GPU-based First Aid for System Faults

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 systems. Therefore, it is important to detect system faults rapidly and accurately and recover from them. Fault recovery is categorized into external one from remote hosts and internal one with processes or the operating system (OS) inside a target system. However, both methods are subject to system faults. If fault recovery fails, a hardware reset is required and can lead to losing system data and states. This paper proposes *GPUfas* for recovering from system faults by *indirectly* controlling OS behavior from a GPU, which is not easily affected by system faults. GPUfas attempts fault recovery by rewriting OS data in main memory and leveraging the capabilities of the OS itself. For example, it can mimic signal sending and process scheduling to force termination of the processes that consume excessive resources. It can also mimic unlocking to recover from some kind of deadlock. We have implemented GPUfas using the Linux kernel, CUDA, and LLVM to enable a GPU to rewrite OS data transparently. Then, we confirmed the effectiveness and efficiency of fault recovery by GPUfas.

CCS CONCEPTS

• **Software and its engineering → Software fault tolerance**; **Scheduling**; **Deadlocks**.

KEYWORDS

fault recovery, GPUs, signals, scheduling, deadlocks

ACM Reference Format:
Kento Kimura and Kenichi Kourai. 2022. GPU-based First Aid for System Faults. In *13th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '22), August 23–24, 2022, Virtual Event, Singapore.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3546591.3547526

1 INTRODUCTION

The root causes of system failures are software faults, poor performance, insufficient capacity, configuration and operation errors, etc. Although system developers should carefully consider software quality, system performance, and capacity, it is difficult to avoid failures completely. For example, AWS caused a failure due to too many threads that exceeded the limits of the operating system (OS) in servers 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 [15]. 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 rapidly and accurately and then recover from them. When admins detect a system fault, they often attempt *external recovery* from the fault by remotely accessing the target system. However, remote access is subject to system faults. For example, it can be prevented by faults in the network stack. Its performance can be affected by available system resources. Without using networks, *internal recovery* can be made by running recovery systems inside the target system in advance. A recovery system can be run as processes or be embedded into the OS kernel. Like remote access, it is also affected by system faults. If these methods cannot recover the system, a hardware reset is a last resort, but it is at high risk for losing the data and states of the system.

To avoid a hardware reset as much as possible, this paper proposes *GPUfas* for running a recovery system on a GPU inside a target host. GPUfas recovers the target system from a system fault by *indirectly* changing OS behavior. Specifically, it rewrites OS data in main memory from a GPU and eliminates the root cause of the fault using the capabilities of the OS itself. As examples of fault recovery, GPUfas can mimic signal sending and process scheduling to force termination of the processes that cause a system fault. It can also mimic unlocking to address some kind of deadlock. Since a GPU is not easily affected by system faults, GPUfas can increase the possibility of correctly running the recovery system.

We have implemented GPUfas using Linux 4.18, CUDA 10.0 [10], and LLVM 8.0 [14]. GPUfas enables the recovery system to rewrite OS data from a GPU by extending our previous work [12]. It maps the entire main memory onto the GPU memory address space using mapped memory in CUDA and extended memory management in Linux. It transforms the program of the recovery system to modify OS data transparently. In the current implementation, GPUfas provides three mechanisms called *pseudo signal sending*, *pseudo process scheduling*, and *pseudo unlocking*. To complement these mechanisms executed on a GPU, GPUfas provides a mechanism for in-kernel recovery support, which can be invoked from a GPU. We conducted several experiments to show the effectiveness and efficiency of GPUfas. As a result, we confirmed that GPUfas could recover from out-of-memory by terminating a process and a deadlock by releasing a spinlock.

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

2 FAULT RECOVERY

Fault recovery is categorized into two types: external and internal. Admins often access the target system remotely, e.g., using SSH, and attempt to recover from a system fault manually. This is called *external recovery*. One advantage of this method is that admins can inspect the root cause of the fault and select the best way of recovery. Instead of admins, a recovery system running at a remote host can automatically perform fault recovery via networks. However, the biggest disadvantage of these methods is that remote access is 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.

As fault recovery without remote access, *internal recovery* is also used. This method runs a recovery system inside a target system in advance. It is more reliable in that the recovery system does not rely on remote access. As an example, the recovery system can be run as processes inside a target system [2]. It periodically checks the system states and performs fault recovery if it detects a system fault. Using processes makes it easy to apply new recovery techniques. However, usable recovery techniques are restricted to the process-level ones. 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 [9, 18]. This method embeds a recovery system into the OS kernel and runs it periodically using timer interrupts. In-kernel recovery systems can tolerate system faults more, compared with process-level ones. They 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, the recovery system is not executed if timer interrupts are not handled by system faults.

If these methods cannot perform fault recovery, admins have to reboot a target system using a hardware reset. In fact, IPMI [8] and remote power management systems are 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. A hardware reset is a powerful recovery method, but it is at high risk for 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 that better fault recovery is impossible. 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 [4–6], it is difficult to salvage all the data.

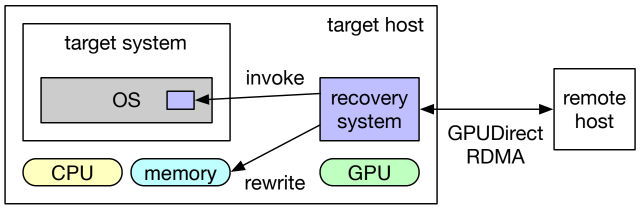


Figure 1: The system architecture of GPUfas.

3 GPU-BASED FIRST AID

In this paper, we focus on recoverable system faults, which do not destroy the integrity of the target system, as the first step. An example of such a fault is excessive resource consumption by processes or the OS kernel.

3.1 GPUfas

GPUfas attempts fault recovery by *indirectly* controlling OS behavior from a GPU. Figure 1 illustrates the system architecture of GPUfas. The recovery system running on a GPU rewrites OS data in main memory to recover the target system by leveraging OS capabilities. Thanks to the direct memory rewrites, developers can implement various recovery techniques without the limitations of processes or kernel modules. Nevertheless, the recovery system in GPUfas runs as a GPU application. This can make it easier to apply new recovery techniques.

Unlike existing methods, the recovery system running 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 recovery system from being affected by the resource shortage of the target system. It is reported that GPUs are subject to hardware failures, compared with CPUs [13], but this paper focuses on software faults. To protect the recovery system from the faults of the other programs running on the same GPU, GPUfas uses one GPU only for the recovery system. This GPU usage is acceptable because GPUfas can use inexpensive, low-end GPUs. The other PCIe devices such as DPUs and FPGAs could also be used for GPUfas, but they are too expensive to use in such a dedicated manner.

GPUfas can achieve both internal and external recovery with higher reliability. The recovery system on a GPU always monitors the target system. When a system fault occurs, the recovery system detects it and identifies its root cause by analyzing OS data in main memory using GPUSentinel [12]. After that, for internal recovery, the recovery system automatically uses one of the pre-set recovery techniques. For external recovery, it communicates with a remote host using GPUDirect RDMA [11] without the help of the OS [12] and uses the best recovery technique. At the remote host, admins can interactively analyze the details of the system fault and select a custom recovery technique. Instead of humans, artificial intelligence could do a better job.

Since GPUfas is a first-aid system, it might achieve only temporal fault recovery. For example, the target system might not provide services correctly after GPUfas terminates abnormal processes. In this case, admins can save data 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, GPUfas can send memory data to a remote host using GPUDirect RDMA without relying on the OS. Admins can analyze the memory data at the remote host and restore the data after a hardware reset.

3.2 Recovery Techniques

To recover from a system fault, GPUfas 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. Processes can send signals using the system call, while the OS kernel can do it directly using the kernel function. However, the recovery system on a GPU cannot invoke the system call or the kernel function. Instead, GPUfas 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, GPUfas provides *pseudo process scheduling*. This mechanism is used for controlling process execution and is indispensable to make pseudo signal sending effective. Pseudo signal sending itself cannot control paused processes because a sent signal is not handled until the target process is scheduled. Pseudo process scheduling enables such a process to be controlled by a signal sent by pseudo signal sending as early as possible. This leads to rapid recovery from a system fault. GPUfas changes the state of the process scheduler by rewriting scheduling data and mimics the adjustment of process scheduling.

GPUfas also provides *pseudo unlocking*. This is required to not only perform mutual exclusion but also recover from some kind of deadlock. A deadlock by missing lock release is a typical and frequent bug in the OS kernel. GPUfas 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 GPUfas 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 lock involved in a deadlock.

As such, GPUfas performs fault recovery by mimicking OS functions, but all the functions are not implementable only by rewriting OS data. For example, 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 implement on a GPU. Even if some of the OS functions are implementable, they could largely degrade the recovery performance due to frequent access to main memory.

To address these issues, GPUfas can cooperate with *in-kernel recovery support*. The recovery system on a GPU 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 because it runs on CPUs. 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 recovery system on a GPU and in-kernel recovery support is more reliable than pure in-kernel

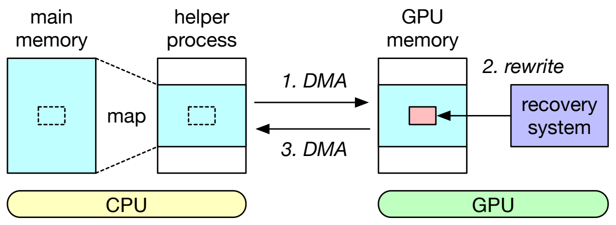


Figure 2: Rewriting main memory from a GPU.

recovery systems. Only the recovery functions running in the kernel can be affected by system faults.

4 IMPLEMENTATION

We have implemented GPUfas using the Linux kernel 4.18.0, CUDA 10.0 [10], and LLVM 8.0 [14].

4.1 Memory Rewrite from a GPU

To rewrite main memory from a GPU, we have extended the GPU-based monitoring mechanism proposed in GPUSentinel [12]. 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, as illustrated in Fig. 2. 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 in the writable mode, the kernel does not increase the reference count of each page so that the page does not become in use.

Developers can write the program of a recovery system using the source code of the Linux kernel. 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. This is transparently done by transforming the program of a recovery system using LLVM. GPUfas searches for the load and store instructions, which are used to read and write data from and to memory in intermediate representation, respectively. Then, it inserts the invocation to the code for address translation just before those instructions.

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 attackers 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.

4.2 Pseudo Control from a GPU

To mimic signal sending on a GPU, GPUfas directly rewrites the data structure used for the signal mechanism in the kernel. First, GPUfas 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 OS kernel checks this flag when it schedules that process and switches the CPU mode from the kernel to user mode. If there is a pending signal, the kernel handles the injected signal.

For pseudo process scheduling, GPUfas indirectly rewrites the data structure used for the process scheduler in the kernel by emulating scheduler functions on a GPU. The reason why GPUfas does not directly rewrite data is that scheduling data is much more complex. Currently, GPUfas support 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. Before this operation, it acquires the spinlock for a per-CPU run queue using in-kernel recovery support. It releases the spinlock using pseudo unlocking after the operation. During the operation, it acquires several necessary spinlocks. Finally, it changes the process state to

Table 1: The used combination of recovery techniques.

method	recovery techniques		
PSIG	pseudo signal sending	–	
PSIG+PSCH		pseudo scheduling/ unlocking	in-kernel locking
PSIG+KSCH		in-kernel scheduling	

TASK_RUNNING. Later, the OS kernel schedules processes in descending order of their virtual run time.

For pseudo unlocking, GPUfas mimics spinlock release by directly rewriting the data structure used for a spinlock in the kernel. It first obtains the qspinlock structure in raw_spinlock_t. Then, it changes the value of the lock variable in it to zero.

4.3 In-kernel Recovery Support

To invoke in-kernel recovery support, a recovery system on a GPU writes a request to a queue allocated in main memory. 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 to 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 on a GPU periodically reads the queue by polling and continues its execution if in-kernel recovery support succeeds.

Currently, GPUfas provides locking and process scheduling as in-kernel recovery support. Pseudo process scheduling has been achieved on a GPU, 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 GPUfas. We used three combinations of recovery techniques, as depicted in Table 1. PSIG used only pseudo signal sending. PSIG+PSCH used pseudo process scheduling and in-kernel recovery support for acquiring a spinlock as well as pseudo signal sending. PSIG+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.

Table 2: The results of pseudo signal sending.

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

5.1 Effectiveness of Pseudo Signal Sending

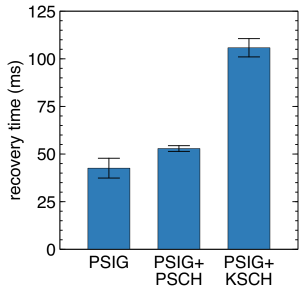
We performed pseudo signal sending to a process for the KILL, TERM, STOP, and CONT signals and examined the behavior of the process. For the CONT signal, we first sent the STOP signal using the kill command to pause the process. Table 2 shows the results of pseudo signal sending. PSIG+PSCH could send all the signals correctly. The KILL signal forced termination of the process, and the TERM signal normally terminated the process. The STOP signal paused the process, while the CONT signal continued the process paused by the kill command. In contrast, PSIG could not continue the paused process because it did not perform process scheduling to wake up that process.

Surprisingly, PSIG+KSCH could not pause the process. After the STOP signal was sent by pseudo signal sending, the kernel paused the process because it periodically scheduled the running process. After that, in-kernel scheduling support changed the process state to runnable again. 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 scheduled in such a case.

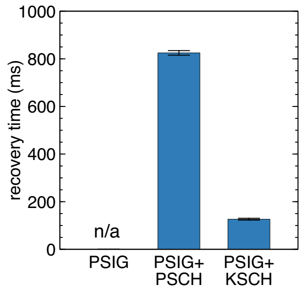
5.2 Performance of Pseudo Signal Sending

We examined the performance of pseudo signal sending. We sent the KILL signals to 1000 processes and measured the time until all the processes were terminated. When we used processes that performed busy waiting, the recovery time was shown in Fig. 3(a). PSIG could terminate all the processes successfully without process scheduling and achieve the fastest recovery. PSIG+PSCH performed pseudo process scheduling as well, but the recovery time only slightly increased because pseudo scheduling did not re-schedule the runnable processes. In contrast, PSIG+KSCH significantly increased the recovery time due to the overhead of invoking in-kernel scheduling support from a GPU.

When we used processes paused by long sleep, the recovery time was shown in Fig. 3(b). PSIG could not terminate any processes because the paused processes could not handle sent signals without process scheduling. Unlike the case of running processes, PSIG+PSCH took much longer than PSIG+KSCH. This is because many invocations of in-kernel locking support suffered from large overhead during pseudo



(a) Running processes



(b) Paused processes

Figure 3: The time to terminate 1000 processes.

process scheduling. Note that we optimized the recovery system to acquire a spinlock per CPU and handle all the processes running on one CPU at once. In PSIG+KSCH, in-kernel scheduling support also suffered from invocation overhead, but it was invoked only once per target process. However, PSIG+KSCH is less reliable due to running complex scheduling code in the kernel.

5.3 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 19 GB of memory in the PC with 16 GB of physical memory. The process sequentially wrote data to the allocated memory to continuously cause swapping. When we attempted remote login to this host, it took about 20x longer due to thrashing. 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, we used a process-level recovery system and an in-kernel recovery system. The recovery process issued the kill system call, while the in-kernel recovery system invoked the kernel function for signal sending.

Figure 4 shows the recovery time from out-of-memory. PSIG in GPUfas was the fastest and the most stable. PSIG+PSCH increased the average only by 32 ms due to pseudo process scheduling, but it was still stable. In PSIG+KSCH, in contrast, the average was 100 ms longer than in PSIG+PSCH. 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 on the in-kernel scheduler also made the variance of the recovery time much larger.

Contrary to our expectation, the recovery process was not largely affected by frequent swapping, but it resulted in lower stability. Surprisingly, the in-kernel recovery system was

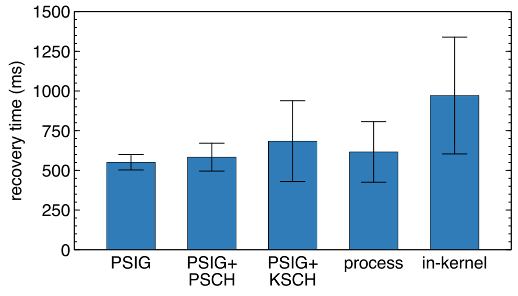


Figure 4: The recovery time from out-of-memory.

worst in both the average and stability. One of the reasons is the large impact on the in-kernel scheduler like PSIG+KSCH. However, it is unclear why the in-kernel recovery system was much worse than the recovery process.

5.4 Recovery from a Deadlock

We made the kernel in the target system cause a deadlock involving all the eight CPUs. We loaded the kernel module in which eight threads attempted to acquire the same 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 eight times 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 recovery process because the deadlock in the kernel prevented that process from running.

Figure 5 shows the recovery time from the deadlock. We confirmed that GPUfas could recover from the 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. The low-level interrupt handler succeeded in fault recovery because it is invoked regardless of the deadlock. GPUfas was slightly slower, but it is 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 [17]. The OOM killer forces termination of the process that consumes excessive memory when the system causes out-of-memory. While it does not consider factors except for the consumption of

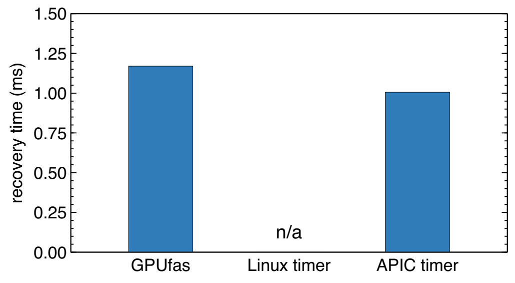


Figure 5: The recovery time from a deadlock.

memory and swap space, GPUfas can select processes more flexibly on a system fault.

SHFH [18] 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 but recovers from the fault only in the kernel. Therefore, it is subject to a system fault and is less reliable than GPUfas, which runs a recovery system on a GPU.

Backdoors [3] 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 GPUfas. 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, GPUfas is more secure because a recovery system on a GPU can execute only recovery functions fixed in advance. GPUfas can also communicate between a GPU and a remote host using GPUDirect RDMA, but a recovery system still runs on a GPU.

EXTERIOR [7] 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. This mechanism assumes the system running in a VM, while GPUfas can be applied to the system running in a physical machine.

Otherworld [6] microreboots the OS kernel when an in-kernel 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 GPUfas. It can be used to minimize the impact of a reboot when GPUfas cannot recover from a kernel fault.

A phase-based reboot [16] can reduce the recovery time from a kernel 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 on a system fault is lost. Also, only the system running in a VM is recoverable.

7 CONCLUSION

This paper proposed GPUfas for enabling fault recovery by running a recovery system on a GPU and indirectly controlling OS behavior. GPUfas rewrites OS data in main memory and attempts to recover from a system fault by leveraging OS capabilities. Currently, it provides recovery techniques called pseudo signal sending, pseudo process scheduling, and pseudo unlocking. For OS functions that are difficult to implement on a GPU, it cooperates with in-kernel recovery support. We have implemented GPUfas by extending the memory management in Linux and using mapped memory in CUDA and program transformation with LLVM. We confirmed that GPUfas could recover from several system faults in a short period.

One of our future work is to optimize pseudo process scheduling. We have implemented it using kernel functions, but we could minimize it. 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 caused by the overflow of performance counters to invoke in-kernel recovery support, instead of timer interrupts. Then, we would like to apply GPUfas to real-world system faults. 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.

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] 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.

[4] 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.

[5] 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.
[6] 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.
[7] 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.
[8] Intel, Hewlett-Packard, NEC, and Dell. 2004. Intelligent Platform Management Specification Second Generation v2.0.
[9] 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.
[10] NVIDIA Corporation. 2018. CUDA Toolkit Documentation v10.0.130. https://docs.nvidia.com/cuda/archive/10.0/.
[11] NVIDIA Corporation. 2022. *Developing a Linux Kernel Module Using RDMA for GPUDirect*. Technical Report TB-06712-001 v11.7. NVIDIA.
[12] 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.
[13] A. Taherin, T. Patel, G. Georgakoudis, I. Laguna, and D. Tiwari. 2021. Examining Failures and Repairs on Supercomputers with Multi-GPU Compute Nodes. In *Proceedings of the 51st Annual IEEE/IFIP International Conference on Dependable Systems and Networks*. 305–313.
[14] The LLVM Foundation. [n.d.]. The LLVM Compiler Infrastructure. https://llvm.org/.
[15] 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.
[16] 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.
[17] 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*.
[18] 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.