Reliable and Accurate Fault Detection with GPGPUs and LLVM

Yuichi Ozaki*, Sousuke Kanamoto*, Hiroaki Yamamoto*, and Kenichi Kourai†
Kyushu Institute of Technology
*{*bushido, k_sousuke, hiroaki*}*@ksl.ci.kyutech.ac.jp*, †*kourai@csn.kyutech.ac.jp*

***Abstract*—As the scale and complexity of cloud systems are increasing, system faults are becoming unavoidable. Therefore, they should be detected as reliably and accurately as possible. Black-box monitoring can reliably monitor a target system from a remote host, but it is often coarse-grained and cannot identify the root causes of system faults. In contrast, white-box monitoring can accurately obtain fault information inside a target system, but it is largely affected by system faults. This paper proposes *GPUSentinel* for more reliable white-box monitoring using general-purpose GPUs. GPUSentinel runs fault detectors in an isolated GPU, which is not easily affected by faults of a target system. For accurate detection, fault detectors in a GPU analyze main memory and directly monitor the state of the operating system. To easily develop such fault detectors, GPUSentinel provides a development environment with LLVM. We have implemented GPUSentinel and seven fault detectors and then confirmed that GPUSentinel could detect various system faults and identify the root causes.**

***Index Terms*—operating systems, system faults, fault detection, GPUs, white-box monitoring**

1. Introduction

Recently, the systems used for cloud computing are getting larger and more complex. As a consequence, it becomes difficult to avoid system faults. Once a system fault occurs, it sometimes results in a system failure and services provided by the system stop. This leads service providers to a huge financial loss. For example, it is estimated that Amazon lost $72 million during Prime Day's one-hour failure [1]. Such a system failure largely affects the users of the services and system administrators as well. The users can also suffer from some loss due to service unavailability. System administrators have to identify the root cause of the system failure and restore the system.

To reduce such loss, system faults should be detected as reliably and accurately as possible before a system failure. Traditionally, *black-box monitoring* is used for fault detection. A typical example is heartbeat monitoring of a target host and services [2]. In this method, fault detectors in a remote host can reliably monitor a target system even when system faults occur. However, it is difficult to obtain detailed information on the target system. For more accurate fault detection, *white-box monitoring* can be used inside a target system [3]–[6]. In this method, fault detectors run on top of the operating system (OS) or are embedded into the OS kernel. This method can obtain the internal state of the target system, but fault detectors can be affected by system faults. This is because fault detectors strongly depend on the target system.

In this paper, we propose *GPUSentinel* for more reliable white-box monitoring using general-purpose GPUs. GPUSentinel runs fault detectors in a GPU, which can execute code independently of CPUs and main memory. Therefore, fault detectors are not easily affected by faults of a target system running in CPUs. In GPUSentinel, fault detectors analyze main memory using the knowledge of data structures used in the OS kernel and obtain the state of the target system. As such, they can use detailed information to detect system faults and identify the root causes. To easily develop such fault detectors like OS kernel modules, GPUSentinel provides a development environment including program transformation with LLVM [7].

We have implemented GPUSentinel using CUDA [8]. GPUSentinel uses the mapped memory mechanism in CUDA and transparently accesses main memory from a GPU. To allow the entire main memory to be mapped in the GPU address space, we have modified the memory management in the Linux kernel. In addition, we have developed a framework called *LLView*, which transparently transforms the programs of fault detectors so as to translate virtual addresses of OS data to GPU addresses. LLView also enables developers to reuse the source code of the OS kernel. Using GPUSentinel, we have developed seven fault detectors for detecting system hangs. Through our experiments, we confirmed that GPUSentinel could detect these system faults successfully.

This paper is an extension of our workshop paper [9]. In the previous paper, we presented only the basic idea of GPUSentinel and preliminary results. In this paper, we have significantly extended our previous paper. First, we have completely implemented the mechanism for fault detection using a GPU. Second, we have developed seven fault detectors for GPUSentinel. Third, we compared the abilities of fault detection between GPUSentinel and the existing CPU-based white-box monitoring system.

The organization of this paper is as follows. Section 2 describes the issues of traditional fault detection. Section 3 proposes GPUSentinel for reliable and accurate fault detection and Section 4 explains its implementation. Section 5 reports the results of our experiments. Section 6 describes related work and Section 7 concludes this paper.

2. Reliable and Accurate Fault Detection

To avoid a system failure, it is necessary to detect system faults as reliably as possible. Fault detectors have to always detect system faults whenever faults occur in a target system. If they cannot work well at that time, missed system faults can cause a system failure and lead to a huge financial loss. In addition, accurate fault detection is also necessary. If possible, it is desirable to detect symptoms of system faults. Earlier detection of system faults can lower the probability of leading to a system failure. Obviously, more information helps system administrators correctly identify system faults and their root causes. If false positives occur, system administrators may have to investigate the root causes of system faults that do not actually occur.

Traditionally, *black-box monitoring* is performed in a remote host to detect system faults. An example of black-box monitoring is the heartbeat monitoring of a target host and services provided by a target system via the network [2]. Fault detectors in a remote host can examine not only the state of the target host but also the state of each service by periodically connecting to all the services. If the responses are slow, that may be a symptom of system faults. If there are no responses, system faults probably occur. However, this monitoring method cannot achieve accurate fault detection. Since fault detection by black-box monitoring is coarse-grained, fault detectors cannot identify which types of system faults exactly occur. Furthermore, it is difficult to obtain detailed information on the target system when system faults occur because fault detectors cannot access the internal state of the system. Therefore, the root causes of system faults cannot be identified as well.

If target hosts equip with hardware monitoring such as IPMI [10], fault detectors in a remote host can obtain more detailed information even outside the target system. They can examine the hardware state and use that information for fault detection. For example, CPU usage, the amount of disk access, and the number of network packets help fault detectors detect system faults more accurately than simple heartbeat monitoring. Similarly, if a target system runs in a virtual machine (VM), fault detectors can obtain the state of virtual hardware from the outside of the VM. However, such extra information may still be insufficient to identify exact fault types and root causes.

For more accurate fault detection, *white-box monitoring* can be used [3]–[6]. In this monitoring method, fault detectors run inside a target system and notify a remote host of system faults. They can obtain more detailed information than black-box monitoring. In particular, when they are embedded into the OS kernel, they can identify fault types and root causes more easily. However, white-box monitoring cannot achieve reliable fault detection. Once a system fault occurs, fault detectors may not work correctly. For example, they cannot detect system faults or identify the root causes if the OS kernel stops. Even if the OS kernel continues to run correctly, detector processes may be terminated by the OS kernel, e.g., the out-of-memory (OOM) killer in Linux, when system memory runs out.

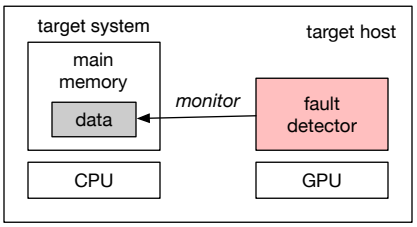


Figure 1: The architecture of GPUSentinel.

3. GPUSentinel

This paper proposes *GPUSentinel* for achieving more reliable white-box monitoring by running fault detectors in commodity GPUs. Fig. 1 shows the system architecture of GPUSentinel. In GPUSentinel, fault detectors start to run at the boot time of the target system, i.e., before any system failures occur. They occupy one GPU and run autonomously. Even if the target system uses GPUs for computing, GPUSentinel is available by installing another low-cost GPU. To monitor the target system from a GPU, fault detectors inspect OS data stored in main memory. If they detect system faults, they analyze the root causes.

Using GPUs, GPUSentinel enables reliable and accurate fault detection. For reliability, fault detectors in a GPU can continue to run with a high probability even when system faults occur in the target system. This is because GPUs are physically isolated from CPUs and main memory, on top of which the target system runs. System faults in the target system are not easily propagated to fault detectors in a GPU. It should be noted that GPUs are usually used as co-processors and are controlled by CPUs. Therefore, GPUSentinel takes control of one dedicated GPU by running fault detectors indefinitely.

For accuracy, GPUSentinel can detect system faults that can be found from OS data stored in main memory if necessary data are not corrupted by faults. System faults that run out of system resources or cause resource starvation are detectable. For example, if the system uses a large amount of memory or memory leaks occur, it cannot allocate necessary memory. If all the CPUs are involved in deadlocks with spinlocks, the entire system hangs. To detect these system faults, fault detectors in a GPU can monitor the amount of free memory and consumed CPU time. If these values are abnormal, GPUSentinel can detect that state as symptoms of system faults.

However, it is a troublesome task to develop such fault detectors that monitor OS data in main memory from a GPU. To obtain OS data, fault detectors have to find the location in main memory and analyze data structures used in the OS kernel. For ease of development, GPUSentinel enables developers to write programs of fault detectors like OS kernel modules. The programming of OS kernel modules is natural to access OS data. In GPUSentinel, a framework called *LLView* provides a development environment for fault detectors. It transparently transforms the programs of fault detectors using LLVM [7] so that developers are not aware

of indirect access to main memory from a GPU. In addition, it enables developers to reuse the source code of the OS kernel as much as possible.

4. Implementation

We have implemented GPUSentinel using CUDA 8.0 [8], LLVM 5.0. To enable GPUs to monitor the entire main memory, we have modified Linux kernel 4.4. For GPUSentinel, we have developed seven fault detectors as one GPU program using multiple GPU cores.

4.1. Main Memory Mapping

GPUSentinel enables fault detectors in a GPU to transparently access main memory using *mapped memory* in CUDA. Mapped memory is used to map main memory onto the GPU address space and make it accessible from GPU kernels. To use mapped memory for the entire main memory, GPUSentinel first maps main memory onto the address space of a host process. However, if GPUSentinel simply maps the entire main memory, free memory runs out. When CUDA maps the memory region onto the GPU address space, it pins all the memory pages so that any pages are not paged out. At this time, all the memory pages are locked and become in use.

To address this issue, GPUSentinel uses a special device added to the Linux kernel. When CUDA pins the memory pages where the device is mapped, the modified Linux kernel neither increases the reference count of each memory page nor locks it to prevent the page from being in use. In addition, GPUSentinel works around the limitation of CUDA, which allows mapping only a bit smaller amount of memory than the size of main memory. It intercepts the sysinfo system call and returns a bit larger size as the size of main memory. For the details of this implementation, see our previous paper [9].

4.2. Transparent Address Translation

To transparently translate virtual addresses of the OS kernel into GPU addresses, LLView compiles the programs of fault detectors using LLVM and transforms the intermediate representation called bitcode. When bitcode reads data from memory, the `load` instruction is used. LLView replaces the `load` instruction so that bitcode invokes the `g_map` function for performing address translation and executes the `load` instruction for the translated address. If the passed address is not the virtual address of the OS kernel, the function just returns the passed address without translation. The threshold address is `0xffff000000000000`. At that time, bitcode accesses local memory in a GPU.

For example, let us consider the following bitcode.

```
%1 = load i64, i64* %jiffies
%2 = udiv i64 %1, 250
```

This bitcode reads the OS global variable `jiffies` into the local variable `%1`, divides it by 250, and stores the result in `%2`. LLView transforms this bitcode as follows:

```
%1 = bitcast i64* %jiffies to i8*
%2 = call i8* @g_map(i8* %1)
%3 = bitcast i8* %2 to i64*
%4 = load i64, i64* %3
%5 = udiv i64 %4, 250
```

This bitcode casts the address of `jiffies` to the pointer to an 8-bit integer and stores it in `%1`. Using that pointer as an argument, the bitcode invokes the `g_map` function and translates the address. Then, it casts the returned address to the pointer to the original 64-bit integer and reads the data from the address to `%4`.

LLView uses the LLVM Pass framework for transforming bitcode. When LLView finds the `load` instruction in bitcode, it obtains the target variable and its type. Using that information, it generates the `call` instruction for invoking the `g_map` function and inserts that instruction just before the `load` instruction. Then, it generates a new `load` instruction that reads data from the translated address, inserts that instruction, and removes the original `load` instruction. At this time, LLView rewrites all the instructions using the local variable in which data is stored by the original `load` instruction. That local variable is replaced with the new one, whose value is stored by the new `load` instruction.

The `g_map` function first translates a virtual address into a physical address using the page tables of the OS kernel. Using the knowledge of the Linux kernel, LLView optimizes this address translation in the following two cases. When a virtual address is in the range of direct mapping of main memory, LLView performs address translation by subtracting the top address of the range from the virtual address. For the address range in which the kernel text area is mapped, LLView does similarly. Next, the `g_map` function translates the physical address into a GPU address. For this translation, LLView simply adds the physical address to the top GPU address in which main memory is mapped.

To enable fault detectors to access global variables in the OS kernel, LLView replaces the kernel variables in bitcode with the corresponding virtual addresses used in the OS kernel. It obtains the mapping between kernel symbols and virtual addresses from the System.map file. The resulting virtual addresses are translated into physical ones by LLView. For example, the `bitcast` instruction for `jiffies` is replaced by the Pass framework as follows:

```
%1 = inttoptr i64 -2113892352 to i8*
```

This virtual address means `0xffffffff82009000` hexadecimally. The `bitcast` instruction is changed to the `inttoptr` instruction because a virtual address is dealt with as a 64-bit integer in bitcode.

4.3. Development like OS Kernel Modules

Using LLView, developers can write programs of fault detectors like OS kernel modules. They can reuse the source

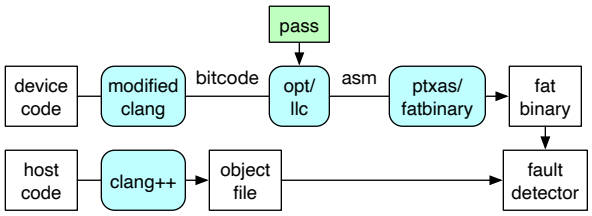


Figure 2: Compilation of a fault detector.

code of the Linux kernel, e.g., data structures, global variables, inline functions, and macros. Let us consider a fault detector that obtains the CPU time consumed by processes. This fault detector traverses the process list, which is a circular list starting with the `init_task` global variable, using the `list_entry` macro. During that traversal, it obtains the system times consumed by each process from the `task_struct` structure, which is defined in `sched.h`. To use the Linux header files, we needed to slightly modify some of them. For example, we had to add the `__device__` qualifier to only used inline functions. We also replaced inline assembly code with C code because LLView could not support assembly code.

GPUSentinel enables developers to write fault detectors in C. CUDA programs consist of device code running on GPUs and host code running on CPUs. Both are usually written in C++. However, it is difficult to reuse the source code of the Linux kernel written in C and compile device code using it as C++. For example, variable names in C can conflict with the reserved words in C++, e.g., `new`. C++ requires type casts that are unnecessary in C and disallows arithmetics for `void` pointers.

Therefore, LLView uses a clang compiler front end modified so that device code is compiled as C. Clang defines specification used for compilation for each type of program. We changed the specification used for CUDA programs to C90 and GCC extensions. In GPUSentinel, CUDA programs are compiled as illustrated in Fig. 2. First, LLView compiles device code using the modified clang. It applies our passes to the generated bitcode using opt. Then, it creates an embeddable binary called fat binary using ptxas and fatbinary. Finally, it compiles host code using the original clang++ and embeds the fat binary into the generated object file.

For several variables and functions that CUDA provides to device code, LLView provides wrapper functions written in C. Since device code is compiled as C in LLView, CUDA variables and functions implemented in C++ cannot be used as is. For example, CUDA provides the `blockDim` variable that returns a unique thread ID in a GPU kernel and the `threadIdx` variable that returns a thread index in a block. For them, LLView provides the C function called `get_thread_id` that returns a thread ID using that variable. For the `__threadfence` function used for exclusive memory access, LLView provides the `__gs_threadfence` function in C.

4.4. Detection of Kernel-level Faults

Using GPUSentinel, we have developed seven fault detectors using some of the metrics proposed for systematically detecting faults that lead to system hangs [4] and additional metrics. The metrics on CPUs are defined as the CPU utilization consumed by the kernel (sys) and processes (usr). For these metrics, a fault detector first obtains the CPU times used for them per CPU from the `kernel_cpustat` structure. Next, it waits for one second by repeatedly reading the value of the clock64 register in a GPU. Then, it obtains these CPU times again and calculates CPU utilization. In addition, we added another metric for timer interrupts (int) because the above two CPU times are not updated without timer interrupts. A fault detector obtains the value of the `jiffies` variable, which is incremented by timer interrupts, and calculates the increase in one second.

The metric on CPU scheduling is defined as the number of context switches per second (cs). A fault detector obtains the number of context switches recorded in the `rq` structure and calculates the increase in one second. In addition, we have added another metric for the number of uninterruptible processes (sleep). A process enters the state of uninterruptible sleep when it issues a system call that cannot be interrupted by signals and waits for I/O. A fault detector traverses the process list from the `init_task` variable and counts the number of uninterruptible processes by examining the `task_struct` structure.

The metrics on memory are defined as the amount of free memory (memfree) and the number of swap-outs per second (pswpout). For memfree, a fault detector obtains the amount of free memory from the `vm_stat` array. For pswpout, a fault detector first obtains the number of swap-out events recorded per CPU in the `vm_event_state` structure and calculates the increase in one second. Then, it accumulates the increases for all the CPUs.

We have added one more metric (panic) because a kernel panic is an explicit fault response from the kernel. Upon a kernel panic, the kernel acquires a spinlock using the `panic_lock` variable. A fault detector obtains the value that represents whether that spinlock is locked or not.

Using the combination of these metrics, our fault detectors detect system faults that are categorized into six types by the detailed analysis of the prior work [4] and one extra fault in Table 1. The first three system faults are related to infinite loops. Fault F1 is defined as an infinite loop with interrupts disabled, e.g., a deadlock with spinlocks. The prior work uses sys, usr, and cs to detect F1, but our fault detector for F1 monitors int because CPU utilization cannot be obtained without timer interrupts. Then, it detects a system fault if the value of int is zero. Fault F2 is defined as an infinite loop with both interrupts and preemption enabled. The fault detector for F2 monitors sys and usr and detects a system fault if the value of sys is more than 95% and that of usr is less than 1%. Fault F3 is defined as an infinite loop with interrupts enabled but preemption disabled. The fault detector for F3 is similar to that for F2,

TABLE 1: Metrics used by kernel-level fault detectors.

	CPU			scheduling		memory		panic
	sys	usr	int	cs	sleep	memfree	pswpout	
F1			✓					
F2	✓	✓						
F3	✓	✓		✓				
F4					✓			
F5	✓	✓		✓				
F6						✓	✓	
F7								✓

but it also monitors cs. It detects a system fault if the value of cs is less than 350 in addition to the condition for F2.

The next three system faults are related to indefinite waits. Fault F4 is defined as an indefinite wait due to resources not being released. The fault detector for F4 monitors sleep and detects a system fault if the value exceeds 32. Fault F5 is defined as an indefinite wait due to sleep while holding a lock. Since this system fault is similar to F3, the fault detector for F5 is the same as that for F3. To distinguish between F3 and F5, additional metrics need to be added. Fault F6 is defined as an indefinite wait due to abnormal resource consumption. In this paper, we focus on memory, although there are various resources. The fault detector for F6 monitors memfree and pswpout and detects a system fault if the value of memfree is less than 256 MB and that of pswpout is more than 3000.

In addition, we newly define a kernel panic as fault F7. A kernel panic is a variant of F1 in that it disables interrupts and slowly executes loops for all the CPUs infinitely or for a while. Therefore, it could be detected by the fault detector for F1, but we have developed the fault detector for F7, which monitors panic instead of int.

4.5. Identification of Root Causes

To identify the processes and kernel threads that cause a system fault on CPUs after detecting the fault, the fault detector calculates the detailed CPU utilization per process. It traverses the process list and, for each process and thread, obtains the CPU times consumed in the user space and the kernel from the `task_struct` structure, respectively. If a process has multiple threads, the fault detector accumulates the CPU times consumed by all the threads for the process. Then, it waits for one second and calculates CPU utilization from the increase in one second. Finally, it sorts the measured CPU utilization of all the processes and kernel threads in a descendant order and regards the ones with higher CPU utilization as the root cause.

For a system fault on memory, the fault detector identifies the processes that cause the fault by calculating the OOM scores. These scores are the values used by the OOM killer in the Linux kernel, which enforces the termination of a process when free memory runs out. To calculate the OOM scores, the fault detector obtains the amounts of consumed main memory, swap space, and page tables from the `mm_struct` structure. On the basis of these values, it calculates the OOM scores. Finally, it finds the processes with the higher score and regards them as the root cause.

TABLE 2: The detectability of system faults.

method	F1	F2	F3	F4	F5	F6	F7
GPUSentinel	✓	✓	✓	✓	✓	✓	✓
OS-based		✓	✓	✓	✓	✓	

5. Experiments

We conducted several experiments to confirm that GPU-based fault detectors in GPUSentinel could detect system faults. For comparison, we used OS-based fault detectors as existing white-box monitoring using CPUs. In addition, we examined the performance impact of GPUSentinel on the target system and that of the target system on GPUSentinel. We used a PC with an Intel Xeon E5-1603 v4 processor, 8 GB of DDR4-2400 memory, and NVIDIA Quadro M4000. We ran Linux 4.4.64 and CUDA 8.0.61.

5.1. Detectability of Kernel-level Faults

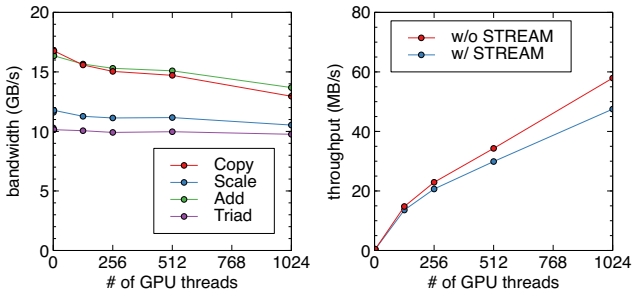
To inject kernel-level faults described in Section 4.4, we have developed seven Linux kernel modules. For F1, the kernel module created the same number of threads as CPUs and caused a deadlock using two spinlocks. For F2 and F3, the same number of kernel threads as CPUs executed an infinite loop. For F4, the kernel module inserted an erroneous mutex lock to cause a deadlock. For F5, one kernel thread acquired a spinlock and indefinitely slept. For F6, the kernel module allocated 16 GB of memory. For F7, the kernel module caused a kernel panic.

The first row in Table 2 shows which faults could be detected by GPUSentinel. We confirmed that GPUSentinel could detect all of the seven kernel-level faults. For F1 to F3 and F5, GPUSentinel could accurately identify the thread names involved in the faults as the root causes.

In addition to fault injection, we caused the actual fault reported in Kernel.org Bugzilla [11] by mounting the special XFS disk image. In Linux 4.4, we could mount it, but we suffered from a deadlock and got stuck when accessing the target directory. After we accessed that directory using more than 32 processes, GPUSentinel could detect fault F4.

5.2. Comparison with OS-based Fault Detection

To show the advantage of GPU-based fault detection, we compared our fault detectors in a GPU with OS-based ones. We have developed OS-based fault detectors that monitor the same metrics as the ones used in GPUSentinel. These detectors were created as Linux kernel modules and were driven by timer interrupts. It should be noted that we needed to modify the Linux kernel to additionally export several kernel variables to the modules. This means that it may not be easy to develop new fault detectors if these modules need kernel variables that are not exported. In GPUSentinel, fault detectors can access any kernel variables.



(a) STREAM on CPUs (b) Fault detector in a GPU

Figure 3: The mutual impact of a CPU-based application and a GPU-based fault detector.

The second row in Table 2 shows which faults could be detected by the OS-based fault detectors. These fault detectors could detect system faults except for F1 and F7. Since F1 and F7 disabled interrupts, timer-driven fault detectors were not invoked.

5.3. Performance Overhead

Since GPUSentinel accesses main memory to monitor OS data, we examined the impact on a memory-intensive application and a fault detector. As a memory-intensive application on CPUs, we ran the STREAM benchmark [12] in the target system. In a GPU, we ran a memory-intensive fault detector that continues to copy OS data from main memory to GPU memory using transparent DMA.

First, we examined the impact of the fault detector on the performance of STREAM. Fig. 3(a) shows the memory bandwidths for four operations in STREAM. As the number of GPU threads increased, the memory bandwidths decreased. For 1024 threads, the overhead reached up to 33%, but actual fault detectors do not just obtain OS data from main memory as in this experiment. Therefore, the performance degradation of memory-intensive applications in the target system could be much smaller in practice.

Next, we examined the impact of STREAM on the performance of the fault detector in a GPU. Fig. 3(b) shows the throughput of copying OS data in the fault detector when we increased the number of GPU threads. As the number of GPU threads increased, the fault detector was affected more largely by the memory pressure of STREAM. The performance degradation was negligible for one thread but 20% for 1024 threads. However, this is also the worst case.

6. Related Work

SHFH [4] classifies the root causes of system faults into six types and detects system faults using only minimum performance metrics. The real-time user process monitors the system in a lightweight manner during normal operation. When the process detects symptoms of a system fault, the kernel module investigates the system state in further detail. However, SHFH cannot detect system faults that make the OS kernel completely hang.

Falcon [3] runs spies in various layers of the system. Then, lower-layer spies monitor higher-layer ones to enable fine-grained fault detection. However, Falcon can detect only a crash failure by black-box monitoring. To obtain more information on system failures, Pigeon [5] runs sensors in system components. Panorama [6] can detect failures by automatically inserting report-detection code into applications. Since these frameworks depend on target components, they are easily affected by component failures.

Backdoors-based remote healing [13] enables a remote monitor to accurately detect system faults in a target system. The target OS periodically stores the OS state in a memory region called a Sensor Box (SB), while the remote monitor obtains the state from the SB using RDMA. However, the SB is not reliable because it strongly depends on the monitored OS. In addition, it requires modification to the target OS.

When the target system runs in a VM, monitoring systems can obtain the internal state from the outside of the VM using VM introspection (VMI) [14]. Vigilant [15] monitors hypervisor-level event counters and detects system faults such as system hangs. HyperTap [16] monitors system events that cause VM exits and detects system faults using hardware architectural invariants. Unlike GPUSentinel, these systems do not analyze OS data in memory.

For security, various mechanisms have been proposed to securely monitor the target system. They could also be used for reliable fault detection. Copilot [17] monitors the integrity of the OS kernel using an ARM evaluation board on a dedicated PCI card. HyperCheck [18] runs a network driver in System Management Mode (SMM) of processors. It transfers memory data to a remote host using the driver and checks the integrity of the hypervisor. HyperSentry [19] communicates with a target host using IPMI and runs a monitoring agent inside the target hypervisor using SMM.

7. Conclusion

This paper proposes GPUSentinel for reliable and accurate fault detection by running fault detectors in GPUs. To detect system faults, GPUSentinel monitors OS data in main memory from GPUs using LLView. Using GPUSentinel, we have developed seven fault detectors and confirmed that they could detect system faults successfully. One of our future work is to support other types of system faults. Another direction is to detect faults in the hypervisor of a virtualized system running VMs.

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] Digital Commerce 360, "The Potential Cost of Amazon's Prime Day Miss? $72 Million," https://www.digitalcommerce360.com/2018/07/17/the-potential-cost-of-amazons-prime-day-miss-72-million/, 2018.

[2] A. Beekhof, "Pacemaker," https://clusterlabs.org/pacemaker/.

[3] J. Leners, H. Wu, W. Hung, M. Aguilera, and M. Walfish, "Detecting Failures in Distributed Systems with the Falcon Spy Network," in *Proc. Symp. Operating Systems Principles*, 2011, pp. 279–294.

[4] Y. Zhu, Y. Li, J. Xue, T. Tan, J. Shi, Y. Shen, and C. Ma, "What is System Hang and How to Handle it," in *Proc. Int. Symp. Software Reliability Engineering*, 2012, pp. 141–150.

[5] J. Leners, T. Gupta, M. Aguilera, and M. Walfish, "Improving Availability in Distributed Systems with Failure Informers," in *Proc. Conf. Networked Systems Design and Implementation*, 2013, pp. 427–442.

[6] P. Huang, C. Guo, J. Lorch, L. Zhou, and Y. Dang, "Capturing and Enhancing in Situ System Observability for Failure Detection," in *Proc. Conf. Operating Systems Design and Implementation*, 2018, pp. 1–16.

[7] The LLVM Foundation, "The LLVM Compiler Infrastructure," https://llvm.org/.

[8] NVIDIA Corporation, "CUDA Toolkit Documentation v8.0," https://docs.nvidia.com/cuda/archive/8.0/, 2017.

[9] Y. Ozaki, S. Kanamoto, H. Yamamoto, and K. Kourai, "Detecting system failures with gpus and llvm," in *Proc. Asia-Pacific Workshop on Systems*, 2019, pp. 47–53.

[10] Intel, Hewlett-Packard, NEC, and Dell, "Intelligent Platform Management Specification Second Generation v2.0," 2004.

[11] W. Xu, "Bug 200027 - Kernel Hangs When Mouting a Crafted XFS Image," Kernel.org Bugzilla, 2018.

[12] J. McCalpin, "STREAM: Sustainable Memory Bandwidth in High Performance Computers," https://www.cs.virginia.edu/stream/.

[13] A. Bohra, I. Neamtiu, P. Gallard, F. Sultan, and L. Iftode, "Remote Repair of OS State Using Backdoors," in *Proc. Int. Conf. Autonomic Computing*, 2004.

[14] T. Garfinkel and M. Rosenblum, "A Virtual Machine Introspection Based Architecture for Intrusion Detection," in *Proc. Network and Distributed Systems Security Symp.*, 2003, pp. 191–206.

[15] D. Pelleg, M. Ben-Yehuda, R. Harper, L. Spainhower, and T. Adeshiyan, "Vigilant: Out-of-band Detection of Failures in Virtual Machines," *SIGOPS Operating Systems Review*, vol. 42, no. 1, pp. 26–31, 2008.

[16] C. Pham, Z. Estrada, P. Cao, Z. Kalbarczyk, and R. Iyer, "Reliability and Security Monitoring of Virtual Machines Using Hardware Architectural Invariants," in *Proc. Int. Conf. Dependable Systems and Networks*, 2014, pp. 13–24.

[17] N. Petroni, Jr., T. Fraser, J. Molina, and W. Arbaugh, "Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor," in *Proc. USENIX Security Symp.*, 2004.

[18] J. Wang, A. Stavrou, and A. Ghosh, "HyperCheck: A Hardware-assisted Integrity Monitor," in *Proc. Int. Symp. Recent Advances in Intrusion Detection*, 2010, pp. 158–177.

[19] A. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. Skalsky, "HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity," in *Proc. Conf. Computer and Communications Security*, 2010, pp. 38–49.