CPU core, memory, and SSD for the backup OS is the minimum for running the backup OS. A system administrator can assign the allocation of the backup OS after considering the amount of computer resources and required resources for migrated applications.

An overview of a failover is shown in Fig. 2. Time for the failover is composed of (A) fixed downtime, (B) variable downtime, and (C) network-dependent time. The failover is started artificially by a *panic()* function, and ends when all operations in Fig. 2 are completed. We measure three types of time shown in Fig. 2.

A. Fixed Downtime

The A in Fig. 2 represents fixed downtime independent of execution statuses of a computer. This time was 415 milliseconds in total. A breakdown of the time is shown in Fig. 3. The

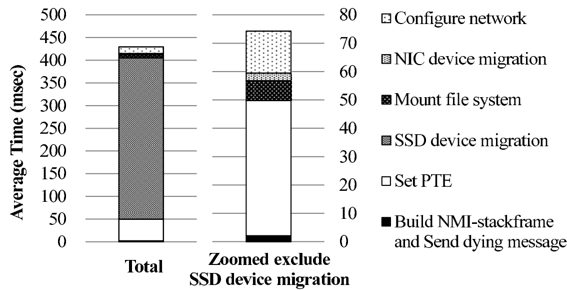

Fig. 3: Fixed downtime

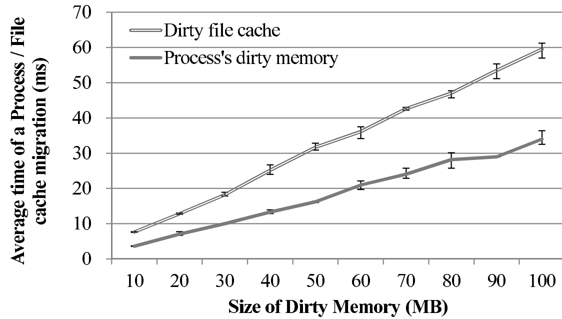

Fig. 4: Time required to migrate a process/filecaches from the amount of dirty memory

left bar shows the whole time of the fixed downtime, and the right bar shows time except for an SSD device migration for clarity. Most of the time is spent in an SSD initialization in the device migration.

B. Variable Downtime

Operations that are represented by B in Fig. 2 spend time depending on sizes of processes and file caches. We call the time of these operations variable downtime, and measure each time individually. Factors that affect the variable downtime are the following four: dirty file cache size, dirty memory size of a process, number of opened files in the process, and number of threads in the process. We measure migration times in these cases. On each measurement, we adjust non-target parameters to minimize them.

First, we investigate the influence of memory copy size on the failover time. The relationships of file cache migration time to the size of dirty file caches and process migration time to the size of dirty memory were measured. The result is shown in Fig. 4. From this result, variable downtimes and sizes of these factors prove to be proportional. Average times per 1 MB are 0.6 milliseconds for the file cache and 0.3 milliseconds for the process. We conclude from the above that there is no influence as long as the size of dirty memory is under a GB order. The reason that the file cache takes longer time than the process memory is that the management structure of a file cache is more complex than that of a process, and thus scanning it incurs large overhead.

Next, the relationship of process migration time with number of opened files is shown in Fig. 5(a), and with number of threads is shown in Fig. 5(b). From this result, variable

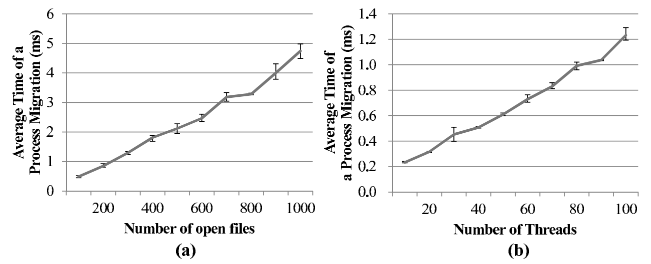

Fig. 5: Time required to migrate process from number of files/threads

times prove to be proportional also to the number of opened files and threads. Average times are 50 microseconds for the file cache and 0.3 milliseconds for the process. This time is not a problem as long as it is a general Linux program.

From the above, it can be seen that the time when we should be careful about variable down time is only the memory copying time when copy size is over GB order. However, the size of dirty memory does not become extremely large for the file caches and the processes. This is because an OS will put out pages to disks when too much memory is used; moreover, the maximum size of dirty pages, which need to be copied, is less than physical memory size. Although this time is impossible to ignore, it is faster than the recovery time of large systems needed to copy GB order.

C. Network-dependent Time

An operation represented by C in Fig. 2 is a waiting period until actually being able to connect to the network after the configuration for the NIC in the resource migration. We call this waiting time network-dependent time. The network-dependent time may vary depending on network configurations of computers and can be concealed from the total migration time if there is a vacant time to start a communication after a network configuration. Therefore, we measure it independently.

In this evaluation, we measured time until the ping (icmp echo) was received by another computer on the same network after the NIC configuration. If an auto-negotiation is used for speed and duplex mode configuration, it takes about 4 seconds. To speed up the process, we disabled the auto-negotiation and manually configured the speed and duplex mode in Orthros. As a result, the recovery of communication required about 1,007 milliseconds in the above configuration. However, we think this time cannot be ignored and must be reduced. We are considering two approaches as reduction methods: migrating device execution statuses of the NIC on the active OS, or optimizing the configuration of network equipment.

D. Total Downtime

From results from sections V-A to V-C, a minimum recovery time when we only preserve a small offline process proves to be about 0.4 seconds. Then, a minimum recovery time is about 1.4 seconds if the migrated process uses the network just after the recovery. Also, a serious failure adds downtime of 1 extra second because the failover is started not by a dying message from the *panic()* function but by a heartbeat message.

In addition, if a dirty memory becomes large, copying time will take maximum 0.6 seconds per 1 GB at a maximum limit of physical memory size.

VI. Application

In this section, we discuss an application possibility of our proposed system, Orthros. We verify advantages of our proposed system for each application and measure actual downtime. The measurements are conducted in the same environment and failover flowchart we showed in Section V.

A. Interactive Applications

An example of an interactive application is *nano*, which is a text editor. *nano* does not save editing contents to a disk unless a user instructs it to save, and always stores them on its memory as temporary data. Further, *nano* does not have an auto-save function, and therefore it takes a long time to reproduce much temporary data if a failure occurs in editing. By applying Orthros to a text editor such as *nano*, a user can preserve editing contents from a sudden OS failure. In an experimental failure, editing contents were maintained against migrating to the backup OS, and were able to be written out to a disk by the user's operation.

We measured the downtime when the failure occurred using *nano* on Orthros. When we opened a file of about 1 MB and edited 128 KB, dirty pages were about 2.5 MB. Time required to copy these pages was less than 2 milliseconds. Then, a user was able to resume editing text in the console of the backup OS in 0.43 second on average.

B. NFS Servers

An NFS does not harm user files by a network disconnection from an OS failure. If an NFS server halts during file operation, an NFS client automatically repeats a request until the request is received. Therefore, the operation is resumed automatically after communication is recovered, and the NFS operation is transparent to users. However, this is not applied when file caches are lost. For example, if an OS failure occurs while writing a file, the file that was being written just then may be destroyed. Therefore, requests to the file may not be processed correctly even after communication recovers with a reboot. It can be said that the NFS server is vulnerable with respect to these file cache losses. In addition, systems that depend on the NFS server stop for a long time until a reboot is completed when a failure occurs on the NFS server. Although this problem can typically be settled by duplicating the server, the duplication causes an increase of management complexity and introduction cost as a trade-off.

Applying Orthros to an NFS server settles the two problems of file destruction and stoppage of client systems for a long time. This is because the backup OS can obtain disks and file caches on the NFS server, which is on the active OS, by resource migration. The NFS server on the backup OS can communicate with an NFS client in the same way as the active OS after the network environment migration. At this time, the NFS server on the active OS does not need to be migrated to the backup OS because it is a stateless application. We just need to start a new NFS server instead of the process migration.

We applied Orthros to an NFS server and measured downtime when a failure occurred. In this measurement, computers that run an NFS server and a client are connected via a 1 GB network switch, and we measured time for resuming an operation after communication was broken. In this case, we ran another NFS server on the backup OS in advance to omit process migration time. As a result, 296KB of dirty file caches was copied and the average downtime was 4.3 seconds. It is significantly longer than the result of section V-D. This time increment reflects the influence of a retransmission interval of the NFS client, and a timeout period of TCP. Thus, we used UDP instead of TCP, set the timeout period to 0.1 seconds, and got a result that downtime became 1.5 seconds. This time is approximately the same as the minimum downtime.

C. Database Servers

If a failure occurs in an OS that runs a database server, sessions between the server and clients are disconnected and executing transactions are rolled back. Therefore, the database clients have to reconnect after waiting until the database server recovers, and also redo the transaction that was executing at the failure. Orthros can solve this problem, which decreases availability.

We applied Orthros to a *MySQL* (*InnoDB*) server, and measured downtime at a failure. Computers that run a *MySQL* server and a client are connected via a 1 GB network switch, and we calculated the downtime by comparing a query execution time at a normal execution with a time at a case of a failure in the middle of the execution. As a result of the measurement, the average downtime was 0.47 seconds. At this time, the size of dirty memory is about 40 MB, and copying time is about 24 milliseconds. In this measurement, a network-dependent time is concealed in a query execution time of a post-failover. Therefore, a delay of a maximum of 1 second comes to the surface when the query execution time becomes shorter.

As another advantage when Orthros is applied to a database server, acceleration becomes possible. Since Orthros improves reliability of execution statuses on memory, it is possible to use techniques that are fast but decrease reliability. For example, we are considering reduction of *sync* to a disk that is performed each time a database is changed, and use of an in-memory database instead.

VII. Conclusion

In this paper, we proposed Orthros, which runs two OSes simultaneously and performs a failover. Orthros can quickly recover and preserve processes and file caches when a failure occurs in the OS. We demonstrated that Orthros has advantages when applied to varied applications, and that a recovery is completed within a few seconds. The recovery time is far faster than both the time to warm-boot a kernel and the time to reboot a computer.

Orthros uses Software LPAR to achieve minimum hardware requirements and minimum performance degradation; moreover, Orthros makes use of memory of a failed OS to preserve processes and file caches without runtime overhead. As a result, the hardware requirements are only a multi-core processor and one extra disk. We are currently developing an implementation that eliminates the need for the extra

disk. Factors that cause performance degradation are only appropriative consumptions of one CPU core and 512 MB of physical memory, and simple interrupts for sending heartbeats.

As future work, we would like to ensure against failure after a recovery. Currently, a reboot is needed when a failure occurs on the backup OS. To solve this problem, we will implement a function that boots a new backup OS after the failover and assigns a different kernel version to the new backup OS.

ACKNOWLEDGMENTS

We would like to thank Tomoaki Tsumura, Toshihiro Matsui, Shinsuke Kajioka, Yudai Kato, and anonymous reviewers for their comments and suggestions.

REFERENCES

[1] Ahmed, M. F. and Gokhale, S. S.: Linux bugs: Life cycle, resolution and architectural analysis, *Inf. Softw. Technol.*, Vol. 51, No. 11, pp. 1618–1627 (2009).

[2] Baker, M. and Sullivan, M.: The Recovery Box: Using Fast Recovery to Provide High Availability in the UNIX Environment, *Proceedings of the USENIX Summer Conference*, pp. 31–43 (1992).

[3] Barham, P., Dragovic, B., Fraser, K., Hand, S., Harris, T., Ho, A., Neugebauer, R., Pratt, I. and Warfield, A.: Xen and the art of virtualization, *Proceedings of the 19th ACM symposium on Operating systems principles*, pp. 164–177 (2003).

[4] Boyd-Wickizer, S. and Zeldovich, N.: Tolerating malicious device drivers in Linux, *Proceedings of the USENIX Annual Technical Conference*, pp. 117–130 (2010).

[5] Candea, G., Kawamoto, S., Fujiki, Y., Friedman, G. and Fox, A.: Microreboot - A technique for cheap recovery, *Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6*, pp. 31–44 (2004).

[6] Chen, P. M., Ng, W. T., Chandra, S., Aycock, C., Rajamani, G. and Lowell, D.: The Rio file cache: surviving operating system crashes, *Proceedings of the 7th international conference on Architectural support for programming languages and operating systems*, pp. 74–83 (1996).

[7] Chou, A., Yang, J., Chelf, B., Hallem, S. and Engler, D.: An empirical study of operating systems errors, *Proceedings of the 18th ACM symposium on Operating systems principles*, pp. 73–88 (2001).

[8] David, F. M., Carlyle, J. C. and Campbell, R. H.: Exploring recovery from operating system lockups, *Proceedings of the USENIX Annual Technical Conference*, pp. 351–356 (2007).

[9] Depoutovitch, A. and Stumm, M.: Otherworld: giving applications a chance to survive OS kernel crashes, *Proceedings of the 5th European conference on Computer systems*, pp. 181–194 (2010).

[10] Giuffrida, C., Cavallaro, L. and Tanenbaum, A. S.: We crashed, now what?, *Proceedings of the 9th international conference on Hot topics in system dependability* (2010).

[11] Hargrove, P. H. and Duell, J. C.: Berkeley lab checkpoint/restart (BLCR) for Linux clusters, *Journal of Physics: Conference Series*, Vol. 46, No. 1, pp. 494–499 (2006).

[12] Ishikawa, H., Courbot, A. and Nakajima, T.: A Framework for Self-Healing Device Drivers, *Proceedings of the 2008 2nd IEEE International Conference on Self-Adaptive and Self-Organizing Systems*, pp. 277–286 (2008).

[13] Kato, Y., Saito, S., Mouri, K. and Matsuo, H.: Faster Recovery from Operating System Failure and File Cache Missing, *Proceedings of the International MultiConference of Engineers and Computer Sciencists*, Vol. 1, pp. 218–223 (2012).

[14] Katori, T., Sun, L., Nilsson, D. K. and Nakajima, T.: Building a self-healing embedded system in a multi-OS environment, *Proceedings of the 2009 ACM symposium on Applied Computing*, pp. 293–298 (2009).

[15] Kivity, A.: kvm: the Linux virtual machine monitor, *OLS '07: The 2007 Ottawa Linux Symposium*, pp. 225–230 (2007).

[16] Le, M. and Tamir, Y.: ReHype: enabling VM survival across hypervisor failures, *Proceedings of the 7th ACM SIGPLAN/SIGOPS international conference on Virtual execution environments*, pp. 63–74 (2011).

[17] Lenharth, A., Adve, V. S. and King, S. T.: Recovery domains: an organizing principle for recoverable operating systems, *SIGPLAN Not.*, Vol. 44, No. 3, pp. 49–60 (2009).

[18] Liao, J. and Ishikawa, Y.: A New Concurrent Checkpoint Mechanism for Real-Time and Interactive Processes, *Proceedings of the 2010 IEEE 34th Annual Computer Software and Applications Conference*, pp. 47–52 (2010).

[19] Liao, J., Shimosawa, T. and Ishikawa, Y.: Configurable Reliability in Multicore Operating Systems, *Proceedings of the 2011 14th IEEE International Conference on Computational Science and Engineering*, pp. 256–262 (2011).

[20] Nomura, Y., Senzaki, R., Nakahara, D., Ushio, H., Kataoka, T. and Taniguchi, H.: Mint: Booting Multiple Linux Kernels on a Multicore Processor, *Proceedings of the 2011 International Conference on Broadband and Wireless Computing, Communication and Applications*, pp. 555–560 (2011).

[21] Palix, N., Thomas, G., Saha, S., Calvès, C., Lawall, J. and Muller, G.: Faults in linux: ten years later, *Proceedings of the 16th international conference on Architectural support for programming languages and operating systems*, pp. 305–318 (2011).

[22] Shimosawa, T., Matsuba, H. and Ishikawa, Y.: Logical Partitioning without Architectural Supports, *Proceedings of the 2008 32nd Annual IEEE International Computer Software and Applications*, pp. 355–364 (2008).

[23] Swift, M. M., Annamalai, M., Bershad, B. N. and Levy, H. M.: Recovering device drivers, *ACM Trans. Comput. Syst.*, Vol. 24, No. 4, pp. 333–360 (2006).

[24] Yamakita, K., Yamada, H. and Kono, K.: Phase-based reboot: Reusing operating system execution phases for cheap reboot-based recovery, *Proceedings of the 2011 IEEE/IFIP 41st International Conference on Dependable Systems&Networks*, pp. 169–180 (2011).

[25] Yoshimura, T., Yamada, H. and Kono, K.: Is Linux kernel oops useful or not?, *Proceedings of the 8th USENIX conference on Hot Topics in System Dependability* (2012).