Xfas: Fault Recovery by Externally Controlling OS Behavior

Kento Kimura
Kyushu Institute of Technology
Iizuka, Fukuoka, Japan
kento027@ksl.ci.kyutech.ac.jp

Kenichi Kourai
Kyushu Institute of Technology
Iizuka, Fukuoka, Japan
kourai@csn.kyutech.ac.jp

ABSTRACT

It is difficult to completely avoid system failures in recent large-scale and complex clouds. Therefore, it is important to detect system faults as symptoms of failures and then recover from them. Admins often attempt fault recovery by remotely logging in to the target system. They can also run recovery systems inside the target system in advance. However, both methods are subject to system faults in the target system. In this paper, we propose *Xfas* for recovering from system faults by indirectly controlling OS behavior from the outside of the target system. Xfas attempts fault recovery by rewriting OS data in the memory of the target system and leveraging the capabilities of the OS itself. For example, it mimics signal sending to force termination of abnormal processes and unlocking to recover from some kind of deadlock. As two instances of Xfas, this paper presents *VMMfas* and *GPUfas*. We confirmed the effectiveness and efficiency of fault recovery by Xfas.

CCS CONCEPTS

• **Computer systems organization → Dependable and fault-tolerant systems and networks**.

KEYWORDS

fault recovery, VMM, GPU, signal, process scheduling, deadlock

ACM Reference Format:
Kento Kimura and Kenichi Kourai. 2023. Xfas: Fault Recovery by Externally Controlling OS Behavior. In *2023 IEEE/ACM 16th International Conference on Utility and Cloud Computing (UCC '23), December 4–7, 2023, Taormina (Messina), Italy*. ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3603166.3632134

1 INTRODUCTION

Recently, clouds are getting larger and more complex. As a consequence, it becomes difficult to avoid system failures completely, although system developers carefully consider software quality, system performance, and capacity. The root causes of system failures are software faults, poor performance, insufficient capacity, configuration and operation errors, etc. For example, AWS caused a failure due to too many threads that exceeded the limits of the operating system (OS) and affected thousands of online services [1]. Even if perfect systems can be constructed, simple configuration errors can cause failures. In the case of Tokyo Stock Exchange, a failure happened because the developer accidentally disabled the function of automatic switchover to the secondary network storage [20]. Once services are disrupted by a system failure, financial loss is large for both the users and providers of services.

Therefore, it is important to detect system faults as symptoms of failures and then recover from them. When admins detect a system fault, they often attempt recovery from the fault by remotely logging in to the target system. However, remote access is subject to system faults in the target system. For example, it can be prevented by faults in the network stack. Its performance can be affected by available system resources. Without using remote access, fault recovery can be made by running recovery systems inside the target system in advance. A recovery system can be run as an OS process or be embedded into the OS kernel. Unfortunately, it is also affected by system faults like remote access. If these methods cannot recover the system, a hardware reset is a last resort, but it is at high risk of losing the data and states of the system.

In this paper, we propose *Xfas* for attempting fault recovery by indirectly changing OS behavior from the outside of the target system. For more reliable fault recovery, Xfas uses the virtual machine monitor (VMM) to recover the system running in a virtual machine (VM) and a GPU to recover the system in a physical machine. It rewrites OS data in the memory of the target system from the external recovery system using a technique extending VM introspection (VMI) [9]. Then, it eliminates the root cause of the fault using the capabilities of the OS itself. As examples of fault recovery, Xfas provides *pseudo signal sending* and *pseudo process scheduling*. These mimic sending and delivering signals to processes to force termination of the processes that cause a system fault. Xfas also provides *pseudo unlocking*, which mimics releasing a spinlock to address some kind of deadlock.

This paper presents two instances of Xfas: *VMMfas* and *GPUfas*. VMMfas enables the recovery system running outside a VM to manipulate the memory of the VM via the VMM. GPUfas maps the entire main memory onto the GPU memory address space using mapped memory in CUDA [14] and extended memory management in Linux to run the recovery system on a GPU. Xfas transforms the program of the recovery system to rewrite OS data transparently. To complement these mechanisms, Xfas provides a mechanism for in-kernel recovery support, which can be invoked to help fault recovery. We have implemented VMMfas and GPUfas and conducted several experiments to show the effectiveness and efficiency of Xfas. As a result, we confirmed that Xfas could recover from out-of-memory by terminating a process and a deadlock by releasing a spinlock.

This paper is an extension of our workshop paper [11]. In the previous paper, we presented the basic concept of only GPUfas and preliminary results. In this paper, we significantly extend our

UCC '23, December 4–7, 2023, Taormina (Messina), Italy

ACM ISBN 979-8-4007-0234-1/23/12…$15.00
https://doi.org/10.1145/3603166.3632134

previous paper as follows. First, we applied the concept of GPUfas to VMs and developed VMMfas. Second, we clarified the differences in usable techniques between GPUfas and VMMfas. Third, we conducted thorough experiments in both GPUfas and VMMfas and compared the results between them.

The organization of this paper is as follows. Section 2 describes current techniques used for fault recovery. Section 3 proposes Xfas, and Section 4 presents its implementation. Section 5 shows experimental results using Xfas. Section 6 describes related work, and Section 7 concludes this paper.

2 FAULT RECOVERY

When a system fault occurs, admins often recover from it by remotely logging in to the target system, e.g., using SSH and VNC. They can also use a serial console, a remote console in IPMI [10], and a KVM switch as other means. If a VM suffers from a system fault, admins can log in to the VM from not only a remote host but also the host running it using the virtual network and para-virtual interfaces such as virtio. One advantage of such *remote access* is that admins can inspect the root cause of the fault and select the best way of recovery. Instead of admins, remotely running recovery systems can automatically perform fault recovery. However, the biggest disadvantage is that remote access can be largely affected by system faults. The network function in the target system can be corrupted. Remote access servers can stop working. If the target system falls into out-of-memory, it could take long to perform remote access due to thrashing.

To enable fault recovery without remote access, admins can run a recovery system inside a target system in advance. This method is more reliable in that the recovery system does not rely on the mechanisms for remote access. As an example, a recovery system can be run as a process inside a target system [2]. It periodically checks the system states and performs fault recovery if it detects a system fault. Using a process makes it easy to apply new recovery techniques because admins can install a new recovery system easily. However, usable recovery techniques are restricted to the process-level ones using system calls. In addition, the recovery system inside the target system still tends to be affected by system faults.

To address these issues, in-kernel recovery systems have been proposed [13, 23]. This method embeds a recovery system into the OS kernel and runs it periodically using timer interrupts. Compared with process-level recovery systems, in-kernel recovery systems can implement various recovery techniques. However, it is not realistic to modify the OS kernel whenever a new recovery technique is needed. Theoretically, this issue is addressed if a recovery system can be dynamically loaded as a kernel module. Unfortunately, kernel modules are less powerful because they are often restricted in terms of available kernel variables and functions and cannot use all the kernel capabilities. This limits implementable recovery techniques. In addition, in-kernel recovery systems can tolerate system faults more than process-level ones, but they still suffer from various system faults. For example, the recovery system is not executed if timer interrupts are not handled correctly by system faults.

If these methods cannot perform fault recovery, admins have to reboot a target system using a hardware reset. For example, IPMI [10] and remote power management systems can be used to perform hardware resets from remote hosts. Hardware watchdog timers can automatically reset the system if the system does not respond for a certain period. In the case of a VM, admins can easily reset a VM by executing a command in the host running it. A hardware reset can usually recover from a system fault because most of the system faults are rare. For example, a deadlock often depends on timing. Out-of-memory is not caused for a long time after a system reboot.

A hardware reset is a powerful recovery method, but it is at high risk of corrupting system states and data. If system states are lost, it becomes difficult to identify the root cause of a system fault. This means that the same system fault occurs again and again. In addition, application data is lost if it exists only in memory. File data only in the buffer cache is also lost if it is not written back to disks. A hardware reset could corrupt filesystems and lead to losing files. Various mechanisms have been proposed to prevent this situation [5–7], it is difficult to salvage all the data.

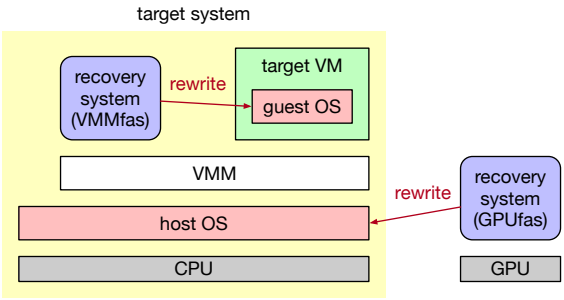


Figure 1: The system architecture of Xfas.

3 XFAS

Xfas attempts fault recovery by indirectly controlling OS behavior from the outside of the target system. Figure 1 illustrates the system architecture of Xfas. The recovery system runs outside the target system, which can run in a VM or a physical machine. When a system fault occurs in the target system, the external recovery system detects it and identifies its root cause using our previous work [16], which monitors resource usage in the OS by a VMI-like technique. Then, it rewrites OS data in the memory of the target system using a technique extending VMI and recovers the target system by leveraging OS capabilities. Thanks to this direct memory rewrite, developers can implement various recovery techniques without the limitations of processes or kernel modules. Since the recovery system in Xfas runs as an application, it is easy to apply new recovery techniques.

This paper presents two instances of Xfas: *VMMfas* and *GPUfas*. VMMfas recovers the target system in a VM from a system fault via the VMM. Since the VMM is logically isolated from a VM, the external recovery system is not affected by a system fault in a VM. Performance isolation, e.g., CPU limits, provided by the VMM can prevent the external recovery system from being affected by excessive resource consumption in the VM. The recovery system can run using resources such as CPUs and memory preserved for the host system. Even if the virtual network of the target VM fails,

the external recovery system can communicate with a remote host using the host network to rely on remote admins for recovery decisions. We assume that cloud providers provide fault recovery with VMMfas as a service or allow cloud users to run their own recovery systems for VMMfas.

On the other hand, GPUfas recovers the target system running in a physical machine from a system fault using a GPU. Although a GPU is one hardware component, the recovery system on a GPU is not easily affected by system faults. A GPU is physically isolated from CPUs and main memory, on which the target system runs. For example, CPUs cannot directly corrupt GPU memory unless they initiate DMA accidentally. In addition, the cores and memory dedicated to a GPU can prevent the external recovery system from being affected by the resource shortage of the target system. The other programmable PCIe devices such as DPUs and FPGAs could be used for Xfas, but GPUs have an advantage in cost because GPUfas can use inexpensive, low-end GPUs. The recovery system on a GPU can communicate with a remote host using GPUDirect RDMA [15] without the help of the OS. GPUfas could be used to recover from system faults in the VMM. It is also useful for the systems running containers, although several types of system faults could be recovered outside containers without a GPU.

The target scope of Xfas is recovery from system faults that do not destroy the integrity of the target system, e.g., excessive resource consumption. To recover from such a system fault, it can perform *pseudo signal sending* to abnormal processes that cause a system fault. For example, it can prevent thrashing due to out-of-memory by sending the KILL signal and terminating processes that consume a large amount of memory. If it pauses processes that consume too much CPU time by sending the STOP signal, it can reduce the CPU load and recover system performance. Xfas changes the process state to the same one as after a signal is sent by rewriting information on pending signals and mimics signal sending to a process. In addition, Xfas provides *pseudo process scheduling*. This mechanism is used for controlling process execution and is indispensable to make pseudo signal sending effective to processes with various states. Xfas changes the state of the process scheduler by rewriting scheduling data and mimics the adjustment of process scheduling.

Xfas can also perform *pseudo unlocking* to recover from some kind of deadlock. A deadlock by missing lock release is a typical and frequent bug in the OS kernel. In particular, a CPU is stuck if a spinlock is not released because it uses busy waiting. Xfas can release such a lock and enable waiting kernel threads to proceed. It changes the lock state by rewriting a lock variable and mimics lock release. This recovery does not lead to data inconsistency because Xfas just releases a lock that should be released by the kernel. Note that pseudo unlocking cannot recover from all types of deadlocks. In general, a consistency problem is caused by releasing one of the locks involved in a deadlock.

However, all the functions are not implementable by rewriting OS data. For example, the recovery system on a GPU cannot acquire a lock used in the kernel because lock acquisition requires an atomic instruction in CPUs to change the value of a lock variable. In addition, there are OS functions that are too complex to re-implement outside the target system. Even if some of the OS functions are implementable, they could largely degrade the recovery performance due to too much access to OS data in the target system.

To address these issues, Xfas can cooperate with *in-kernel recovery support* inside the target system. The external recovery system communicates with the mechanism embedded into the target OS kernel and executes necessary functions inside the kernel. In principle, the mechanism in the kernel can do anything including hardware access. However, such in-kernel recovery support is subject to system faults. It is necessary to consider a trade-off between fault tolerance, ease of implementation, and performance. It should be noted that the combination of the external recovery system and in-kernel recovery support is more reliable than pure in-kernel recovery systems. Only the recovery functions running in the kernel can be affected by system faults.

Since Xfas is a first-aid system, it might achieve only temporal fault recovery. For example, the target system might not provide services correctly after Xfas terminates abnormal processes. In this case, admins can save data and then reboot the system if they can access the target system remotely, thanks to temporal fault recovery. This can prevent important data from being lost. If even temporal fault recovery is impossible, e.g., on a fault of in-kernel recovery support, Xfas can send memory data to a remote host without relying on the target OS. Admins can analyze the memory data at the remote host and restore the data after a hardware reset.

4 IMPLEMENTATION

We have implemented Xfas using the Linux kernel 4.18.0 and LLVM 8.0 [19]. We used CUDA 10.0 [14] for GPUfas and QEMU-KVM 2.11.0 [3] for VMMfas. GPUfas could use any OSes and GPU libraries that support for mapping main memory onto a GPU memory address space, although we need to overcome several implementation issues. VMMfas can easily be applied to other virtualized systems.

4.1 Pseudo Signal Sending

When a process such as the kill command uses the signal mechanism provided by the OS, it sends signals to a target process using the kill system call. As another method, the kernel can send signals directly using its function. According to the received signal, the target process executes a pre-registered function, performs a default action, or ignores the signal. Note that the default actions are always performed for the KILL and STOP signals. When a signal is sent to a process, the kernel records the signal number in the target process and sets the flag for pending signals. When the target process is scheduled, the kernel checks this flag. If there is a pending signal, it delivers the signal to the process. However, the recovery system running outside the target system cannot directly invoke the system call or the kernel function.

To mimic this signal sending outside the target system, Xfas directly rewrites the data structure used for the signal mechanism in the target kernel, as illustrated in Fig. 2. First, Xfas searches for the task_struct structure used for the target process in the kernel memory. Then, it finds the signal bitmap (sigset_t) in the sigpending structure, which is included in task_struct. Next, it finds the thread_info structure in task_struct and sets the TIF_SIGPENDING flag. Later, the kernel checks this flag when it schedules that process

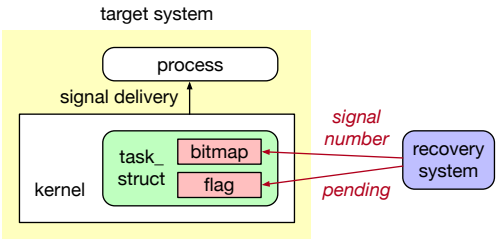


Figure 2: Pseudo signal sending.

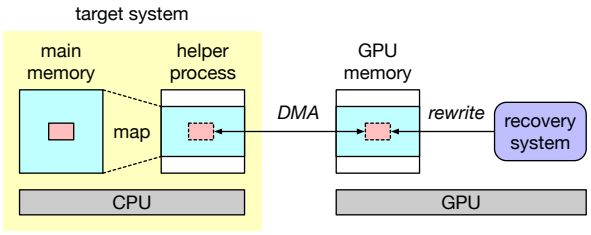


Figure 3: Rewriting main memory from a GPU.

and switches the CPU mode from the kernel to user mode. If the flag is set, the kernel handles the injected signal.

4.2 Pseudo Process Scheduling

After the kill system call or the kernel function sets the flag for pending signals, the kernel schedules the target process so that the sent signal is delivered to the process as early as possible. In particular, this is mandatory if the target process is paused in the sleeping state. Signals are delivered only when a process is scheduled. To schedule a process, the kernel first adds the process to the run queue of the process scheduler. Then, it changes the process to the runnable state. When the process scheduler selects that process and changes it to the running state, pending signals are delivered to the process. However, the external recovery system cannot directly schedule the target process.

To mimic this process scheduling outside the target system, Xfas indirectly rewrites the data structure used for the process scheduler in the kernel. This is done by emulating the kernel functions of the process scheduler because scheduling data is too complex. Currently, Xfas supports CFS, which is the most popular process scheduler in Linux. First, it adds the sched_entity structure in task_struct to the red-black tree in the cfs_rq structure. At this time, it searches for the most appropriate position using the virtual run time recorded in sched_entity. Then, it changes the process state to TASK_RUNNING. Later, the process scheduler schedules processes in descending order of their virtual run time.

Before manipulating the red-black tree, Xfas acquires the spinlock for a per-CPU run queue using pseudo locking or in-kernel locking support. It releases the spinlock after the operation using pseudo unlocking. During the operation, it acquires and releases several necessary spinlocks. To reduce the number of locking and unlocking, Xfas adds all the processes assigned to one CPU to the red-black tree at once when it sends signals to multiple processes at the same time. Using this optimization, it can acquire and release the spinlock for a per-CPU run queue as many times as the number of CPUs.

4.3 Pseudo Locking and Unlocking

Locking is used to achieve mutual exclusion in many kernel functions. In particular, this paper focuses on a spinlock, which is used when a kernel thread is unlikely to wait for a long period. A thread waits for lock acquisition using busy waiting if another thread acquires the spinlock. After the thread acquires the spinlock, it executes mutually excluded code and then releases the spinlock. If it misses this lock release, other threads have to wait for the spinlock forever. Even the thread itself has to wait if it attempts to acquire the spinlock again. A spinlock cannot also be acquired or released outside the target system.

To mimic the acquisition and release of a spinlock, Xfas directly rewrites the data structure used for a spinlock in the kernel. It first obtains the qspinlock structure in the raw_spinlock structure, which is included, e.g., in the rq structure and the task_struct structure. For pseudo locking, Xfas changes the value of the lock variable to one and can acquire the spinlock if the old value is zero. If the value is not zero, Xfas repeatedly attempts this spinlock acquisition. Since this operation needs to be atomic, only VMMfas can support pseudo locking by using the atomic instruction. Since a GPU cannot use such an instruction of a CPU, GPUfas supports spinlock acquisition using in-kernel locking support. For pseudo unlocking, Xfas changes the value of the lock variable in it to zero. Since this operation does not need the atomic instruction, GPUfas can also support spinlock release using pseudo unlocking.

4.4 Manipulation of OS Data

Xfas enables developers to write the program of a recovery system using the source code of the Linux kernel. Specifically, they can use kernel data structures, global variables, inline functions, and macros by including kernel header files. Xfas transforms the written program using LLVM so that a recovery system running outside the target system accesses the memory of the target system. Xfas first compiles the program to LLVM intermediate representation. Then, it searches for the load and store instructions, which are used to read and write data from and to memory, respectively. For each instruction, it inserts the invocation to the code for accessing the memory of the target system just before those instructions. We have implemented this framework by extending our previous work [16].

4.5 Memory Rewrite from a GPU

To rewrite main memory from a GPU, GPUfas uses CUDA's mapped memory, which is a mechanism for mapping main memory onto the GPU memory address space. A recovery system on a GPU can access main memory after a system fault through this advance memory mapping. Since CUDA provides only a function for mapping process memory, GPUfas first maps the entire main memory onto the address space of a helper process in the writable mode, as illustrated in Fig. 3. However, this runs out of free memory because the entire memory becomes in use. To avoid this issue, GPUfas extends memory management in the Linux kernel and provides a special device file. When a helper process maps this file, the kernel

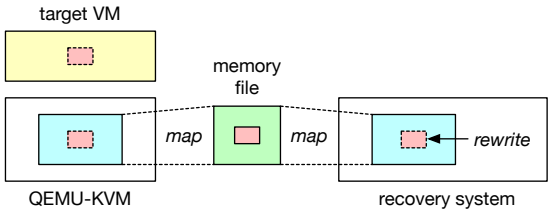


Figure 4: Rewriting the memory of a VM via the VMM.

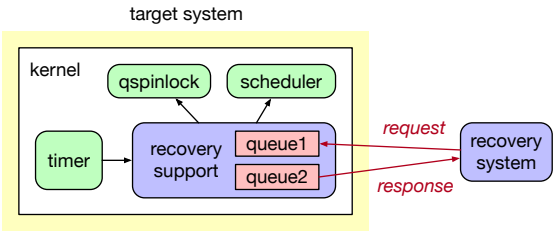


Figure 5: In-kernel recovery support.

does not increase the reference count of each page so that the page does not become in use. For the implementation details, see our previous paper [16].

When the recovery system on a GPU rewrites OS data in main memory, GPUfas translates its virtual address into a physical one using the page tables in the target system. Then, it translates that address into a GPU one. When the recovery system accesses the translated GPU address, the GPU automatically transfers only accessed data from main memory to GPU memory using DMA. Since it temporarily keeps transferred data, that data does not put pressure on GPU memory. After the recovery system modifies the data, the GPU automatically transfers the modified data back to main memory using DMA. As a result, the kernel in the target system can access the modified data in main memory and change its behavior.

It is not secure if intruders could rewrite the entire main memory using the mechanism provided by GPUfas. Therefore, GPUfas provides two access restrictions. First, the kernel prevents the helper process itself from accessing the mapped main memory by modifying the page tables of that process. Even if attackers compromise that process, they cannot read or rewrite main memory through that process. Second, the kernel permits only privileged processes to map main memory. Since a recovery system occupies a GPU in GPUfas, attackers need to terminate the recovery system before launching their malicious GPU programs. At this time, the helper process is also terminated. As a result, GPU programs launched by attackers cannot map main memory via new helper processes unless they can gain administrative privileges. Note that attackers can access main memory by installing a kernel module if they can take administrative privileges. If attackers hijack a recovery system running on a GPU, they could rewrite already mapped main memory, but this is not easy.

4.6 Memory Rewrite via the VMM

To rewrite the memory of a VM via the VMM, VMMfas creates a memory file for a VM on the host OS and maps it onto the address space of the QEMU-KVM process, as illustrated in Fig. 4. QEMU-KVM assigns the mapped file to a VM, instead of allocating memory by malloc. To read and rewrite the memory of the VM, a recovery system also maps the memory file onto its process address space in the writable mode. When a recovery system rewrites OS data in the memory of a VM, VMMfas first translates the virtual address of the OS data into a physical one in the VM. For this address translation, VMMfas obtains the address of the page tables from the CR3 register of the VM by communicating with QEMU-KVM and then traverses the page tables of the guest OS. Next, it translates the physical address in the VM into the virtual address of the recovery system. Finally, it writes modified data to the memory address. We have implemented this mechanism by extending our previous work [12].

4.7 In-kernel Recovery Support

To invoke in-kernel recovery support, a recovery system writes a request to a queue allocated in the memory of the target system, as illustrated in Fig. 5. In-kernel recovery support periodically reads the queue using timer interrupts. For simple support such as locking, it runs in the interrupt handler for the local APIC timer. For complex support such as process scheduling, it runs in the callback function registered to the Linux timer. This is because the low-level interrupt handler is more tolerant of system faults but should not run for a long time. Then, in-kernel recovery support executes the function corresponding to the request and writes a response to the other queue. The recovery system periodically reads that queue by polling and continues its execution if in-kernel recovery support succeeds.

Currently, Xfas provides locking and process scheduling as in-kernel recovery support. Pseudo process scheduling has been achieved outside the target system, but we have also implemented process scheduling as in-kernel recovery support for comparison.

5 EXPERIMENTS

We conducted several experiments to show the effectiveness of Xfas, specifically GPUfas and VMMfas. We used four combinations of recovery techniques, as depicted in Table 1. PSIG used only pseudo signal sending. PSCH+PLK used pseudo process scheduling and pseudo locking and unlocking in addition to pseudo signal sending. This is applicable only in VMMfas because GPUfas cannot perform pseudo locking. PSCH+KLK was similar to PSCH+PLK but used in-kernel recovery support for acquiring a spinlock, instead of pseudo locking. KSCH used pseudo signal sending and in-kernel recovery support for process scheduling.

We used a PC with an Intel Core i7-9700 processor, 16 GB of memory, a 2-TB HDD, and NVIDIA GeForce GTX 960. We ran Linux 4.18.0 and assigned 7 GB of swap space. For VMMfas, we created a VM with 3 virtual CPUs, 2 GB of memory, and 4 GB of swap space on top of QEMU-KVM 2.11.2. We ran Linux 4.18.0 as the guest OS.

5.1 Effectiveness of Pseudo Signal Sending

We performed pseudo signal sending to a running process for the KILL, TERM, STOP, and CONT signals and examined the behavior of the process. The KILL signal forced termination of the process, whereas the TERM signal normally terminated the process. The

Table 1: The used combination of recovery techniques.

<table>
<tr><th>method</th><th colspan="3">recovery techniques</th></tr>
<tr><td>PSIG</td><td rowspan="4">pseudo signal sending</td><td colspan="2">–</td></tr>
<tr><td rowspan="2">PSCH+PLK</td><td rowspan="2">pseudo scheduling & unlocking</td><td>pseudo locking</td></tr>
<tr><td rowspan="2">PSCH+KLK</td><td rowspan="2">in-kernel locking</td></tr>
<tr><td>KSCH</td><td colspan="2">in-kernel scheduling</td></tr>
</table>

Table 2: The results of pseudo signal sending in GPUfas.

signal	PSIG	PSCH+PLK	PSCH+KLK	KSCH
KILL	✓	n/a	✓	✓
TERM	✓	n/a	✓	✓
STOP	✓	n/a	✓	
CONT		n/a	✓	✓

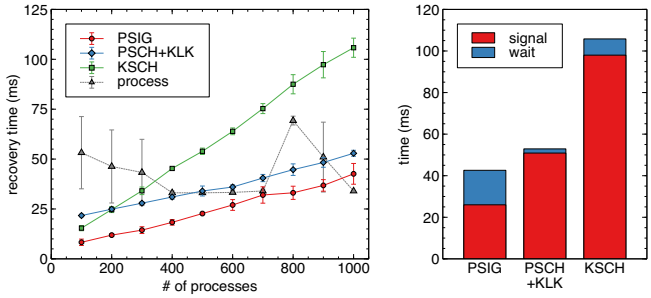
Table 3: The results of pseudo signal sending in VMMfas.

signal	PSIG	PSCH+PLK	PSCH+KLK	KSCH
KILL	✓	✓	✓	✓
TERM	✓	✓	✓	✓
STOP	✓	✓	✓	✓
CONT		✓	✓	✓

STOP signal paused the process, whereas the CONT signal continued the paused process. For the CONT signal, we first sent the STOP signal using the kill command to pause the process.

Table 2 shows the results of pseudo signal sending in GPUfas. PSIG could not continue the paused process because it did not perform process scheduling to wake up that process. In contrast, PSCH+KLK could send all the signals successfully. Surprisingly, KSCH could not pause the process. After GPUfas sent the STOP signal by pseudo signal sending, the kernel paused the process because it periodically scheduled the running process. After that, GPUfas changed the process state to runnable again by in-kernel scheduling support. As a result, the paused process was continued. This is due to the time lag between pseudo signal sending and in-kernel scheduling support. We need to fix in-kernel scheduling support so that the process is not re-scheduled in such a case.

Table 3 shows the results of pseudo signal sending in VMMfas. In addition to PSCH+KLK, PSCH+PLK could send all the signals successfully. This means that VMMfas can completely achieve signal sending without any help of in-kernel recovery support, thanks to enabling pseudo locking. Unlike in GPUfas, KSCH could pause the process as expected. This is because VMMfas was faster than GPUfas and the time lag between pseudo signal sending and in-kernel scheduling support decreased. VMMfas could change the process state to runnable by in-kernel scheduling support before the kernel re-scheduled the running process.



(a) Recovery time **(b) 1000 processes**

Figure 6: The average time to terminate running processes in GPUfas.

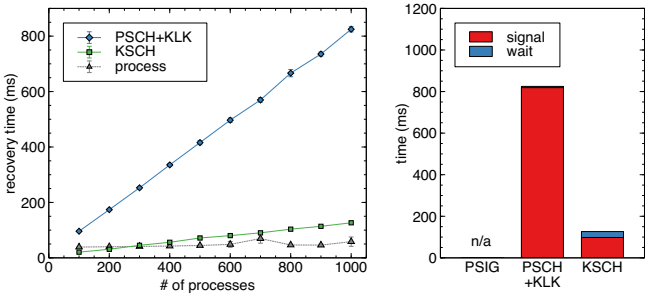
5.2 Signal Performance in GPUfas

We examined the performance of pseudo signal sending in GPUfas. We sent the KILL signals to various numbers of processes and measured the time until all the processes were terminated.

First, we sent the signals to running processes that performed busy waiting. The average recovery time of 10 attempts is shown in Fig. 6(a). For GPUfas, it was proportional to the number of processes, and the standard deviation was relatively small. PSIG could terminate all the processes successfully without process scheduling and achieve the fastest recovery. PSCH+KLK performed pseudo process scheduling as well, but the recovery time increased only by 8.5-13.5 ms because all the processes were runnable and were not actually re-scheduled by pseudo process scheduling. In contrast, KSCH significantly increased the recovery time due to the overhead of invoking in-kernel scheduling support from a GPU. For comparison, we sent the signals using the process-level recovery system, which issued the kill system call inside a target system. The average recovery time was not proportional to the number of processes, and the standard deviation was large. GPUfas was often faster than the process-level recovery system.

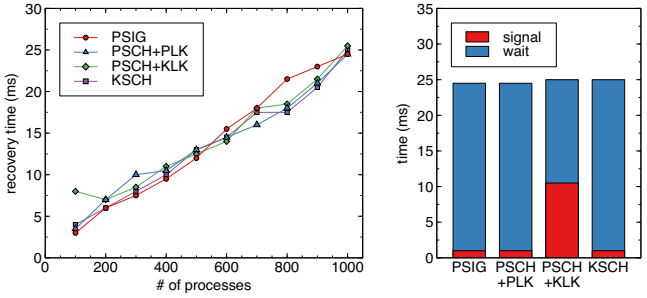
Figure 6(b) shows the breakdown of the recovery time for terminating 1000 processes. The recovery time consists of the time to send the KILL signals to all the processes from a GPU (signal time) and the time to wait for the termination of all the processes in a GPU (wait time). When we used PSIG, the ratio of the signal time was relatively small because pseudo signal sending is lightweight. When we sent signals to fewer processes, the ratio was smaller. This is because fewer processes were terminated before GPUfas completed sending signals. In PSCH+KLK and KSCH, in contrast, the signal time always occupied a larger portion.

Next, we sent the signals to processes paused by a long sleep. Figure 7(a) shows the average recovery time with a very small standard deviation. PSIG could not terminate any processes because the paused processes could not handle sent signals without process scheduling. Unlike the results for running processes, PSCH+KLK took much longer than KSCH. This is because many invocations of in-kernel locking support suffered from large overhead during pseudo process scheduling. As shown in Fig. 7(b), PSCH+KLK spent most of the time performing pseudo process scheduling, unlike



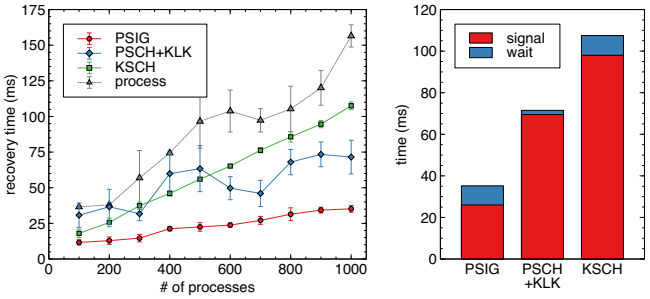
(a) Recovery time (b) 1000 processes

Figure 7: The average time to terminate paused processes in GPUfas.



(a) Recovery time (b) 1000 processes

Figure 9: The median time to terminate running processes in VMMfas.



(a) Recovery time (b) 1000 processes

Figure 8: The average time to terminate intermittent processes in GPUfas.

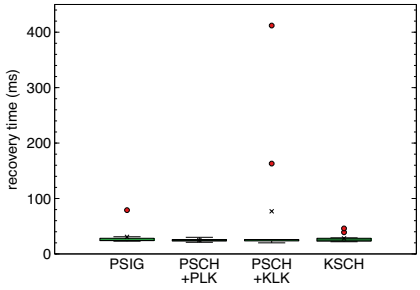


Figure 10: The distribution of the recovery time for 1000 running processes in VMMfas.

KSCH. In KSCH, in-kernel scheduling support also suffered from invocation overhead, but it was invoked only once per process. If the number of processes was less than 300, KSCH was faster than the process-level recovery system. Note that KSCH is less reliable due to running complex scheduling code in the target kernel.

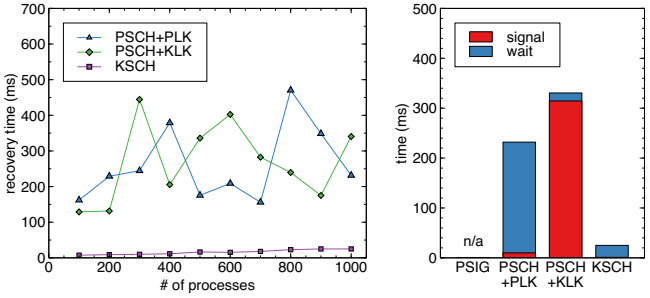
Finally, we sent the signals to intermittent processes that ran every 10 ms. Figure 8(a) shows the average recovery time. For PSIG and KSCH, the results were almost the same as those for running processes in Fig. 6(a). However, the recovery time in PSCH+KLK varied largely, and the standard deviation was quite large. This is because an intermittent process has the characteristics of both running and paused processes. If the target process was running during pseudo process scheduling, it was not re-scheduled, so that the overhead of pseudo process scheduling was small. Otherwise, the process was re-scheduled with a large overhead. Nevertheless, the breakdown of the recovery time was similar to that for running processes, as shown in Fig. 8(b). Surprisingly, GPUfas was always faster than the process-level recovery system in this case.

5.3 Signal Performance in VMMfas

In VMMfas, we also examined the performance of pseudo signal sending. First, we sent the KILL signals to various numbers of processes running in a VM via the VMM. Figure 9(a) shows the median recovery time because the average was not stable enough. The recovery time was proportional to the number of processes. Unlike GPUfas, all the methods including PSCH+PLK could terminate processes in almost the same time. To be exact, PSCH+PLK, PSCH+KLK, and KSCH were 4.3%, 19%, and 1.7% slower than PSIG on average, respectively. Compared with GPUfas, VMMfas was 1.5-2.7x faster for PSIG, thanks to a CPU core much faster than a GPU core. For PSCH+KLK, it was 2-3.6x faster because more complex process scheduling could be executed in a faster CPU core. For KSCH, it was 4-4.9x faster due to much less overhead of invoking in-kernel scheduling support for a VM.

Figure 9(b) shows the breakdown of the recovery time. Unlike GPUfas, PSIG, PSCH+PLK, and KSCH spent most of the time waiting for the termination of processes. In contrast, PSCH+KLK needed a longer signal time because it invoked in-kernel locking support many times. This means that the overhead of communicating with in-kernel recovery support is still large even in VMMfas.

As shown in Fig. 10, the recovery time was basically stable, but there were several extreme outliers. It sometimes took much long time to terminate processes, particularly when the number of processes increased. One of the reasons is that the invocation time of in-kernel recovery support was largely affected by virtualization.



(a) Recovery time (b) 1000 processes

Figure 11: The median time to terminate paused processes in VMMfas.

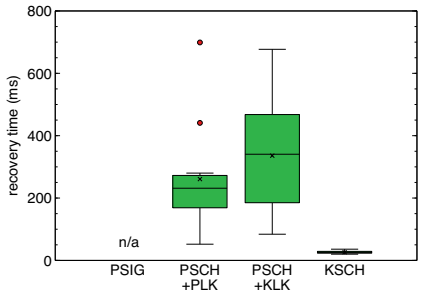
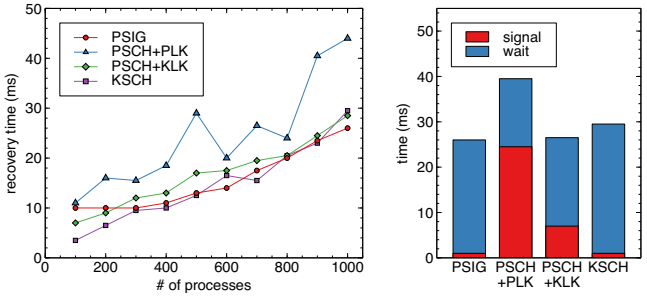


Figure 12: The distribution of the recovery time for 1000 paused processes in VMMfas.



(a) Recovery time (b) 1000 processes

Figure 13: The median time to terminate intermittent processes in VMMfas.

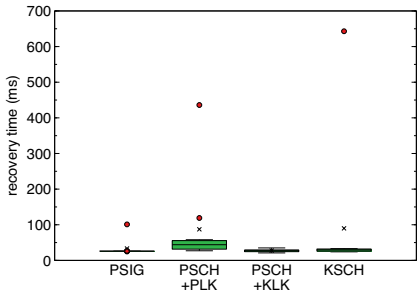


Figure 14: The distribution of the recovery time for 1000 intermittent processes in VMMfas.

Since PSCH+PLK was the most stable for any number of processes, it is the best for running processes.

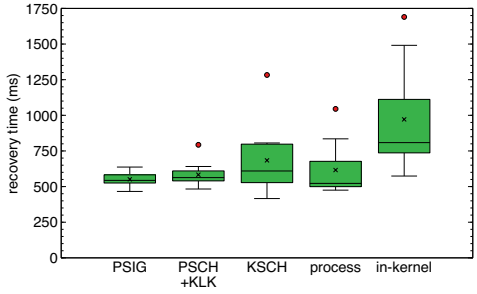
Next, we sent the signals to processes paused in a VM. Figure 11(a) shows the median recovery time, and Fig. 12 shows the distribution of the recovery time. Like GPUfas, KSCH was the fastest and the most stable. In contrast, the recovery time in PSCH+PLK and PSCH+KLK was not proportional to the number of processes, and the variance was very large. This is probably due to virtualization, e.g., VM scheduling, but the details are under investigation. For 1000 processes, the recovery time was 2.4x and 4.8x faster than GPUfas in PSCH+KLK and KSCH, respectively. For PSCH+KLK, the ratio of the signal time was very large due to the frequent invocation of in-kernel locking support, as shown in Fig. 11(b).

Finally, we sent the signals to processes that intermittently ran in a VM. Figure 13(a) shows the median recovery time. It was basically proportional to the number of processes. PSIG, PSCH+KLK, and KSCH could terminate processes in almost the same time. To be exact, PSCH+KLK was 8.2% slower than PSIG on average, and KSCH was 11% faster. However, PSCH+PLK was 57% slower than PSIG. This is because PSCH+PLK was largely affected by pseudo process scheduling if the target process was not running. In fact, Fig. 13(b) shows that PSCH+PLK spent a relatively long time executing pseudo process scheduling. This is largely different from the breakdown for running and paused processes. In contrast, PSCH+KLK needed a shorter signal time than that for the other types of processes. This is probably the reason why PSCH+KLK was faster than PSCH+PLK. As shown in Fig. 14, PSCH+KLK had almost no outliers.

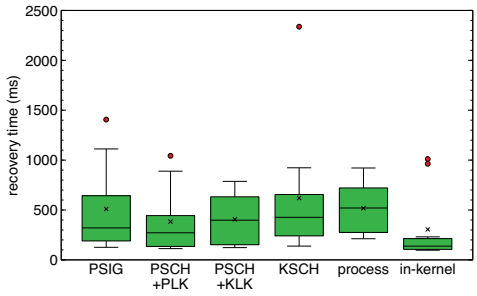
5.4 Recovery from Out-of-memory

To show the recoverability from a real system fault, we made the target system use up physical memory. We ran one process that allocated a large amount of memory and sequentially wrote data to the memory to continuously cause swapping. For GPUfas, the process allocated 19 GB of virtual memory in the physical machine with 16 GB of memory. When we attempted remote login to this host, it took about 20x longer due to thrashing. For VMMfas, the process allocated 5 GB of virtual memory in the VM with 2 GB of memory. When we attempted remote login to this VM, it took about 15x longer.

In this experiment, the recovery system detected a system fault if the amount of memory consumption exceeded 80% and sent the KILL signal to the target process for recovery. In addition to GPUfas and VMMfas, we used a process-level recovery system and an in-kernel recovery system, which are described in Section 2. The in-kernel recovery system invoked the kernel function for signal sending.



(a) GPUfas



(b) VMMfas

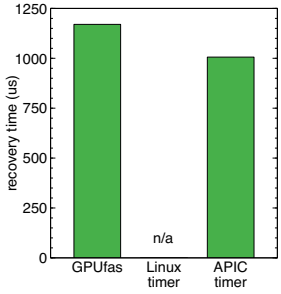
Figure 15: The recovery time from out-of-memory.

Figure 15(a) shows the recovery time from out-of-memory in GPUfas. PSIG was the fastest and the most stable. PSCH+KLK slightly increased the average due to pseudo process scheduling, but it was still stable. In KSCH, in contrast, the average was 100 ms longer than in PSCH+KLK. This is due to the invocation of in-kernel scheduling support from a GPU and the impact of continuous swapping on the process scheduler in the kernel. This impact also made the variance of the recovery time much larger. Contrary to our expectation, the process-level recovery system was not largely affected by frequent swapping, but it resulted in lower stability. Surprisingly, the in-kernel recovery system was worst in both the average and stability. One of the reasons is the large impact of swapping on the in-kernel scheduler like KSCH.

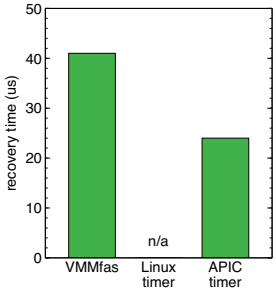
Figure 15(b) shows the recovery time in VMMfas. PSCH+PLK was the fastest among the four methods in VMMfas. Its median recovery time was 2x faster than that of the fastest method in GPUfas. Unlike GPUfas, however, no methods were so stable. The process-level recovery system was the slowest for a VM. In contrast, the in-kernel recovery system was the fastest and the most stable, although there were several outliers. This is completely different from the results in the physical machine. The reason is currently unclear.

5.5 Recovery from a Deadlock

As another real system fault, we made the kernel in the target OS cause a deadlock involving all the CPUs. We loaded the kernel module in which all the threads attempted to acquire the same



(a) GPUfas



(b) VMMfas

Figure 16: The recovery time from a deadlock.

spinlock without disabling interrupts. This module caused a deadlock by failing to release the acquired spinlock. In this experiment, the recovery system released the spinlock using pseudo unlocking after the deadlock occurred. For comparison, we used two in-kernel recovery systems. One released the spinlock in a callback function registered to the Linux timer. The other did in the interrupt handler for the local APIC timer. We did not use a process-level recovery system because the deadlock in the kernel prevented that process from running.

Figure 16 shows the recovery time from the deadlock. We confirmed that both GPUfas and VMMfas could recover from this type of deadlock. The reason why the high-level Linux timer failed to recover is that the kernel thread used by the Linux timer was not scheduled due to no idle CPU. The low-level interrupt handler succeeded in fault recovery because it was invoked regardless of the deadlock. GPUfas was 164 μs slower than the in-kernel recovery system. In contrast, VMMfas was only 17 μs slower and 29x faster than GPUfas. In both cases, Xfas was comparable to the in-kernel recovery system.

6 RELATED WORK

The Linux kernel provides several features for fault recovery such as a kernel oops and the out-of-memory (OOM) killer. A kernel oops terminates the process that causes a system fault when the kernel detects the fault and enables the execution of the system to be continued. However, the kernel state is not always restored to the normal one [22]. The OOM killer forces termination of the process that consumes excessive memory when the system causes out-of-memory. Whereas it does not consider factors except for the consumption of memory and swap space, Xfas can select processes more flexibly on a system fault.

SHFH [23] detects various system hangs and recovers from the faults. It provides three recovery techniques. One is to force termination of the process or thread that causes a system fault. The other two are to send a non-maskable interrupt (NMI) to a stalled CPU and to reboot the system. SHFH detects a system fault using both the process and the kernel and then recovers from the fault in the kernel. Therefore, it is subject to a kernel-level system fault and is less reliable than Xfas, which runs a recovery system outside the target system.

Backdoors [4] performs fault recovery by modifying OS data using RDMA from a remote host. As an example, it mimics sending the KILL signal to a process like Xfas. However, it is necessary to modify the OS so that a remote host can access the process table using RDMA. In addition, Backdoors needs to permit direct access to the kernel memory from a remote host. This can introduce a new attack surface to the target system. In contrast, Xfas is more secure because the recovery system can execute only recovery functions fixed in advance.

VMI [9] is a well-known technique for obtaining the internal state of a VM from the outside of it. It is used mainly for security but is also applied to fault detection [17, 18]. However, there are few systems to modify the internal state. EXTERIOR [8] enables the system in a VM to be recovered when the system is attacked. It prepares a different VM that runs the same OS kernel as the target VM and seamlessly reflects memory updates by the commands executed in this VM to the target VM. For example, it can terminate processes using the kill command and unload kernel modules using the rmmod command. EXTERIOR can be used to run reliable process-level recovery systems for VMs. However, usable recovery techniques are restricted to process-level ones, as described in Section 2. Xfas can use more powerful kernel-level recovery techniques. In addition, it can be applied to not only VMs but also physical machines by using a GPU.

Otherworld [7] microreboots the OS kernel when a kernel-level system fault occurs. Unlike a normal reboot, a microreboot reboots the system without corrupting the states of running applications on top of the kernel. After a microreboot, Otherworld restores the memory of applications, opened files, and the states of the other resources. This mechanism is orthogonal to Xfas. It can be used to minimize the impact of a reboot when Xfas cannot recover from a system fault.

A phase-based reboot [21] can reduce the recovery time from a system fault using a VM. It divides the boot sequence into three phases and saves the system state for each phase. Upon fault recovery, it restores the system state of the most appropriate phase to reduce the reboot time. However, the system state that is not saved before a system fault is lost. Also, only the system running in a VM is recoverable. Xfas also supports the recovery of the system in a physical machine.

7 CONCLUSION

This paper proposed Xfas for enabling fault recovery by running a recovery system outside the target system and indirectly controlling OS behavior. Xfas rewrites OS data in memory and attempts to recover from a system fault by leveraging OS capabilities. Currently, Xfas provides recovery techniques called pseudo signal sending, pseudo process scheduling, and pseudo locking and unlocking. It can cooperate with in-kernel recovery support if necessary. We have implemented two instances of Xfas and confirmed that Xfas could recover from several system faults in a short period.

One of our future work is to minimize pseudo process scheduling, although we implemented it by emulating kernel functions. In addition, we need to recover from various types of system faults, e.g., a deadlock due to spinlocks with interrupts disabled. In this case, we could use the NMI instead of timer interrupts. Another direction is to use remote hosts with GPUDirect RDMA for advanced fault recovery.

ACKNOWLEDGMENTS

This work was partially supported by JST, CREST Grant Number JPMJCR21M4, Japan. These research results were partially obtained from the commissioned research (No.05501) by National Institute of Information and Communications Technology (NICT), Japan.

REFERENCES

[1] Amazon Web Services, Inc. 2020. Summary of the Amazon Kinesis Event in the Northern Virginia (US-EAST-1) Region. https://aws.amazon.com/message/11201/.
[2] A. Beekhof. [n. d.]. Pacemaker. https://clusterlabs.org/pacemaker/.
[3] F. Bellard. [n. d.]. QEMU. https://www.qemu.org/.
[4] A. Bohra, I. Neamtiu, P. Gallard, F. Sultan, and L. Iftode. 2004. Remote Repair of Operating System State Using Backdoors. In *Proceedings of the 1st International Conference on Autonomic Computing*. 256–263.
[5] P. Chen, W. Ng, S. Chandra, C. Aycock, G. Rajamani, and D. Lowell. 1996. The Rio File Cache: Surviving Operating System Crashes. In *Proceedings of the 7th International Conference on Architectural Support for Programming Languages and Operating Systems*. 74–83.
[6] F. David, J. Carlyle, and H. Campbell. 2007. Exploring Recovery from Operating System Lockups. In *Proceedings of the 2007 USENIX Annual Technical Conference*. 351–356.
[7] A. Depoutovitch and M. Stumm. 2010. Otherworld: Giving Applications a Chance to Survive OS Kernel Crashes. In *Proceedings of the 5th European Conference on Computer Systems*. 181–194.
[8] Y. Fu and Z. Lin. 2013. EXTERIOR: Using a Dual-VM Based External Shell for Guest-OS Introspection, Configuration, and Recovery. In *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments*. 97–110.
[9] T. Garfinkel and M. Rosenblum. 2003. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of Network and Distributed Systems Security Symposium*. 191–206.
[10] Intel, Hewlett-Packard, NEC, and Dell. 2004. Intelligent Platform Management Specification Second Generation v2.0.
[11] K. Kimura and K. Kourai. [n. d.]. GPU-based First Aid for System Faults. In *Proceedings of the 13th ACM SIGOPS Asia-Pacific Workshop on Systems*. 38.
[12] K. Kourai and K. Nakamura. 2014. Efficient VM Introspection in KVM and Performance Comparison with Xen. In *Proceedings of Pacific Rim International Symposium on Dependable Computing*. 192–202.
[13] J. Leners, H. Wu, W. Hung, M. Aguilera, and M. Walfish. 2011. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*. 279–294.
[14] NVIDIA Corporation. 2018. CUDA Toolkit Documentation v10.0.130. https://docs.nvidia.com/cuda/archive/10.0/.
[15] NVIDIA Corporation. 2022. *Developing a Linux Kernel Module Using RDMA for GPUDirect*. Technical Report TB-06712-001 v11.7. NVIDIA.
[16] Y. Ozaki, S. Kanamoto, H. Yamamoto, and K. Kourai. 2019. Detecting System Failures with GPUs and LLVM. In *Proceedings of the 10th ACM SIGOPS Asia-Pacific Workshop on Systems*. 47–53.
[17] D. Pelleg, M. Ben-Yehuda, R. Harper, L. Spainhower, and T. Adeshiyan. 2008. Vigilant: Out-of-band Detection of Failures in Virtual Machines. *SIGOPS Operating Systems Review* 42, 1 (2008), 26–31.
[18] C. Pham, Z. Estrada, P. Cao, Z. Kalbarczyk, and R. Iyer. 2014. Reliability and Security Monitoring of Virtual Machines Using Hardware Architectural Invariants. In *Proceedings of the 44th Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 13–24.
[19] The LLVM Foundation. [n. d.]. The LLVM Compiler Infrastructure. https://llvm.org/.
[20] Tokyo Stock Exchange, Inc. 2020. Report on the Cash Equity Trading System Failure on Oct. 1. https://www.jpx.co.jp/english/corporate/news/news-releases/0060/20201019-01.html.
[21] K. Yamakita, H. Yamada, and K. Kono. 2011. Phase-based Reboot: Reusing Operating System Execution Phases for Cheap Reboot-based Recovery. In *Proceedings of the 41st IEEE/IFIP International Conference on Dependable Systems and Networks*. 168–180.
[22] T. Yoshimura, H. Yamada, and K. Kono. 2012. Is Linux Kernel Oops Useful or Not?. In *Proceedings of the 8th USENIX Workshop on Hot Topics in System Dependability*.
[23] Y. Zhu, Y. Li, J. Xue, T. Tan, J. Shi, Y. Shen, and C. Ma. 2012. What is System Hang and How to Handle it. In *Proceedings of the 23rd IEEE International Symposium on Software Reliability Engineering*. 141–150.