Detecting System Failures with GPUs and LLVM

doi: http://dx.doi.org/10.1145/3343737.3343749

Detecting System Failures with GPUs and LLVM

Yuichi Ozaki
Kyushu Institute of Technology
bushido@ksl.ci.kyutech.ac.jp

Sousuke Kanamoto
Kyushu Institute of Technology
k_sousuke@ksl.ci.kyutech.ac.jp

Hiroaki Yamamoto
Kyushu Institute of Technology
hiroaki@ksl.ci.kyutech.ac.jp

Kenichi Kourai
Kyushu Institute of Technology
kourai@ksl.ci.kyutech.ac.jp

ABSTRACT

Since system failures cause a huge financial loss, they should be detected as early and accurately as possible and then be recovered rapidly. To detect system failures, there are mainly two methods: *black-box* and *white-box monitoring*. However, external black-box monitoring cannot obtain detailed information on system failures, while internal white-box one is largely affected by system failures. This paper proposes *GPUSentinel* for more reliable white-box monitoring using general-purpose GPUs. In GPUSentinel, system monitors running in a GPU analyze main memory and indirectly obtain the state of the target system. Since GPUs are isolated from the target system, system monitors are not easily affected by system failures. For easy development of system monitors, GPUSentinel provides a development environment including program transformation with LLVM. In addition, it also provides reliable notification mechanisms to remote hosts. We have implemented GPUSentinel using CUDA and the Linux kernel and confirmed that GPUSentinel could detect three types of system failures.

ACM Reference Format:
Yuichi Ozaki, Sousuke Kanamoto, Hiroaki Yamamoto, and Kenichi Kourai. 2019. Detecting System Failures with GPUs and LLVM. In *10th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys '19), August 19–20, 2019, Hangzhou, China.* ACM, New York, NY, USA, 7 pages. https://doi.org/10.1145/3343737.3343749

1 INTRODUCTION

Recently, the scale of computer systems is being larger and the complexity is increasing. As a consequence, system failures are unavoidable. Once a system failure occurs, services provided by the system often 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 [2]. The users of such services can also suffer from some loss due to service unavailability. To reduce such loss, system failures should be detected as early and accurately as possible and then be recovered rapidly.

Traditional failure detection is mainly categorized into two methods: *black-box* and *white-box monitoring*. For external black-box monitoring, heartbeat monitoring of target hosts and services is often used. This method can monitor the target system even when system failures occur, but it is difficult to obtain detailed information on the target system. For internal white-box monitoring, system monitors run on top of the operating system (OS) or are embedded into the OS kernel. This method can detect system failures more accurately using the internal state of the target system, but it is largely affected by system failures because system monitors strongly depends on the target system.

In this paper, we propose *GPUSentinel* for more reliable white-box monitoring using general-purpose GPUs. Since GPUs can run code independently of CPUs and main memory on top of which the target system runs, they are not easily affected by system failures. In GPUSentinel, system monitors in a GPU analyze main memory using the knowledge of data structures used in the OS kernel and indirectly obtain the state of the target system. As such, they can use detailed information to detect system failures. For easy development of system monitors, GPUSentinel provides a development environment including program transformation using LLVM [14]. In addition, it provides reliable notification mechanisms of failure occurrences and root cause to remote hosts.

We have implemented GPUSentinel using CUDA [10]. 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 and the GPU driver. In addition, we have developed a framework called *LLView*, which transparently transforms the programs of system monitors so as to translate virtual addresses of OS data to GPU addresses. LLView also hides the differences between GPU programming and OS kernel programming. Using GPUSentinel, we have developed three system monitors for detecting CPU anomaly, out-of-memory, and deadlocks. Through our experiments, we confirmed that GPUSentinel could detect system failures successfully.

The organization of this paper is as follows. Section 2 describes issues of traditional failure detection. Section 3 proposes GPUSentinel 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 FAILURE DETECTION

To reduce a huge financial loss due to system failures, rapid recovery from system failures is important. For this purpose, it is necessary to detect system failures as early as possible after system failures occur. If possible, it is desirable to detect symptoms of system failures before systems completely stop services. In addition, it is necessary to detect system failures as accurately as possible. System administrators could not prevent next system failures unless they cannot identify failure types or their root causes. If false positives occur, system administrators may have to investigate the root causes of system failures that do not really occur and stop services for that.

Traditionally, there are mainly two methods for detecting system failures: *black-box* and *white-box monitoring*. An example of black-box monitoring is heartbeat monitoring of target hosts and services provided by target systems via networks. External monitors can examine not only the state of the entire systems but also the state of each service in a finer-grained manner by periodically connecting to all the services. If the responses are slow, they may be symptoms of system failures. If there are no responses, a system failure probably occurs. However, it is difficult to obtain detailed information on target systems when system failures occur because external monitors cannot access the internal state of the systems. Therefore, they cannot identify which types of failures occur or the root causes.

If target hosts equip with hardware monitoring such as IPMI [5], external monitors can obtain more detailed information even outside the target systems. They can examine the hardware state and use that information for failure detection. For example, CPU usage, the amount of disk access,

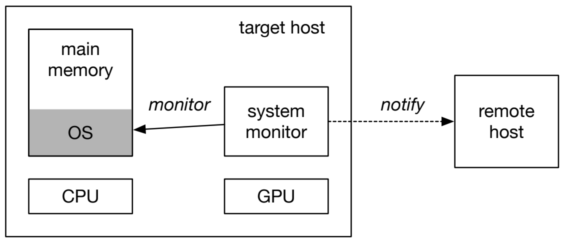


Figure 1: The architecture of GPUSentinel.

and the number of network packets help external monitors detect system failures more accurately than simple heartbeat monitoring. Similarly, if target systems run in virtual machines (VMs), external monitors can obtain the state of virtual hardware from the outside of the VMs. However, such extra information may be still insufficient to identify exact failure types and root causes.

In white-box monitoring, on the other hand, internal monitors run inside target systems and notify remote hosts of system failures. They can obtain more detailed information than black-box monitoring. In particular, when they are embedded into the OS kernel, they can identify failure types and root causes more easily. However, once a system failure occurs, internal monitors would not work correctly in a high probability. For example, they cannot detect system failures or identify the root causes if the OS kernel stops. Even if the OS kernel continues to run normally, internal monitors may be terminated by the OS kernel, e.g., the OOM killer in Linux, when system memory runs out.

3 GPUSENTINEL

In this paper, we propose GPUSentinel for achieving more reliable white-box monitoring by running system monitors in GPUs. Figure 1 shows the system architecture of GPUSentinel. In GPUSentinel, system monitors 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 graphics or computing, GPUSentinel is available by installing another GPU. To monitor the target system from a GPU, system monitors analyze OS data stored in main memory. Using detailed information at the OS level, GPUSentinel enables accurate failure detection.

Using GPUs for failure detection has three advantages. First, system monitors in a GPU can continue to run in a high probability even when system failures occur. This is because GPUs run independently of CPUs and main memory on top of which the target system runs. GPUs are usually used as co-processors and are controlled by CPUs, but GPUSentinel takes control of one dedicated GPU by running system monitors indefinitely. Second, a GPU enables various system monitors to efficiently run at the same time because it has many cores. Therefore, system monitors in a GPU can investigate various symptoms of system failures in parallel. In addition, one system monitor can rapidly examine the symptom of one system failure using many cores. Third, GPUs are general-purpose hardware and we can choose low-cost ones.

GPUSentinel can detect system failures that can be found from data stored in main memory. For example, system failures caused by running out system resources is detectable. If processes use a large amount of memory or memory leaks occur, the system cannot allocate necessary memory. Also, system failures caused by resource starvation is detectable. If all the CPUs cause deadlocks with spinlocks, the entire system freezes. To detect these system failures, system monitors in a GPU can examine the amount of free memory and consumed CPU time. If these values are abnormal, GPUSentinel can detect that state as symptoms of a system failure. Note that GPUSentinel cannot detect hardware failures because GPUs cannot access hardware except for main memory, e.g., CPUs and NICs.

To support easy development of system monitors running in GPUs, GPUSentinel provides a framework called *LLView*. When system monitors obtain OS data, they have to indirectly access main memory from a GPU and analyze data structures used in the OS kernel. In addition, the development of GPU programs is largely different from that of normal system programs. To solve these issues, LLView transparently transforms the programs of system monitors using LLVM [14] so that developers are not aware of such indirect memory access. Using a dedicated development environment including this program transformation, LLView enables developers to write the programs of system monitors as if they develop OS kernel modules.

GPUSentinel provides two reliable notification mechanisms of system failures to remote hosts. Since the OS kernel may stop on system failures, it is not guaranteed that notification using the OS functions is available. In GPUSentinel, system monitors in a GPU can directly write data to the VRAM allocated in main memory and display information on the screen. Using remote console in IPMI [5] and KVM switches, remote system administrators can receive notification of system failures. In addition, GPUSentinel can notify remote hosts of system failures more flexibly using RDMA. Remote hosts send requests to system monitors by directly writing data to GPU memory and receive responses by directly reading data stored in GPU memory.

4 IMPLEMENTATION

We have implemented GPUSentinel using CUDA 8.0 [10] and LLVM 5.0 [14]. To enable GPUs to monitor OS data, we have modified Linux kernel 4.4 and NVIDIA GPU driver 375.66. System monitors in a GPU are implemented as one GPU kernel.

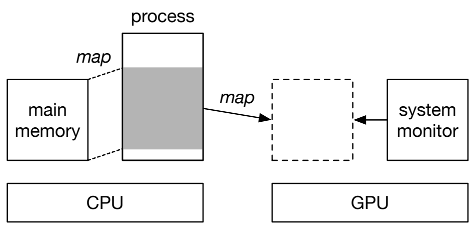


Figure 2: Mapping main memory with mapped memory.

4.1 Mapping Main Memory

GPUSentinel enables system monitors in a GPU to autonomously access main memory using DMA. In CUDA, DMA transfers can be initiated explicitly only from the CPU side. After a system failure, the target system may not be able to perform such DMA transfers. Therefore, GPUSentinel uses the *mapped memory* mechanism provided by CUDA. Mapped memory is used to map main memory onto the GPU address space and make it accessible from GPU kernels. When a GPU kernel accesses the mapped area, a GPU performs DMA transfers transparently. Since GPUSentinel sets up mapped memory before a system failure, system monitors in a GPU can access main memory even after the target system does not work correctly.

To use mapped memory for the entire main memory, GPUSentinel first maps main memory onto a process address space, as illustrated in Fig. 2. This is because CUDA can map only process memory onto the GPU address space. However, if GPUSentinel simply maps the entire main memory, free memory runs out and the system stops working. When CUDA maps the mapped main memory 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. It should be noted that `/dev/mem` can be mapped without any pages being in use but cannot be pinned.

To solve this problem, we have modified the memory management of the Linux kernel so as to provide a special device called `/dev/pmem`. When `/dev/pmem` is mapped onto the process address space, the modified Linux kernel does not increase the reference count of each memory page to prevent the page from being in use. The reference count of a free memory page is kept to zero. This does not raise a problem because the process itself does not use the mapped main memory in GPUSentinel. In addition, when CUDA pins the mapped memory pages, the modified Linux kernel does not lock them.

To enable such memory pages to be unmapped correctly, we have modified not only the Linux kernel but also the GPU driver. When the mapped memory pages are unmapped, they are usually unpinned and unlocked. Then, the reference count of each pages is decreased. Since the main memory mapped with `/dev/pmem` is not locked, GPUSentinel does not unlock those pages. Similarly, it does not decrease the reference count. Note that the source code of the NVIDIA GPU driver is basically closed-source but the code for memory pinning is open.

To work around the limitation of CUDA and enable the entire main memory to be mapped, GPUSentinel hooks the sysinfo system call. CUDA limits the size of mapped memory only to a bit smaller amount of memory than the size of main memory. The reason is probably that CUDA prevents the system from stopping by pinning all the memory pages. This is unnecessary limitation for GPUSentinel because mapped memory pages are not really pinned. Therefore, GPUSentinel intercepts the sysinfo system call and returns a bit larger size as the size of main memory. As a result, it can map the entire main memory.

4.2 Transparent Address Translation

To transparently translate virtual addresses of the OS kernel into GPU addresses, LLView compiles the programs of system monitors 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 address translation and executes the `load` instruction for the translated address. The `g_map` function returns the passed address as is if the passed address is not the virtual address of the OS kernel.

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 `bitcast` and `call` instructions for invoking the `g_map` function and inserts them 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 system monitors 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.

4.3 Development Environment

LLView enables developers to write the programs of system monitors as OS kernel modules using the source code of the Linux kernel. Let us consider an example of a system monitor that obtains process information. This system monitor traverses the process list, which is a circular list starting with `init_task`, using the `list_entry` macro. During that traversal, it obtains information such as the IDs and names of all the processes through the `task_struct` structure, which is defined in `linux/sched.h`.

GPUSentinel enables developers to write the programs of system monitors in C. CUDA programs consist of device code running in GPUs and host code running in 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.

For LLView, we have modified the Clang compiler front end 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 follows. First, LLView compiles device code using modified clang. It applies our passes to the generated bitcode using opt. Then, it creates 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 `threadIdx` variable that returns a thread index in a block. For this variable, LLView provides the C function called `get_thread_id` that returns a thread ID using that variable.

4.4 Failure Notification

During normal time, system monitors in a GPU notify a host process in the target system of failure occurrences and root causes using a ring buffer. In preparation for system failures, GPUSentinel provides a mechanism called *direct VRAM output* for failure notification without relying on the target system. This mechanism enables system monitors to display images and characters on the screen by writing graphics data to the VRAM allocated in main memory. For characters, the font data stored in the Linux kernel is used. One limitation is that this mechanism requires a GPU that allocates VRAM in main memory.

Therefore, GPUSentinel also provides a mechanism called *direct GPU communication* to notify a remote host of failure information. This mechanism uses GPUDirect RDMA [11], which enables a remote host to directly access GPU memory without CPU intervention. GPUSentinel maps GPU memory into the physical address space of the target system using the GPUDirect mechanism in advance and uses the RDMA feature provided by NICs. A dedicated thread in a GPU polls GPU memory and waits for a remote host to write a request with an RDMA write. If it receives a request, it writes requested information to GPU memory. Then, the remote host reads that information by polling GPU memory with RDMA reads.

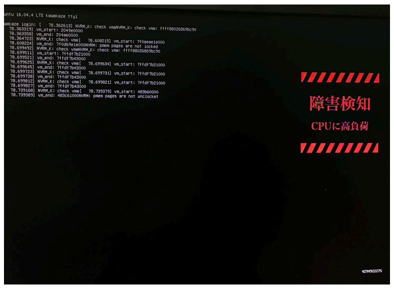
5 EXPERIMENTS

We conducted several experiments to confirm that system monitors in GPUSentinel could detect system failures. In this experiment, we used a PC with an Intel Core i7-7700 processor, 8 GB of DDR4-2400 memory, and two GPUs of NVIDIA GeForce GTX 960 and Intel HD Graphics 630. We ran Linux 4.4.67, NVIDIA GPU driver 375.66, and CUDA 8.0.61.

5.1 Detection of CPU Anomaly

We have developed a system monitor that calculated CPU utilization every second in a GPU. This system monitor detected anomaly if the utilization of all the CPUs exceeded 90% for more than 5 seconds. To confirm that this system monitor could detect CPU anomaly, we ran a program that intensively used CPUs in the target system. This program ran 4 processes for a while, stopped them, and ran 8 processes. After a while, the system monitor displayed the red image for failure notification, as shown in Fig. 3.

Figure 4 shows changes in CPU utilization calculated by this system monitor. After the program increased the number of processes at time 17 second, the system monitor could



```
untu 16.04.4 LTS kawakaze tty1

wakaze login: [   78.362613] NVRM_K: check vmaNVRM_K: check vma: ffff88026869bc90
  78.363319] vm_start: 2049e0000
  78.363358] vm_end: 204ae0000
  78.364723] NVRM_K: check vma[   78.608215] vm_start: 7f0aeae1e000
  78.608224] vm_end: 7f0d69e1e000NVRM: pmem pages are not locked
  78.699492] NVRM_K: check vmaNVRM_K: check vma: ffff88026869bc90
  78.699511] vm_start: 7ffdf7b21000
  78.699521] vm_end: 7ffdf7b43000
  78.699625] NVRM_K: check vma[   78.699634] vm_start: 7ffdf7b21000
  78.699645] vm_end: 7ffdf7b43000
  78.699723] NVRM_K: check vma[   78.699731] vm_start: 7ffdf7b21000
  78.699738] vm_end: 7ffdf7b43000
  78.699812] NVRM_K: check vma[   78.699821] vm_start: 7ffdf7b21000
  78.699827] vm_end: 7ffdf7b43000
  78.739168] NVRM_K: check vma[   78.739379] vm_start: 483b60000
  78.739389] vm_end: 483c61000NVRM: pmem pages are not unclocket
```

Figure 3: Failure notification on the screen.

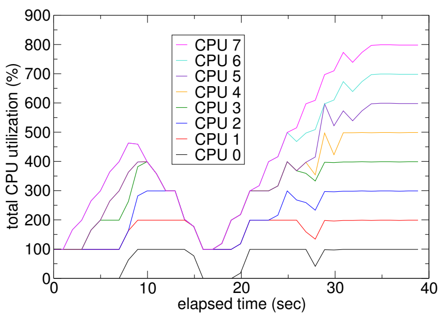


Figure 4: Changes in CPU utilization.

detect CPU high loads in 22 seconds. Note that this does not always mean a system failure because normal heavyweight tasks may run. Therefore, system administrators need to consider other system states as well.

5.2 Detection of Out-of-memory

We have developed a system monitor that obtained the amounts of free memory and swap space and calculated the ratios to the total amounts, respectively. This system monitor detected out-of-memory if both the ratios were less than 30%. To confirm that this system monitor could detect out-of-memory, we ran a program that used a large amount of memory in the target system. This program allocated 20 GB of memory in total and write data to it. After a while, the response of the target system slowed down. Finally, this system monitor displayed the notification image and the ID of the process consuming the largest amount of memory.

Figure 5 shows changes in amounts of free memory and swap space. The log output by the host process temporarily stopped at time 1 second because the host process slowed down extremely and could not write a log to a file. From this point, the system monitor could detect the failure in 57 seconds.

5.3 Detection of Deadlocks

We have developed a system monitor that calculated in-kernel CPU utilization per process and obtained the number

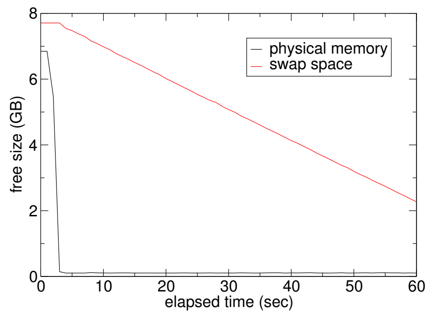


Figure 5: Changes in free memory and swap space.

of context switches. This system monitor detected a deadlock if this CPU utilization was larger than 95% for more than 5 seconds or there were no context switches for 1 second. To confirm that this system monitor could detect a deadlock, we created a Linux kernel module that ran two threads acquiring two spinlocks, respectively. This module made the specified number of threads wait for the release of these spinlocks. After we ran the kernel module using 2 threads, the notification image was shown on the screen in 5.1 seconds. When we ran the kernel module using 8 threads, the entire system stopped the response immediately and the notification image was shown in 1.7 seconds. At this time, the number of context switches did not change for 1 second. From this result, it was shown that GPUSentinel could detect even a system hang.

6 RELATED WORK

SHFH [16] detects system failures using only minimum performance metrics. It classifies the root causes of system failures into six types, e.g., infinite loops, deadlocks, and resource shortage. Then it finds performance metrics on CPUs, processes, memory, and disk I/O to detect system failures. SHFH is implemented as a real-time user process and a kernel module. In a normal state, the process monitors the system in a lightweight manner. When the process detects symptoms of a system failure, the kernel module investigates the system state in further detail. Therefore, SHFH cannot detect system failures that make the OS kernel hang.

For reliable failure detection, Falcon [8] runs spies in different layers of the system such as the OSes and network switches. Using the spies, it detects the failure of a higher layer from the lower layer, but the target is only a crash failure. Pigeon [7] enables applications to obtain more information about failures by running sensors in system components. Panorama [4] can detects gray failures, whose manifestation is subtle, by automatically inserting report-detection code into applications. Since these frameworks depend on target components, they are easily affected by component failures.

For security, various mechanisms have been proposed to monitor the system without being affected by attacks. Using such mechanisms, it is possible to achieve failure detection that is not affected by system failures. Copilot [12] obtains the contents of kernel memory using a dedicated PCI card. SPE Observer [6] runs a monitoring system on an SPE in a Cell/B.E processor, which can be isolated from a PPE running the OS. Also, monitoring systems using Intel processors have been proposed. HyperCheck [15] runs a network driver in System Management Mode (SMM), transfers memory contents to a remote host, and monitors them. HyperSentry [1] enables a monitoring agent to securely run in the target hypervisor. Flicker [9] uses Intel TXT and runs a monitoring system securely.

When the target system runs in a VM, monitoring systems can obtain OS data outside the VM using a technique called VM introspection [3]. If this technique is applied to the detection of system failures, the detection system is not easily affected by system failures and has high detection ability. However, this technique cannot be used for systems that do not use VMs. GPUSentinel applies VM introspection to not-virtualized systems by using GPUs.

7 CONCLUSION

This paper proposes GPUSentinel for reliable failure detection by running system monitors in GPUs. To detect system failures, GPUSentinel monitors OS data in main memory from GPUs. It uses modified Linux kernel and GPU driver to enable the entire main memory to be mapped onto the GPU address space without pinning. To support easy development of system monitors, GPUSentinel provides a framework called LLView, which enables developers to use the source code of the OS kernel as much as possible. In addition, GPUSentinel provides reliable failure notification mechanisms. Using GPUSentinel, we have developed three system monitors and confirmed that they could detect system failures.

One of our future work is to support all of the performance metrics proposed in SHFH [16]. Then, we need to detect system failures more accurately on the basis of more information to reduce false positives. Another direction is to recover from system failures by rewriting OS data in main memory. For example, CPU high loads can be mitigated by temporarily removing processes from run queues of the process scheduler [13].

ACKNOWLEDGMENT

The research results have been achieved by the "Resilient Edge Cloud Designed Network (19304)," the Commissioned Research of National Institute of Information and Communications Technology (NICT), Japan.

REFERENCES

[1] A. Azab, P. Ning, Z. Wang, X. Jiang, X. Zhang, and N. Skalsky. HyperSentry: Enabling Stealthy In-context Measurement of Hypervisor Integrity. In *Proceedings of ACM Conference on Computer and Communications Security*, pages 38–49, 2010.

[2] 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.

[3] T. Garfinkel and M. Rosenblum. A Virtual Machine Introspection Based Architecture for Intrusion Detection. In *Proceedings of Network and Distributed Systems Security Symposium*, pages 191–206, 2003.

[4] P. Huang, C. Guo, J. Lorch, L. Zhou, and Y. Dang. Capturing and Enhancing in Situ System Observability for Failure Detection. In *Proceedings of the 12th USENIX Conference on Operating Systems Design and Implementation*, pages 1–16, 2018.

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

[6] K. Kourai and T. Nagata. A Secure Framework for Monitoring Operating Systems Using SPEs in Cell/B.E. In *Proceedings of Pacific Rim International Symposium Dependable Computing*, pages 41–50, 2012.

[7] J. Leners, T. Gupta, M. Aguilera, and M. Walfish. Improving Availability in Distributed Systems with Failure Informers. In *Proceedings of the 10th USENIX Conference on Networked Systems Design and Implementation*, pages 427–442, 2013.

[8] J. Leners, H. Wu, W. Hung, M. Aguilera, and M. Walfish. Detecting Failures in Distributed Systems with the Falcon Spy Network. In *Proceedings of the 23rd ACM Symposium on Operating Systems Principles*, pages 279–294, 2011.

[9] J. McCune, B. Parno, A. Perrig, M. Reiter, and H. Isozaki. Flicker: An Execution Infrastructure for TCB Minimization. In *Proceedings of European Conference on Computer Systems*, pages 315–328, 2008.

[10] NVIDIA Corporation. CUDA Toolkit Documentation v8.0. https://docs.nvidia.com/cuda/archive/8.0/.

[11] NVIDIA Corporation. Developing a Linux Kernel Module Using RDMA for GPUDirect. Technical Report TB-06712-001 v10.1, NVIDIA, 2019.

[12] N. Petroni, Jr., T. Fraser, J. Molina, and W. Arbaugh. Copilot – a Coprocessor-based Kernel Runtime Integrity Monitor. In *Proceedings of USENIX Security Symposium*, 2004.

[13] H. Tadokoro, K. Kourai, and S. Chiba. A Secure System-wide Process Scheduler across Virtual Machines. In *Proceedings of Pacific Rim International Symposium Dependable Computing*, 2010.

[14] The LLVM Foundation. The LLVM Compiler Infrastructure. https://llvm.org/.

[15] J. Wang, A. Stavrou, and A. Ghosh. HyperCheck: A Hardware-assisted Integrity Monitor. In *Proceedings of International Symposium Recent Advances in Intrusion Detection*, pages 158–177, 2010.

[16] 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 *Proceedings of International Symposium on Software Reliability Engineering*, pages 141–150, 2012.