博士論文

A Study on Operating System Virtualization Optimized for Functional Requirements

(機能要件に最適化されたオペレーティングシステムの仮想化に関する研究)

味曽野 雅史

Dissertation

A Study on Operating System Virtualization Optimized for Functional Requirements

Masanori Misono

Email: misono@os.ecc.u-tokyo.ac.jp

Thesis Committee

Supervisor and Chair: Dr. Takahiro Shinagawa
Associate Professor of Information Technology Center, The University of Tokyo

Examiner: Dr. Shigeru Chiba
Professor of Department of Creative Informatics, Graduate School of Information Science and Technology, The University of Tokyo

Examiner: Dr. Hiroshi Nakamura
Professor of Department of Information Physics and Computing, Graduate School of Information Science and Technology, The University of Tokyo

Examiner: Dr. Hideki Takase
Associate Professor of Department of Information Physics and Computing, Graduate School of Information Science and Technology, The University of Tokyo

Examiner: Dr. Yuji Sekiya
Professor of Center for Education and Research in Information Science and Technology, Graduate School of Information Science and Technology, The University of Tokyo

The thesis is submitted to the Department of Information Physics and Computing, Graduate School of Information Science and Technology, The University of Tokyo in partial fulfillment of the requirements for the degree of Doctor of Philosophy in the field of Information Science and Technology.

Abstract

Virtualization technology introduces a new abstraction layer between an OS and the hardware, allowing new functionalities to be added to the OS transparently. A general-purpose hypervisor can provide richer functions, but it also requires more overhead. We can reduce virtualization overhead by specializing in a specific function.

In this thesis, we studied optimizing virtualization for functional requirements through several use cases. First, we optimized nested virtualization for hypervisor device drivers testing. Focusing on the fact that the security features required by normal virtualization are unnecessary for testing purposes, we improved virtualization performance by eliminating them. Second, we presented the efficient IOMMU virtualization method for device protection. We achieved higher performance than a regular IOMMU virtualization by only shadowing the necessary area for protection. Third, we presented a detailed performance evaluation of the NUMA-visible virtual machines on Linux. The evaluations revealed several problems with vNUMA scheduling and we fixed the incorrect paravirtualization feature that caused severe performance degradation. Finally, we proposed a method to improve the flexibility of hypervisors without compromising performance by using a secure and lightweight language virtual machine. An example of the use of the language virtual machine, we presented a source-side DDoS prevention scheme using virtualization. Through these use cases, we showed several effective ways of optimizing virtualization for functional requirements.

Acknowledgement

I would like to thank my advisor, Associate Prof. Takahiro Shinagawa, for his kind and careful guidance for six years, including the master's course. Also, I would like to express my sincere gratitude to the examiners, Prof. Shigeru Chiba, Prof. Hiroshi Nakamura, Associate Prof. Hideki Takase, and Prof. Yuji Sekiya, for their detailed comments on my thesis.

As the first master's student in Shinagawa Laboratory, I have experienced many things, and the people in the laboratory helped me in many ways. In particular, Dr. Keiichi Matsuzawa and Dr. Takaaki Fukai often discussed with me and taught me many technical and academic things. Tomoyuki Nakamura, Ryosuke Yasuoka, Satoru Takekoshi, Iori Yoneji, Kohei Azuma, Yoshida Kaito, Hu Siyi, Masahiro Ogino, Suzuki Yuki, Toshiki Hatanaka, Hiromu Yamasaki, Liao Zihao, Itta Toda, Takashi Ogino, Hidehito Yabuuchi, Ryo Hayashi, Akira Moroo, Yosuke Ozawa, Shotaro Gotanda, Junnosuke Mizutani, Shu Anzai, Shoi Takahashi, Rikima Mitsuhashi and Aoki Katsunori, I also would like to thank you for their valuable discussions and supports.

In my university life, I was especially helped by my fellow transfer students from technical colleges. Masaru Matsunaga, Kenta Watanabe, Taihei Oki, Iori Yanokura, Takuma Yoshitani, Tan Van Vu, Haruki Ejiri, Shuhei Yoshida, Yuto Nakajima, Yuto Kondo and Hiroki Kuga, they are much better than I am, and they are always inspiring. I would also like to thank the people in Matsuo Labolatry, where I belong in undergraduate. Especially Prof. Yutaka Matsuo and Assistant Prof. Yusuke Iwasawa occasionally discussed me even after graduating from undergraduate school. Although the content of my research differed between graduate school and undergraduate school, I am certain that the experiences I had as an undergraduate are very useful to me today.

Outside of the university, I would like to thank Mr. Hideki Eiraku, who taught me about many detailed things of virtualization technology of x86. They were very helpful to implement my idea. I would also like to thank lecturers from my days of National Institute of Technology, Gunma Collage. In those days, at first, I was thinking of starting working after graduation, but a lecturer Dr. Keita Ushida (currently Associate Prof. at Kogakuin University), a graduate of the Graduate School of Information Science and Technology, taught me about the University of Tokyo. I was also influenced by Prof. Toshiaki Ohmameuda and Prof. Hideaki Ujino, who are also graduates of the university. I was very fortunate to be able to talk to Ph.D. researchers when I was a teenager. There is no doubt that studying under them had an impact on going on to university. I would also like to thank Prof. Yoshiaki Hachitori (currently at Matsuyama University) for always giving me advices and encouraging me to go on to higher education.

Last but not least, I am grateful for devoted support from my family. As the youngest of six siblings, I have always been indebted to my brothers, sisters, and parents. They never interfered with my career path and always supported me. During my university life, especially graduate school, my mother and sisters provided me with financial support to obtain the degree. I would like to express my gratitude to all of you once again.

March 2022, Masanori Misono

Publications

Parts of this thesis are based on the following publications. IEEE holds the copyright of the publications [ii] and [iii], potions of which are reprinted in this thesis in accordance with its copyright policy [179]. The authors hold the copyrights of the publications [i] and [iv].

International Conference (peer-reviewed)

[i] **Masanori Misono**, Toshiki Hatanaka and Takahiro Shinagawa.
DMAFV: Testing Device Drivers against DMA Faults. In *Proceedings of the 37th ACM/SIGAPP Symposium On Applied Computing (SAC 2022)*, Apr 2022 (To Appear). doi:10.1145/3477314.3507082

[ii] **Masanori Misono**, Masahiro Ogino, Takaaki Fukai and Takahiro Shinagawa.
FaultVisor2: Testing Hypervisor Device Drivers against Real Hardware Failures. In *Proceedings of the 10th IEEE International Conference on Cloud Computing Technology and Science (CloudCom'18)*, pp.204-211, Dec 2018. doi:10.1109/CloudCom2018.2018.00048

[iii] **Masanori Misono**, Kaito Yoshida, Juho Hwang and Takahiro Shinagawa.
Distributed Denial of Service Attack Prevention at Source Machines.
In *Proceedings of the 16th IEEE International Conference on Dependable, Autonomic and Secure Computing (DASC'18)*, pp.488-495, Aug 2018. doi:10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00096

International Conference Poster (peer-reviewed)

[iv] **Masanori Misono** and Takahiro Shinagawa.
OS Independent Fuzz Testing of I/O Boundary.
In *Proceedings of the 2021 ACM Conference on Computer and Communications Security (CCS'21)*, Nov 2021. doi:10.1145/3460120.3485359

The authors received following awards regarding the publications.

1. Best Paper Award at IEEE CloudCom 2018 (Acceptance Ratio: 19.8%. Only one paper is chosen.)

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation

1.1.1 Virtualization Technology

Today, virtualization technology[1] is utilized in a variety of systems. One of the typical applications that utilize the technology is cloud services. They provide virtual machines (VMs) without being aware of physical hardware limitations. By using cloud services, users can utilize computational resources without being aware of actual machines. On the other hand, virtualization technology is sometimes perceived to run multiple virtual machines on a single physical machine like cloud services, but this is a limited view.

The history of virtualization dates back to around the 1970s. The classic definition of virtualization is that "a virtual machine is an efficient, isolated duplicate of a real machine." [1] A hypervisor (also called a virtual machine monitor; VMM for short) runs a virtual machine. The hypervisor provides the virtual devices necessary for the operation of the virtual machine and emulates specific instructions to be executed by the virtual machine if necessary. When it comes to running multiple virtual machines on a single physical machine, virtualization technology in the cloud follows this classical definition.

However, today's virtualization technology is not confined to this classical definition of virtualization. What is important about virtualization is that it introduces a new abstraction layer between the operating system and the hardware. The hypervisor can use this abstraction layer to add new functionality to the OS. One of the primary advantages of using hypervisors to add functionality is their transparency. A hypervisor can transparently trap and modify OS activities such as memory accesses and device manipulations In addition, since the hypervisor runs in a more privileged state than OS, virtualization is suitable for security enhancement because the hypervisor can continue processing even if the OS is attacked.

[1] The term "virtualization" and "virtual machine" have been used in several contexts. A language's virtual machine is dedicated to processing the language like Java Virtual Machine (JVM) [207]. Recently, "virtualization" may also refer a container technology (lightweight virtualization), which utilize Operating System functionalities for process isolation, such as Docker [174]. In this thesis, the term virtualization refers to Operating System virtualization [117] unless explicitly noted.

1.1.2 Example Use Cases of Virtualization

The functions realized by hypervisors range from verifying the integrity of the kernel to system maintenance functions. Here, we briefly introduce several use cases.

Operating System Monitoring: A hypervisor has a higher privilege level than a virtual machine and can observe the virtual machine's behavior. [10] proposed virtual machine introspection (VMI), which analyzes the behavior of applications, such as malware from outside the OS, by referring to the memory in the VM from the hypervisor. Some malware may change its behavior when it detects that it is being monitored. VMI reduces the possibility of being detected by the monitored target by monitoring from outside the OS. As related works, there are also hypervisor-based forensics [42, 121, 125], intrusion detection [46], and kernel integrity verification [25].

I/O Enhancement: The hypervisor is capable of intercepting the I/O of the VM. I/O functions can be added transparently by modifying the whole or part of the I/O processing using this property. Some of the exmaples include applying VPN transparently [35], background storage encryption [57, 58], malware detection [59] and rootkit detection [40] through disk IO monitoring, and non-volatile memory write protection [119]. The parapass-through architecture [35] reduces overall virtualization overhead by only virtualizing certain I/Os and letting pass through others.

Maintenance: Virtualization is also used for machine maintenance. To reduce downtime during machine maintenance, Microvisor [13] uses virtualization to launch the maintenance OS while keeping running a OS. BMCast [94] proposes a method of transparently caching the results of network boot to local disks to speed up the provisioning of bare-metal clouds. [89] uses a dedeicated thin hypervisor to achieve migration in bare-metal clouds.

1.1.3 Overhead of Virtualization

As stated above, there are many use cases achieved by virtualization. However, the introduction of a hypervisor inevitably imposes an operational overhead on the hypervisor. The operational overhead of virtualization depends on how much the system is virtualized. In this thesis, we call this concept the degree of virtualization. Figure 1.1 shows the overview of the degree of virtualization. First, a bare-metal machine without virtualization has the lowest degree of virtualization. On the other hand, a conventional general-purpose hypervisor, which is designed to run multiple virtual machines, has a high degree of virtualization. The higher the degree of virtualization, the more functions can be added by virtualization, but this also increases the overhead. When adding function with virtualization, it does not necessarily require the rich functionality of a general-purpose hypervisor. Several studies have attempted to achieve bare-metal like performance by limiting the virtualization functions. For example, some studies have used the parapass-through architecture [35], which virtualizes only a portion

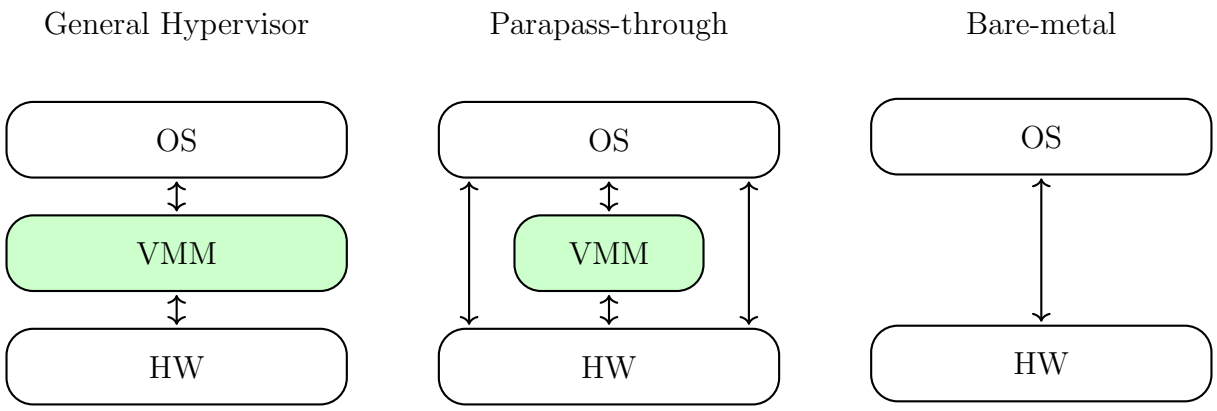


Figure 1.1: The Degree of the Virtualization

of the devices, to reduce the virtualization overhead.

Figure 1.2 shows the performance overhead comparison between a general-purpose hypervisor (KVM [21]) and a parapass-through hypervisor (BitVisor [35]) in SPEC2017 benchmark[2]. That experiment was performed using an Intel Core i7-4790K processor with hyper-threading disabled and 16 GB memory. We created a single VM and allocated the same number of CPU cores and amount of memory as the host machine to the VM. In this experiment, we are not running any programs other than the virtual machine, so ideally, the performance on the VM will be equivalent to that of bare metal. Here, with KVM, seven out of 19 workloads showed greater than 10% execution time overhead compared to the baremetal, and the maximum overhead was 21%. On the other hand, as for BitVisor, only two workloads demonstrate greater than 10% overhead, and most workloads have less than 5% overhead.

There are several reasons that a KVM has a higher overhead. One of the main reasons is the overhead of memory virtualization. The hypervisor manages the memory used by the VM, and the VM manages the memory used by itself. Therefore, memory management is performed by both the hypervisor and the VM.

The same dual management also exists for CPUs. This dual management causes several problems known as double scheduling problems. One of the typical double scheduling problems is the lock holder preemption problem. If a vCPU running a thread that holds a lock is preempted, other vCPUs that are waiting for the lock will wait until the vCPU with the lock is scheduled again. In addition, cache utilization efficiency decreases when vCPUs are scheduled on various CPUs.

Another factor that contributed to overhead was interrupts including periodic timer. When an interrupt occurs while executing a VM, a context switch happens, and the hypervisor's interrupt handler is executed. Then, the hypervisor will return a VM. When returning a VM,

[2]The experiment result is excerpt from [138] (© 2018 IEEE). The author conducted the experiment.

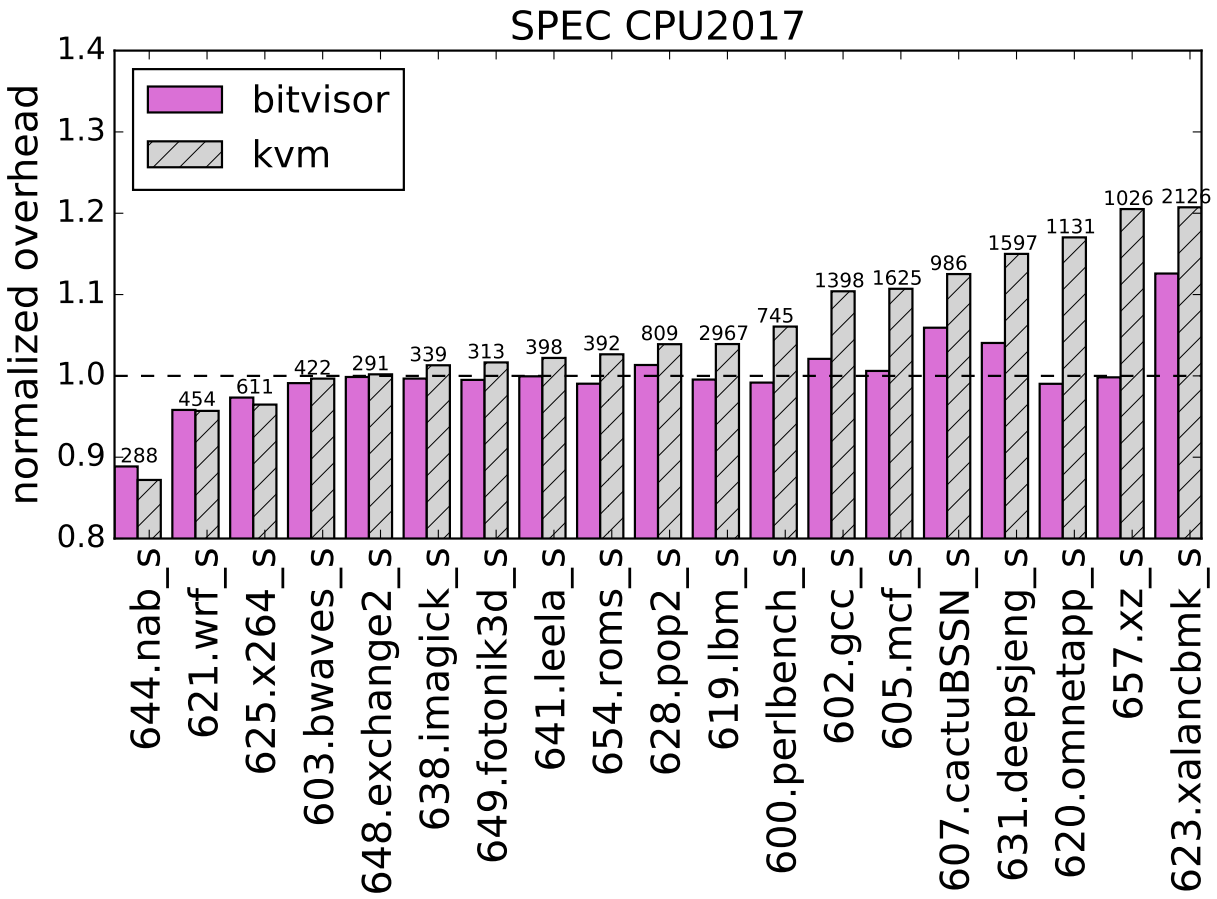


Figure 1.2: Comparison between a general purpose hypervisor (KVM) and a parapass-through hypervisor (BitVisor) in SPEC CPU2017 (CPU and memory intensive workloads). The baseline is a baremetal machine. The numbers above bars are baseline runtimes (sec). The lower is better. (© 2018 IEEE)

it may insert interrupts into the VM as needed. The context switch between the VM and the hypervisor will degrade the performance. This becomes especially a problem when using a high-performance device such as NIC whose bandwidth is more than 10Gbps, and recent fast storage devices like NVMe.

A parapss-through hypervisor (BitVisor) solves these problems by adopting the strategy of supporting only one VM and making most operations pass through. This removes most memory operations and CPU scheduling on the hypervisor side. Most interrupts are also directly delivered into a VM without context switches. As a result, the overhead of BitVisor is smaller than that of KVM.

1.2 Research Objectives and Overview

It is important to reduce the overhead for applying virtualization to real systems. Figure 1.2 shows that a virtual machine performance can be greatly improved by removing unnecessary functions and optimizing virtualization for functional requirements. In this thesis, we study optimizing virtualization for functional requirements from the following perspectives.

1. Optimizing Virtualization for Adding Functionality to a Hypervisor: Adding functionality using virtualization can also be applied to a hypervisor itself. By running a hypervisor on top of a hypervisor, the lower hypervisor can modify some of the processing of the upper hypervisor. However, current CPUs do not natively support running a hypervisor on top of a hypervisor. Nested virtualization [37] solves this problem by having the hypervisor emulate some parts of the CPU virtualization instructions. However, normal nested virtualization includes the functions of supporting multiple hypervisors and therefore is known for its overhead. On the other hand, for some applications, only one hypervisor needs to be supported. What kind of optimization can be done when we consider nested virtualization, where only one hypervisor support is required?

2. Virtualization for IOMMU Protection: IOMMU is a device that manages to DMA (Direct Memory Access) address transformation. IOMMU can limit the area that devices can DMA. Originally, IOMMU was used to pass through devices safely to virtual machines. It has also been used to protect OS from malicious devices in recent years. On the other hand, with the evolution of devices in recent years, methods for verifying OS consistency and forensics using external devices have emerged. However, if the IOMMU is enabled in the environment, an attacker may use IOMMU to disable the device's functionality. This attack can be defended against by virtualizing the IOMMU in the hypervisor (vIOMMU), but regular vIOMMU is slow because it virtualizes the entire IOMMU functionality. What optimizations could be made by specializing in IOMMU protection?

3. Efficient Virtualization on NUMA machine: NUMA (Non-Uniform Memory Ac-

cess) is a memory topology where the CPU and memory are divided into pairs called nodes, and the speed of memory access from the CPU on one node differs from that on other nodes. The host's NUMA configuration is hidden to the guest virtual machine in a normal virtualization environment. As a result, it is difficult to extract NUMA performance from the guest. The solution to this problem is to show NUMA to the guest. This is known as a virtual NUMA (vNUMA) or a NUMA-visible virtual machine. Although several studies have reported the usefulness of vNUMA performance, few reports have evaluated vNUMA on a variety of workloads. Is an existing hypervisor able to extract NUMA-like performance with vNUMA?

4. Improving Hypervisor Flexibility while Keeping Safety and Performance: Updating hypervisor functions usually require pausing virtual machines running on it. Suppose some of the hypervisor's functions were performed using a dynamic language virtual machine. In that case, the processing of the hypervisor can be changed by modifying the script executed by that language virtual machine. However, what kind of configuration is necessary to safely and efficiently execute the processing by the language processor in the hypervisor?

To answer these questions, we conducted the following research.

1.2.1 Chapter 3. Nested Virtualization for Hypervisor Device Driver Testing

We apply a device driver testing method using fault injection by a hypervisor to the hypervisor's own device drivers inspection. Focusing on the fact that the security features required by normal virtualization are unnecessary for testing purposes, we improve the performance by removing them.

1.2.2 Chapter 4. IOMMU Virtualization for Device Protection

We presents efficient IOMMU virtualization method dedicated for device protection. It achieves higher performance than conventional IOMMU virtualization by only virtualizing necessary parts of IOMMU for device protection and passing through others.

1.2.3 Chapter 5. Investigating and Improving Scheduling Performance of NUMA-visible Virtual Machines

We evaluated the scheduling performance of a NUMA-visible virtual machine on Linux using various benchmarks. We found several problems that cause severe performance degradation due to a paravirtualization function which is desirable for non-NUMA-visible VMs. We propose the fix and show the effectiveness of the proposed method.

1.2.4 Chapter 6. Improving Hypervisor's Flexibility with Safe and Lightweight Language VM

We propose a method to improve the flexibility of hypervisors without compromising performance by using a secure and lightweight language virtual machine. An an example of the use of the language virtual machine, we present DDOS prevention scheme using virtualization.

1.3 Contributions

The main contributions of this thesis are as follows.

1. We propose several methods to optimize virtualization for three use cases: (1) Nested virtualization for hypervisor device drivers testing, (2) IOMMU virtualization for device protection, and (3) NUMA-visibale virtualization.

2. We propose using a safe and lightweight language virtual machine in a hypervisor to improve flexibility while keeping the performance. An an example of the use of the language virtual machine, we present DDOS prevention scheme using virtualization.

3. We implement the proposed methods and perform detailed experiments and show the usefulness of the proposed methods.

1.4 Thesis Organization

The remainder of this thesis is organized as follows. First, chapter 2 summarizes the related works to this thesis. Chapter 3 shows how we optimize nested virtualization for hypervisor device drivers testing. Chapter 4 presents efficient IOMMU virtualization method dedicated for device protection. In chapter 5, we evaluate the scheduling performance of NUMA-visible virtual machines on Linux and presents the several fixes for performance improvements. Chapter 6 propose a method to improve the flexibility of hypervisors without compromising performance by using secure and lightweight language processing systems. Finally, Chapter 7 concludes the thesis.

2 Related Works

This chapter presents the prior works related to this thesis. First we present optimization methods used in virtualization to improve performance. We also present the double scheduling problems, which is one of the main performance problems in virtualized environments, and its mitigation methods.

2.1 Optimizing Virtualization Performance

2.1.1 Paravirtualization

Paravirtualization optimizes virtualization processing by having a hypervisor and virtual machines work together. Originally, paravirtualization was used to realize virtualization where CPUs provide no hardware-assisted virtualization technology [9]. In such an environment, paravirtualization was essential to trap sensitive instructions executed by virtual machines. Today, most CPUs support hardware-assisted virtualization technology (e.g., Intel VT-x and AMD-v). Therefore, it is possible to run a virtual machine without paravirtualization. However, paravirtualization is still important for performance optimization. For example, virtio [30] is a one of the most known paravirtualized method, which provides a simplified device interface for a virtual device. KVM also has several paravirtualization features [208] and these features can used with hardware-assisted virtualization. We discuss the details of KVM's paravirtualization features in subsection 5.2.4.

The disadvantage of paravirtualization is that it requires modification of the guest.

2.1.2 Device Pass-through

Device pass-through [12] is a method of assigning a physical devices to a virtual machine without virtualizing it. Compared to virtual devices, device pass-through let a single virtual machine to occupy a specific device, but on the other hand, the virtual machine can directly manipulate the device in the same way as bare-metal. Device pass-through is useful for maximizing device performance within a virtual machine. However, the hypervisor cannot modify the I/O if the guest passes through a device.

2.1.2.1 Partition-based Hypervisor

Some hypervisors statically partition hardware resources and allocate them to a virtual machine. Such hypervisors include the jailhouse [217] and ACRN [145]. In this case, it can be seen that the entire hardware used by the virtual machine is pass-through. In such a hypervisor, the virtual machine runs directly on the non-virtualized hardware and achieve the same performance as bare metal. Such a hypervisor is suitable for high-performance computing and for running applications that require real-time performance.

2.1.3 Parapass-through

Device pass-through can bring out the performance, but if all the functions are passed through, it is impossible to manipulate the virtual machine's device operations in a hypervisor. To cope with the problem, BitVisor [35] proposes parapass-through architecture. In parapass-through architecture, most of the operations are pass-through, but some crucial parts are virtualized.

One of the main examples of parapass-through is descriptor shadowing. Descriptor shadowing only shadows descriptors used by devices. This allows the hypervisor to manipulate device data while the device's control plane is pass-through to the guest.

The parapass-through architecture has been used in several studies, including the application of transparent VPN [35], background storage encryption [57, 58], malware detection [59] and rootkit detectiojn [40] through disk IO monitoring, and non-volatile memory write protection [119]. Also, [140] proposed "device masquerade" that reduces the effort of creating device drivers on the OS side by showing a physical device as a virtio device [30] to the OS by appropriately converting OS I/O access.

2.1.4 On-demand Virtualization

In some cases, if the functionality provided by virtualization is only needed temporarily, then on-demand virtualization can reduce the virtualization overhead during normal operations. For example, Microvisor [13] uses on-demand virtualization to launch the maintenance OS while keeping running a OS to reduce downtime during machine maintenance. After the maintenance is completed, Microviser devirtualize the machine, thus reducing the overhead during normal operation. BMCast [94] proposes a method of transparently caching the results of network boot to local disks to speed up the provisioning of bare-metal clouds. In BMCast, disk I/O is captured by the hypervisor at boot time and the untransferred data is retrieved remotely, while data that is already local is accessed directly. After the OS is started, the overhead of the operation is removed by devirtualization. [122] extends [89], which achieve migration in bare-metal clouds, and uses on-demand virtualization during migration to completely eliminate overhead during normal operation.

On-demand virtualization cannot be used for security applications that require constant monitoring and protection by the hypervisor.

2.2 Double Scheduling Problem

In the virtualization environment, there are two intrinsic schedules; the hypervisor performs vCPU scheduling, and the guest performs its thread scheduling. This double scheduling causes several performance problems.

One of the most famous is the lock holder preemption (LHP) problem, which occurs when a vCPU holding a spinlock is preempted [16, 27]. Other vCPUs that want to acquire the lock need to wait until the preempted vCPU is rescheduled and releases the lock. The lock waiter preemption (LWP) problem occurs when the very next waiter for the lock is preempted [16]. The blocked-waiter wakeup (BWW) problem happens when the cost of the waking blocking primitives is high, which is common in the virtualization environment, since sending IPIs introduces several VMEXITs [80]. Preemption during the interrupt or RCU context also causes performance delay or increases in the memory footprint, since the other threads need to wait to complete these contexts [124, 134]. Even if vCPU is not preempted when the guest is in the critical section, vCPU scheduling essentially cause I/O delays since vCPUs may not be scheduled when interrupts occur [76]. These problems lead to performance degradation, especially when the CPUs are heavily oversubscribed.

2.2.1 Mitigating Double Scheduling Problems

Co-scheduling tries to avoid synchronization delays between vCPUs by scheduling vCPUs of the same VM simultaneously [36, 51, 74]. However, co-scheduling may introduce other problems, including CPU fragmentation or priority inversion. Several studies try to increase the I/O performance in a virtualization environment by prioritizing I/O event [53, 68, 69], using shortend time slices [76, 113, 130], performing active vCPU-aware scheduling [133], reserving CPU time for I/O task [126], or offloading the I/O processing to the hypervisor [80, 81, 129].

Paravirtualization (PV) is a well-known approach to reduce the semantic gap. PV spinlock avoids unnecessary spin by cooperating with the hypervisor [71, 104, 127, 213]. PV TLB shootdown reduces the latency of the TLB shootdown by delaying injection of IPIs to non-active vCPUs [61] or utilize special HW instructions to flush the guest TLB [109, 135]. eCS [134] uses the shared memory region to share the scheduler information of the guest and the hypervisor and avoid preemption if the guest is in the critical section. eCS needs to manually annotate the critical section in the guest kernel.

2.2.2 Dedicated CPU Resource Assignment

The double scheduling problem can essentially be solved by allocating a dedicated CPU to a virtual machine, replicating hardware configuration of the host, and avoiding overcommitment.

Song et al. [72] first proposed the idea of the vCPU ballooning (VCPU-BAL), which dynamically adjusts the number of vCPU according to the available pCPU. By exclusively assigning the CPU to one vCPU, we can eliminate the vCPU scheduling in the hypervisor, thus double scheduling problem can be avoided. Understandably, light-weight vCPU tuning mechanism is important to achieve maximum performance. VCPU-BAL is later implemented on QEMU/KVM and Linux using Linux's CPU hotplug/unplug mechanism [92].

Even in the virtualization environment, Linux's CPU hotplug/unplug takes around several tens of milliseconds [102], which is longer than both the normal scheduling time slice (1000 HZ = 1 ms or 250 HZ = 4 ms) and the boot time of the light-weight VMs (several milliseconds) [123]. One reason is CPU hotplug/unplug require several global oprations which need to acuire spinlocks.

There are several studies to shorten the CPU hotplug/unplug time. Chameleon [60] use proxy processor, which perform tasks instead of the offline CPU to enable rapid reconfiguration. Bolt [96] refactor the CPU hotplug/unplug design and reduce the latency by only performing a critical task preferentially and deferring other tasks.

vScale [102] proposed a lightweight vCPU freezing mechanism for the vCPU ballooning. vScale carefully investigates the characteristics of user and kernel threads and shows that by migrating the migratable threads and suppressing IPIs and interrupts by utilizing the paravirtualized interface, effectively a vCPU become offline within several microseconds.

2.2.3 Virtual NUMA (vNUMA)

On a NUMA machine, a virtual machine whose host NUMA configuration is reproduced is calld vNUMA (Virtual NUMA). It also called NUMA-visible virtual machine. In most cases, when creating vNUMA, resources are exclusively assigned to a VM and therefore the VM can avoid double scheduling A number of studies have shown the performance benefits of vNUMA [48, 55, 79, 110, 143, 155].

One of the weaknesses of vNUMA is that it cannot handle dynamic changes in NUMA configuration. Such changes can occur owing to overcommitment, VM consolidation, and migration, among other issues. XPV [143] proposed a mechanism that supports dynamic changes in NUMA configuration, and virtflex [155] put forward an OpenMP infrastructure that supports such alterations.

2.3 Summary

We presented several studies that improves virtualization performance. Basically, performance can be improved by omitting the some parts of virtualization processes. However, which processes can be omitted depends on security and functional requirements. In the following chapters, we present specific optimization methods for virtualization through several use cases. We also presented the double scheduling problem and its mitigation methods. We present the detailed perform evaluation of vNUMA in chapter 5.

3 Nested Virtualization for Hypervisor Device Drivers Testing

In this chapter[†], we study how nested virtualization can be optimized for hypervisor device driver inspection. We apply a device driver testing method using fault injection by a hypervisor to the hypervisor's own device driver inspection. Focusing on the fact that the security features required by normal virtualization are unnecessary for testing purposes, we improve the performance by removing them.

3.1 Introduction

In cloud environments, typically, tens of thousands of servers are operated in a cloud and they run hypervisors on them. Since the number of servers is very large, the probability of hardware failures at one of the servers is not so low, even if individual servers are reliable. As an example, Baidu reported that they encountered more than 300,000 hardware failures in their data center in the past four years [128]. BackBlaze, a cloud storage service company, operated 116,833 hard disk drives (HDD) from 2013 to 2017 and reported that a failure occurred in 6,795 HDDs, which was only a little less than 5% of them [193]. Therefore, it is crucial for hypervisors to tolerate hardware failures to improve the reliability of the cloud. If hypervisors do not handle hardware failures properly, the hypervisors could unexpectedly crash and all virtual machines (VMs) running on it would also crash immediately. In that sense, the reliability of hypervisors is more important than normal operating systems (OS).

Hardware failures are caused by various factors [29]. In addition to permanent failures due to wear, aging, and deterioration, temporal failures due to electromagnetic interference or overheating could also occur. Unfortunately, device drivers of OSs and hypervisors often do not assume such failures and make a wrong assumption that hardware devices always work as defined in the specifications [32, 112]. For example, Listing 3.1 shows a code fragment of device drivers that makes such a wrong assumption. This code fragment performs a busy-wait loop until the most significant bit of the status register becomes 0. If this bit does not become

[†]This chapter is based on [137] () and [159, 161] ().

```
1 while(ioread(STATUS_REGISTER) & 0x8000);
```

Listing 3.1: Example code that does not assume hardware failures.

0 due to a hardware failure, the device driver will hang up. Such code actually exists in Linux device drivers [32, 112]. Since a hardware failure is a rare event, it is not easy to test device drivers against such failures.

We aim at providing a testing platform for cloud vendors to test device drivers of the closed-source hypervisors, which they are going to use for their cloud, against hardware failures that could occur in the real hardware the vendors have before they are in operation. To this end, the testing platform must not require source code, must be able to test against real hardware, and must be able to be applied to hypervisors.

In order to efficiently test device drivers, several methods have been proposed. One representative is static code analysis that examines the source code to detect inappropriate error handling without actually executing it [19, 23, 26, 32, 33, 44]. However, it requires driver's source code and cannot be applied to closed-source systems. Another is symbolic execution that detects the condition in which the device driver becomes an illegal state [41, 54, 62, 67]. However, since symbolic execution performs verification in a fully-virtualized environment, it cannot test device drivers against real hardware. Software fault injection (SFI) inserts pseudo faults to the target code to inspect whether device drivers handle the faults appropriately [20, 22, 49, 63, 75, 88, 101, 112, 196, 200]. Although SFI can test the driver code in a real environment, most SFI still need to modify source code.

FaultVisor [112] takes a unique approach to perform SFI without requiring source code. It virtualizes a part of device registers of real hardware and injects pseudo faults at the hypervisor layer when device drivers access the registers. By doing this, FaultVisor can test real closed-source device drivers against hardware failures. FaultVisor virtualizes only the target device registers and access to all other hardware is pass-through, allowing device drivers to be tested in a real environment. Additionally, FaultVisor is OS independent. Unfortunately, FaultVisor cannot be used for testing hypervisor device drivers as it is because it cannot run a hypervisor on it. Since hypervisors (Type I virtual machine monitors) have their own device drivers and do not have a host OS, testing device drivers separately with their host OS is not possible.

This section proposes FaultVisor2, a small hypervisor for testing hypervisor device drivers against hardware failures that exploits SFI and nested virtualization. To test closed-source hypervisors and device drivers, we inject faults to the value returned from real hardware devices by intercepting access to the hardware from the target hypervisor with the FaultVisor2 hypervisor. To test in a real hardware environment, we allow the target hypervisor to ac-

cess hardware in the pass-through manner and only intercept access to the target part of the devices. To run the target hypervisor on FaultVisor2, we exploit the concept of nested virtualization [37]. To reduce nested virtualization overhead and make our test environment close to real, we omit some of nested virtualization functions, including nested paging virtualization, and incorporate minimal functions that are necessary to inject faults.

We evaluated FaultVisor2 by testing two closed-source production hypervisors: the VMWare ESXi hypervisor [228] and the vThrii hypervisor [180]. As a result of evaluation, we detected three kinds of errors caused by improper error handling concerning the storage device driver in the VMWare ESXi hypervisor. Moreover, based on the evaluation result of the overhead, we confirmed that FaultVisor2 can test device drivers in close to a real environment.

The contribution of this research is as follows.

- We proposed a framework to test error handling of hypervisor device drivers against real hardware failures by performing fault injection to hypervisor device drivers with nested virtualization.
- We show the design and implementation of the framework.
- We evaluated our framework by testing existing production hypervisors and found several errors which led to critical system failures.

3.2 Device Driver Testing Methods

Device drivers are written primarily by third-party developers and known to be less reliable than the other part of the kernel. In addition, error checking of device drivers is difficult due to their characteristics that they communicate with both the kernel and devices with complicated interfaces. Therefore, various studies have been conducted for efficient inspection of device drivers. The target of inspection ranges from the error handling against hardware failures to the use of appropriate interfaces and confirmation of resource release. The purpose of inspection differs depending on methods.

3.2.1 Static Code Analysis

Static code analysis analyzes the source code of device drivers. Several works proposed to use theorem proving or model checking to ensure that device drivers follow the specifications [19, 23, 26, 33, 44]. Carburizer [32] analyzes the source code of Linux device drivers and finds errors that implicitly assume that the devices never fail. In addition, it can automatically correct a part of the errors. However, static code analysis cannot be applied to closed-source device drivers and does not execute the driver code in real environments.

3.2.2 Symbolic Execution

Symbolic execution is a method of comprehensively searching paths that a program can execute. Several works proposed to use symbolic execution for inspecting device drivers without using corresponding hardware devices [41, 54, 62, 67]. DDT [41] uses selective symbolic execution to detect resource leaks, race condition, null pointer reference, and so on. SymDrive [62] combines static code analysis and symbolic execution to make symbolic execution more efficient. Symbolic execution has high code coverage. However, since it takes time to search for executable paths, programmer effort is necessary for an efficient path search. Also, since the examination is performed in a virtual environment, various behaviors including side effects in the real environment cannot be accurately reproduced. For example, it is difficult to detect errors that are caused by subtle timing problems such as that related to interrupts and DMA.

3.2.3 Software Fault Injection

SFI verifies the operation of software by inserting pseudo faults. Several works used SFI for device driver testing [20, 22, 49, 63, 75, 88, 101, 112, 196, 200]. In these methods, a fault is inserted mainly to the return values of a function, and the behavior of the device driver is observed. EH-Test [101] automatically generates test cases primarily for detecting resource leaks in the error handling code of device drivers. However, EH-Test does not test against hardware failures. In addition, the code tested is slightly different from the original one, leaving the possibility that errors that occur in a real environment do not reproduce.

FaultVisor [112] uses SFI to test device drivers against hardware failures. FaultVisor uses a small hypervisor to intercept access to a part of device registers from the device driver and inject a pseudo fault into the value returned from the target device. Therefore, it can be used for closed-source device drivers and OSs. FaultVisor allows pass-through access to other part of hardware, allowing the device driver and OS to run in a real hardware environment. Unfortunately, since FaultVisor relies on hardware-assisted virtualization functions (such as Intel VT-x and AMD SVM) and does not virtualize them, it cannot be applied to hypervisors that also require hardware-assisted virtualization functions and embed their own device drivers in them without using host OSs. In this thesis, we extend FaultVisor so that it can be applied to such hypervisors.

3.3 Design

We extend FaultVisor so that it can be used for verifying the fault tolerance of Type I hypervisors against hardware failures. In this section, we first describe the overview of FaultVisor and then describe the extensions in FaultVisor2.

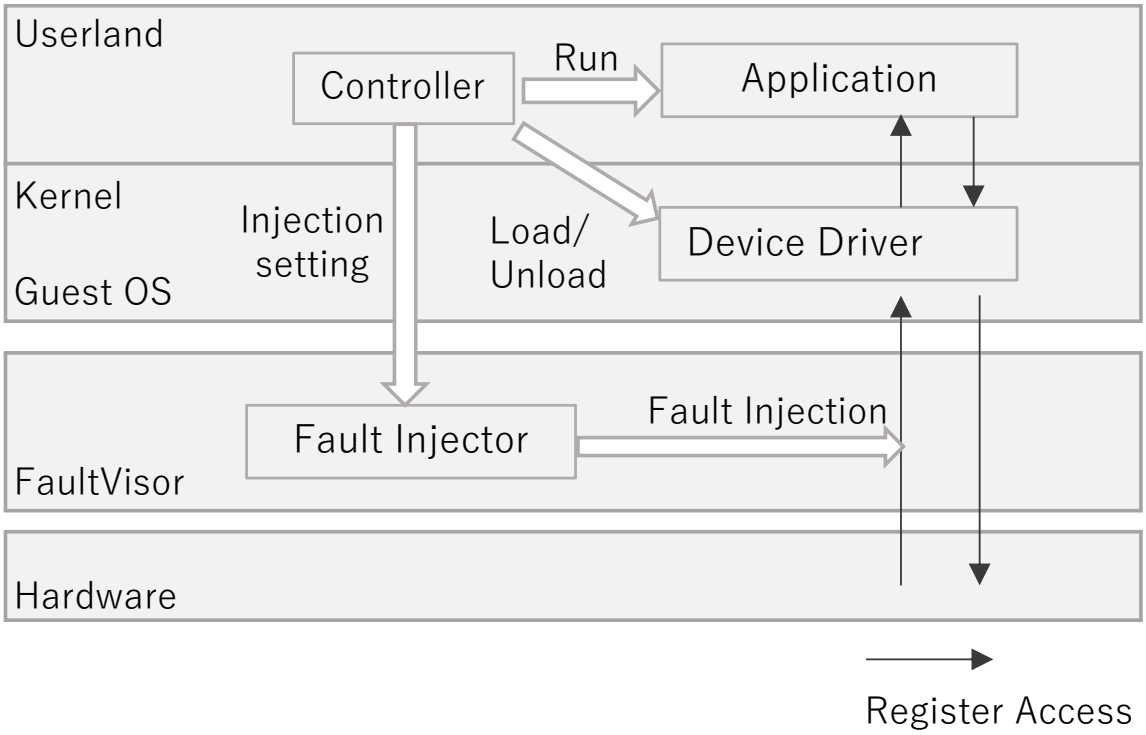


Figure 3.1: Overview of FaultVisor

3.3.1 Overview of FaultVisor

Figure 3.1 shows the overview of FaultVisor. The OS with the device driver to be inspected is run on FaultVisor as a guest. When the device driver tries to read values from device registers, FaultVisor intercepts it and performs fault injection by returning values different from the original ones. FaultVisor can intercept programmable I/O and memory-mapped I/O (MMIO) accesses. Faultvisor does not support intercepting fault to a DMA area.

To control the overall testing, a controller program that runs at the userland of the OS is used. First, the controller determines the target device and the fault injection mode (described below). The controller directs the fault injection configuration to the FaultVisor hypervisor by an API call. FaultVisor then starts fault injection in accordance with the configuration. After that, the controller executes the pre-defined applications that use the device, and load / unload the device driver. If the kernel crashes (due to, e.g., an unintentional kernel panic) or hangs up (e.g., there is no response for a fixed time) at this time, it is judged that there is a problem in the device driver error handling code.

FaultVisor has two modes to perform fault injection. One is “fixed,” which always returns a predefined value, and the other is “xormask,” which applies the xor mask operation to the value read from the device with a predefined mask value. During the inspection, the controller determines the register number to be fault-injected and whether to use “fixed” or “xormask” for modification. The test pattern (the mask value) to be used in injection is programmable

by the controller, thus the inspection becomes flexible.

To assist deciding the registers to be fault-injected, FaultVisor has a mode called "monitor mode." When FaultVisor is operated in this mode, fault injection is not performed but the registers read by the device driver is recorded. By using this mode, it is possible to investigate the device registers that is used in the workload without analyzing the source code or device specification. To reduce virtualization overhead, FaultVisor uses the parapass-through hypervisor [35]; it intercepts only a part of register access and allow pass-through access to other hardware. Using the parapass-through hypervisor allows the inspection in the situation close to the real environment.

3.3.2 Proposed Method

We propose FaultVisor2, an extended version of FaultVisor that can be used to test hypervisor device drivers. Figure 3.2 shows the overview of the proposed framework. In this research, we denote the FaultVisor2 hypervisor as L0, the hypervisor to be tested as L1, and the OS operating on L1 as L2.

In this framework, we run the L1 hypervisor on the L0 hypervisor using nested virtualization [37]. By using nested virtualization, L1 access to hardware can be intercepted by L0. As shown in Figure 3.2, the L2 application accesses the virtual device provided by L1. The L1 hypervisor then uses its device driver to control the corresponding real hardware device. The L0 hypervisor intercepts the access to the real device from L1 and performs fault injection.

The controller of FaultVisor2 operates on the L2 OS. The controller selects and executes an appropriate L2 application so that fault injection is performed on the actual device to be inspected. The reason why we do not run the controller on the L1 directly is because many hypervisors do not support running an application on L1 itself. In addition, running workloads on L2 allows reproducing device usages close to the real environment. Some hypervisors support dynamic device driver loading and unloading via L1 management tools. In this case, it is possible to verify the device driver in that part by using the tools, although this process needs a hypervisor-dependent implementation to access the management tools.

3.3.3 Advantages of Proposed Method

Since this method does not depend on the hypervisor to be tested, it is easy to apply our method to various hypervisors. By operating the controller on L2, inspection can be performed on closed-source hypervisors. Furthermore, using the monitor mode allows the inspection without source code or device register information.

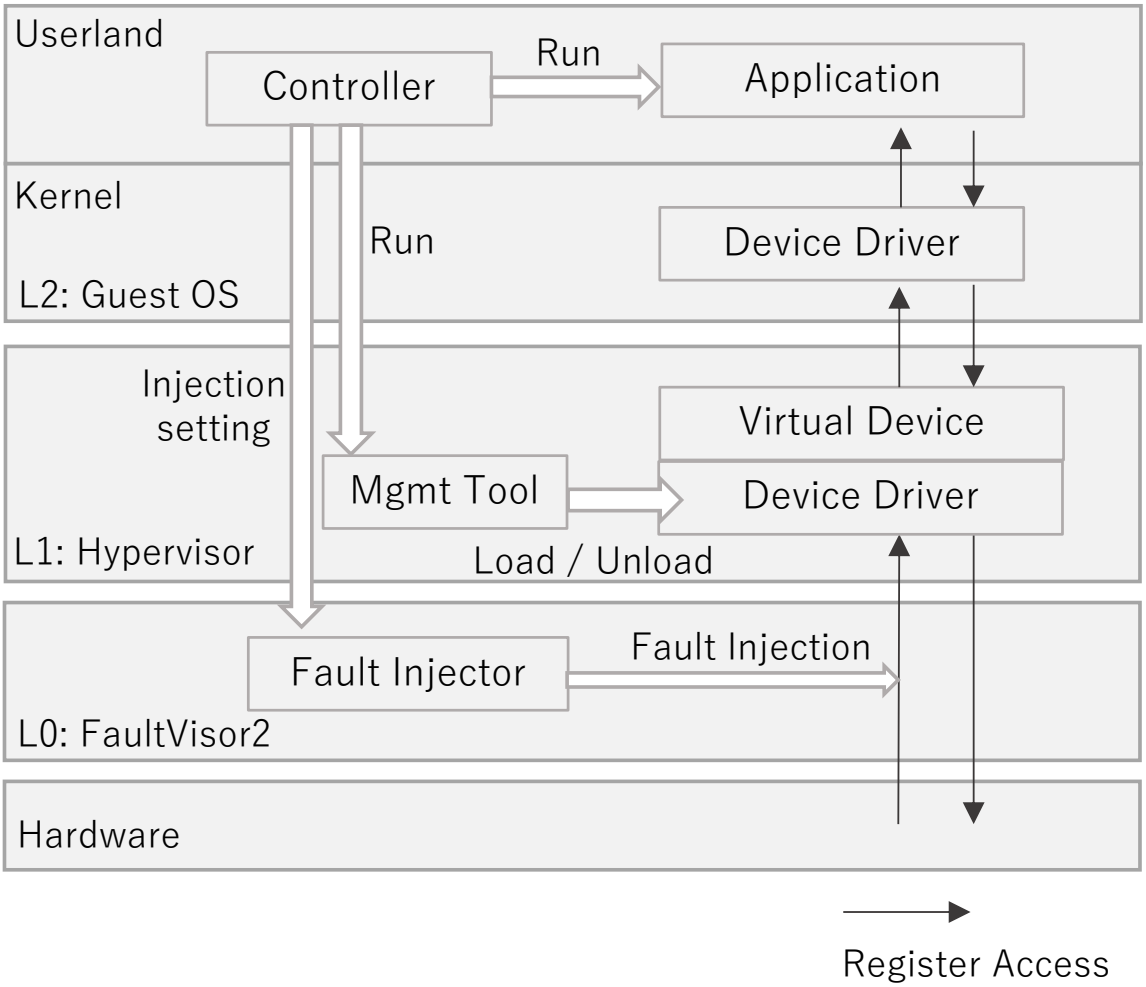


Figure 3.2: Proposed Method

3.4 Implementation

We implemented FaultVisor2 by adding nested virtualization support to FaultVisor and constructed an inspection system for nested environments. FaultVisor is based on BitVisor [35], which supports Intel and AMD CPUs. In our current implementation, FaultVisor2 supports only Intel CPUs.

3.4.1 Fault Injection by Nested Virtualization

Intel VT-x does not fully support native nested virtualization. To realize nested virtualization, L0 needs to trap the VMX instructions (virtualization instructions) executed by L1 and perform appropriate emulation [37].

Intel VT-x has two modes to operate virtualization. One is the VMX root mode and the other is the VMX non-root mode. The former is used for the host and the latter is used for guests. Intel VT-x manages host and guest states with a data structure called VMCS. The

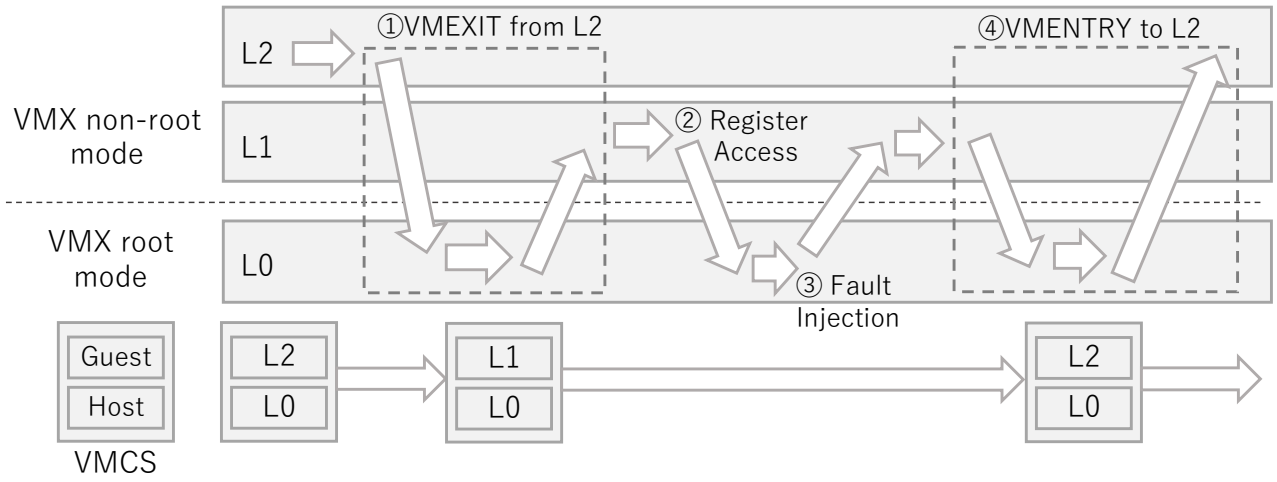


Figure 3.3: Fault Injection Process

information used during the execution of the guest OS, such as CPU registers, is stored in the guest state field of the VMCS, and information of the host hypervisor is stored in the VMCS host state field.

To switch to the VMX non-root mode, the hypervisor first sets the VMCS appropriately and then issues the VMLAUNCH or VMRESUME instruction. When the guest OS executes a pre-determined instruction in the VMX non-root mode, an VMEXIT event occurs and the mode is switched to the VMX root mode. The instructions that cause VMEXIT is defined in the VMCS. Extended page table (EPT) is used to convert the guest physical address to the host physical address. In the nested virtualization, L0 runs in the VMX root mode, and L1 and L2 run in the VMX non-root mode. We need to set the VMCS appropriately to switch between L1 to L2.

Figure 3.3 shows a typical fault injection process. At the first state, L2 is running and the VMCS guest state is for L2.

1. When an L2 application accesses a virtual device via a device driver, a VMEXIT event occurs since the VMCS is appropriately configured to cause VMEXIT. At this time, the control is transferred to L0, not to L1. To pass the control to L1, L0 switches the VMCS for L2 to the VMCS for L1, and issues the VMENTRY instruction to enter L1.

2. L1 processes the virtual device and eventually accesses the target real device. This causes an VMEXIT event due to the EPT violation, since EPT access permissions are configured to cause EPT violation when registers to be inspected are accessed. Then, the control is passed to L0.

3. L0 accesses the device register instead of L1. If the access operation is read and the register is a target of fault injection, L0 modifies the read value in accordance with the predefined settings and returns it to L1.

4. L1 processes the value of the register and finally tries to enter to L2. Since it is impossible for L1 to directly switch to L2, the control is first passed to L0 and then L0 changes the VMCS to enter L2.

This is a very simplified form of nested virtualization. Since FaultVisor2 runs only a single hypervisor on it, it can omit virtual CPU scheduling. Therefore, FaultVisor2 needs to manage only two VMCSs (one is for L1 and the other is for L2), and switch them in response to the VMEXIT events.

3.4.2 Controller

The controller is implemented as an application running on L2. Communication from the controller to the FaultVisor2 hypervisor is performed by using the VMCALL instruction, which explicitly transfers control from the VMX non-root mode to the root mode. When VMCALL is executed at L2, VMEXIT occurs and the control is passed to L0. Arguments can be passed via registers. In short, VMCALL is used by the guest OS to send some messages to the hypervisor.

A typical implementation of nested virtualization always transfers the control to L1 if the L2 issues VMCALL. To get a control message from L2, L0 intercepts the VMCALL and handle it, then return to L2 without transitioning to L1. However, intercepting all VMCALL messages from L2 will cause a problem since most L1 hypervisors also need to receive messages from L2 via VMCALL. It is not easy for L0 to determine whether the VMCALL is for L1 or L0, since we need to examine the state of the L2 when VMCALL occurs. In our current implementation, L0 only intercepts VMCALLs whose arguments contain a predefined magic value.

3.4.3 EPT Pass-through

In our implementation, L0 directly uses the EPT that L1 configured; we do not perform EPT shadowing (nested paging virtualization). This is possible because we run only a single guest hypervisor on FaultVisor2, and do not virtualize most of hardware and allow the OS pass-through access to hardware. Therefore, the physical address of L1 is identical to the physical address of L0, and there is no need for address translation. This approach reduces the overhead of nested virtualization.

EPT pass-through has the following restrictions.

1. L1 can create an EPT mapping with which L2 can access the L0 memory region.
2. L1 can create a device pass-through setting to allow L2 direct access to a physical device.
3. L0 cannot run multiple hypervisors.

However, these restrictions are not a problem for testing hypervisor device drivers for the following reasons. Regarding 1), L0 manipulates the memory map returned by BIOS so that the memory area used by L0 becomes a reserved area. As a result, L1 will not try to use the memory area. Furthermore, L0 configures EPT so that L1 cannot access memory regions that L0 uses. In theory, it is possible for L1 to create an EPT that allows L2 to access the L0's region if L1 intentionally tries to do that. However, we assume that L1 is not malicious and will not intentionally destroy L0.

Regarding 2), if L1 creates an EPT in which no EPT violation occurs when accessing device registers, VMEXIT events never occur and L0 cannot capture the access from L2 to the device. However, in device pass-through settings, no hypervisor device driver is used, and therefore, there is nothing to check for L0. If we need to check the device driver of L2, we can use FaultVisor (not FaultVisor2).

Regarding 3), since our purpose is to check the device drivers of a hypervisor, we do not need to do it.

3.5 Evaluation

In this section, we first present the results of the hypervisor device driver testing. We then present the results of the performance evaluation. We tested device drivers of the VMWare ESXi [228] and the vThrii Seamless Provisioning (vThrii) [180], both of which are closed-source production hypervisors.

3.5.1 VMWare ESXi

VMWare ESXi is a Type I hypervisor and widely used in cloud environments.

3.5.1.1 Setup

Table 3.1 shows hardware and software setup in this experiment. Table 3.2 shows the physical devices we tested. We created a single VM on the target hypervisor and allocated all of the available host memory and CPU cores to the VM. To test drivers, we assign a virtual device to the VM which internally uses a target physical device.

3.5.1.2 Experiment Procedure

Experiments were performed through the following procedure. Details of the workload executed are described in the next section.

1. Run workloads under the monitor mode and record register access. The recorded registers become the test targets.

Table 3.1: Setup of VMWare ESXi

Name	Information
Motherboard	ASRock Z170 Extreme4
CPU	Intel Core i7-6700 (hyper-threading disabled)
RAM	DDR4 2133MHz 8GB × 2
Guest OS	Ubuntu 18.04 (Linux 4.15.0-23)
Hypervisor	VMWare ESXi 6.5.0 (VMKernel Build 5310538)

Table 3.2: Target Device of VMWARE ESXi

Device	Driver	Version
Intel 82574L	ne1000	0.8.0-11
Intel X540-T2	ixgbe	4.4.1
Fusion IO IoDrive2	iomemory-vsl	3.2.15
LSI MegaRAID	lsi_mr3	6.910.18.00

2. Perform steps 3) to 6) three times for the fixed mode and the xormask mode for each target register to be fault injected until bugs are found.

3. Choose one target register. Determine the injection parameter with random number and call FaultVisor2 to start the fault injection.

4. Execute the workload.

5. After executing the workloads, stop the fault injection by calling FaultVisor2.

6. If an error is detected, reboot the machine and start inspection from the next target register.

3.5.1.3 Workload

For the workload of network devices, we ran a ping command to a machine connected via a hub. For the workload of storage devices, we ran fio [189]. fio measures the reads and writes performance of the target storage device.

VMWare ESXi supports dynamic device driver loading. Therefore, we also executed the driver load and unload workload. For this workload, we first logged into the ESXi host from the guest OS, and then ran the `vmkload_mod` command. We separately executed a ping or fio workload and a driver load and unload workload.

3.5.1.4 Result

Table 3.3 shows the test result. "Used" indicates the number of registers read by the driver during the monitor mode. Note that it does not include the number of registers written by the

Table 3.3: The Test Result of ESXi

Drivers	Workload	Used	Tested	Error	Test Time
ne1000	ping	3	3	0	9m
ixgbe	ping	231	231	0	1h08m
ixgbe	driver	1077	564	0	5h17m
iomemory-vsl	fio	5	5	1	14m
iomemory-vsl	driver	77	77	8	2h57m
lsi_mr3	fio	4	4	0	12m

Table 3.4: Detected Errors in iomemory-vsl

Error type	Number of registers	
	fio	driver load / unload
TIMEOUT	0	4
No heartbeat	1	3
VMKernel Exception	0	1

driver. "Tested" is the number of registers to which we performed the fault injection. Mainly, we performed fault injection to each of the detected registers. As of the ixgbe driver, we found that the driver accessed large contiguous register regions (DMA registers and multicast table arrays) when initializing the device. To shorten the inspection time, we excluded these registers from the test. "Error" indicates the number of registers which caused a failure during the test. "Test Time" indicates the total test time including machine reboots due to a failure. We did not perform the driver load and unload workload for ne1000 and lsi_mr3 since we found that once we unloaded the drivers, VMWare ESXi failed to load the driver unless the machine was rebooted.

We confirmed three types of errors in iomemory-vsl. Table 3.4 shows the error types and the number of registers which caused that error. Note that the register that caused the no heartbeat error was the same in the fio and the driver load / unload workload. The detailed status of each error type is provided below.

TIMEOUT The device driver loading process did not finish after a certain period of time. When the process is forcibly terminated, the device driver cannot recognize the target device. The state of the device driver still remains in use, and it can not be unloaded. This type of error can be caused by a coding error such as shown in Listing 3.1

No heartbeat A VMware ESXi kernel panic occurred because a virtual CPU did not respond for a certain period of time.

VMKernel Exception An exception in the VMWare ESXi kernel occurred, which led to the VMWare ESXi kernel panic. According to the error message, a DE exception (possibly

Table 3.5: Setup of vThrii

Name	Information
PC	Macbook Pro 13-inch Late 2017
Guest OS	macOS High Sieera 10.13.4
Hypervisor	vThrii-P 1.5.3

Divide-by-zero Exception) occurred. This type of error can be caused by using a device register value without verification.

When no heartbeat or VMKernel Exception occurs, the stack trace can be acquired. Since the iofusion-vsl driver is a closed source, we could not identify the detailed cause at this time. However, such information can be useful for locating the problem for driver developers.

3.5.2 vThrii

vThrii is a special type of hypervisor for OS provisioning. When booting, vThrii at first only fetches the minimum disk data from a provisioning server that is enough to start booting the OS. After the OS booted, vThrii fetches the remaining data in the background. If the OS accesses an unfetched region, vThrii traps the access and transparently fetches the data from the server. This enables a fast OS startup compared to fetching all the disk data in advance while eventually all OS data are fetched and can be accessed without network access. vThrii allows pass-through access to hardware devices except for network devices. vThrii itself is based on BitVisor [35] and is a product version of BMcast [95].

3.5.2.1 Setup

Table 3.5 shows the setup in this experiment. We tested two device drivers of vThrii: "bnx" for Broadcom BCM57762 which is in a Thunderbolt-to-Gigabit Ethernet adapter and "nvme" for Apple NVMe SSD which is the built-in SSD in MacBook Pro.

3.5.2.2 Workload

We used ping and fio in the same way as the ESXi testing. vThrii does not support dynamic device driver loading. Hence, to test device initialization codes, we also performed fault injection when booting the machine. We boot FaultVisor2 from the UEFI shell and pass the injection parameters as UEFI shell arguments. FaultVisor2 performed the injection from its startup. In this experiment, we performed fault injection per target register with the fixed and xormask mode for one each. We restarted the machine for each test.

Table 3.6: The Test Result of vThrii

Drivers	Workload	Used	Tested	Errors	Test Time
bnx	ping	0	-	-	-
bnx	boot	36	36	11	1h55m
nvme	fio	0	-	-	-
nvme	boot	12	12	5	34m

3.5.2.3 Result

Table 3.6 shows the test result. We found that both bnx and nvme drivers only write to device registers during the benchmark workload (ping and netperf). Therefore, no injection is performed as of these workloads.

During the injection when booting, we observed that several fault injection patterns led to vThrii panic. According to the log messages of vThrii, the panics can be classified into three types. One is an initialization error (occurred during a very early stage of booting and any log messages were not outputted), another is a connection lost error (occurred three times during injection to bnx), and the other errors (occurred during booting with no panic message logs). Since a hardware error can be temporal, retrying device accesses instead of causing a sudden panic would improve the reliability of vThrii. In addition, checking the register values and reporting error messages if it finds an fault would help diagnose problems.

3.5.3 Performance Evaluation

To measure the overhead of FaultVisor2, we performed three kinds of benchmarks on ESXi, ESXi on KVM [21] and ESXi on FaultVisor2. We used the same hardware and software shown in the Table 3.1. We used Ubuntu 16.04 (Linux 4.13.0-45) for the KVM host OS and used the pass-through configuration using VFIO [197] when running ESXi on KVM. KVM allocated all of the available host memory and CPU cores to the ESXi and the ESXi had a single VM which had all available ESXi's resources.

3.5.3.1 SPEC CPU2017

SPEC CPU2017 [220] is a benchmark package containing memory and CPU intensive workloads. Figure 3.4 shows the execution time overhead of the SPEC CPU2017 (intspeed, fpspeed). The reported values are the median values among the three measurements. We excluded the 627.cam4_s and 657.xz_s workloads since we have a compilation problem not related to virtualization.

As shown in Figure 3.4, 17 out of 19 workloads on the ESXi on KVM had greater execution time overhead than ESXi on Faultvisor2, and 13 workloads had more than 10% overhead

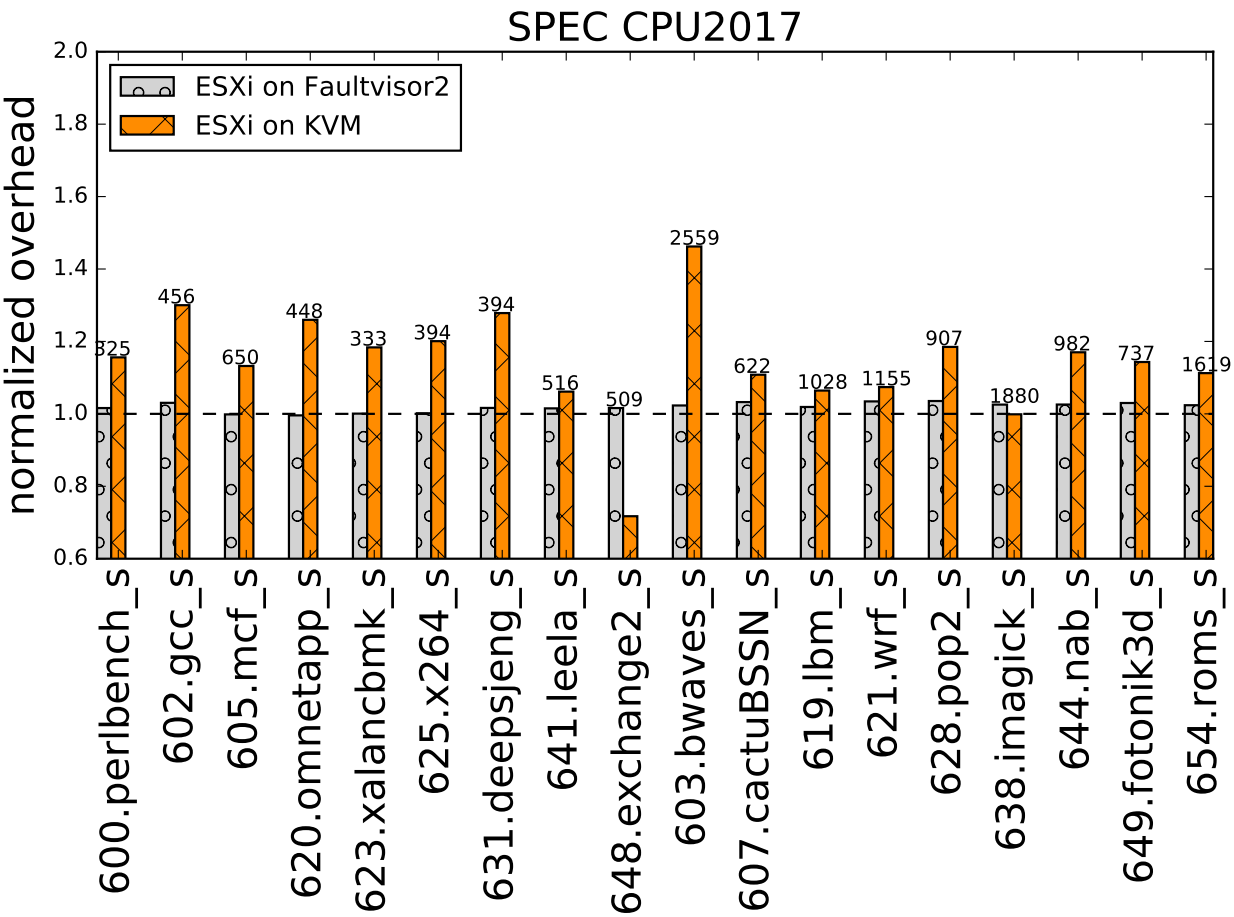


Figure 3.4: SPEC2017 benchmark result. The baseline is ESXi. The numbers on the bars are baseline runtimes (sec). The lower is better.

compared to the baseline. The maximum overhead of ESXi on Faultvisor2 was only 1.035 (628.pop2_s).

3.5.3.2 Network

We connected two machines via a switch and measured network throughput and latency. The device used was Intel 82574L. To measure throughput, we used netperf TCP_STREAM test with various MTUs. The machine running hypervisor ran the netperf server. Throughput was measured five times and the mean values were reported. We measured the latency 30 times using a ping command.

Figure 3.5 shows the results. In the figure, "ESXi on FaultVisor2" means that FaultVisor2 did not perform any fault injection. "ESXi on FaultVisor2*" performed xor injection to a certain register with 0 value (therefore the register value was not altered). ESXi on KVM had some throughput degradation whereas ESXi on FaultVisor2 (with or without injection) had no decline. The latency of the ESXi on KVM was almost six times longer than ESXi. On

Table 3.7: VMEXIT Counts

	spec	ping	fio
ESXi on KVM	9,682,183	115,452	49,163,515
(EPT Violation)	872,930	7,070	7,913,130
(VMREAD)	1,486,707	11,295	10,149,187
(VMWRITE)	72	0	0
ESXi on FaultVisor2	10,939,984	82,654	43,617,261
(EPT Violation)	18,873	281	9,577
(VMREAD)	6,570,637	47,273	26,467,241
(VMWRITE)	2,928,982	20,653	12,386,678

the other hand, the latency of ESXi on FaultVisor2 was only about 30 µs longer than ESXi. Comparing ESXi on Faultvisor2 with ESXi on FaultVisor2*, we can see that performing fault injection did not affect the latency.

3.5.3.3 Storage

We measured the random read-write completion latency and IOPS (input/output per second) of a virtual disk on an ioDrive using fio. We set the block size at 4KB, iodepth at 16, a number of jobs at 4, read ratio at 40%, and file size at 1 GB. Figure 3.6 shows the result. As shown in the figure, we observed some performance degradation when using nested virtualization. At 90 percentile of read latency, ESXi on FaultVisor2 had 514 µs longer latency than the baseline and had almost half the IOPS. Performing fault injection slightly affected the performance. However, FaultVisor2 had much lower overhead compared to ESXi on KVM.

3.5.3.4 VMEXIT counts

To investigate the factor of the performance degradation, we measured the number of VMEXIT from both L1 and L2 during workloads. Workloads are (1) SPEC (600.perlbench_s), (2) ping to an connected machine and (3) fio with filesize 256MB. Table 3.7 shows the result. As shown in the table, KVM had a large number of EPT violation compared to FaultVisor2. We also found that FaultVisor2 had many VMREAD/VMWRITE exits. These exists can be reduced using VMCS shadowing technology, which currently FaultVisor2 does not utilize.

3.6 Discussion

3.6.1 Detected errors

The main purpose of FaultVisor2 is to detect device driver coding errors such as that shown in the Listing 3.1. However, we found that errors detected by FaultVisor2 do not necessarily

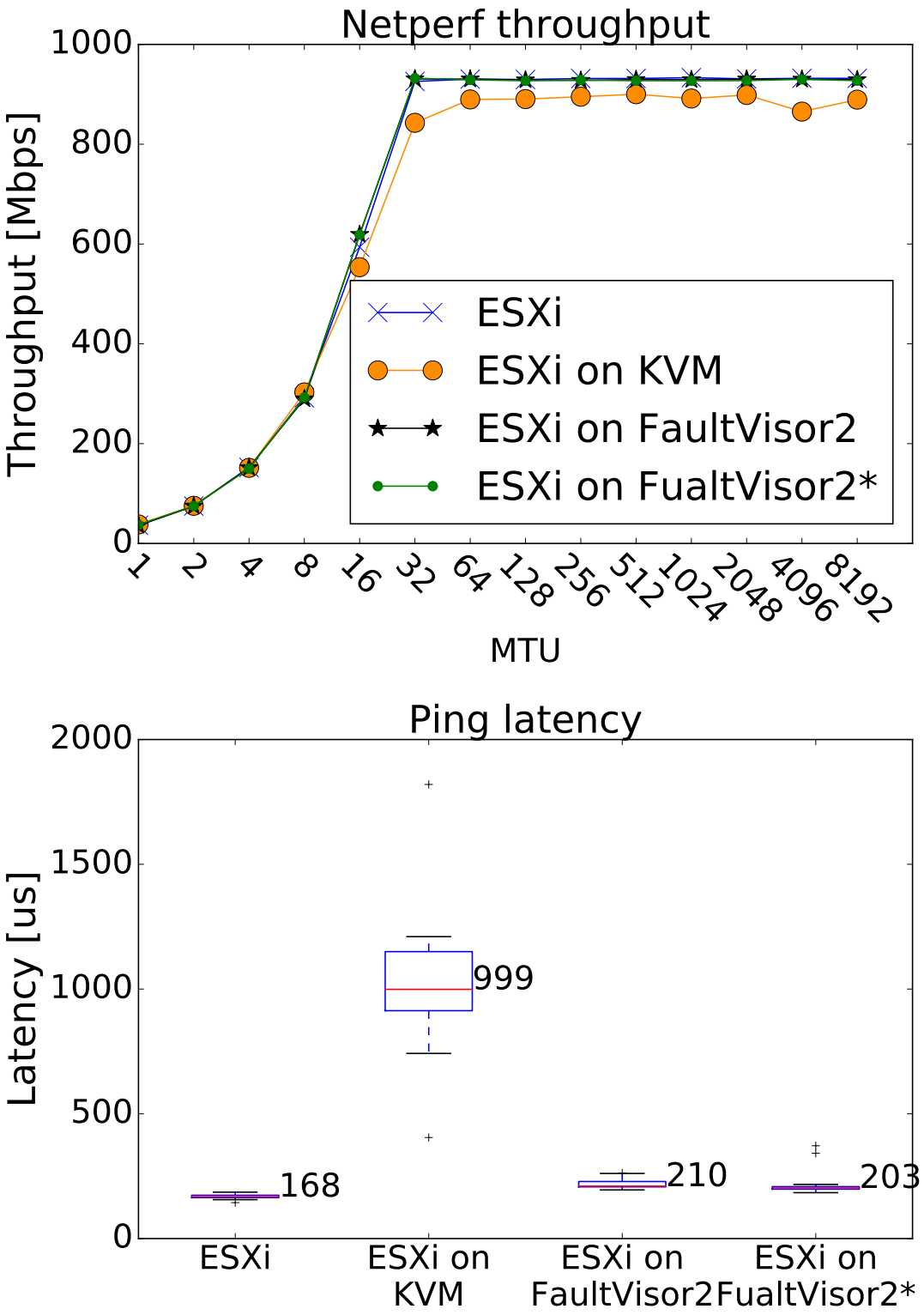


Figure 3.5: netperf throughput (upper) and ping latency (lower).

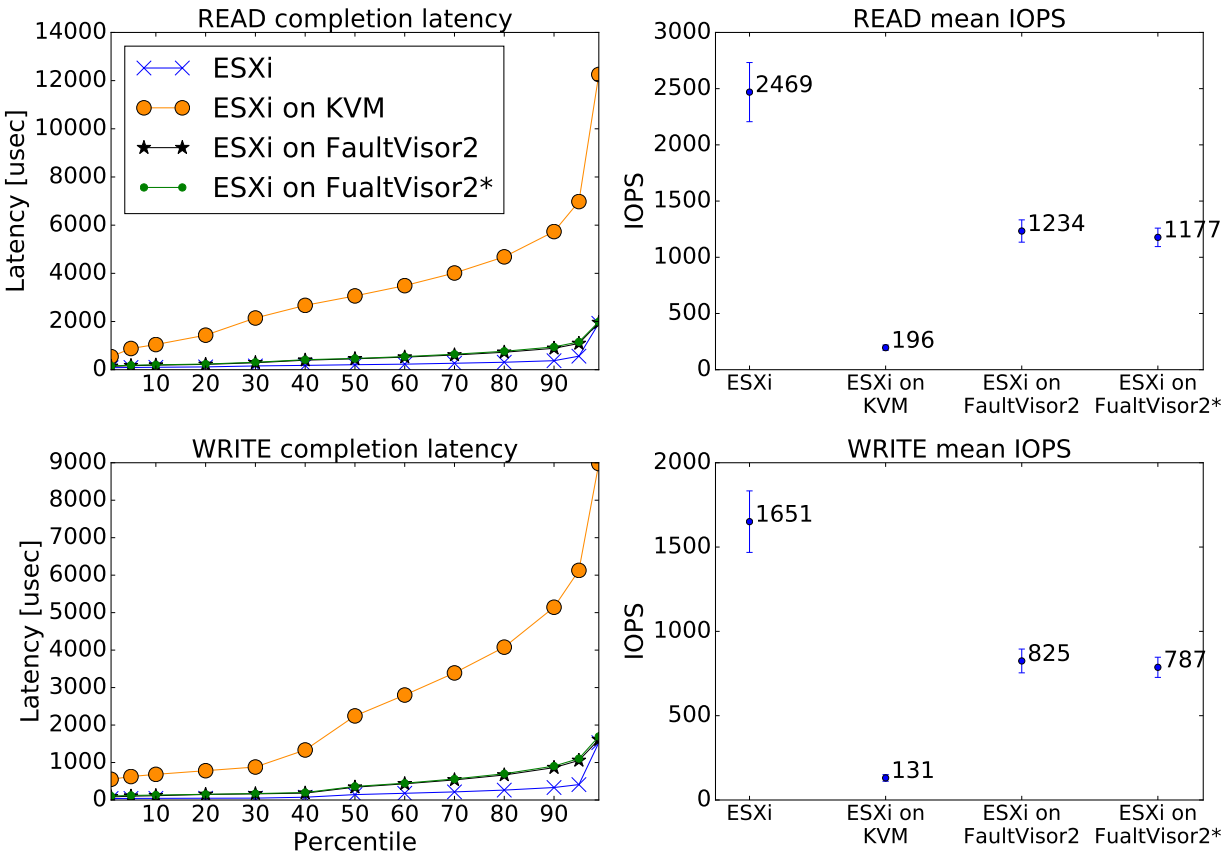


Figure 3.6: fio benchmark. Left: percentile plots read/write completion latency (lower is better). Right: mean read/write IOPS with standard deviation (higher is better).

indicate a coding error directly. For example, error handling code may have a bug that leads to a system crash. In vThrii, the device driver seems to check device register values, but the error handling is not enough in some cases. In this sense, our method can detect errors that cannot be found by static code analysis that only checks if the error handling is performed.

3.6.2 DMA support

Currently, FaultVisor2 does not support injecting faults to a DMA area. However, inspection in the DMA is crucial. For example, NICs (network interface cards) manage the location of packets in the queue through descriptors that are read and written by DMA. In such devices, the device driver uses the information of the DMAed values for processing. Therefore, failure to check for errors in DMAed values can cause serious problems. In addition, DMA attacks from malicious devices have become a problem in recent years [146, 156], which makes error handling of device input values by device drivers increasingly important.

We can extend FaultVisor to support inspection of the DMA area in the following ways. Figure 3.7 shows the overview of the proposed method.

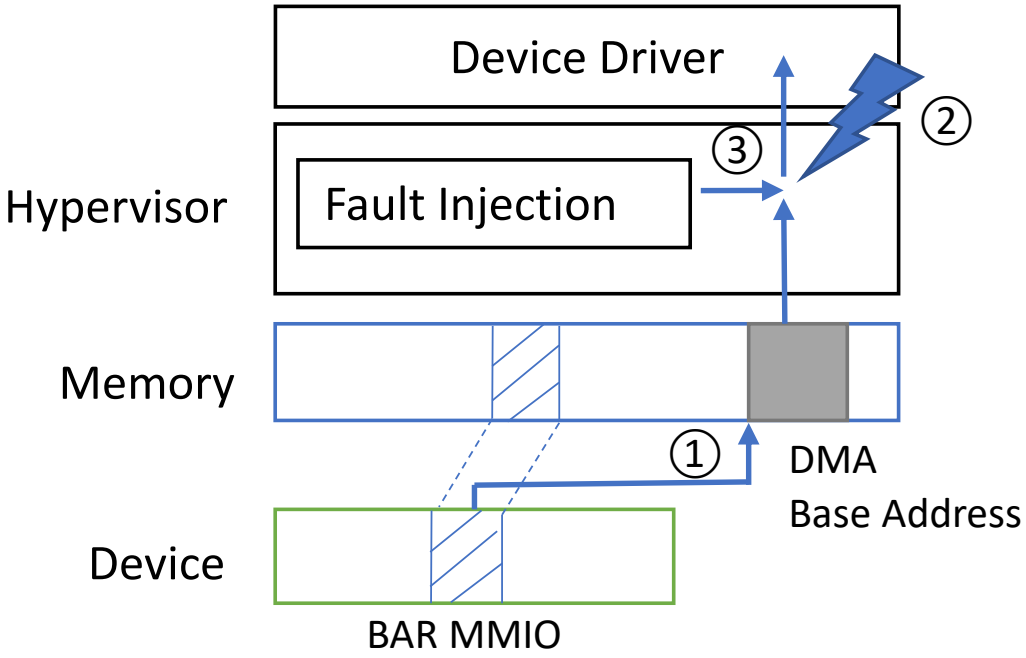


Figure 3.7: Fault Injection to the DMA Region

First, the DMA region that will be used by the device to be inspected is first identified using device-specific driver code (①). Then, the hypervisor applies memory access protection to the DMA area using the nested paging and detects memory reads from the device driver (②). When the hypervisor detects a memory read from the device driver, it performs fault injection by returning a random value other than the actual value to check if the device driver correctly handles errors in the DMA area (③). The details of each step are described below.

3.6.2.1 Identifying the DMA region

The address of the DMA area used by a device does not exist in the PCI configuration space and is device-dependent. The proposed method solves this problem by creating a device-specific driver code that specifies the DMA area used by the device based on the device specifications. The address of the DMA area used by the device is stored in a specific MMIO register. The driver code obtains the address of the DMA area by referring to this register.

Specifically, if the device has already been initialized at the start of the fault injection, the hypervisor reads the register. If the device has not yet been initialized at the start of the fault injection (e.g., when the device driver is loaded during the fault injection), the hypervisor uses nested paging to capture the register access and obtains the value when the OS stores it. In addition, the driver code decides which parts of the DMA area to fault-inject (i.e., data that the device driver does not use is excluded from the injection), which leads to more efficient device driver testing.

3.6.2.2 Detecting accesses to the DMA area of a device driver

The hypervisor uses nested paging to detect access to the DMA area of a device driver. It applies access protection to the area identified by the driver code. If the device driver accesses the area, control is transferred to the hypervisor. Note that nested paging does not respond to memory access by DMA from the device.

3.6.2.3 Inspection through fault injection

When the hypervisor traps a device driver access using the method described above, it performs fault injection by returning a different value to the device driver if it tries to read the value. The value returned as a fault injection can be a random value or a boundary value (0 or the maximum value of the field). The actual flow of the test is as follows. First, we booted the OS on the hypervisor of the proposed method. Then, we executed a specific workload and performed fault injection to observe the behavior of the device driver. If a kernel panic or hangup is observed, it is judged that the error handling of the device driver is inappropriate. We also monitor kernel messages ("demsg" for Linux) and check if any error occur. We repeat this process several times. If the OS clash, we reboot a machine and continue the inspection.

We implemented our prototype and found one bugs in Linux NVMe driver [161]. Applying these DMA inspection method proposed to FaultVisor2 is one of the future works.

3.7 Summary

In this chapter, we proposed a hypervisor device driver testing framework that combines fault injection and nested virtualization. Focusing on the fact that the security features required by normal virtualization are unnecessary for testing purposes, we optimized nested virtualization performance by removing them. In our experiment, we found three kinds of errors concerning the storage device driver in the VMWare ESXi. We also found several errors in the vThrii hypervisor's device drivers. The performance experiments showed that the proposed method had a much lower overhead than the traditional nested virtualization scheme and could test the hypervisor device drivers in close to the real environment.

4 IOMMU Virtualization for Device Protection

In this chapter, we presents how IOMMU virtualization can be optimized for device protection purpose.

4.1 Introduction

While computers have become an indispensable part of our daily lives, the threat of cyber-attacks has only increased. It is thus important to take countermeasures to protect against, detect, and analyze the content of such attacks. Memory acquisition is one of the most valuable techniques for detection and analysis in this regard. It is the process of storing the contents of the main memory in another storage medium or transferring them to an analysis system. By analyzing the contents, it is possible to verify the system's integrity, detect malware, and perform memory forensics.

Memory acquisition can be broadly divided into two types: software-based [42, 43, 52, 64, 86, 93, 99, 121, 125, 141, 144, 154, 160] and hardware-based [15, 56, 66, 70, 78, 105, 111, 132, 147]. In the past, it was mainly software-based. An example of software-based memory acquisition is running memory acquisition software as a process or a kernel module on the OS [64]. While such software-based methods have the advantage of being easy to use, they also have a problem in that the memory acquisition function may be disabled by stopping the process when an attacker is in control of the system.

As a way to increase the attack resistance of software-based methods, acquiring memory at a higher privilege level than the OS has been proposed. A typical method is to use a VMM (Virtual Machine Monitor) [42, 93, 121, 125, 144, 154, 160]. Some methods use x86 System Management Mode (SMM) [181] or Trusted Execution Environments (TEEs) [115, 149] such as ARM TrustZone [170] to run the memory acquisition mechanism in an environment that operates at a higher privilege level than a VMM [43, 52, 86, 99, 141]. However, these methods incur a significant overhead due to virtualization and/or sharing CPU time and, in the case of the VMM method, the TCB is too large for an ordinary general-purpose VMM. Moreover, the method using SMM requires rewriting BIOS/UEFI firmware, which is challenging to use

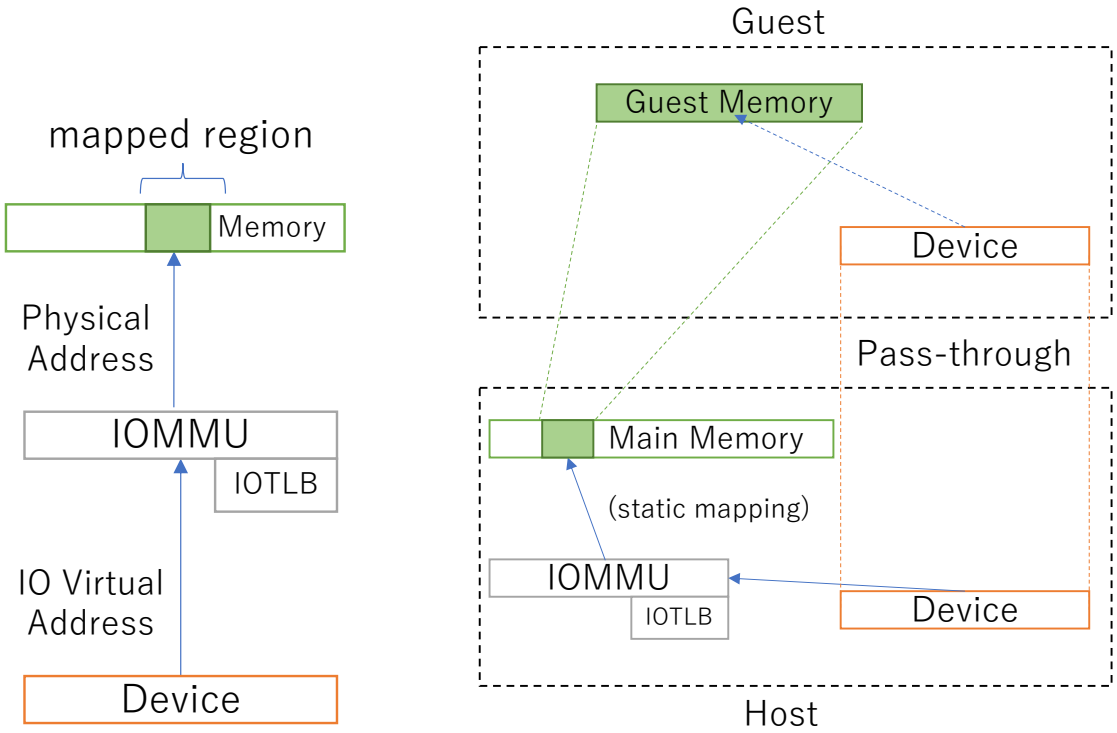
in general.

In addition to memory acquisition via software, there is also a memory acquisition method that uses hardware [15, 56, 66, 70, 78, 105, 111, 132, 147, 150]. Among them, one promising approach is using a coprocessor [15, 66, 105, 111, 147, 150]. The coprocessor operates independently of the CPU and reads the memory contents using DMA (Direct Memory Access), thus enabling high-speed memory acquisition. Furthermore, the coprocessor can be connected to the PCIe bus and can be used transparently in existing systems. A typical disadvantage of coprocessor-based memory acquisition is that it is more expensive and difficult to obtain than other methods. However, recent FPGAs have made it easier to obtain coprocessors [98] for memory acquisition. For example, the SCREAMER M.2 USB-C (R04) [216], which uses the XC7A35T Xilinx 7 Series FPGA and can perform memory acquisition, is commercially available for 249 Euros (at the time of writing.)

On the other hand, coprocessor-based memory acquisition methods still have several challenges. One is the possibility of an attacker disabling the DMA function of the coprocessor. One example of such an attack involves the rewriting of the IOMMU (Input Output Memory Management Unit) configuration. IOMMU is a mechanism that translates addresses used by PCIe devices for DMA. IOMMU makes it possible to DMA non-contiguous regions and to limit the range of DMA. The latter is used to minimize bugs in device drivers and protect memory from malicious devices [146, 156, 171, 214]. IOMMU is also used to pass through a device to to a virtual machine. Figure 4.1 shows example usage of IOMMU. IOMMUs thus improve device flexibility and security; from the perspective of a memory acquisition coprocessor, however, attackers can exploit this feature. In other words, if an attacker takes control of a machine with an IOMMU, they can disable the DMA functionality of the memory acquisition coprocessor by rewriting the IOMMU configuration.

One solution is to disable the IOMMU at the BIOS level, making it impossible for any user to use it. However, from a security point of view, it is desirable to enable IOMMU as described above. Previous studies using coprocessor-based memory acquisition [15, 66, 105, 111, 147, 150] either do not assume the existence of the IOMMU in the first place [15, 66] or assume that the IOMMU is properly configured to work with the coprocessor [105, 147, 150] or disabled [111].

In addition, coprocessor-based memory acquisition methods cannot do some of the things that a VMM and SMM methods can do. One of the things that cannot be done with the coprocessor-based method is acquiring register values that are useful for memory analysis. Furthermore, since DMA operates asynchronously with the CPU, there is a possibility that the guest may rewrite the memory contents during memory acquisition, therefore consistent memory acquisition cannot be guaranteed. On top of that, event-based memory acquisition [56, 70] is not possible for a coprocessor alone. An attacker may take advantage of these characteristics to hide their existence [24, 82, 100].



(a) Intra-OS Protection

(b) Inter-OS Protection

Figure 4.1: Examples of IOMMU Usage. A device can DMA only in a mapped region. (a) Intra-OS Protection. Protect from DMA attacks or buggy device drivers by limiting DMA-able region. (b) Inter-OS Protection. Mainly used for device pass-through. A device assigned to a guest can only DMA to guest memory region. Note that to limit DMA-able region in a guest memory, vIOMMU is required.

In this study, we propose a software-based method to solve these problems of coprocessor-based memory acquisition by cooperating with a VMM. The VMM performs shadowing of the minimum necessary IOMMU configurations in the proposed method while letting the guest use the IOMMU. The VMM also protects the memory space of the PCI configuration space so that the coprocessor-based memory acquisition can work reliably even if the attacker gains control of the OS. In addition, the VMM makes it possible to retrieve register values and enables consistent memory acquisition by pausing the vCPUs. By cooperating with the VMM, it is also possible to perform memory acquisition only when a specific event occurs such as memory writing to a specific region. Using a VMM specializing in the above processing, the overall TCB (Trusted Computing Base) and overhead is kept small, and the possibility of attacks on the VMM is minimized.

The contributions of this research are as follows:

1. We organize and present the problems of coprocessor-based memory acquisition methods,
2. We propose a method for coprocessor-based memory acquisition to work reliably in the IOMMU environment by cooperating with a VMM. The proposed method gives not only enough protection from an attacker but also enables register acquisition, consistent and event-based memory acquisition, and
3. We implement a prototype of the proposed method and conduct a detailed performance evaluation.

4.2 Background

4.2.1 Memory Acquisition

Memory acquisition is an important technique used in, for example, software integrity verification [15, 43, 56, 70, 93, 105, 147], memory forensics [42, 52, 86, 125, 154], and malware detection [99, 111, 141]. If the memory acquisition mechanism runs on the OS, the function may be disabled by an attacker who has taken control of the OS. Therefore, to increase the certainty of memory acquisition, research has been conducted on memory acquisition methods from outside the OS[15, 42, 43, 52, 56, 66, 70, 78, 86, 93, 99, 105, 111, 121, 125, 132, 141, 144, 147, 154, 160].

Table 4.1 depicts a rough classification of such memory acquisition methods. The meaning of the terms of each property in the table is as follows:

Tamper Resistance The memory acquisition method must continue to operate even if an attacker gains control of OS. The vulnerability of the memory acquisition method itself is not considered here.

Table 4.1: Memory Acquisition Methods from Outside the OS

Method↓ \ Property→	Tamper Resistance	Consistency	Performance Isolation	Small TCB	Availability	Research
VMM	✓	✓	✗	✗[†]	✓	[42, 93, 121, 125, 144, 154, 160]
SMM	✓	✓	✗	✓	✗	[43, 52, 99, 141]
TEE (TrustZone)	✓	✓	✗	✓	✓	[86]
Coprocessor	✗	✗	✓	✓	✓	[15, 66, 105, 111, 147, 150]
Special HW	✓	✓	✓	✓	✗	[56, 70, 78, 132]
Coprocessor + VMM	✓	✓	✓	✓	✓	Shielded Copilot (Our Work)

[†] When using a general-purpose VMM.

Table 4.2: Related Works Using Coprocessor-based Memory Acquisition

Research↓ \ Property→	Tamper Resistance	Register Value Acquisition	Consistency	Event-based Acquisition	Accessible Region			Main Goal
					OS	VMM	SMM	
Copilot (2004) [15]	✗	✗	✗	✗	✓	✓	✗	Integrity Check
Balogh, et al. (2013) [66]	✗	✗	✗	✗	✓	✗	✗	Memory Acquisition
GRIM (2016) [105]	✗	✗	✗	✗	✓	✓	✗	Integrity Check
LO-PHI (2016) [111]	✗	✗	✗	✗	✓	✓	✗	Malware Analysis
Nighthawk[†] (2019) [147]	✗[‡]	✓[‡‡]	✗	✗	✓	✓	✓	Introspection
BMCLeech (2020) [150]	✗	✗	✗	✗	✓	✓	✗	Memory Acquisition
Shielded Copilot (Our Work)	✓	✓	✓	✓	✓	✓	✗	Memory Acquisition

[†] This uses Intel ME, not an external device. [‡] It is possible to check IOMMU integrity. [‡‡] By cooperating with SMM.

Consistency A property that prevents the CPU from rewriting a part of the memory during memory acquisition.

Performance Isolation Memory acquisition is executed separately from the CPU.

Small TCB TCB is small.

Availability The availability of the method to the public.

One of the most common methods of acquiring memory from outside the OS involves using a VMM [42, 93, 121, 125, 144, 154, 160]. Using a VMM makes it possible to keep the memory acquisition function running even if an attacker has taken control of the OS. However, traditional VMMs have significant overhead and TCB, making them a potential target of attack.

The x86 CPU has a unique mode of operation called the System Management Mode (SMM) [181]. SMM operates with higher privileges than a VMM, and all physical memory ranges can be accessed from it. Several studies use SMM for memory acquisition and memory analytics [43, 52, 99, 141]. The advantage of the SMM over the VMM is that it is more difficult to attack and the SMM can verify the code of the VMM. However, as with a VMM, overhead is a problem because all CPUs stop running when switching on the SMM mode. In addition, since the SMM code is contained in the BIOS/UEFI firmware, it must be modified before it can be used, making it more difficult to work with for ordinary users.

Arm CPUs have a function to provide a TEE called a TrustZone [170]. The TrustZone also operates with higher privileges than OS, so it can continue to operate even if control of the OS is lost. TrustDump [86] utilizes TrustZone for memory acquisition. However, this method has the same problem as the one using VMMs.

Some studies use special hardware for memory acquisition [56, 70, 78, 132]. Vigilare [56] and KI-MON [70] achieves efficient memory acquisition using SoC to snoop bus traffics to detect memory write events and perform memory acquisition at the time. Ziyi et al., [78] uses programmable DRAM to perform kernel and VMM integrity checking transparently. SnipSnap [132] proposes memory snapshot system based on on-package DRAM technologies. These methods enable to acquire memory consistently and quickly, but these require special hardware and are not generally available at present.

Another approach to memory acquisition involves using a coprocessor [15, 66, 105, 111, 147, 150]. Typically, coprocessors are connected as external hardware via PCIe. These methods use DMA to acquire memory without involving the CPU. Therefore, they have the advantage of being fast and system-transparent because their operation does not depend on the CPU.

Copilot [15] is a pioneer in memory acquisition by coprocessors, and it verifies the integrity of the kernel at runtime. Balogh et al., [66] proposed a method of memory acquisition in cooperation with NIC device drivers in the OS. LO-PHI [111] uses memory acquisition from

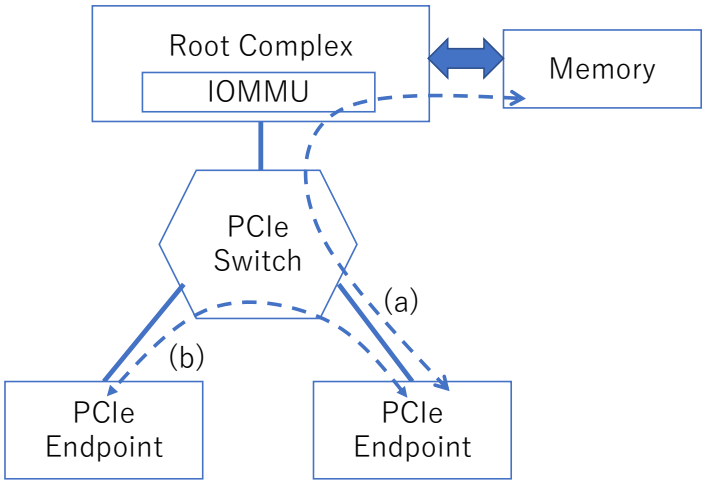


Figure 4.2: Example of PCI Express Structure. IOMMU is in a Root Complex. Actual PCIe structure can be more complex; e.g., Root Complex has several endpoints in it, etc. Also it is possible to have multiple IOMMU and PCIe hierarchies. (a) DMA. IOMMU converts address when accessing memory if necessary. (b) P2P DMA without routing to Root Complex. In this case, address translation by IOMMU is not performed.

a coprocessor to realize a malware analysis environment that is as close as possible to the natural environment. GRIM [105] uses the GPU, and NightHawk [147] uses Intel ME for memory acquisition. BMCLeech [150] proposes a method of memory acquisition from the Baseboard Management Controller (BMC) used for server management.

However, a major problem with the coprocessor-based method is that the IOMMU may disable the DMA function. In addition, since the coprocessor operates asynchronously with the CPU, it is impossible to retrieve register values or acquire memory consistently, which is possible with a VMM and SMM. This issue will be discussed in detail in section 4.5.

4.2.2 PCI Express (PCIe)

PCI Express (PCIe) is the most common interconnect for connecting peripherals in today's computers. A schematic diagram of PCIe is shown in Figure 4.2. PCIe has a Root Complex (RC), which is connected to the CPU and memory. Devices (PCIe Endpoints) are connected via PCIe switches.

In PCIe, data and messages are exchanged in units of packets called Transaction Layer Packets (TLPs). The device sends a TLP of DMA requests for DMA, as shown in Figure 4.2 (a). The PCIe switch routes the packet to the RC. The RC retrieves the memory data by DMA and sends back a response to the device. Also, as shown in Figure 4.2 (b), DMA can be performed directly between devices without an RC if the devices support it. This is called P2P DMA.

Each PCIe Switch or PCIe Endpoint has an area for a configuration called the PCI configuration space. The PCI configuration space can be accessed via I/O instructions and also from the MMIO area. For a device to perform DMA, the "Bus Master Enable" bit of the command register in the PCI configuration space of the device and the PCIe switches through which the TLP passes must be set to 1. Otherwise DMA will be aborted.

4.2.3 IOMMU

The IOMMU is a memory management unit in the PCIe RC that handles address translation for DMA performed by devices. Using IOMMU, it is possible to translate the IO virtual address used by the device to a specific physical address. This functionality is also known as DMA remapping. There are several implementations of IOMMU, such as Intel VT-d [185], AMD IOMMU [165] and ARM SMMU [169].

The IOMMU can be used to DMA to non-contiguous regions or prevent DMA to unexpected ranges due to bugs in the device driver. IOMMU can also be used to limit the DMA range of a device to that of the guest memory when passing through a device to a virtual machine.

The mapping settings of IOMMU are located in the memory. In addition, IOMMU can set mapping for each PCI device. IOMMU also has several functionalities such as Interrupt Remapping, which converts interrupt vector, and Posted Interrupts, which enables to post interrupt to a VM without VMEXIT [165, 185]. These functionalities also important for security. For example, interrupt remapping can block interrupts whose sender is not a legitimate device [65, 85]. Although this research focuses on DMA remapping, disabling IOMMU means that users cannot use not only DMA remapping but also these functionalities.

4.2.3.1 Address Translation Service (ATS)

ATS [34] provides a mechanism for devices to cache the results of IOMMU address translation. A device that supports ATS notifies the OS of its support using the capability area of the PCI configuration space. If the OS enables the ATS, the device can DMA using addresses without IOMMU conversion. Specifically, the device can receive the result of address translation from the IOMMU by sending a TLP with the Address Translation (AT) field set to `01` (Translation Request TLP). If the device performs DMA using a TLP with `10` in the AT field (Translated TLP) , the address is not translated by the IOMMU and is used as is. Note that a device can send a Translated TLP without using a Translation Request. Therefore, the use of ATS should be limited to trusted devices only. TLP with `00` in the AT field is a normal TLP and `11` is reserved.

4.2.3.2 Access Control Service (ACS)

ACS [18] is a mechanism to check TLPs sent by devices in RC and PCIe switches. ACS can force P2P DMA to always go through an RC and block Translated TLPs using ATS. ACS is essential for enforcing address translation by IOMMU.

4.3 Memory Acquisition in the Presence of IOMMU

Several works tried to acquire memory in the presence of IOMMU. IO-Trust [142] and ThunderClap [146] showed that because Linux unconditionally enables the PCIe ATS feature before 5.x, the device can bypass the IOMMU setting and access the memory by sending a Translated TLP. However, this method is not reliable for memory acquisition because an attacker who has taken control of the OS can block a Translated TLP by using ACS or disable ATS itself. In addition, since Linux 5.x, the ATS of externally connected devices ("untrusted" devices) is uniformly disabled [188], so memory acquisition using this method is no longer possible. This method also cannot be used on Windows, macOS, and FreeBSD, as they do not support ATS in the first place [146].

IOCHECK [87] proposes to use SMM to verify the integrity of the IOMMU configuration. This can be used to detect if an attacker rewrites the IOMMU configuration; after the rewrite, however, memory acquisition may not be possible. Also, there is a possibility of transient attacks. In addition, it is difficult to use in general because it needs to modify SMM.

Morgan et al., [139] proposes an attack method that bypasses the IOMMU settings set by the OS by rewriting the IOMMU settings by DMA during the short time between the creation of the IOMMU page table in the memory and setting the base address of the page table in the MMIO register. In principle, this method can be used to overwrite the IOMMU page table if an attacker tries to do so. However, this method assumes that the OS always places the IOMMU configuration at the same physical address, which is challenging to use in practice.

4.4 Assumption and Threat Model

The assumptions and threat models used in this study are described below.

4.4.1 Assumption

This study assumes that memory acquisition is performed on a machine (the monitored machine) using a coprocessor. The memory analysis is performed on the coprocessor or a different machine (the analytics machine) than the monitored machine. The target machine uses PCIe,

the most standard interconnects, and the coprocessor is connected via PCIe. The target machine has several peripherals such as NICs and SSDs. The coprocessor has a function to read memory via DMA. The coprocessor also has an out-of-band communication channel for communication with the analytics machine and can send the acquired memory to the analytics machine, receive messages from the analytics machine, and perform processing accordingly. The target machine has an IOMMU with ACS, and the OS can freely configure IOMMU. Each device including IOMMU is assumed to work correctly according to the specification, and hardware bugs or misconfigurations (such as [24, 100]) are not assumed to exist. At system boot time, we assume that the proposed method is correctly executed by using Trusted Boot.

4.4.2 Threat Model

An attacker may gain control of the monitored OS via the network and carry out arbitrary memory rewriting or code execution on the OS. An attacker also may gain a control of a device other than the memory acquisition coprocessor and execute DMA to arbitrary memory region. We do not assume attacks on a VMM or SMM, which have a higher privilege level than the OS. As discussed in section 4.6, the VMM we use in this research is a lightweight VMM specializing in extending the functionality of memory acquisition devices, thus its attack surface is small and may possibly be formally verified [157, 158, 162]. As for the SMM, the attack surface can be sufficiently reduced via existing methods, as discussed in subsection 4.9.1. We trust boot process, and attacks during the system boot time before the proposed system runs are out of the scope of this research. Physical attacks by attackers, DoS attacks, and side-channel attacks are out of the scope of this research as well.

4.5 Problems with Coprocessor-based Memory Acquisition Methods

In this chapter, we summarize the main problems with memory acquisition methods using coprocessors:

1. An attacker could disable the DMA function of the memory acquisition coprocessor,
2. CPU register values cannot be acquired,
3. Consistent memory acquisition is not guaranteed, and
4. Event-based memory acquisition is not possible.

The details of each are described below.

4.5.1 Problem: Disabling the DMA Function of a Coprocessor

The DMA function of a coprocessor may be disabled within the scope of OS authority by modifying the IOMMU or PCI settings. The IOMMU page table exists in the memory and the IOMMU itself has MMIO registers. To prohibit DMA to the memory acquisition coprocessor, an attacker can simply disable IOMMU or delete the entry for the memory acquisition coprocessor in the IOMMU page table. Even worse, by rewriting the IOMMU page table of the memory acquisition coprocessor, the memory acquired by the coprocessor can be set to desired values to hide an existence of an attacker.

In order to use DMA, the Bus Master feature of the device and the PCIe switch through which the device's TLP passes must be enabled. An attacker can also stop the DMA of a memory acquisition coprocessor by disabling the Bus Master function from the PCI configuration space.

The problem described here can be avoided by disabling the IOMMU at the BIOS level at system startup. However, in that case, IOMMU will not be available to legitimate system users. IOMMU is essential for device pass-through when using virtualization and is also used to protect against buggy device drivers. In addition, DMA attacks on devices have become a widespread problem in recent years [146, 156, 171, 214], and it is desirable to enable IOMMU for security reasons. An existing countermeasure is to disguise the device information reported by the memory acquisition device via the PCI configuration space to hide its existence. This is not a perfect solution, however. Also, it is possible to check IOMMU page tables integrity from a coprocessor by reading page table entries via DMA. However, such page table entry is not reliable due to the reason described above, and there is also a possibility of a transient attack.

4.5.2 Problem: Register Values Cannot Be Acquired

The memory acquisition device alone can acquire data in the memory, but it cannot acquire the register values of the CPU. Some register values are helpful for analysis. For example, it is possible to know which physical address is being used by referring to the page table, but the base address of the page table is stored in the CR3 register.

Existing studies use information from the OS or static analysis of memory to estimate the location of page tables. However, the former method is unreliable because the attacker may have take control of the system. The latter method is also unreliable since an attacker can change page table structures to hide their presence [82].

4.5.3 Problem: Consistent Memory Acquisition Cannot Be Performed

Memory acquisition devices are fast because they acquire memory contents via DMA without using the CPU; on the other hand, memory contents may be rewritten by the CPU or DMA of other devices during memory acquisition. It is known that DMA can be estimated by measuring the bus bandwidth using performance counters [73]. An attacker may take advantage of this, and it is possible that they attempt to conceal their presence when they detect DMA.

4.5.4 Problem: Event-Based Memory Acquisition Cannot Be Performed

The memory acquisition method using a coprocessor is classified as a method that periodically acquires and processes the memory contents (the snapshot-based method). Vigilare [56] and KI-MON [70] proposed an event-based monitoring method that monitors signals flowing in the memory bus and invokes processing in a callback when a non-specified value is written to a memory area registered in advance. The advantage of this event-based method is that the memory can be acquired at the time when the data is modified by the attacker, significantly reducing the possibility of the intrusion being hidden using transient attacks. In addition, the number of DMAs can be kept to a minimum, resulting in minimum memory bandwidth.

GRIM [105] showed that even a snapshot-based method using a coprocessor could achieve the same detection performance as an event-based method if the memory is acquired at short intervals. Therefore, the snapshot-based method may not be inferior to the event-based method in terms of security, though the event-based method still has an advantage in terms of memory bandwidth. Since [56, 70] uses special hardware to monitor the bus contents, the coprocessor alone cannot perform this function.

4.5.5 Summary

Table 4.2 provides a summary of the research on coprocessor-based memory methods. Except for Nighthawk [147], existing methods that use coprocessor-based memory acquisition have all the same problems as described here. Nighthawk uses Intel ME [229], a special coprocessor embedded in the Intel chipset with higher privileges than the SMM, to verify the integrity of the IOMMU configuration and fetch register values in cooperation with the SMM. However, using Intel ME requires rewriting the firmware like SMM-based methods, which is difficult in general. In addition, even Nighthawk does not provide complete IOMMU protection, which can lead to problems such as DMA being stopped by the OS during DMA execution.

Table 4.3: Comparison of CPU States which are higher than OS

Capability↓ \ CPU State→	VMM	SMM	Intel ME†
Higher Priviledge than OS	✓	✓	✓
Get OS's Register Values	✓	✓	✗
Stop OS Temporary	✓	✓	✗
Trap-and-Emulate	✓	✗	✗
Availability	✓	✗‡	✗‡

† Technically Intel ME is an coprocessor. ‡ Require firmware modifications.

4.6 Proposed Method

This research aims to solve the problems of existing coprocessor-based memory acquisition methods in a software manner. The four problems of coprocessor-based memory acquisition described in section 4.5, i.e., 1) the coprocessor cannot protect its IOMMU settings; 2) DMA cannot access CPU registers; 3) The coprocessor cannot stop CPU activity; and (4) The coprocessor cannot detect CPU events, are due to hardware limitations.

In order to deal with these problems, cooperation with the CPU side is essential. Therefore, in this research we solve these problems by cooperatively using a VMM with the coprocessor. The reason for using a VMM is that a VMM is the most flexible and easy to use among the CPU states that operate with higher privileges than the OS, as shown in Table 4.3. Namely, a VMM can get OS's register values, stop OS temporarily, and detects OS's event by trap-and-emulate. Also, a VMM does not require firmware modification and can be used transparently to OS. Traditional VMM is known for its large TCB and overhead. To use the method in practice, the TCB and overhead of the method needs to be as small as possible. We solve this problem by focusing the VMM on a specific purpose and narrowing down its functions.

4.6.1 Overview

Figure 4.3 shows the overview of the proposed method, which we call Shielded Copilot. In the proposed method, a memory acquisition coprocessor and a VMM work together. The VMM is responsible for protecting the IOMMU configuration and the PCI configuration space. In addition, the VMM pauses the OS vCPU and detects OS memory write events in response to requests from the coprocessor (or the analytics machine). The details of each of these functions are described below.

4.6.2 Guaranteed Operation of Memory Acquisition Coprocessor

We use IOMMU shadowings to guarantee operation of memory acquisition coprocessor. IOMMU shadowings ensure that a memory acquisition coprocessor always works irrespective of the

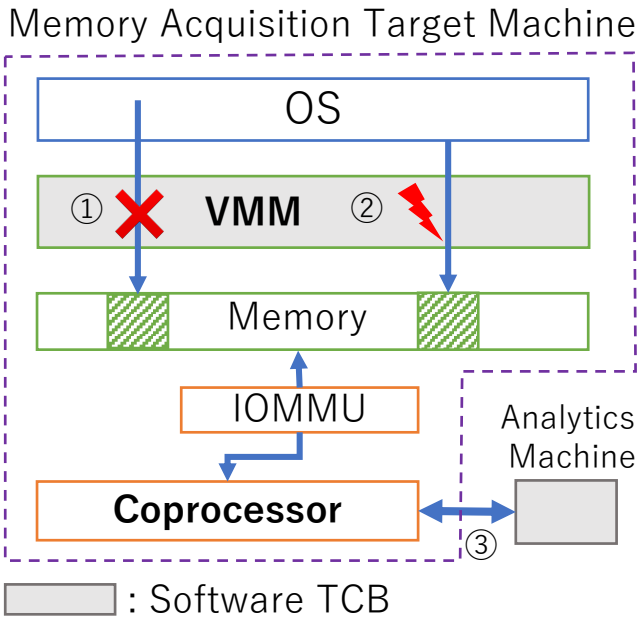


Figure 4.3: Proposed Method: Shielded Copilot. Gray parts are Software TCBs. The VMM does several tasks that a coprocessor cannot do alone. ① VMM protects several memory regions to prevent attacks against the VMM and a coprocessor. ② VMM can detect several OS's events and notify them to a coprocessor and the analytics machine. ③ VMM communicates with a coprocessor and the analytics machine using ouf-of-band communication channel. The analytics machine send a request to the coprosessor or the VMM such as memory acquisition, register value acquisition, temporal vCPUs suspension, and OS's event detection. The VMM or the coprocessor response the request.

IOMMU settings that OS creates. Figure 4.4 shows an overview of the IOMMU shadowings. First, when the OS does not use IOMMU, the IOMMU setting created on the VMM side is used to ensure that the memory acquisition coprocessor always operates. Also, DMA to the VMM area of devices other than the memory acquisition coprocessor is prohibited.

When the OS sets up the IOMMU, the VMM will shadow the IOMMU configuration as in Figure 4.4. In this case, we must create a configuration where the memory acquisition coprocessor can access all memory no matter how OS configures IOMMU. The IOMMU settings for devices other than the memory acquisition coprocessor are the same as those set by the guest. However, if there are any mappings in the IOMMU page table created by the guest that allow access to the VMM area, we must delete those mappings so that devices other than the memory acquisition coprocessor cannot access the VMM. IOMMU shadowing is performed every time when the OS invalidates IOTLB entries.

Some IOMMU has a pass-through mode [165, 185]. If a pass-through mode is enabled for a device, IOMMU does not perform any address translation for the device. If OS tries to set a device pass-through in IOMMU, VMM creates an identity mapping in shadow IOMMU page tables except for the VMM memory region. This prevents IOMMU pass-through devices from

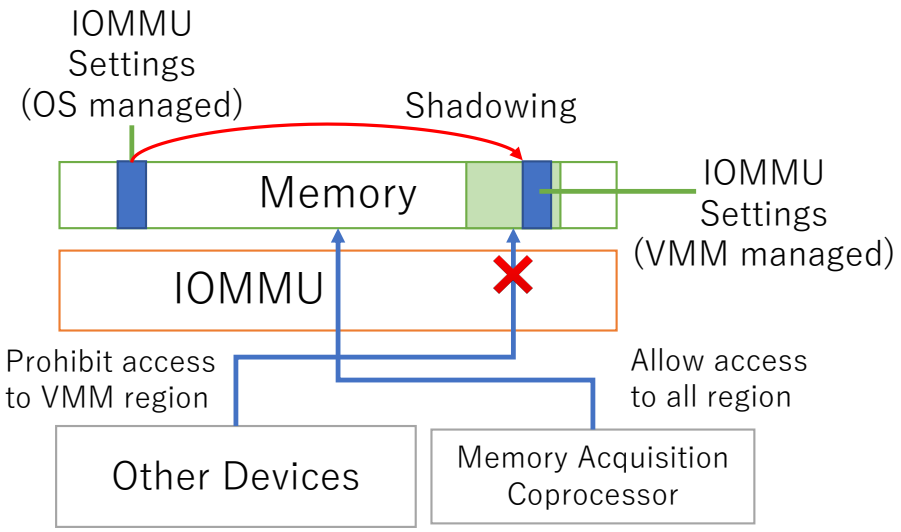


Figure 4.4: IOMMU Shadowing. This ensures that 1) memory acquisition coprocessor always can DMA, and 2) other devices cannot DMA to VMM regions. Not like traditional vIOMMU, the VMM only shadows necessary IOMMU settings and lets other part pass-through to the OS. Users can use other IOMMU functionalities freely.

attacking the VMM.

4.6.3 Protecting the PCI Configuration Space

The PCI configuration space can be accessed by PIO or MMIO. For PIO, configure a VM so that issuing I/O instructions cause VMEXIT. For MMIO, use nested paging to cause VMEXIT when accessing the PCI configuration space. The VMM prohibits the modification of the settings of the memory acquisition coprocessor and PCIe bus so that DMA from the coprocessor always work. The VMM also protects from PIO and MMIO overlapping attacks [65, 85].

When OS enumerates PCI devices, the VMM hides the coprocessor from the OS by concealing PCI information. This reduces the probability of an attacker detecting that a memory acquisition coprocessor is running when the OS is hijacked.

4.6.4 Register Value Acquisition

The VMM holds the register values of the OS. When acquiring registers, the analytics machine first sends a message to the VMM through the coprocessor. The message is written to a queue in the VMM by DMA, and an interrupt notifies of the arrival of the message. The VMM receives the interrupt and reads the message from the queue. If the message content is a register value request, the register value is returned to the coprocessor as a message response. The coprocessor returns the received message to the analytics machine and, finally, the analytics machine obtains the register value.

4.6.5 Consistent Memory Acquisition

When performing an consistent memory acquisition, the VMM suspends the vCPU of the OS. In this case, as in register value acquisition, the analytics machine first sends a message to the VMM, ordering it to stop the vCPU. When the VMM receives the message, it sends an IPI to all vCPUs to force VMEXIT and temporarily stop the vCPU. Then the VMM returns a reply to the coprocessor. By performing memory acquisition while the vCPU is stopped, memory can be acquired consistently. When the analytics machine finishes acquiring memory, it sends a message to the VMM to restart the vCPU. The VMM receives the message and restarts the OS vCPU.

Limitation The VMM can stop the vCPUs, but there is a possibility that DMA by devices other than the memory acquisition coprocessor will rewrite the memory contents. Note that this problem is not only for the proposed method, but also exist other VMM or SMM -based approach. DMA can be stopped by using IOMMU or ACS, but it is difficult to restart the DMA properly in such cases. If DMA should not write the data for the area to be monitored, one possible way is deleting IOMMU table entries that point to the region when shadow IOMMU tables. Also it may be possible to use the DMA Protected Region (the detail is described in section 4.7), which is available for VT-d and prohibit any DMA to that region, to protect the area from DMA.

4.6.6 Event-Based Memory Acquisition

Since the proposed method uses a VMM, it can detect any event that can be trapped by the VMM and perform memory acquisition on it. For example, the VMM can use nested paging to see if a value other than the pre-specified value is written to a specific memory area and if so stop vCPUs and start memory acquisition. This significantly reduces the possibility of transient attacks.

4.6.7 Challenges

In this section, we describe the challenges in implementing the proposed method and how to overcome them.

4.6.7.1 Dealing with the Increased TCB and Operational Overhead Associated with the Use of VMM

The disadvantage of using a VMM for memory acquisition is the increase in TCB and overhead due to virtualization. When using a general-purpose VMM such as KVM or Xen, the TCB can reach millions of lines, which increases the possibility of attacks on the VMM itself. Therefore, in our method, we use a parapass-through VMM [35] instead of a general-purpose VMM. The

parapass-through VMM supports only one guest OS and passes through most of the devices but traps some devices and memory accesses to perform necessary operations. In this way, the parapass-through VMM can achieve a TCB several hundred times smaller than a conventional general-purpose VMM by limiting its functions. The pass-through nature of the VMM also reduces the overhead of the operation. From the standpoint of the VMM, the proposed method can be regarded as offloading the primary function of memory acquisition from the VMM to the hardware side.

4.6.7.2 Reducing IOMMU shadowing overhead

IOMMU is known for its overhead [90, 91, 97, 108, 136], let alone vIOMMU [45, 114, 116, 152, 153]. To reduce its overhead, we do not virtualize all IOMMU functionalities. Instead, we shadow only some parts of IOMMU where necessary with parapass-through VMM. Parapass through VMM lets OS use most of the hardware as is, thus significantly reducing the overhead and TCB. We present the details of the implementation in the next section.

4.6.7.3 Using interrupts without interfering with the OS

The VMM or the coprocessor must use an interrupt vector that does not affect the OS when sending interrupts. Since we assume arbitrary operation of OS, it is not easy to know which interrupt vector will be used by the OS. Thus, we use NMI (Non-Maskable Interrupt) to cope with this problem. In the x86, NMI has a higher priority than the normal interrupts, and we can configure VM so that it VMEXIT when receives NMI. Furthermore, if a subsequent NMI occurs during the NMI handler processing, the NMI is inserted after the `iret` instruction.

The concrete way of using NMI is as follows. First, the VMM or the coprocessor who wants to send an interrupt writes a value to a specific memory area in the VMM before sending the NMI and then sends the NMI. Next, a NMI handler of the VMM checks the VMM memory area to determine if the NMI is from the VMM or the coprocessor. If so, the VMM performs proper processing. Otherwise, the VMM inserts the NMI into the guest. In this way, the VMM and the coprocessor can send an interrupt without affecting other devices.

4.6.7.4 Dealing with P2P DMA and Translated TLP

As shown in Figure 4.2 (b), if a device is connected to the same switch as the memory acquisition coprocessor, an attacker may take control of the device and attack the memory acquisition coprocessor via P2P DMA. An attacker also may take a control of a device which has ATS capability and try to DMA using Translated TLPs to attack the VMM or the memory acquisition coprocessor. To prevent such an attack, we use the ACS feature so that 1) TLPs always pass through the IOMMU, and 2) PCIe switches block Translated TLPs. This setting is protected by nested paging.

4.7 Implementation

We implemented the prototype of the proposed method on Intel's IOMMU (VT-d) using BitVisor [35], a lightweight parapass-through VMM. BitVisor only supports one guest OS, and its main usage is enhancing security by protecting some memory regions and I/O with nested paging. BitVisor can boot an existing OS image without any modification. Our prototype supports IOMMU shadowing and basic operations including register value acquisition, temporal vCPUs suspension, and monitor specific memory region. The core parts of BitVisor is about 36KLOC. We added about 3KLOC for our implementation.

We use PCIe Screamer R02 [209] for a memory acquisition coprocessor. PCIe Screamer is built on top of XC7A35T Xilinx 7 Series FPGA and has a USB3 interface that can be used to communicate and send acquired memory data to an analytics machine. The analytics machine use PCILeech [226] to control PCIe Screamer. We also create a controller that manage communication between the analytics machine and the VMM in Python.

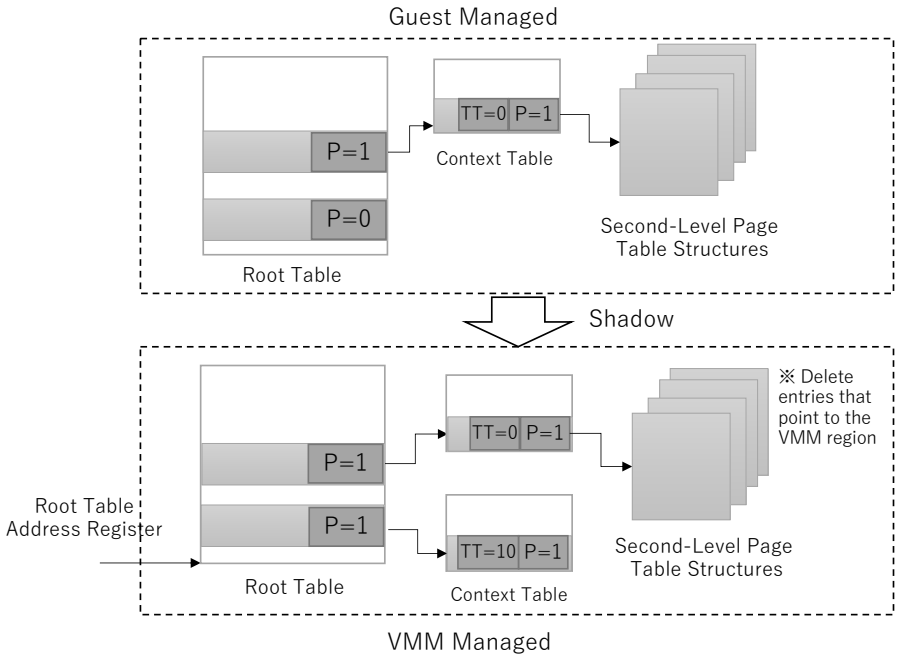
4.7.1 IOMMU Shadowing

Figure 4.5 shows an overview of the shadowing of the IOMMU configuration on Intel VT-d. VT-d has two operation modes [185]. One is legacy mode and the other is scalable mode. The scalable mode is not commercially available at the time of writing, so we use the legacy mode. The basics of the shadowing would be the same for the scalable mode.

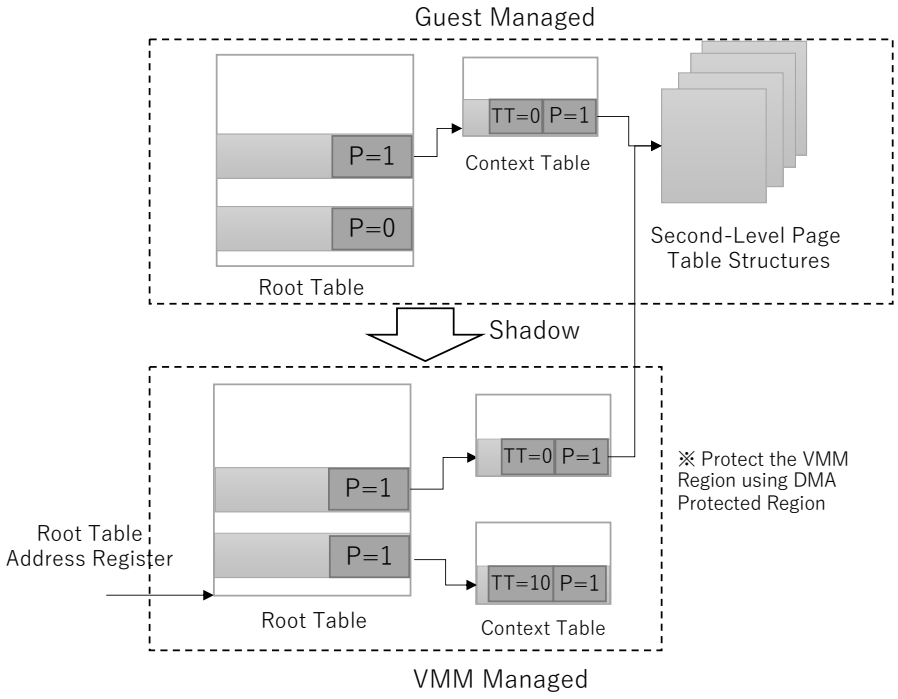
In VT-d, the base address of the IOMMU page table in memory is set in the Root Table Address Register. ACPI DMAR tables contain information on the VT-d, including MMIO location of the Root Table Address Register. The IOMMU page table consists of a Root table, Context table, and Second-Level Page Table Structures. The bus, device, and function numbers of a PCIe device uniquely determine the Root table and Context table entries, and the Second-Level Page Table Structures define the DMA address translation for that device. The Second-Level Page Table Structures resemble 4-level page tables for CPU but no the same.

In VT-d, if the "Present bit (P)" of the entry in the Root Table and Context Table is 1 and the "Translation Type (TT)" of the entry in the Context table is 10, VT-d is in pass-through mode. In this mode, the address in a TLP which the device send is used as-is for DMA.

There are two possible IOMMU shadowing implementations, depending on whether shadowing the Second-Level Page Table Structures or not. Both implementations shadow the root and context entry and create the entry for the memory acquisition coprocessor, which is set to pass-through mode. If we shadow Second-Level Page Table Structures (Figure. 4.5a), at the time of shadowing, we checked table entries and delete entries that point to the VMM region, if any, to protect VMM. PCI specification disallow DMA access to PCI configuration



(a) Shadow All Tables (Shadow)



(b) Shadow Only Context and Root Tables (Shallow Shadow)

Figure 4.5: IOMMU Shadowing Implementation for Intel VT-d (legacy mode). The upper part is a guest managed IOMMU page table and the other a VMM managed shadow IOMMU page table. There are two possible implementations. (a) Shadow all IOMMU tables. When shadowing, 1) add a pass-through entry (`tt=10`) for a memory-acquisition coprocessor in a context table, and 2) delete entries that point to the VMM region in second-level page tables if any. (b) Only shadow root and context tables. In this case the VMM memory including shadowed IOMMU tables is protected using DMA Protected Region mechanism.

space. However, apparently, there are chipsets that allows such DMA operation [50]. In this case, we also delete entries that point to a PCI configuration space of a memory acquisition coprocessor. This approach is the most straightforward and generic way and could implement on any other architectures. If an architecture does not have pass-through mode, then we can create an identity map for a memory acquisition coprocessor. Note that removing such malicious page table entries in a guest memory (without shadowing) is insufficient because there is a possibility of TOCTTOU (Time-Of-Check-To-Time-Of-Use) attacks, in which an attacker rewrites configurations after the VMM rewriting the guest OS's IOMMU page table. So, we need to shadow Second-Level Page Table Structures.

We can eliminate shadowing of the Second-Level Page Table Structures by utilizing VT-d's DMA Protected Region (Figure. 4.5b). DMA Protected Region specifies region which device cannot DMA. VT-d can have two DMA Protected Regions: one in less than 4GB memory region and the other in above 4GB memory region. By using DMA Protected Region, we ensure that the VMM cannot be DMA-ed by a malicious device without checking Second-Level Page Tables.

However, there are two downsides of using DMA Protected Region. Obvious one is the guest cannot use DMA protected region. DMA Protected Region is mainly used for protecting IOMMU page tables before enabling IOMMU. Without DMA Protected Region, the IOMMU page table can be modified by DMA from malicious hardware, as demonstrated in [139]. However, we assume that there is no malicious device at the boot time, and in that case, there is no problem not having DMA Protected Region. Besides, Linux does not use DMA Protected Region at all.

A more serious issue introduced by using DMA Protected Region is that the DMA Protected Region prevents the VMM from communicating with the coprocessor or analytics machine via DMA within the VMM memory. To cope with this problem, the VMM place the memory used for communication between the VMM and the coprocessor or analytics machine outside the DMA Protected Region. Then, the communication messages are encrypted and signed to ensure that messages are not compromised by an attacker.

Shadowing is performed when OS invalidates IOMMU table entries. VT-d specification does not require IOTLB invalidation if an entry is not on the IOTLB. However, without IOTLB invalidation we need to monitor every IOMMU page tables using EPT to detect entry changes, which is costly operation. Thus, we use VT-d's Caching Mode (CM) to request a OS to invalidate IOMMU tables whenever it changes an IOMMU entry even if the entry is not on the IOTLB. CM is reported via VT-d's capability register. The hardware IOMMU implementation reports CM as zero. This field is for aiding vIOMMU implementation. An attacker may ignore CM, but it is not the problem. Because IOMMU page tables are shadowed, and an IOMMU entry for a memory acquisition coprocessor is protected, ignoring CM merely results in incorrect IOMMU page tables for the OS.

Invalidations are performed via register-based interface or queue-based interface. Register-based interface is legacy interface, so we use queue-based interface. Invalidation granularity ranges from device-level to a global level for root and context tables. When we rewrite the root and context table so that a memory coprocessor can DMA, we must invalidate the entry to ensure the changes are in effect. Therefore, the VMM also shadows the invalidation queue, and issues an invalidation when creating an IOMMU entry for a memory acquisition coprocessor. Other than IOMMU shadowing, we let the guests use the IOMMU freely. This reduces the overhead of IOMMU shadowing.

4.7.2 VMM and PCI Configuration Space Protection

EPT (nested paging) protects the VMM memory region including the shadowed IOMMU configuration. This means that the IOMMU configuration of the memory acquisition device is always in pass-through mode, regardless of the IOMMU configuration of the guest, and the memory acquisition device can retrieve the memory contents. MMIO regions of PCI Configuration Space is also protected by the same way so that bus mastering is always enabled. The VMM also intercept PIO access by enabling Unconditional I/O Exiting of VMCS, and protect the PCI Configuration Space from malicious rewritings.

4.7.3 Register Value Acquisition

Intel VT-x's VMCS holds a guest state including register values and the VMM can read values using the `vmread` instruction. The VMM reads register values and send it to the analytics machine using the communication channel described below.

4.7.4 Communication between VMM and Analytics Machine

PCIe Screamer does not have interfaces for communication between the VMM and the analytics machine. So, we used a dedicated NIC (Intel 82574L) and a serial cable for communication between them. The serial cable is used for asynchronous output from the VMM. In our implementation, we do not use NMI for NIC interrupts. Instead, we poll NIC descriptors every time when a vCPU VMExits. We confirmed that this approach worked practically in the experiments. The NIC is protected in the same way as the coprocessor protection.

BitVisor has an ability to build an server using lwIP. We built a simple JSON-RPC [190] server using it. The analytics machine uses this server to achieve several things such as obtaining register values, requesting temporal vCPUs suspension and monitoring memory region for event-based acquisition. The server is easy to extensible and we can add new operations if necessary. Table 4.4 shows the supported operations.

Table 4.4: Supported Operations

Command	Operation
stop_vcpu	Stop vCPUs
start_vcpu	Start vCPUs
get_cr3	Get CR3 value of a vCPU
monitor (addr)	Monitor memory address and notify if accessed

4.8 Evaluation

We use a machine with Intel Core i7 8700 (6 cores, hyper-threading disabled), 16GB memory, and ACS supported PCIe bus as a target machine. The machine runs Linux 5.13.0 and has PCIe-connected NVMe (OCZ RD400/400A) and 40GbE NIC (Intel XL710). PCIe Screamer (R02) is connected to the target machine via a PCIe slot. PCIe Screamer is also connected to an analytics machine using USB3. The analytics machine and the target machine are connected using a 1GbE NIC for the communication with the VMM. These machines are also connected with 40GbE NIC directly. The analytics machine has Intel Core i7 6700 (4 cores 8 threads), 16GB memory.

In Linux, we can use `iommu` and `intel_iommu` kernel parameters [224] to control how OS uses IOMMU. We uses the following three settings in the experiments.

iommu=off
: Linux does not use IOMMU (Linux's default)

iommu=nopt intel_iommu=on
: Linux uses IOMMU (create IOMMU page table per device)

iommu=nopt intel_iommu=on,strict
: Same as "on" except that every `unmap_single` operation will result in a IOTLB flush

iommu=nopt means that Linux uses iommu for intra-memory protection. Note that to improve performance, Linux batches IOTLB flushes by default. However, this causes several security problems [136, 156]. "strict" option forces to flush IOTLB every time unmap region. When using Caching Mode, "strict" option is enforced no matter if the option is specified.

4.8.1 Memory Acquisition in the presence of an IOMMU

iommu=off In this case, Linux does not use IOMMU. So we can extract memory using PCIe Screamer as is.

iommu=nopt and intel_iommu=on(,strict) In this case, Linux creates IOMMU page

Table 4.5: Memory Acquisition Time (1GB)

Name	Time (s)
bare-metal (iommu=off)	17.018
shadow (iommu=nopt, intel_iommu=on,strict)	16.728

Table 4.6: Experiment Settings

Name	Dscription
noiommu	bare-metal machine
on	bare-metal with intel_iommu=on
strict	bare-metal with intel_iommu=on,strict
hook	run OS on a VMM w/o shadowing (iommu is strict)
shallow	only shadow root and context tables
shadow	shadow all IOMMU tables
qemu	run OS on a KVM/QEMU with its vIOMMU [175]

tables per IOMMU group [1]. The PCIe Screamer looks a ethernet controller made by Xilinx in a Linux's point of view[2]. Linux does not create IOMMU page tables for the PCIe Screamer because there is no driver for it. As a result, all DMA request from PCIe Screamer is blocked by IOMMU.

Next, we employed our proposed method and running OS on the VMM. We confirmed that in this case, the PCIe screamer successfully DMA and acquire memory. As the same as the `iommu=off`, Linux does not create page tables for it. We confirmed it by checking IOMMU page tables that Linux created. However, our VMM successfully shadowed IOMMU page tables, thus the PCIe Screamer could DMA.

Table 4.5 shows times taken to acquire 1GB of memory by our method. Since our method uses DMA, we did not see any noticeable differences between bare-metal and our proposed method. In our environment, PCIe Screamer fetched memory around 60MB/s.

4.8.2 Overhead Evaluation

Table 4.6 shows the experiment settings used in this evaluation. When experimenting using VMM, iommu is used with strict mode. When experimenting on QEMU/KVM, NVMe and 40GbE NIC devices are pass-through-ed using VFIO [197]. We use "iommu=on intel_iommu=on,strict" for "hook", "shallow", "shadow", "qemu".

[1] Devices belonging to the same IOMMU group can communicate with each other without IOMMU intervention. For example, a multiple-function device belongs to an IOMMU group containing each function as an endpoint. We can configure ACS to enforce every DMA transaction going through IOMMU, and in that case, each endpoint belongs to each IOMMU group.

[2] Linux see the device's class id and vendor id in a PCI configuration space.

Table 4.7: IOMMU Map and Unmap Time Comparison (32MB region)

Name	Map (ns)	Unmap (ns)
noiommu	N/A	N/A
on	2,921,377	4,116,167
strict	2,926,536	4,621,587
hook	2,900,304	14,976,643
shallow	14,309,282	11,715,443
shadow	17,406,628	21,976,681
qemu†	N/A	N/A

† qemu's results are omitted because we encountered VFIO errors.

4.8.2.1 Microbenchmark

To measure the shadowing overhead, we create a program that maps and unmaps IOMMU tables using VFIO for 32MB memory region for 500 times. Table 4.5 shows the median times of the experiments. When using IOMMU on bare-metal machine, map and unmap operation takes about 2 to 4 milliseconds. Our proposed method takes about 10 to 20 milliseconds.

4.8.2.2 NVMe and 40GbE NIC throughput and latency

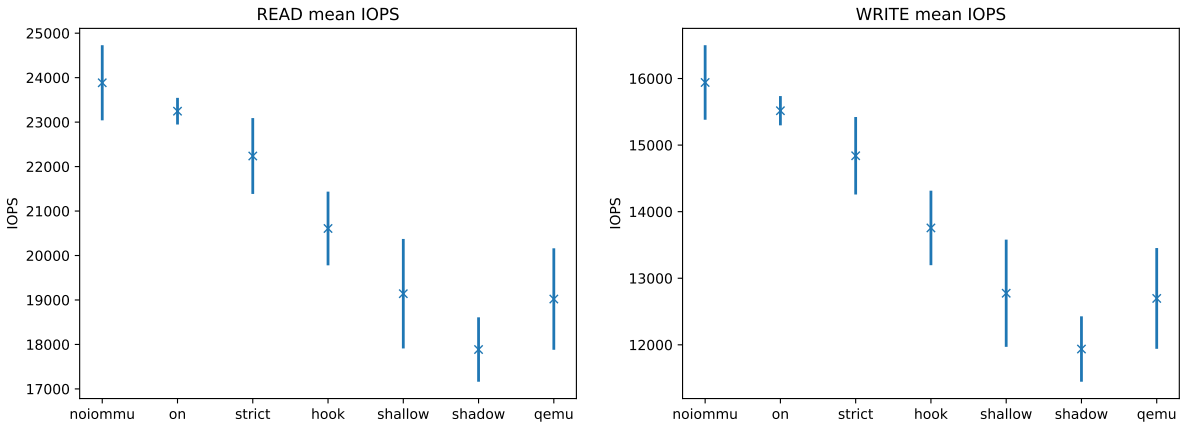
We use NVMe and 40GbE NIC to measure IOMMU shadowing overhead during normal operations. When experimenting on QEMU/KVM, these devices are pass-throgh-ed, and we assigned 6 vCPUs and 14GB memory. Figure 4.6 and Figure 4.7 shows NVM's throughput and latency measured by fio [189]. The fio performs random read and write with block size 4KB, numjobs 4, and iodepth 16. 4.8a shows 40GbE NIC throughput measured by netperf [178] with one thread and 4.8b shows latency measured by ping.

4.8.2.3 NAS Parallel Benchmark

Table 4.8 shows the normalized execution time of NAS Parallel Benchmark (NPB) [201] to demonstrate the CPU overhead of the proposed method. "spincount" is a value of "GOMP_SPINCOUNT", which determines how long to try to spin to get a lock.

4.8.2.4 Benchmark Summary

As shown in the graph, our implementation generally better than the QEMU/KVM's vIOMMU. This is because our IOMMU shadowing is parapass-through approach. Although our proposal has a non-negligible overhead, we think our method works practically enough compared to a traditional vIOMMU mechanism (qemu's vIOMMU). VT-d has added new features to increase IOMMU performance. For example, a recent VT-d has a cache-coherency for an IOMMU page table (our VT-d does not have it.) Therefore, when creating IOMMU page tables, we do not



(a) Read (b) Write

Figure 4.6: NVMe fio IOPS with error bars of standard deviation (higher is better)

Table 4.8: NPB Normalized Result (baseline: bare-metal (iommu=strict))

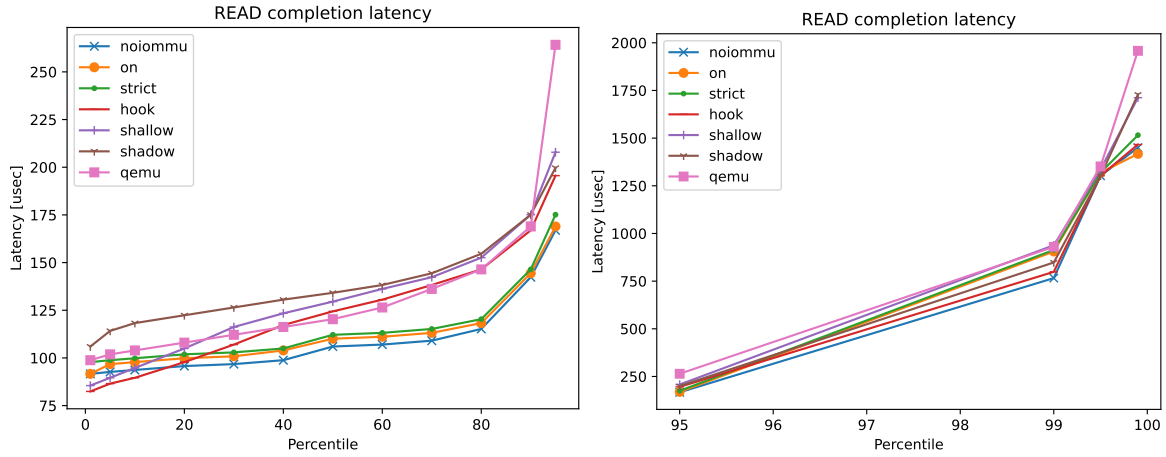
	shadow			qemu		
spincount→	0	300k	30b	0	300k	30b
bt	1.00	1.01	1.07	1.08	1.00	1.08
cg	1.02	1.02	1.00	1.02	1.01	1.00
ep	0.94	1.00	1.07	1.11	1.00	1.13
ft	0.99	0.99	1.06	1.46	1.42	1.47
is	1.00	1.00	1.00	1.00	1.00	1.00
lu	1.24	1.02	1.05	1.05	1.02	1.24
mg	1.02	1.02	1.00	1.01	1.02	1.01
sp	1.00	1.02	1.00	1.01	1.02	1.01
ua	1.03	1.03	1.00	1.05	1.01	1.01
gmean	1.02	1.01	1.03	1.08	1.05	1.10

have to flush CPU caches. We expect hardware improvement will decrease performance overhead more and more.

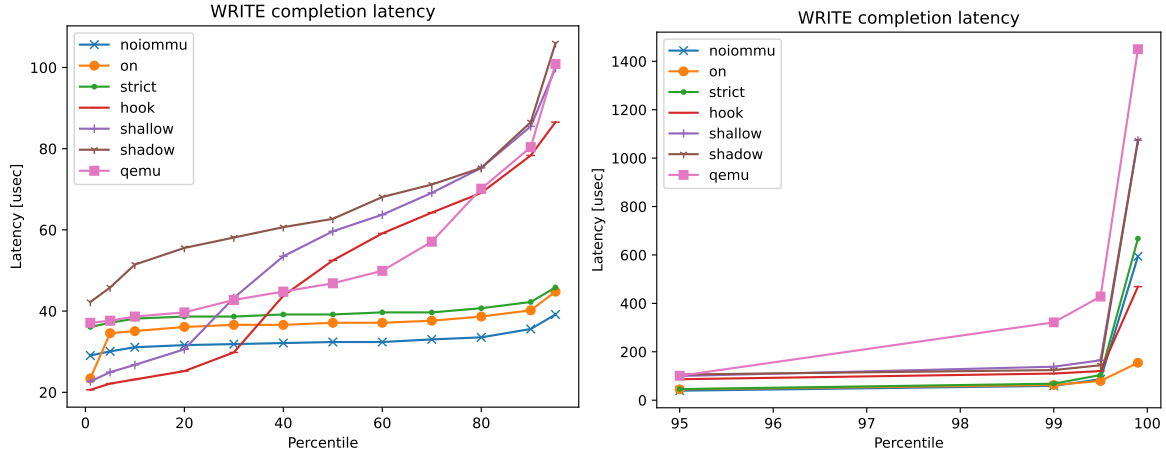
4.9 Discussion

4.9.1 SMM Monitoring

Since the proposed method uses a VMM, the monitoring of SMM – i.e., states with higher privilege levels than a VMM – is out of the scope of the proposed method. Chevalier et al., [118] proposes a method to monitor the integrity of the SMM from the coprocessor by using the coprocessor and the compile-time instrumentation of the SMM program. Nighthawk [147] propose a method to monitor the integrity of the SMM from the coprocessor by using a special coprocessor called Intel ME [229], which has a higher privilege than the SMM. Our proposed



(a) Read



(b) Write

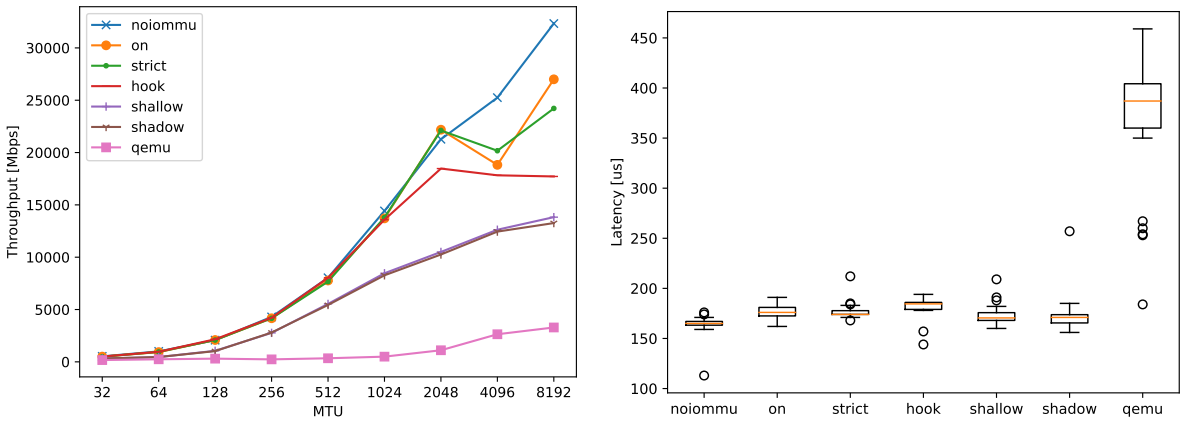
Figure 4.7: NVMe fio Latency (up to 99.9%-tile) (lower is better)

method is orthogonal to these methods; by combining them, we can monitor the SMM.

In recent years, the Platform Runtime Mechanism (PRM) [211] has been proposed to make most of SMM functions implemented outside the SMM. Intel also introduced the SMI Transfer Monitor (STM) [218], used for increasing the security of the SMM by virtualizing the SMM and running the main processing of the SMM in VMX non-root mode. By using these methods together, it is possible to significantly reduce the attack surface of the SMM.

4.9.2 Guest Hypervisor Support

Coprocessors can acquire VMM's memory region. However, since the proposed method uses virtualization, the guest cannot use the virtualization function as is. If the VMM used by the guest is a Type-1 VMM, the proposed method can be implemented in the VMM, but the TCB will increase. If the VMM used by the guest is a Type-2 VMM, the proposed method cannot be implemented in the VMM because the OS itself runs outside the VMM.



(a) Netperf Throughput (higher is better)

(b) Ping Latency (lower is better)

Figure 4.8: 40GbE NIC Experiments

In order to solve this problem, we can utilize nested virtualization [38]. With nested virtualization, the OS can use the virtualization function even with the proposed method while keeping the TCB small.

4.9.3 Hardware-Based Memory Encryption

In recent years, some CPUs have been equipped with, or are scheduled to be equipped with, a function that constantly encrypts part or all of the memory [166, 168, 182, 183, 184]. In this case, memory can be divided into two types: DMA-able (and decryptable), and non-DMA-able. When verifying the integrity of the memory, the data acquired from DMA can be left encrypted, but when analyzing the contents, the acquired data must be decrypted.

Intel TME [182] and AMD SME [166] are DMA-able and have decryptable memory contents. One of the primary applications is the encryption of non-volatile memory. This provides page-by-page encryption functions. Specifically, the upper bits of the physical address that are not used in the page table are used to specify whether to decrypt or encrypt data when reading or writing to the memory. By setting the upper bits of the physical address correctly, encrypted data can be decrypted and read from the DMA. It is possible to find out which page is encrypted by tracing the page table from CR3 and looking at the physical address settings in the page table. However, in the event of a timing attack, there is a possibility that the information in the page table could be rewritten by the attacker and thus could not be decrypted correctly. One possible countermeasure is to shorten the interval between memory fetches or DMA by trying all possible combinations of the upper bits of the encryption setting until the correct data can be read.

The memory encryption technologies that cannot be DMA-ed are Intel SGX [183], Intel TDX [184], Arm CCA [168], and AMD SEV [166]. These are features that provide a TEE

(Trusted Execution Environment). If we want to obtain the data or code executed in these systems, one approach is to run an agent inside the monitored system and have the agent periodically copying the memory of the part we want to obtain to the shared memory that can be DMA-ed, after encrypting and signing it with keys shared with the coprocessor in advance. Then, the coprocessor can perform the desired processing by retrieving data from the shared memory. However, in this case, if the internal agent itself is attacked, the data copying may not be performed correctly, so it is essential to take thorough preventive measures to protect the agent itself from attacks.

4.9.4 Possible Hardware Improvement

In this section, we discuss how to extend the hardware functions to realize the above goals.

4.9.4.1 Device-Selective DMA Protected Region

We presented how to use VT-d's DMA Protected Region to implement our proposed method. The DMA Protected Region is applied to all devices, so we need to place communication massage outside of a VMM, and encrypt and sign messages when communicating between a VMM and a coprosessor or an analytics machine. If the DMA Protected Region can be selected for a device, the memory acquisition coprocessor can be set as DMA-able and all other devices as DMA-disabled, allowing communication between the memory acquisition coprocessor and the VMM while preventing DMA attacks on the VMM from other devices. Although Intel plans to deprecate DMA Protected Region[3] [185], we believe the functionality is useful for a coprocessor cooperating with a VMM.

4.9.4.2 Hypervisor-Managed IOMMU Translation

If hardware-based nested IOMMU can be available in the same way as nested paging in CPUs, VMMs do not need to perform software-based shadowing of IOMMU page tables as we did. AMD has been providing a hardware-based two-stage IOMMU page table feature since Zen2, called hardware-based vIOMMU [165]. Intel is also planning to include such a feature in its next-generation VT-d Scalable I/O Virtualization [185]. However, these features are based on PCIe's PASID specification, and the structure of a first IOMMU page tables is the same structure as the CPU's page tables. Therefore we cannot directly use it to override IOMMU settings that a guest creates. From the security point of view, it is handy if the first IOMMU page table is the same as the second IOMMU page table.

[3]Intel plans to introduce alternative way to prevent DMA attack to IOMMU page tables when setting Root Table Address Register by temporary disable all DMA transactions.

4.9.4.3 Configuration Lockdown

There are several methods available to prevent some settings from being changed in the software in the BIOS. For example, when the Serial Peripheral Interface (SPI) Configuration Lockdown function is enabled, the SPI settings written up to at the point cannot be changed until the PC is rebooted [219]. In this study, we used nested paging for the memory area that protects the configuration, such as the PCI configuration space, but if such a lockdown mechanism exists, memory protection by a VMM can be omitted in that area.

4.9.4.4 Summary of Possible Hardware Improvements

In summary, we believe that our proposed method can be implemented most efficiently if the following hardware features are available:

1. DMA Protected Region configurable per device and
2. Lockdown of DMA Protected Region and PCIe settings.

These features eliminate the need for shadowing the IOMMU page table and implementing PCI configuration space protection. However, they do not eliminate the need for the VMM. Communicating with the memory acquisition coprocessor, hooking specific events, and stopping the vCPU temporary are processes that the VMM performs. Of course, it may be possible to implement these processes in hardware, but considering their complexity, it would be better to use a VMM.

4.10 Summary

Due to the progress of hardware technology, memory acquisition by coprocessors has become a practical method. On the other hand, there has been little discussion of its challenges, especially about IOMMU. In this study, we organized and presented the four problems of coprocessor-based memory acquisition. Then we proposed Shielded Copilot, which enables trustworthy coprosessor-based memory acquisition in the presence of an IOMMU. Our method cooperate a coprocessor with a thin VMM, which is in charge for operating what a coprocessor cannot achieve. We implemented the prototype of proposed method for Intel's VT-d. Our evaluation showed that optimizing IOMMU for device protection achieves higher performance than traditional vIOMMU and the proposed method worked practically.

5 Investigating and Improving Scheduling Performance of NUMA-visible Virtual Machines

In this chapter, we evaluated the scheduling performance of a NUMA-visible virtual machine on Linux using various benchmarks. To our surprise, we found several problems that cause severe performance degradation due to a paravirtualization function which is desirable for non-NUMA-visible VMs. We propose the fix and show the effectiveness of the proposed method.

5.1 Introduction

In modern computers, the CPU and memory are structured hierarchically, as shown in the Figure 5.1. A CPU consists of LLC and multiple cores that each contain L1 and L2 caches[1]. Machines with multiple CPUs generally use an architectural configuration called NUMA (non-uniform memory access). In a NUMA machine, the CPU and memory are divided into pairs called nodes, and the speed of memory access from the CPU on one node differs from that on other nodes.

To derive the best performance on NUMA machines, the operating systems, and the scheduler should consider the hardware's characteristics. For example, improving the speed of memory access necessitates that the data used by an application be placed in memory on a node located as close as possible to the CPU where the application runs. Scheduling threads in the same node can also improve the efficiency of cache utilization. This hardware hierarchy is considered in task scheduling performed by Linux's completely fair scheduler (CFS) [172].

On the other hand, a host's hardware configuration is typically hidden from a guest in a virtualized environment, such as clouds. In such an environment, the guest cannot operate in consideration of the host's hardware configuration. This problem can be solved by revealing the host's hardware configuration to the guest. These virtual machines are particularly used for requiring high-performance tasks, such as machine learning and high-performance computing. For example, AWS EC2 [163], a leading cloud provider, rents out virtual NUMA (vNUMA)

[1] In some CPUs, the L2 cache is shared among cores.

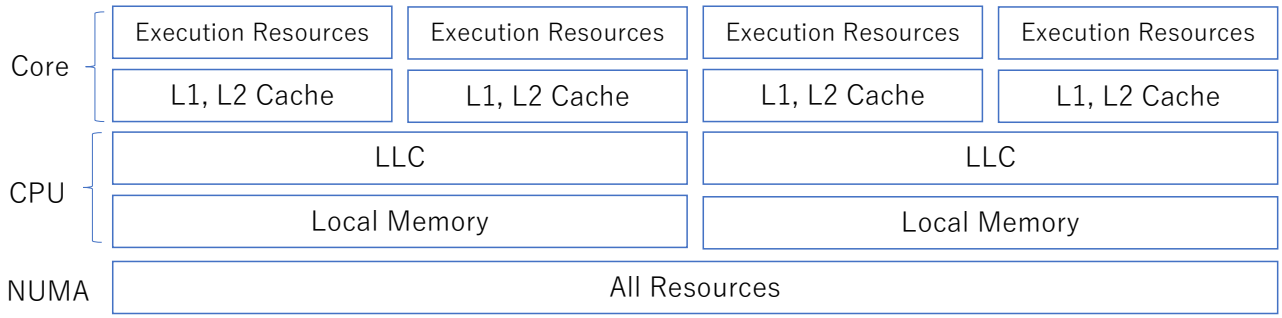


Figure 5.1: Hierarchical Hardware Structure

instances with 96 vCPUs, 192 GiB of memory.

Several studies have reported that VM performance improves when a guest replicates a host's hardware configuration [48, 55, 79, 110, 143, 155]. However, all such research evaluated only a small number of workloads, and few studies have conducted detailed experiments.

The hardware configuration of a host can be reproduced in several ways, depending on the extent of the intended configuration. For example, when creating a virtual machine with 24 vCPUs, it is possible to create a machine with 24 sockets, or not, a machine with 24 cores per socket. These virtual machines may look the same, but in reality, they have different hardware configurations, which result in different scheduling behavior. In addition, recent years have seen the common use of paravirtualization functions in virtualized environments, but most of these functions are designed to improve performance in an overcommitted environment. How effective they are for virtual machines that reproduce a host configuration is not evaluated in the previous studies.

In this study, we created several virtual machines which reproduced (parts of) the hardware configuration of a host using Linux's QEMU/KVM and conducted the following detailed performance evaluations:

1. Evaluating the effectiveness of paravirtualization for vNUMA machines
2. Comparing the vNUMA performance with various virtual hardware configurations
3. Evaluating the performance of VMs that partially replicate host hardware configuration in an overcommitted environment

Contrary to expectations, we found that some workloads cause significant performance degradation relative to that occurring in bare-metal and virtual machines that do not replicate a host's hardware configuration. We analyzed these problems and found several causes.

The contributions of this research are as follows.

1. We performed the detailed performance evaluation on virtual machines with several virtual hardware configurations on Linux, and found several performance degradation problems.

2. We analyzes several problems we found and proposed solutions.

5.2 Background

5.2.1 NUMA

NUMA architecture is a type of hardware design in which the CPU and memory are divided into multiple units called nodes. It is distinct from UMA (uniform memory access) architecture, and in NUMA architecture, the distance (access speed) from the CPU to memory differs, depending on the location of both nodes. Memory located on the same node as the CPU is called local memory, and another type of memory is called remote memory. Access to these is called local and remote access, respectively. Partitioning by node leads to less contention for memory access, thereby enabling memory scaling. Today, NUMA architectures are commonly used on machines with sizable CPUs and memory. To ensure the best performance on a NUMA machine, it is crucial for the CPU on which an application runs and the memory that contains the application data to be located on the same or closely positioned nodes.

5.2.1.1 Memory Access Policies for NUMA

Two typical policies are implemented to access memory on NUMA machines: First Touch and Interleave. The First Touch policy allocates memory from the node where the CPU that accesses memory is located. This policy is effective for applications in which a separate thread runs on each CPU, but it can be ineffective for applications wherein a specific thread initializes all memory or wherein threads are often migrated across nodes because of overcommitment. In contrast, the Interleave policy allocates memory alternately from each node when it issues memory. It is, therefore, suitable for applications in which data are accessed from multiple nodes.

In Linux, the First Touch policy is used by default, but it is possible to run applications with the Interleave policy by using the `numactl` command [203]. Linux also has automatic NUMA balancing (ANB), which dynamically improves NUMA memory access by periodically and intentionally causing page faults and migrating data to local memory when necessary.

5.2.2 Reproducing NUMA in a Virtualized Environment

5.2.2.1 Motivation

In a normal virtualized environment, the hardware configuration of the host is hidden. This makes it easy to change the virtual machine's hardware configuration and migrate between

machines with different hardware configurations. However, from the performance point of view, there are several problems.

First of all, since the guest cannot see the host hardware configuration, it is hard to exploit NUMA architecture from the guest. In addition, in an overcommitted environment, "double scheduling problems", which is caused by the dual scheduling in the guest and the host, can occur. Most common double scheduling problems is the lock holder preemption (LHP) problem [16, 27, 127], which occurs when a vCPU that holds a spinlock is preempted. At this stage, other vCPUs that want to acquire the spinlock must wait until the preempted vCPU is rescheduled and the lock is released; scheduling the vCPU that is waiting for the lock first wastes CPU time. This problem is especially significant when parallel programs with many exclusive controls are executed.

Other issues include the lock waiter preemption (LWP) problem [16, 127], which arises from the preemption of a vCPU that is waiting for the highest priority in an ordered lock; the blocked-waiter wakeup (BWW) problem [80], which occurs when a vCPU waiting for a block to be released given a slow IPI in a virtualized environment is woken up; and problems caused by the preemption of a vCPU in an interrupt or RCU context [124, 134]. In this context, the BWW problem can also occur. Even if a vCPU is not preempted in a critical section, failing to schedule its I/O completion via a virtual machine causes I/O delays.

These problem can essentially be solved by allocating a dedicated pCPU to a virtual machine, replicating hardware configuration of the host, and avoiding overcommitment. Such virtual machines are used in where performance is important.

5.2.2.2 Reproducing a host's CPU configuration

CPU configuration information is included in the Processor Properties Topology Table of the ACPI (Advanced Configuration and Power Interface). Therefore, a specific CPU hierarchy can be reproduced on a guest by appropriately configuring the ACPI at the time of guest startup. In this research, the CPU created on a guest is called a "vCPU," whereas that located on a host is called a "pCPU." We explicitly refer to the CPU core as "vCore" or "pCore."

CPU pinning is the process of narrowing down the number of host pCPUs that schedule a guest's vCPU to one and ensuring a host pCPU and vCPU correspondence of 1:1. CPU pinning reproduces an environment closer to that of a host for a guest.

5.2.2.3 Reproducing a host's NUMA configuration

NUMA configuration information is included in the ACPI Static Resource Affinity Table (SRAT). When a virtual machine initiates, NUMA can be configured on this machine by appropriately constructing this SRAT. A virtual machine with NUMA configured on a guest

is called virtual NUMA (vNUMA), or a NUMA-visible virtual machine. Major hypervisors, such as Hyper-V, QEMU/KVM, Xen, and VMware, support vNUMA.

When vNUMA is used, the NUMA structure is normally reproduced on a guest in accordance with a host's NUMA configuration. In other words, vCPU pinning and memory allocation are performed so that the NUMA reproduced on the guest corresponds to the NUMA of the host. In this study, vNUMA refers to the correspondence between the NUMA configurations of a guest and a host. With vNUMA, the guest can take advantage of the physical NUMA configuration for scheduling and memory management. This process is desirable in terms of application performance, given that the semantic gap [6] is smaller than that generated when the NUMA configuration is hidden from a guest and when vCPU scheduling and memory allocation are devised on a host.

5.2.3 Scheduling in Linux

In Linux, there are several scheduler classes and each thread belongs to a specific scheduler class. Each scheduler class has a given priority, and Linux schedules threads starting from the scheduler class with the highest priority. The scheduler searches for the next task to be executed for each scheduler class and switches tasks when the next task is available.

5.2.3.1 Complete Fair Scheduler (CFS)

Normal threads belong to the `SCHED_NORMAL` (also called `SCHED_OTHER`) scheduling class. This section describes the CFS used in this class.

The CFS [172], which is the main scheduler for Linux, was introduced in Linux 2.6.23 and remains in use today. It uses a weighted fair queueing algorithm and does not have the concept of a fixed time slice. Instead, it allocates execution time in accordance with the weight of each thread so that each thread is scheduled at least once during a period called the latency target. Note that a minimum time slice is set, thus preventing an excessively short execution time. The weight used here is called the nice value, and the weighted execution time assigned to a thread is called vruntime. Threads with the same weight are assigned the same amount of execution time. Each executable thread is stored in the run queue of a CPU. The CFS selects the thread with the smallest vruntime in the run queue as the next thread to be executed.

If a thread goes to sleep, the minimum vruntime is subtracted from the thread's vruntime. Then, if the thread wakes up from sleep, the minimum vruntime of the run queue is added to the vruntime. This adjustment prevents excessive CPU allocation to sleeping threads and enables the rapid scheduling of threads that frequently sleep (interactive threads) at wake up.

In the case of multi-threaded (or multi-process) programs, simply allocating CPU time to each thread results in excessive CPU time for a program. Using the cgroup function introduced in Linux 2.6.38 [173] enables the combination of multiple threads (or threads) into one and

their treatment as a single scheduling entity, thereby preventing excessive CPU time from being allocated to a specific program.

5.2.3.2 Load balancing

The CFS runs on each CPU. To eliminate unbalanced run queues on each CPU, the scheduler performs load balancing as needed. The basic idea of load balancing is to balance the load calculated from CPU utilization and thread weights on each CPU. Another essential task is to avoid CPU resource contention to improve performance. For example, in the case of a multi-threaded program that shares memory, load balancing on the same LLC core is more efficient in cache utilization.

In the CFS, load balancing is performed from the lower domain of a hierarchical domain called the scheduler domain [215] to evenly distribute loads across domains. For the architecture shown in the Figure 5.1, the scheduler domain consists of three domains from the top: Core, CPU, and NUMA.

There are four types of CFS load balancing:

Fork/exec Balancing This load balancing occurs when thread forks or execs. The scheduler selects which CPU's run queue to insert a thread into.

Wakeup Balancing This process involves the selection of which CPU's run queue a thread will be inserted into when the thread wakes up. No balancing across NUMA is performed to improve cache efficiency.

Idle Balancing Idle balancing is performed when the run queue of a CPU becomes empty. If there is an executable thread in the run queue of another CPU, the thread is migrated to its own run queue. If no thread is available, the CPU becomes idle. The CPU is eventually woken up by an iterrupt or periodic load balancing.

Periodic Balancing Periodic balancing is performed in the SOFTIRQ context in response to periodic timer interrupts. In the case of NOHZ kernels that do not generate periodic timer interrupts, which are the current mainstream occurrence, the kernel checks whether there is an idle CPU when an active CPU receives a scheduler tick. If so, the kernel sends an IPI to that CPU to request load balancing. After that, the load balancing process is basically the same as that occurring during the idle balancing.

The fork/exec and wakeup balancing processes entail selecting a run queue to store threads. Idle and periodic balancing are processes of fetching threads from the run queues of other CPUs to one's run queue. The triggers and targets of each load balancing round are summarized in Table 5.1. To avoid unnecessary repetition, `load_balance()` is performed by the first idle

Table 5.1: Summary of Linux Load Balancing

Type of Load Balance	Maximum Scheduler Domain	Main function
fork/exec	CPU	`find_idlest_cpu()`
wakeup	CPU	`select_idle_sibling()`
idle	NUMA	`load_balance()`
periodic	NUMA	`load_balance()`

CPU found when the target CPU is iterated in the domain or only by the first CPU if there is no idle CPU.

5.2.4 KVM

KVM is the default virtualization mechanism for Linux. KVM has the following paravirtualization features.

Asynchronous Page Fault Allowing the guest to run while the guest page swapping in.

PV EOI Omit accessing EOI (End of Interrupt) register (thus omitting VMEXIT) when handling a virtual interrupt.

PV IPI Insert IPI when performing VMENTRY if the vCPU is not running at the time.

PV TLB Flush Flush TLB when performing VMENTRY if the vCPU is not running at the time.

PV UNHALT Use paravitualized spinlock. Instead of doing busy loop, a vCPU waiting a lock sleeps. When the lock holder release the lock, it wakes up the vCPU.

Stealtime The mechanism to notify the guest information on how much CPU time is spend in a hypervisor. Combining with PV UNHALT, this feature used for dectection of vCPU preemption, which is described in the next section.

5.2.4.1 Detection of vCPU preemption

KVM also has a paravirtualization feature to detect whether a vCPU is preempted or not. The function `available_idle_cpu()` (Listing 5.1) is used to determine whether a CPU is idle when load balancing is performed. This function is enabled when KVM's paravirtualization feature "steal time" and "PV UNHALT" are activated. For a guest to tell a host whether a vCPU is preempted, the former sets a memory address in a dedicated MSR (`MSR_KVM_STEAL_TIME`) at boot time, after which the host writes information about whether the CPU is preempted to this memory area. With this feature, the CFS on a virtual machine does not execute load

balancing for the vCPU preempted by the host, even if the vCPU is idle. This feature was introduced in Linux 4.18.

```
int available_idle_cpu(int cpu)
{
    if (!idle_cpu(cpu))
        return 0;

    if (vcpu_is_preempted(cpu))
        return 0;

    return 1;
}
```

Listing 5.1: `available_idle_cpu()` function (defined in kernel/core/sched.c)

5.2.5 Research Questions

Although intuitively it is believed that constructing a NUMA-visible virtual machine is the best way to elicit performance on a NUMA machine in a virtual environment, few studies have evaluated the details performance. In this study, we created several virtual machines which reproduced (parts of) the hardware configuration of a host using Linux's QEMU/KVM and conducted the detailed performance experiments to evaluate the followings:

1. The effectiveness of paravirtualization for vNUMA machines
2. The vNUMA performance on various virtual hardware configurations
3. The performance of VMs that partially replicate host hardware configuration in an overcommitted environment

5.3 Experimental Setup

In this section, we briefly summarize the experiment environment and benchmarks we used.

5.3.1 Experimental Environment

The evaluation machine has two NUMA nodes, with one node having 32 GB of memory. The machine has two Intel Xeon Platinum 8160 CPUs. Each CPU has 24 cores, and hyper-threading is disabled. Functions that dynamically change the CPU frequency, such as C State

and Turbo Boost, are also disabled. We use Linux 5.13 for both the host and the guest OSs. We use QEMU [212] 6.1.0 and libvirt [195] 7.7.0 for creating virtual machines.

5.3.2 Virtual Machines

We use virtual machines depicted in Figure 5.2 for the main evaluation. The description of each VM is as follows:

UMA A traditional virtual machine.

CPU Pinning (pinning) Only CPU pinning is performed.

vNUMA (non-LLC-shared) vNUMA is created. However, there is no CPU core which shares LLC.

vNUMA (LLC-shared) vNUMA with LLC-shared among CPU cores.

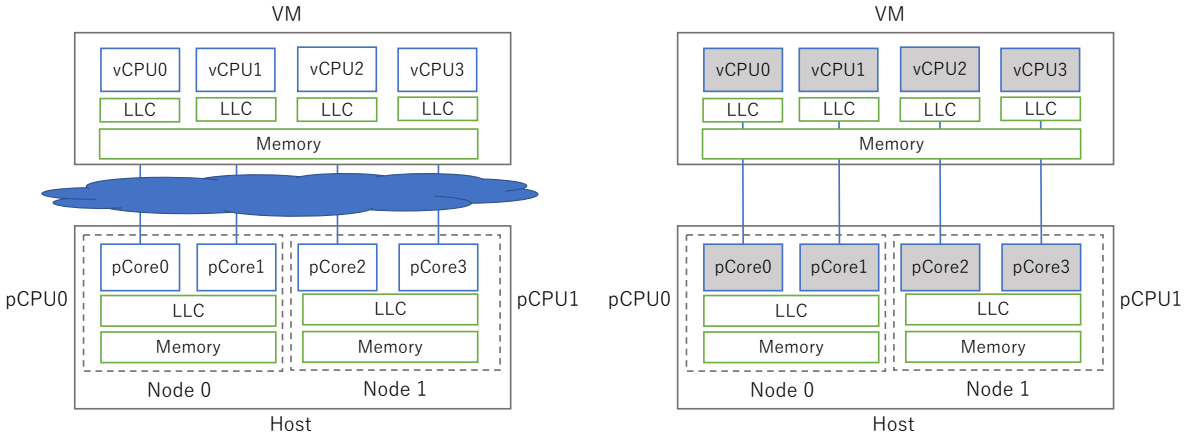
5.3.3 Benchmarks

We used the following benchmarks in the experiments:

5.3.3.1 Performance Evaluation

To evaluate performance throughput, we used the OpenMP version of NAS Parallel Benchmarks (NPB) [201] 3.4.1 and the pthread version of parsec [225] 3.0. The NPB and parsec contains more than a dozen parallel computing programs. We executed all the programs in NPB, except "dc," which requires substantial disk I/O. We use data size C and GNU "libomp" for the OpenMP library. We excluded some Parsec programs that failed to compile in the experimental environment.

In OpenMP, the `OMP_WAIT_POLICY` environment variable [204] can be used to specify how long a thread waits in a user space (i.e., whether a spin loop is performed) for a lock release. Valid values for `OMP_WAIT_POLICY` are "active" or "passive," and its behavior is implementation dependent. In our experimental environment, the number of spin loops (SPINCOUNT) is 0 when `OMP_WAIT_POLICY` is passive, 300,000 when it is not set (300k), and 30,000,000,000 (30b) when it is active. We ran the experiment for each spinlock setting. The larger the number of spin loops, the higher the probability that a lock will be released during a spin loop; acquiring a lock during this loop advances the fastest lock acquisition, which generally enhances software performance. When the spin loop count reaches the upper limit, a thread sleeps until the lock becomes available through the use of the futex [177] system call.



(a) UMA

(b) CPU Pinning

(c) vNUMA (non-LLC-shared)

(d) vNUMA (LLC-shared)

Figure 5.2: Virtual Machines used in the vNUMA Experiments

5.3.3.2 Scheduler Evaluation

We used the perf bench sched messaging [210] to evaluate performance of scheduler and IPC. In perf bench sched messaging, the parameter "groups" controls the total number of threads used in the experiment. We also used schbench [191] to evaluate the latency distribution for the scheduler wakeups. In schbench, the parameter "threads" controls the total number of threads used in the experiment.

5.3.3.3 Measurement Methodology

Except schbench, which reports distribution, we run experiments three times and reports the median value. We use tsc clock source in a virtual machine to measure time. We disable automatic NUMA balancing during the experiments.

5.4 Evaluation of Paravirtual Features on a NUMA-visible Virtual Machine

In this section, we evaluate KVM paravirtual features on a vNUMA (LLC-shared) machines to investigate the effectiveness of paravirtualization for vNUMA machines. We use First Touch NUMA policy for this experiment. We ran experiments on VMs each of which enables the following each KVM paravirtual feature.

nopv No paravirtual festures.

asyncpf Enable Asynchronous Page Fault.

pv_eoi Enable PV EOI.

pv_ipi Enable PV IPI.

pv_tlbflush Enable PV TLB Flush.

steltime Enable stealtime.

unhalt Enable PV UNHALT.

unhalt_stealtime Enable both PV UNHALT and stealtime.

unhalt_stealtime* Enable both PV UNHALT and stealtime with the fix presented in subsection 5.4.2.

vnuma* Enable all paravirtualization features with the fix presented in subsection 5.4.2.

We report relative values against "nopv" for NPB, parsec, and perf sched benchmarks. Lower is better for all graphs.

5.4.1 Result

Figure 5.3 shows the result of NPB benchmark and Figure 5.4 shows the result of parsec. Figure 5.5 and Figure 5.6 shows the result of perf bench sched and schbench, respectively. From the figtures, we can observed that the "unhalt_stealtime" virtual machine showed the severe performance degradation especially for NPB SPINCOUNT 0.

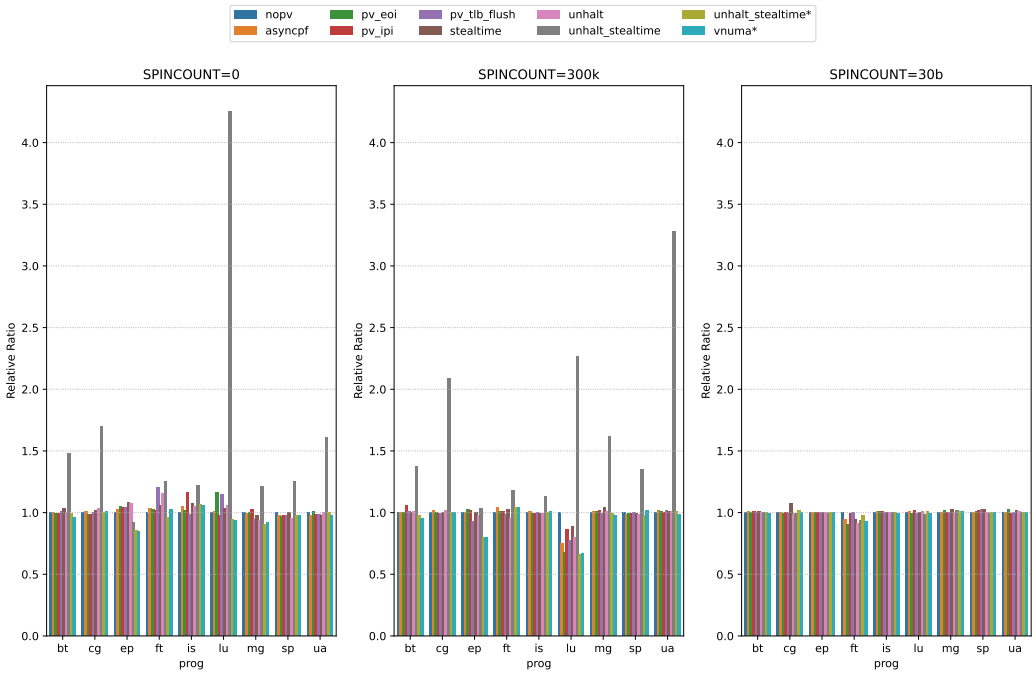


Figure 5.3: NPB Benchmark

5.4.2 False Preempted Problem

We investigated the problem of the "unhalt_stealtime" and found the cause.

5.4.2.1 The Cause

This problem occurs under the following conditions:

- PV SPINLOCK (`KVM_FEATURE_PV_UNHALT`) is enabled.
- "steal time" (`KVM_FEATURE_STEAL_TIME`) function is enabled.
- The CPU in a VM is multi-core. (This problem does not happen on a non-LLC-shared vNUMA machine.)

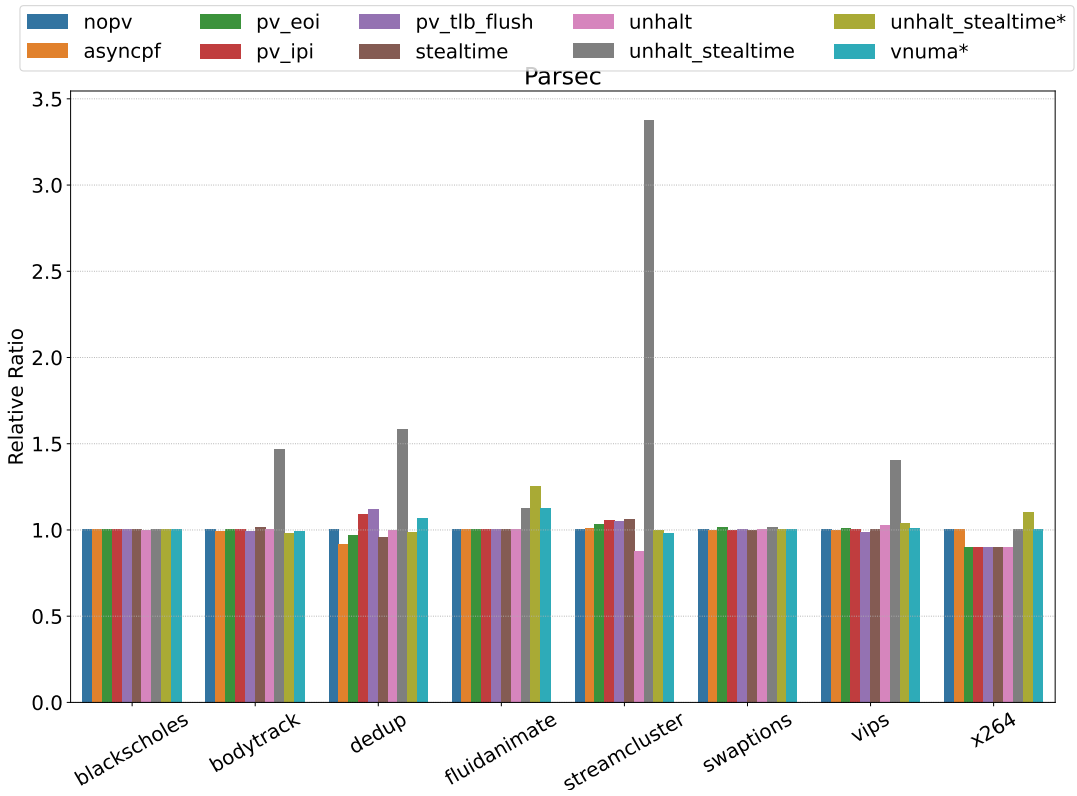


Figure 5.4: Parsec

As described in subsubsection 5.2.4.1, under these condition, a preempted vCPU is excluded from candidates of wake up load balancing. This seems reasonable because a thread migrated to a preempted vCPU needs to wait until it is scheduled on the host side. However, when a vCPU becoming idle and doing HLT VMEXIT, the problem occur. This vCPU is marked as preempted and will wait for a rescheduling IPI from another vCPU or an external interrupt. However, the vCPU is flagged as preempted and is exempted from wakeup load balancing in the CPU scheduler domain. As a result, even if nothing else is running on the host side, the guest assumes that the vCPU is preempted and that the vCPU is not utilized, thereby significantly degrading performance. This problem especially occurs frequently when running parallel programs with a small OpenMP SPINCOUNT because a vCPU tend to become idle in such an environment due to the lock contentions.

5.4.2.2 The Fix

To fix the problem, we created a per-cpu kthread. The `SCHED_IDLE` scheduling class is assigned to the kthread. This means that the kthread only runs when the CPU becomes idle. When the kthread is scheduled, the kthread deactivated the preempted flags from vCPUs that sleeps on the CPU. This way, HLT VMEXIT vCPU becomes candidates of wakeup load balancing when the host CPU is idle.

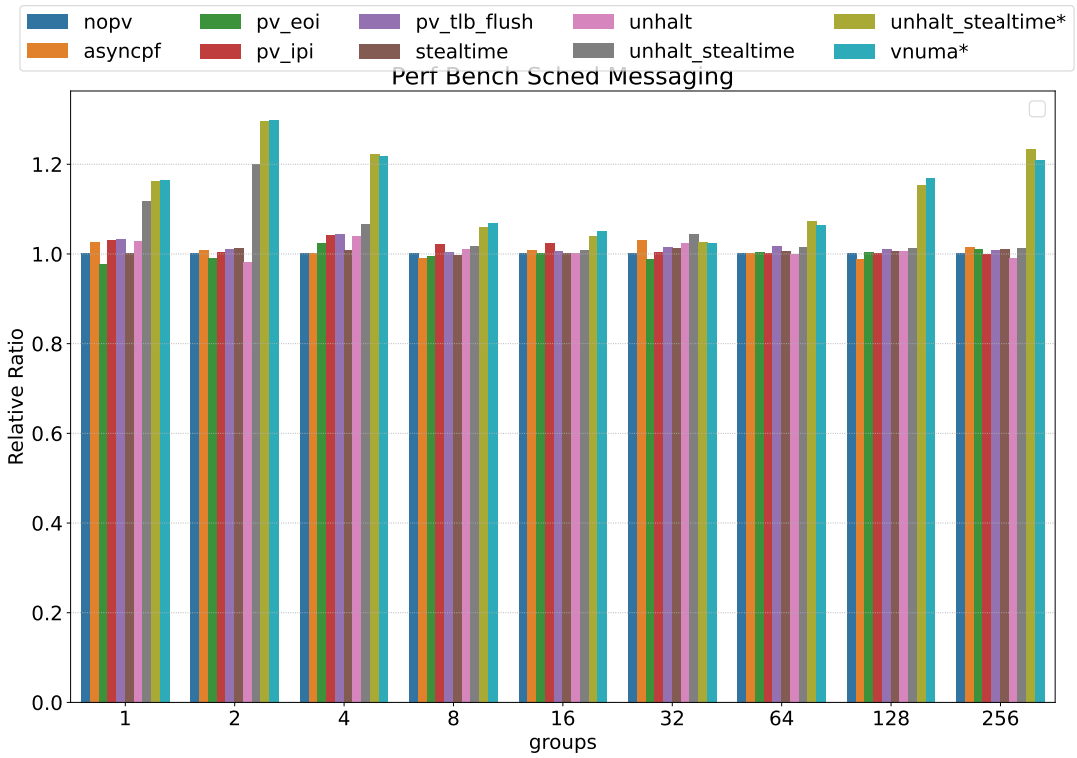


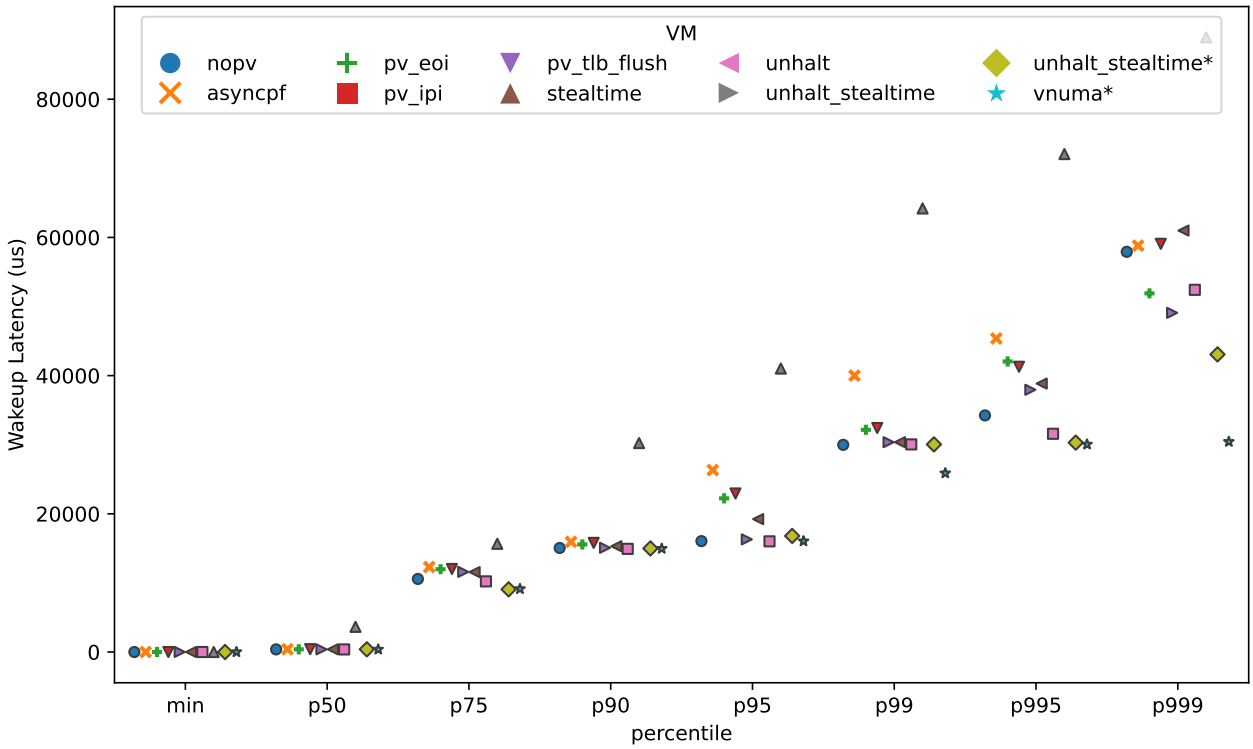
Figure 5.5: Perf Bench Sched Messaging

Figure 5.7 shows the visualization of each CPU's run queue length when running NPB lu program with SPINCOUNT 0. The graph is created using tracing results obtained by ftrace [176]. The y axis is the CPU number and the x axis is time. The color depth indicates the number of threads (`nr_running`) in the run queue. There are 48 threads in total. As shown in the figure, before the fix, there are several run queue contention. After the fix, these contentions are solved. Figure 5.8 shows which CPU the threads are scheduled. From this figure, it can be confirmed that unnecessary migration occurred due to the limited number of available CPUs before the modification.

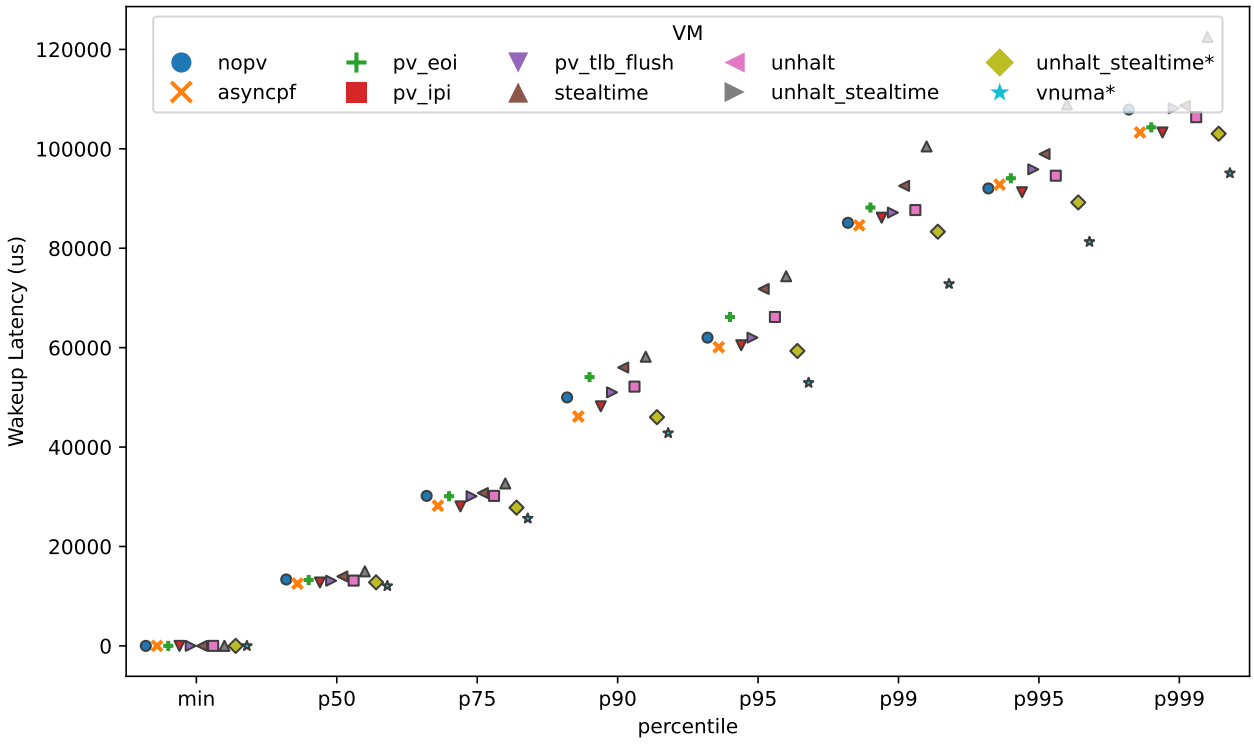
In Figure 5.3, Figure 5.4, Figure 5.5 and Figure 5.6, "unhalt_stealtime" and "vnuma*" are experimental results with this fix applied. As shown in the graph, the performance degradations of NPB and parsec are solved.

5.5 Evaluation of NUMA-Visible Virtual Machines

In this section, we evaluate performance of virtual machines shown in Figure 5.2 to investigate perormance of vNUMA machines. We used First Touch policy in the experiment. We enabled all KVM paravirtualizartion features with the fix presented in subsection 5.4.2. We report relative values against bare-metal for NPB, parsec, and perf sched benchmarks.

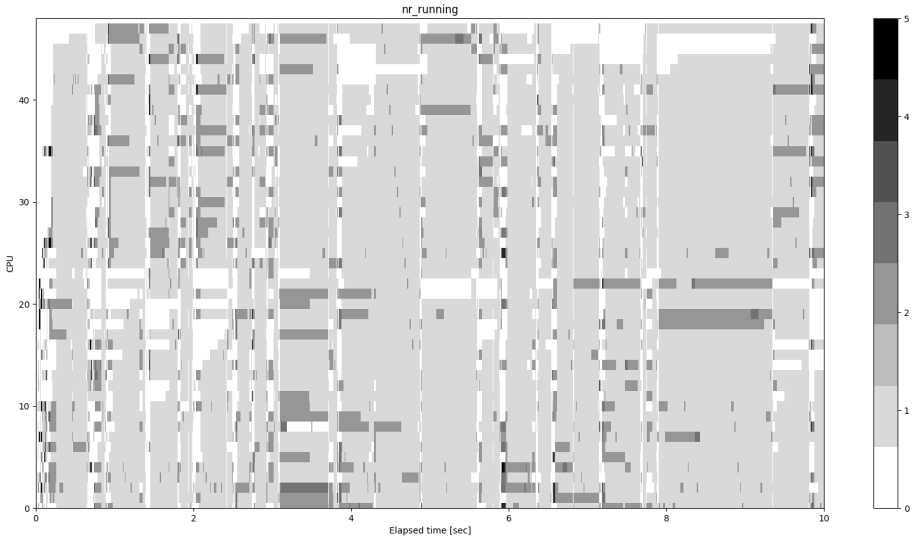


(a) threads=64

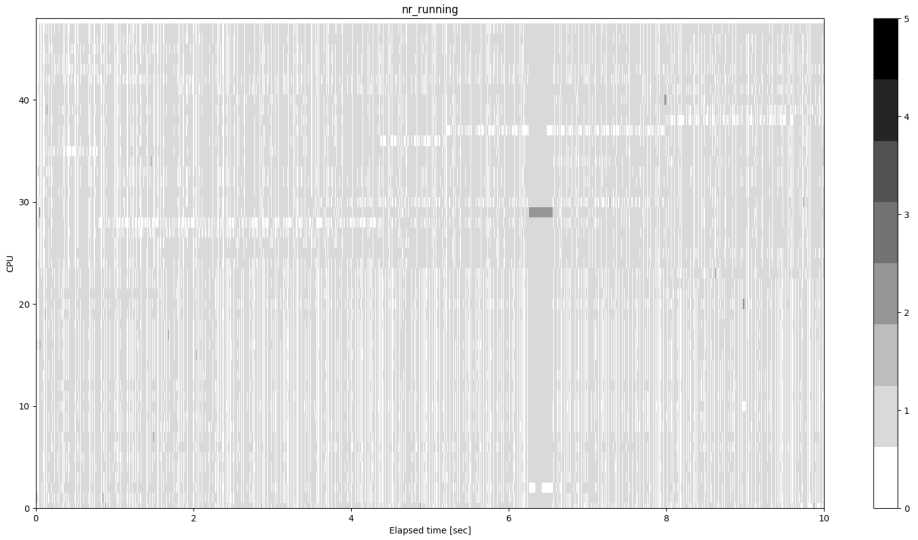


(b) threads=128

Figure 5.6: Schbench

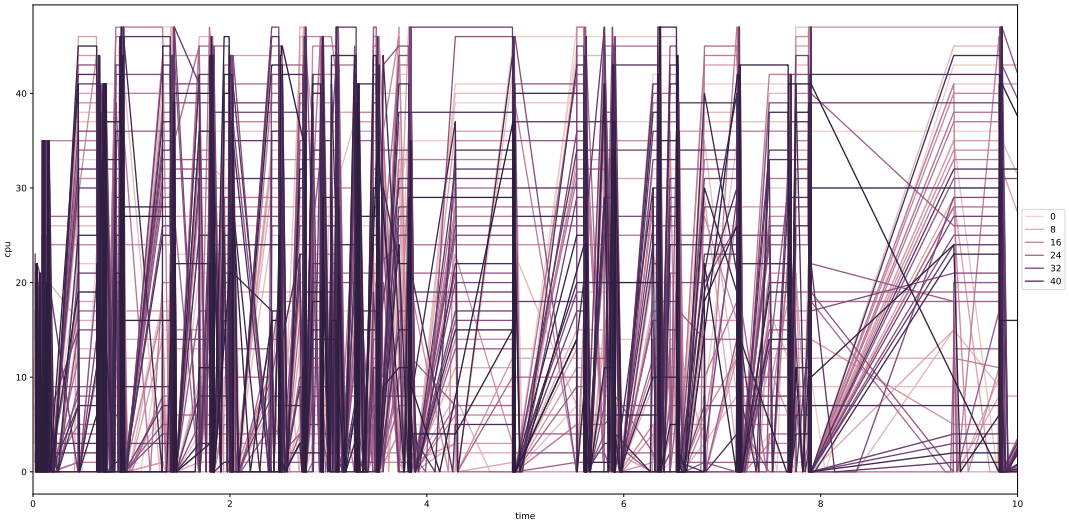


(a) Before the fix

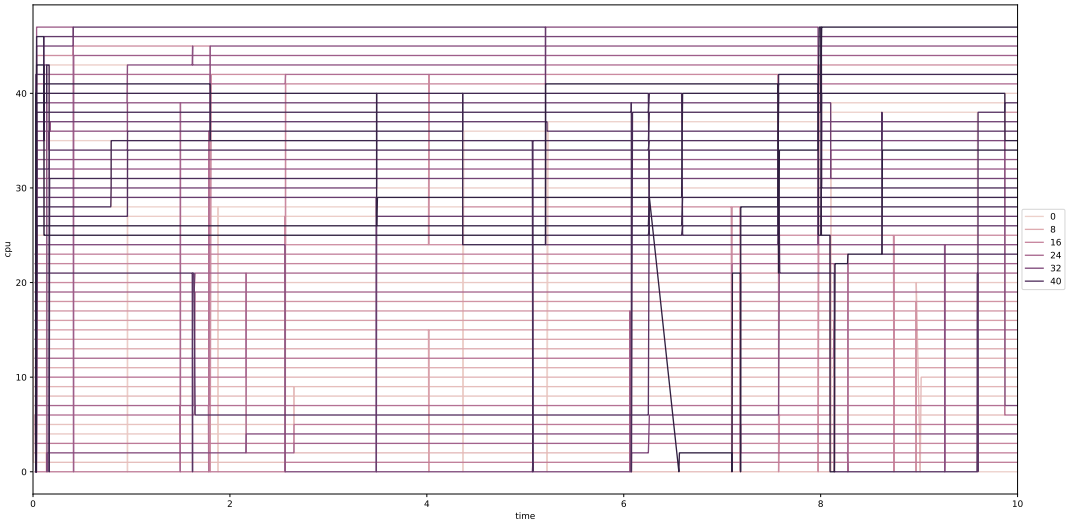


(b) After the fix

Figure 5.7: Visualization of each Run queue Length. The vertical axis is vCPU number, and the horizontal axis is time. The color intensity indicates the length of the run queue.



(a) Before the fix



(b) After the fix

Figure 5.8: Visualization of on vCPU Scheduling. The vertical axis is vCPU number, and the horizontal axis is time. This graph shows which vCPU is scheduled to which pCPU.

5.5.1 Result

Figure 5.9 shows the result of NPB benchmark and Figure 5.10 shows the result of parsec. Figure 5.11 and Figure 5.12 shows the result of perf bench sched and schbench, respectively. From the figtures, we can observed the following things:

- As of NPB, when SPINCOUNT is higher (30b), vNUMA (shared LLC) is the closest to the bare-metal performance in most programs.
- However, when SPINCOUNT is lower, vNUMA (LLC-shared) (and sometimes also bare-metal) are slower than others. This is especially noticeable in "sp", and "lu" with SPINCOUNT 0.
- As of parsec, vNUMA (LLC-shared) is slower than UMA and vNUMA (non-LLC-shared) on "dedup" and "streamcluster".
- As of perf bench, the vNUMA's execution time become larger when the number of threads increases.
- As of schbench, the scheduler latencies of bare-metal and vNUMA (LLC-shared) become worse than UMA when threads=128.

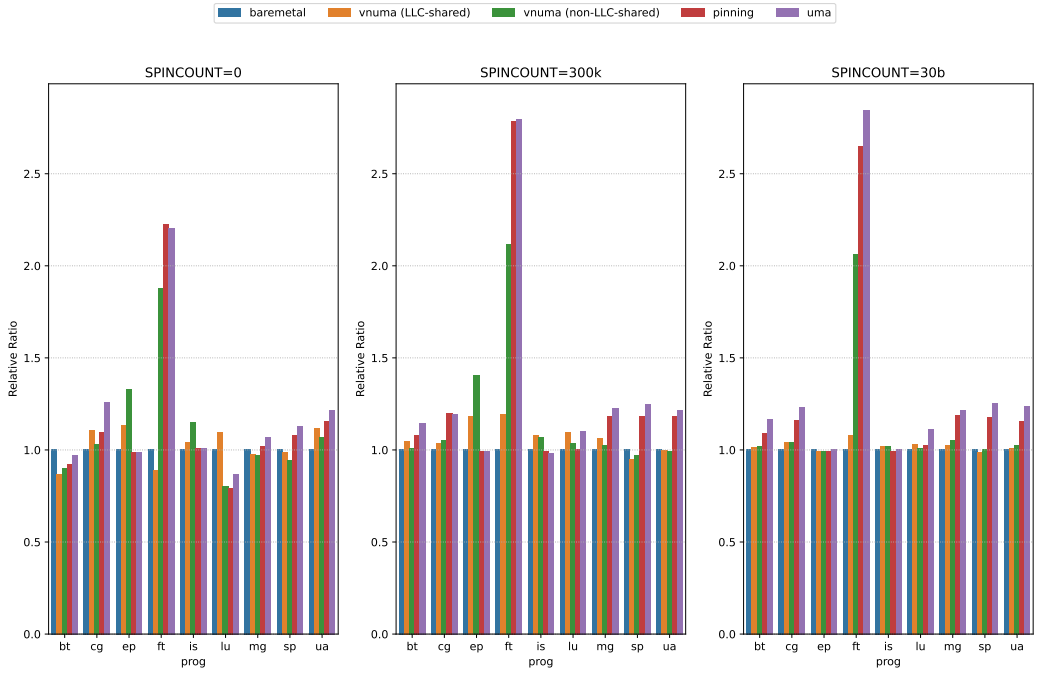


Figure 5.9: NPB Benchmark

[t]

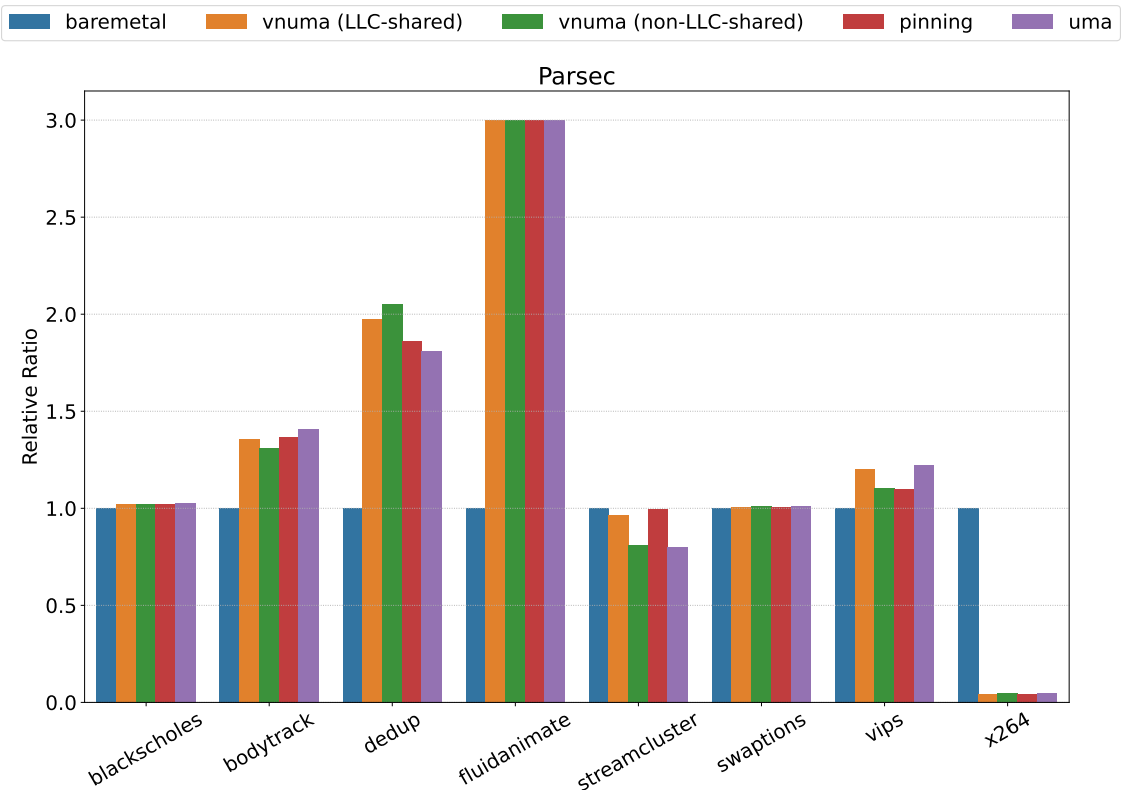


Figure 5.10: Parsec

5.5.2 Overload Wake-on-Bug (OWB)

We investigated the performance degradation problems of NPB "lu" and "sp". From the fact that vNUMA (non-LLC-shared) does not have the performance problem, we presume that this problem is caused by the wakeup load balancing. Specifically, we presume that this problem is caused by the following procedure:

- A thread sleeps as it waits for a lock.
- Idle load balancing causes the migration of executable threads from a vCPU in another NUMA domain.
- The first thread on this vCPU wakes up, clogging the vCPU run queue.

This problem is related not to virtualization but to the load balancing of the NUMA domain of the CFS. [107] reports similar problems and they called it "Overload Wake-on-Bug". [107] improved the performance by performing wakeup load balancing among the NUMA scheduling domains if there is no idle cores in the CPU scheduling domains. We also implemented the same fix and re-evaluated benchmarks. Figure 5.13, Figure 5.14, Figure 5.15 and Figure 5.16 shows the results.

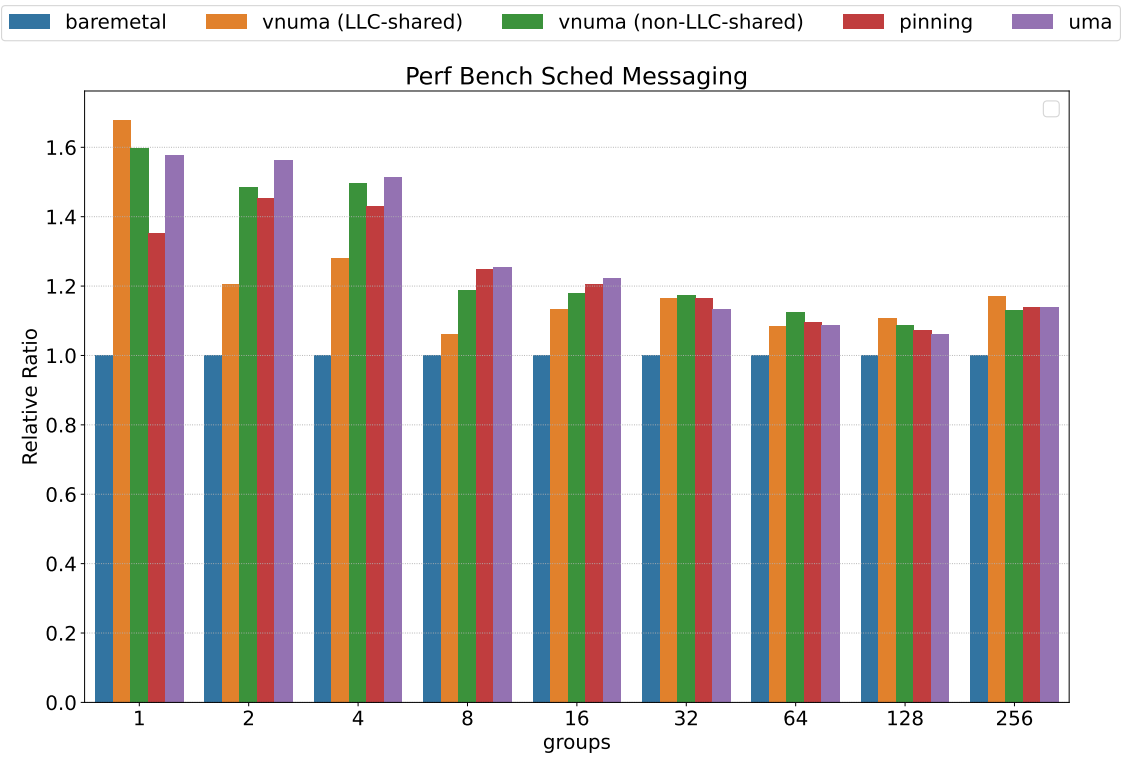


Figure 5.11: Perf Bench Sched Messaging

As shown in the Figure 5.13, the performance degradations of "lu" and "sp" are fixed. However, the fix is not a panacea; some performance degradation is still observed (e.g., parsec's "dedup" and "fluidanimate"). Further performance improvement is one of the future works.

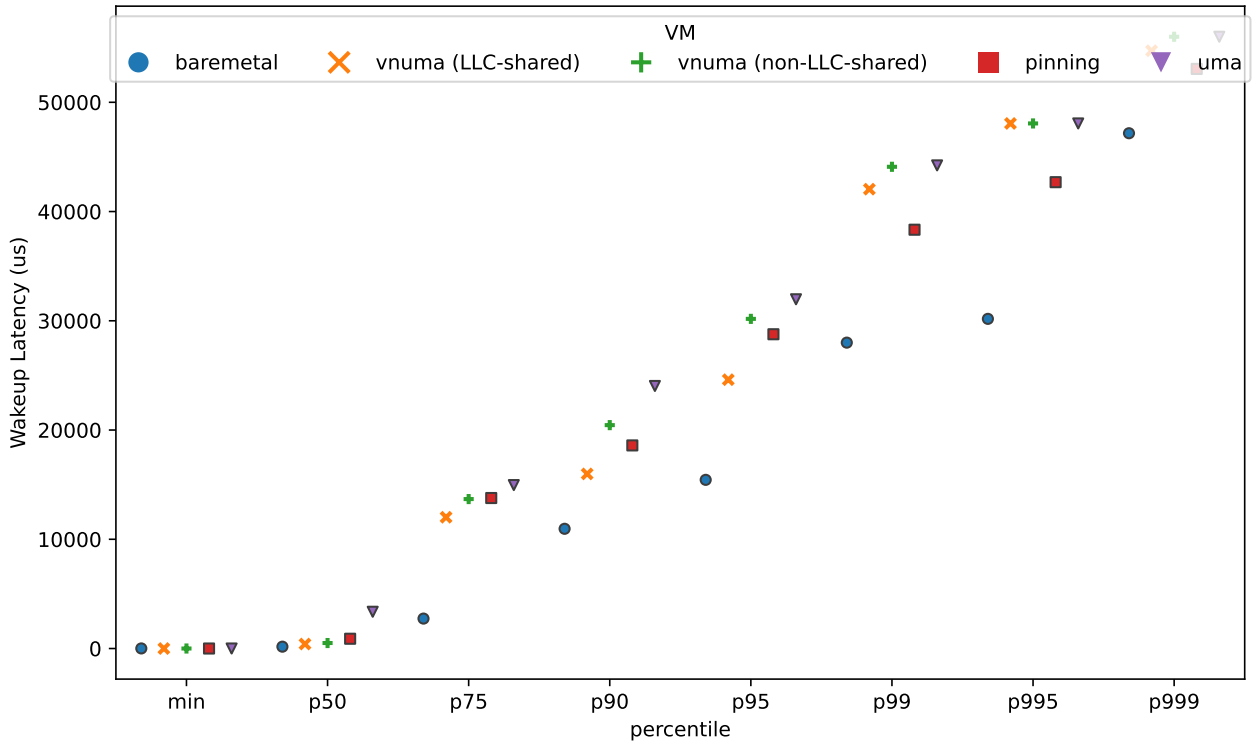
5.6 Related Work of Linux Scheduling

5.6.1 Analyzing Linux Scheduling

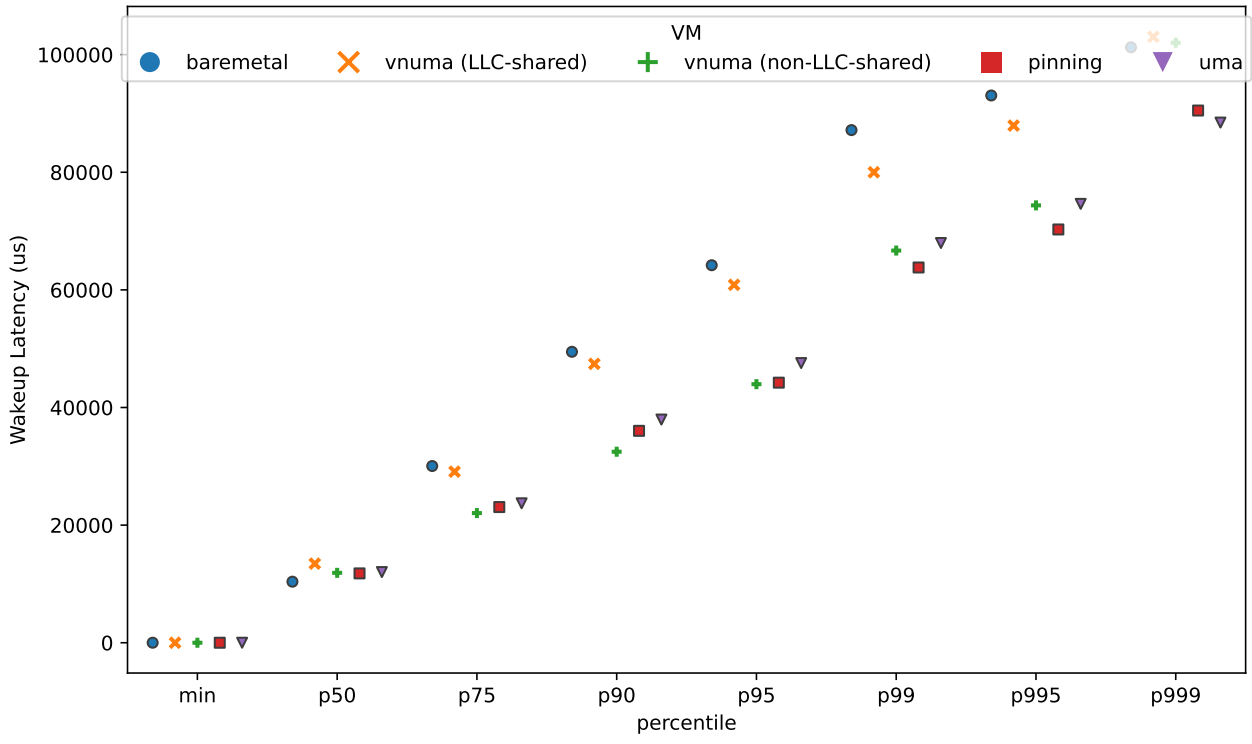
[107] reports four bugs in the Linux CFS. All four are related to NUMA, and three are associated with NUMA load balancing. The overload wake-on-bug reported in the report was also confirmed in this study. As shown in the evaluation, the fix proposed in [107] increases some program throughput but does not solve all scheduler problems found in this paper. [131] compared the performance of Linux CFS with that of FreeBSD Scheduler ULE. Both studies did not evaluate the scheduler in a virtualized environment.

5.6.2 Improving (NUMA) Scheduling

There are several studies on improving scheduling such as [84, 110, 148, 151]. Recently, to improve scheduler performance, [148] proposed a load balancing method that uses machine



(a) threads=64



(b) threads=128

Figure 5.12: Schbench

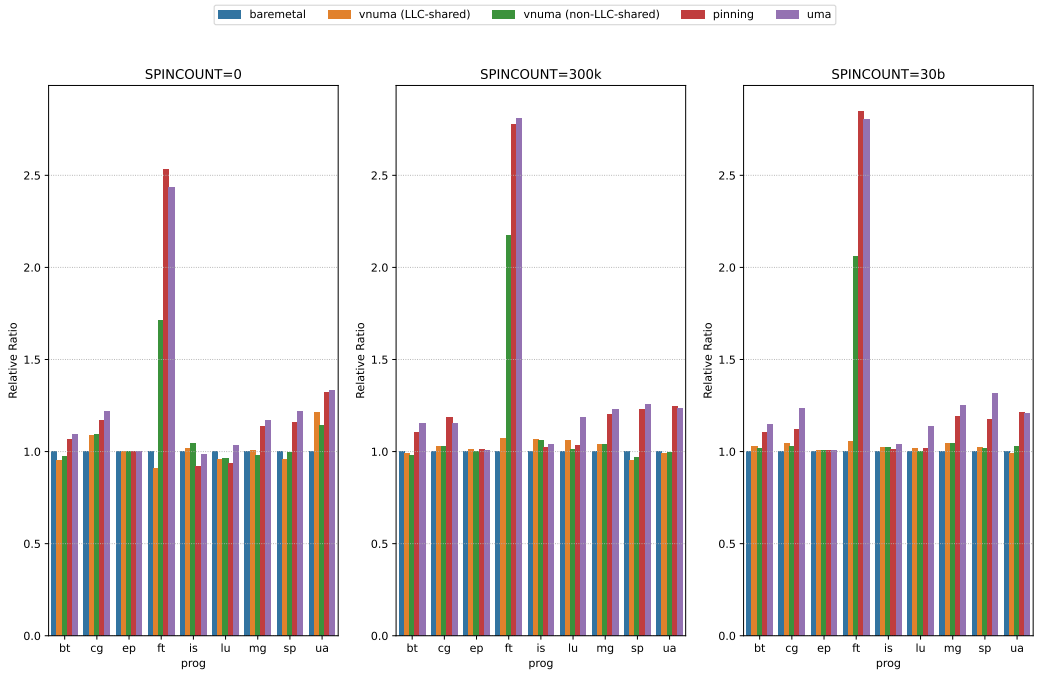


Figure 5.13: NPB Benchmark (with OWB fix)

learning trying to consider hardware characteristics. [151] propose a scheduler that always preserves specific properties using formal verification. These studies did not evaluate the performance of the scheduler in vNUMA environment. As shown in this research, there is still room for improvement for the (v)NUMA scheduler.

5.7 Summary

In this chapter, we conducted detailed performance experiments on virtual machines in which guests replicated the hardware configuration of the host. Contrary to our expectations, the performance of virtual machines that replicate a host's hardware configuration is sometimes lower than that of VMs that do not carry out such replication. The comparison of our experimental results with those realized using bare-metal indicated that part of the reason for these findings is essentially the slow scheduling of NUMA. We expect our findings to help improve the performance of the scheduler.

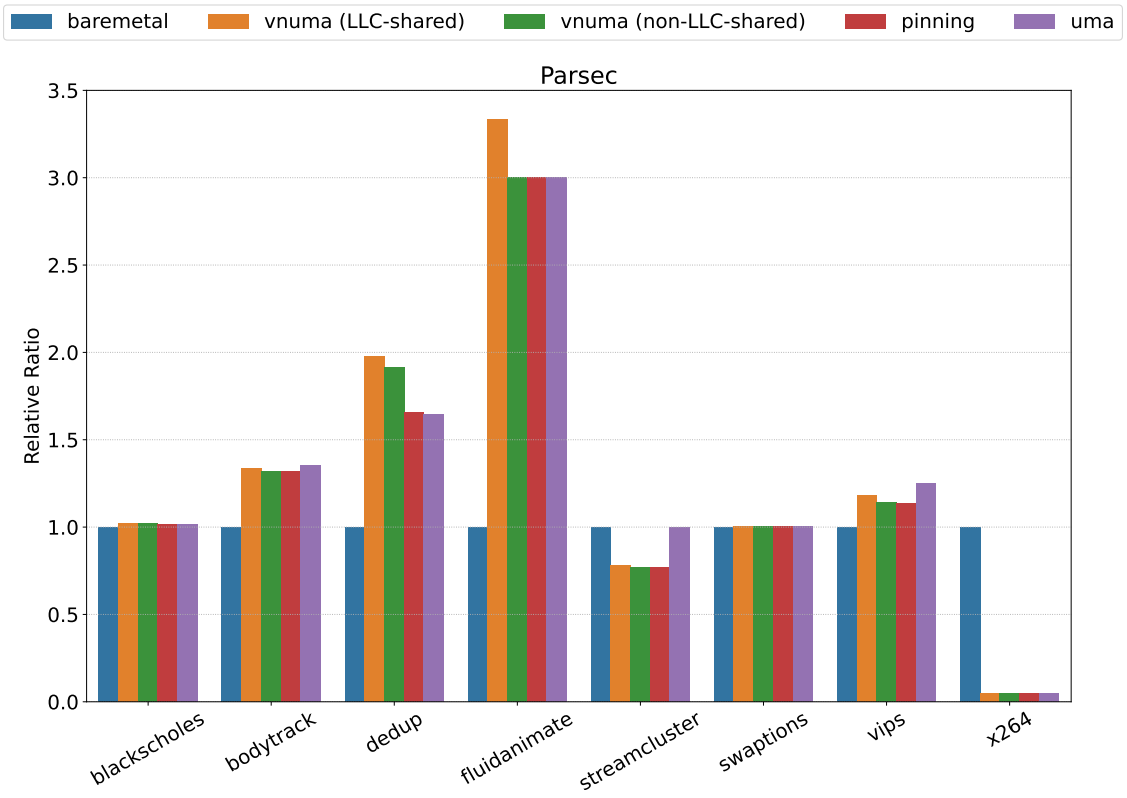


Figure 5.14: Parsec (with OWB fix)

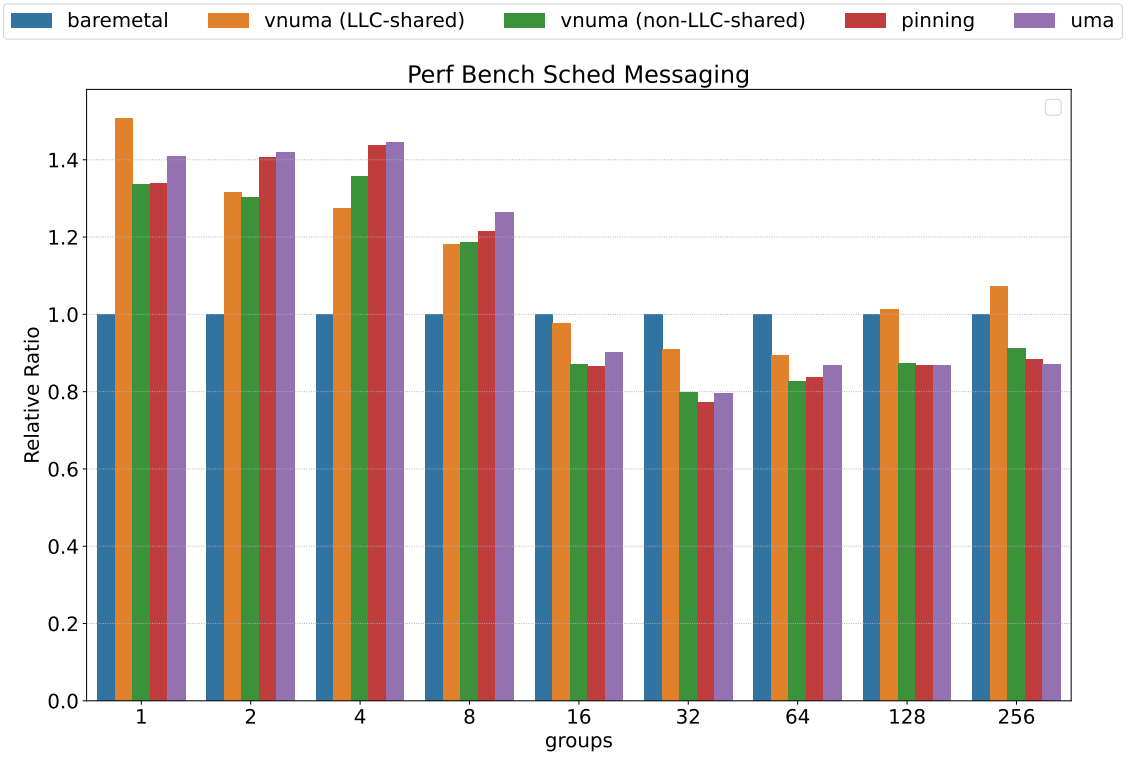
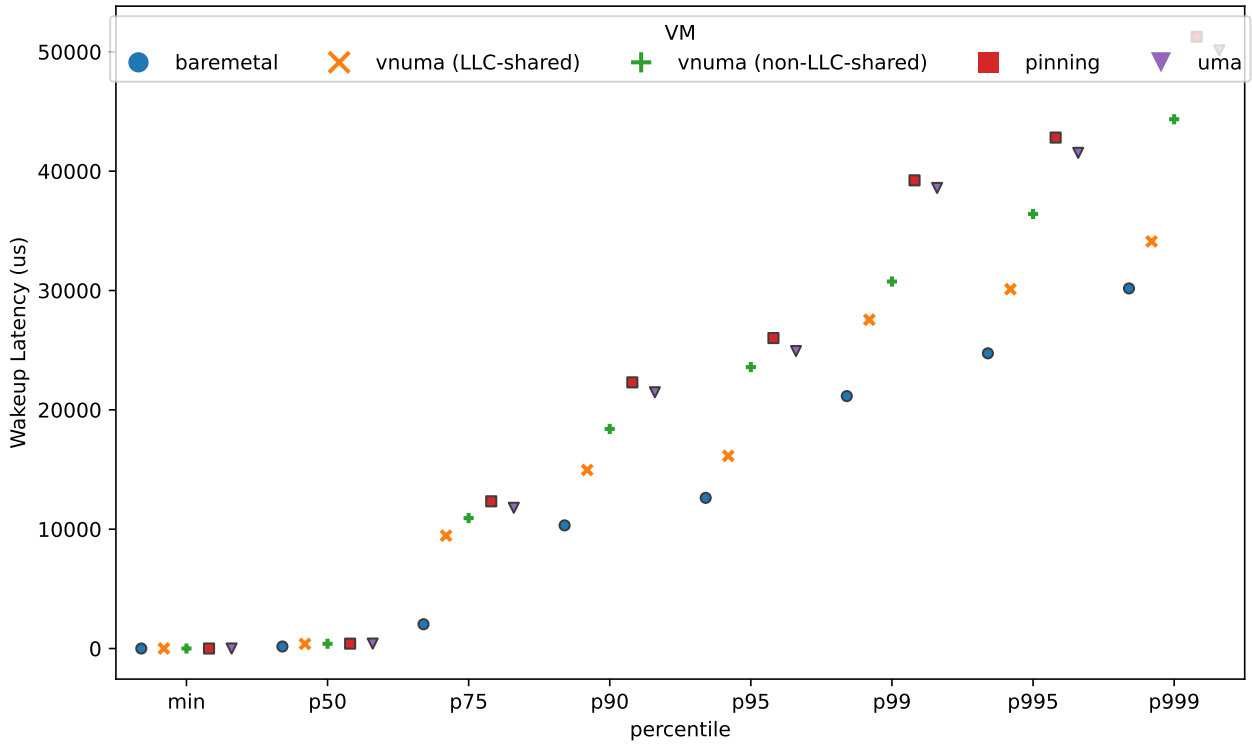
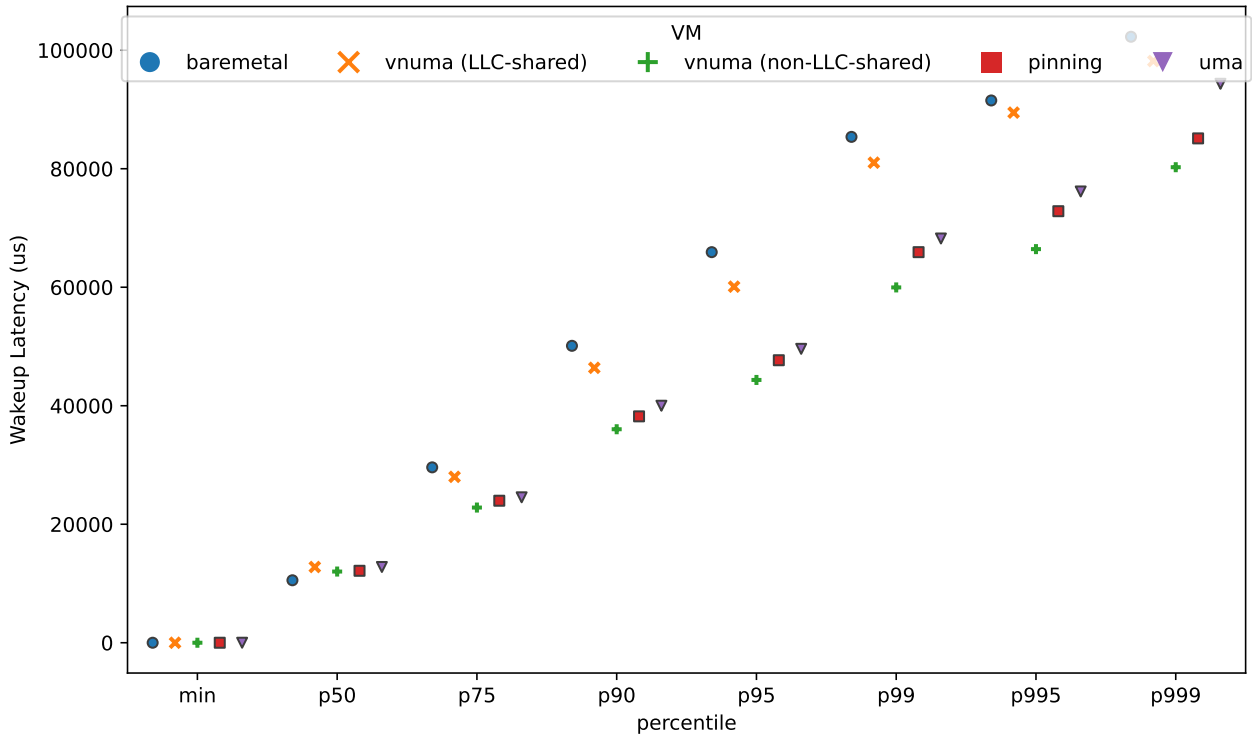


Figure 5.15: Perf Bench Sched Messaging (with OWB fix)



(a) threads=64



(b) threads=128

Figure 5.16: Schbench (with OWB fix)

6 Improving Hypervisor's Elasticity with Safe and Lightweight Language VM

In this chapter[†], we propose a method to improve the flexibility of hypervisors without compromising performance by using a secure and lightweight language virtual machine. An an example of the use of the language virtual machine, we present source side DDoS prevention scheme using virtualization.

6.1 Introduction

As computers and the Internet play a critical role in modern society, cyberattacks become a significant security issue. For example, denial of service (DoS) attacks make services to legitimate users unavailable by occupying server resources (e.g., CPU and memory resources) or exhausting network resources (e.g., bandwidth) that provide routes to the target service [77]. Most DoS attacks come in the form of distributed DoS (DDoS) attacks. In DDoS attacks, attackers use viruses or similar techniques to hijack a large number of personal computers to distribute the attack [5]. According to a survey by the Kaspersky Lab, the number of DDoS attacks per day ranged from 296 to 1508 in Q3 2017 [192]. It is important for system administrators to protect machines they manage from being hijacked and used as attacking machines. If an administrator detects that a managed machine is carrying out a DDoS attack, the administrator must attempt to stop the attack immediately.

In this study, we focus on situations wherein a system administrator manages a large number of computers and individual users, e.g., in schools and companies. Our goal is to develop a reliable, lightweight, transparent, and flexible DDoS attack prevention scheme to easily suppress packet transmissions from the machines involved in DDoS attacks. DDoS attacks have been studied extensively [14]. For example, as DDoS attack detection schemes at the source side, [47, 106, 120] proposed DDoS attack detection systems in the cloud using machine learning techniques. However, few studies have focused on DDoS attack prevention mechanisms at source machines that are suitable for personal computers having less computing power than servers.

[†]This chapter is based on [138] (© 2018 IEEE. Reprinted, with permission).

A firewall is a primary countermeasure against DDoS attacks. By installing a firewall on the boundary of a managed network and configuring it properly, a system administrator can prevent the transmission of attack packets. However, this method requires significant computational power at the firewall because many packets must be handled at the network edge. Furthermore, attack packets consume significant network resources when they traverse the network path to the firewall. Therefore, the best solution is to suppress packet transmissions at the attacking machines. Most operating systems (OS) have firewall functionalities. If a system administrator can configure a firewall remotely, they can stop the transmission of attack packets. However, as the machines carrying out a DDoS attack are often controlled by the attacker, the attacker can disable the OS firewall. Therefore, we require a packet filtering scheme that cannot be disabled by the attackers.

Several previous studies have proposed using hypervisors to ensure firewall functionality reliably [28, 31, 83, 103]. However, traditional hypervisors demonstrate two main disadvantages. First, as discussed in subsection 1.1.3, virtualization incurs significant overhead. Second, users cannot use devices that are not supported by the hypervisor. For example, there is less chance of full functionality support of a laptop's touchpad by hypervisors because touchpads have diverse functionalities compared to mice and keyboards. It may be possible to use a device pass-through technique. However, this requires additional configuration operations, which is troublesome for non-expert users and is unsuitable for personal computers. Furthermore, additional procedures, such as booting the hypervisor and initializing virtual devices, are required prior to booting the guest OS. In particular, if a Type II hypervisor is employed, it is difficult to force the users to use only the guest OS because they can access the host OS as well. Using two OSs is troublesome for ordinary users in daily use. Therefore, it is important to provide a protection scheme that can enforce packet filtering while it is transparent from users.

We propose a reliable, lightweight, transparent, and flexible DDoS attack prevention scheme that is suitable for managed personal computers. In this scheme, the administrator installs a lightweight hypervisor on each managed machine to achieve reliable packet filtering. This hypervisor does not virtualize hardware, except for network interface cards (NIC), thereby, significantly reducing virtualization overhead and making the hypervisor transparent from users. To make our scheme flexible, we integrate a configurable packet filtering mechanism into the hypervisor that can be controlled by the administrator. Figure 6.1 shows an overview of the proposed scheme. Here, the dotted line shows the network area managed by the administrator. When one or several machines in the managed network perform a DDoS attack on a target server (indicated by the red arrows), the administrator sends a filtering policy to these machines. By sending the filtering policy to only the attacking machines, a legitimate user can still send packets to the server.

To facilitate flexibility of the packet filtering mechanism, we allow the system administrator to send a filtering policy as executable code. Therefore, the administrator has considerable flex-

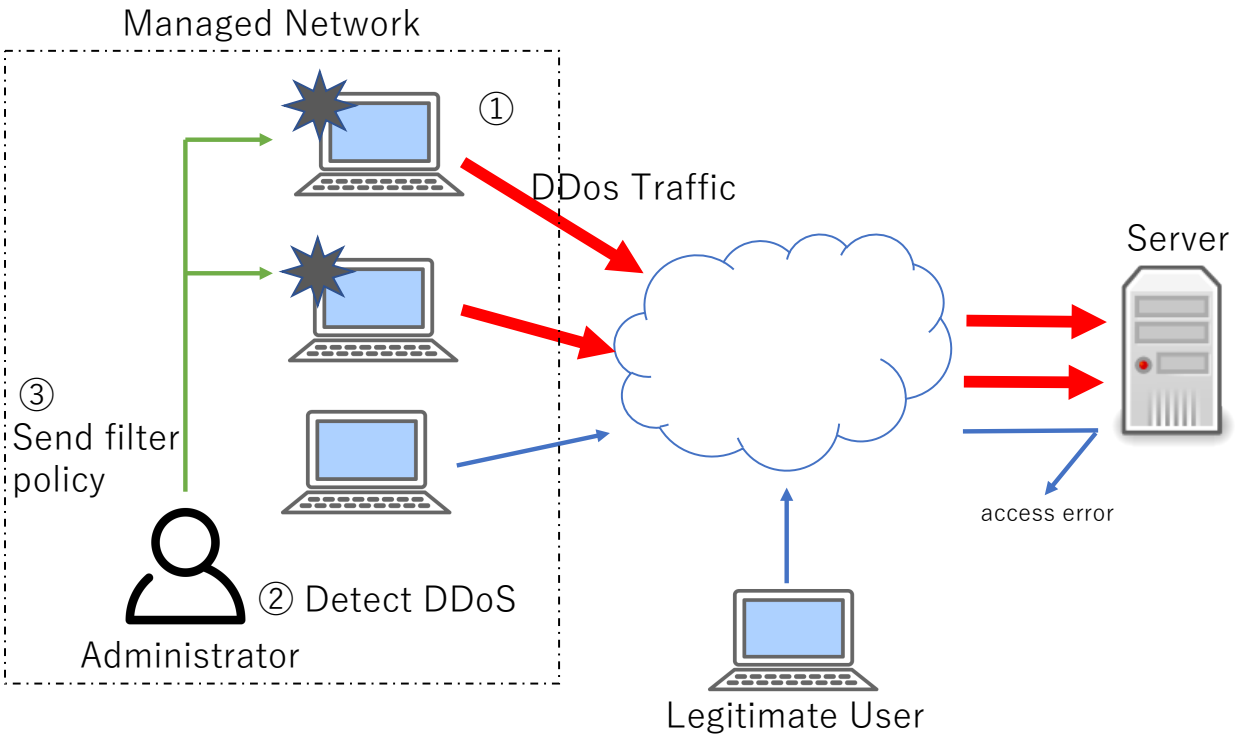


Figure 6.1: Overview of the Proposed Scheme

ibility to implement an arbitrary policy on the managed machines. To prevent programming mistakes from affecting the security of the managed machines, the verifier in the hypervisor checks the filtering behavior to guarantee security prior to execution. This mechanism gives the administrator greater flexiblity in enforcing filtering policies without compromising the security of the managed machines.

We implemented the proposed scheme using BitVisor [35] and the Berkeley Packet Filter (BPF) [222]. We made BitVisor to monitor only the NIC I/Os and integrated the BPF execution environment into BitVisor such that network packets could be filtered based on the execution results of the BPF program. The experimental results show that the proposed scheme can suppress the transmission of packets upon request with negligible latency and throughput overhead compared to a bare metal machine.

We assume that DDoS attack detection is achieved using existing methods. Automatically creating filtering code based on the detected attacks will be the focus of future work.

The contributions of this research are as follows.

- We propose a reliable, lightweight, transparent, and flexible DDoS attack prevention scheme that can enforce a configurable packet filtering policy even if the OS is compromised while the prevention scheme is lightweight and transparent from users.
- We show a specific implementation of our scheme that exploits BitVisor and BPF to

achieve negligible virtualization overhead and configurable packet filtering policies as safe executable code.

- We demonstrate the feasibility and performance of our scheme using the implementation.

6.2 Design

In this section, we outline the threat model and our assumptions. We then describe the system objectives and the proposed scheme, including its limitations and advantages.

6.2.1 Threat Model and Assumptions

We assume that malicious remote attackers can gain complete control of the target machines' OSs via software-based attacks, e.g., through viruses and exploiting application vulnerabilities. Since the attackers can obtain OS administrator privilege, the OS firewall functionalities can be disabled. However, we do not assume that attackers can gain control of the underlying hypervisor; since our hypervisor is sufficiently small, we can eliminate security vulnerabilities in the hypervisor. We also do not assume attacks against hardware, such as exploiting the system management mode of processors or the management engines embedded in processors. We do assume that attackers cannot gain physical access to the managed machines and network.

In addition, we assume that the administrator is a trusted entity that can detect that managed machines are performing a DDoS attack. Note that DDoS detection methods are beyond the scope of this study. We also assume that the administrator's management machine is isolated from the managed network and is not compromised by the attackers. Therefore, the administrator can securely send filtering code to the managed machines.

6.2.2 System Objectives

Our goal is to create a reliable, lightweight, transparent, and flexible DDoS attack prevention scheme that allows a system administrator to easily and reliably prevent packet transmissions from managed machines. This scheme should satisfy the following practical properties.

1. Packet filtering must be enforced regardless of the state of the OS.
2. The system administrator can control the filtering policy of managed machines remotely.
3. The scheme should not pose any other security concerns.
4. The user can use a machine in a transparent manner.

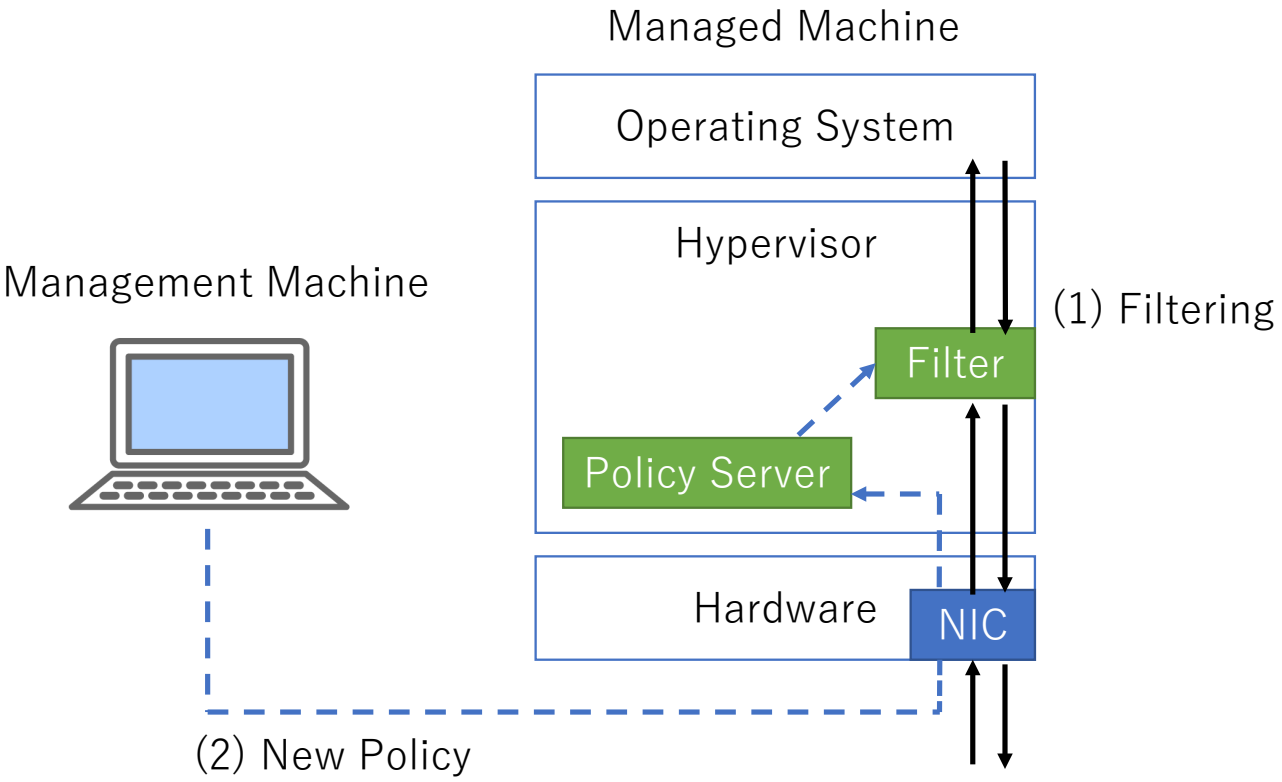


Figure 6.2: Proposed Scheme

5. The protection scheme should not impair performance, i.e., the scheme should be lightweight.

6. The filtering rule should be sufficiently expressive to stop DDoS attacks and reconfigurable without requiring a machine reboot, i.e., the scheme should be flexible.

6.2.3 Proposed Scheme

We propose a DDoS attack prevention scheme that satisfies all of the abovementioned objectives. Figure 6.2 shows the details of the proposed scheme. This scheme involves managed machines controlled by an administrator. A managed machine runs a hypervisor under the guest OS. The hypervisor has two built-in components: a filtering mechanism and a filter policy server that receives filtering policies.

In the proposed scheme, we use a hypervisor to enforce packet filtering even if the guest OS is compromised (Objective 1). As indicated by the black arrows in Figure 6.2 (1), network packets sent to or received from the NIC are filtered by the hypervisor. We exploit a parapass-through hypervisor [35] for the following reasons. First, a parapass-through hypervisor allows a guest OS to access most hardware directly and intercepts only some I/O accesses. Using this mechanism, we only intercept NIC accesses to filter packets, while other I/Os are performed

as if no hypervisor is present, which reduces overhead (Objective 5). This mechanism also reduces boot time, and the user can use the machine in a transparent manner (Objective 4). Second, the parapass-through hypervisor code size is much smaller than that of traditional hypervisors, such as Xen and KVM [35], which means that the trusted computing base of the parapass-through hypervisor is small and there is less chance that the hypervisor will contain security vulnerabilities (Objective 3).

The filtering mechanism in the hypervisor filters network packets based on a filtering policy. To allow the filtering policy to be altered remotely, the hypervisor has a filter policy server that listens for new policies from the administrator. The server updates the filtering policy accordingly when receiving a new policy. As indicated by the blue dotted lines in Figure 6.2 (2), the system administrator can change the filtering policy by sending a new policy to the hypervisor (Objective 2).

To meet our objectives, the filtering mechanism must be safe, fast, and flexible. The fastest and most flexible way to achieve this is to directly run a program written in machine code. However, since such a program can comprise arbitrary code, it is difficult to guarantee the safety of the program, such as eliminating access to external memory regions other than those storing the given packet data and avoiding infinite loops. Another option is to define filtering rules in the hypervisor in advance and selecting which rule to use based on the given policy. This option is safe in the sense that the filtering will only perform predefined processing. However, this lacks flexibility and the hypervisor would need to be restarted to update the filtering rules.

We use a specialized language-based virtual machine aimed at fast packet filtering. The administrator writes a filtering policy as code in the language used by the virtual machine. The hypervisor performs packet filtering by running the code using an interpreter. Note that security is guaranteed by the verifier prior to execution. This mechanism has the following advantages. First, by changing the program, we can change the filtering policy, which facilitates greater flexibility than a static setting (Objective 6). Second, by designing simple and restricted ISA, verifying program safety becomes easier than verifying native code (Objective 3). Optionally, by employing JIT compilation, we could generate native code and increase the speed of filtering.

Figure 6.3 shows the flow of setting a new policy and filtering. Here, the black dotted line represents the filtering setting flow. When setting a new filtering policy, the verifier first checks the program's safety. If safety is verified, the hypervisor sets the policy as the filtering rule with optional JIT compiling. Then, the hypervisor runs filtering code (blue lines) when it receives new packets. To increase program flexibility and performance, optionally we can create an interface that allows the program to call predefined hypervisor functions, e.g., a function that looks up the IP address blacklist table.

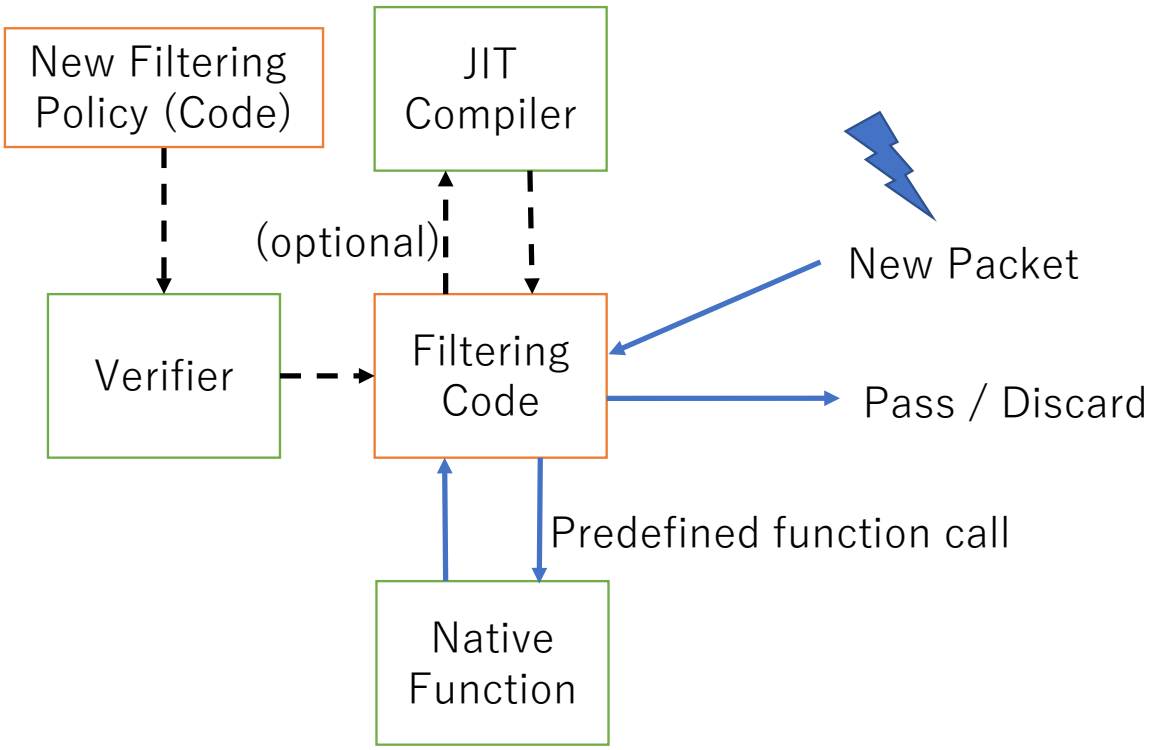


Figure 6.3: Filtering flow

6.2.4 DDoS Attack Prevention Workflow

The typical workflow of preventing DDoS attacks using the proposed scheme is as follows.

1. The system administrator introduces the proposed scheme (the hypervisor with the filtering mechanism) to their managed machines. The default policy is no filtering.
2. The attacker takes control of some of these machines to perform DDoS attacks.
3. When the system administrator identifies that a managed machine is performing a DDoS attack, they create a new policy to stop DDoS attack packets (e.g., to stop packets whose destination is a specific host) and send the policy to the attacking machine(s).
4. The transmission of DDoS attack packets is suppressed by the hypervisor according to the new policy.
5. If the DDoS attack is stopped, the system administrator can create a new policy with no filtering and send it to the target machine. Thus, the system administrator can reset the filtering settings without rebooting the given system.

6.2.5 Discussion of the Proposed Scheme

Recent CPUs have a mechanism to create a security subsystem, such as Intel SGX [183] or ARM TrustZone [167]. By utilizing this mechanism, one can create a secure region which cannot be accessed from untrusted regions. We may create a packet filtering mechanism which runs in such a secure region so that attackers cannot read or modify the filtering policy. However, we cannot enforce packet filtering with this mechanism alone. Therefore, we use a hypervisor in the proposed scheme.

Our scheme will work with any arbitrary OS since it does not depend on any particular OS functionalities. It is easy to install the proposed scheme because the hypervisor can be inserted into existing machines without reinstalling the OS.

Typical DDoS attacks involve many machines spanning several networks. Hence, many administrators would need to enforce filtering policies for each machine. To efficiently and effectively create a policy, it is nice to have a platform where administrators can share DDoS attack information and cooperate to create policies, which will be the future work.

It is possible for clients to regularly pull the filter policy from a policy server. However, in this case, clients need to remember the server address and unnecessary traffic is generated. In our scheme, the management machine pushes the policy to each client. This scheme may not scale enough if there is a large number of clients. We can adopt P2P communication protocols such as gossip protocols [2] to distribute filtering policies, which will also the future work.

Note that the proposed scheme does not utilize virtual machine introspection (VMI) [10] to obtain information about the packet sending process. Therefore, the hypervisor cannot perform packet filtering based on such process information. However, we consider that such process information is not necessarily required, i.e., packet information is sufficient to stop DDoS attacks. In addition, we can obtain a performance advantage by not employing VMI architecture. Furthermore, since we do not need to implement a process analysis mechanism, the implementation is independent of the guest OS.

6.3 Implementation

This section describes the implementation of the proposed scheme. First, the packet interception method and filtering mechanism are described. Then, we describe how we create BPF programs and how the policy server receives them.

6.3.1 Packet Interception

We implemented the proposed scheme using BitVisor [35] as the hypervisor. BitVisor employs a parapass-through architecture wherein, in exchange for supporting only one guest OS, the

guest OS can essentially access hardware directly and the hypervisor can intercept some of the I/O accesses. In our configuration, only access to a NIC is intercepted by the hypervisor, and access to other devices is pass-through. Each interrupt is delivered directly to the guest OS without exiting the VM, thereby improving network performance.

BitVisor intercepts network packets with shadow descriptors. Figure 6.4 illustrates how shadow descriptors work. BitVisor shadows the NIC's descriptor by intercepting MMIO accesses from the guest OS device driver. NIC's descriptor base register points to a memory region inside BitVisor. When transmitting a network packet, the guest OS first sets up its descriptor table and buffers. Then, the guest OS attempts to write the NIC's MMIO register to request the NIC start transmitting packets. BitVisor intercepts this MMIO access and copies data to its descriptor table and buffers, and then makes a transmission request to the NIC. When receiving a packet, an interrupt is first delivered to the guest OS directly. Then, the guest OS device driver attempts to access the NIC's MMIO register to check the interrupt status. BitVisor intercepts this MMIO access and copies data from the shadow buffer to the guest OS buffer. Note that this interception method does not depend on any OS functionality and works for any arbitrary OS. BitVisor has this mechanism for common NICs, including the Intel Pro1000, Realtek 8169, and Broadcom 43xx NICs. We modified BitVisor such that, when copying packet data from the guest OS buffer to the shadow buffer, BitVisor runs packet filtering code per packet and discards a packet if the code does not allow it to be transmitted.

6.3.2 Filtering Mechanism

In the proposed scheme, we use the BPF [3] as the filtering mechanism. The BPF is a virtual register machine that performs filtering by running a BPF program. By changing the BPF program, we can change the filtering policy. The BPF instruction set is designed such that it can be configured for various filtering conditions. For example, we can make a BPF program that suppresses transmission of packets to a specific destination host and port. The filtering method is designed to reduce packet reference times as much as possible.

In our implementation, we used the extended BPF (eBPF) [198], which is used in the Linux kernel. Compared to the traditional BPF, the eBPF has more registers with 64-bit width, ISA similar to the x86-64 and ARM-64 architectures, which allows easy JIT compilation, and a mechanism to call predefined external functions.

We used a userspace implementation of the eBPF VM called ubpf [187], which has an eBPF interpreter, a JIT compiler, and a simple verifier. The ubpf verifier can detect invalid instructions and infinite loops; however, its functionality is not perfect compared to the Linux verifier [221]. For example, the current ubpf checks memory bounds dynamically when accessing memory. In the future, we will implement a verifier that is comparable to the Linux verifier.

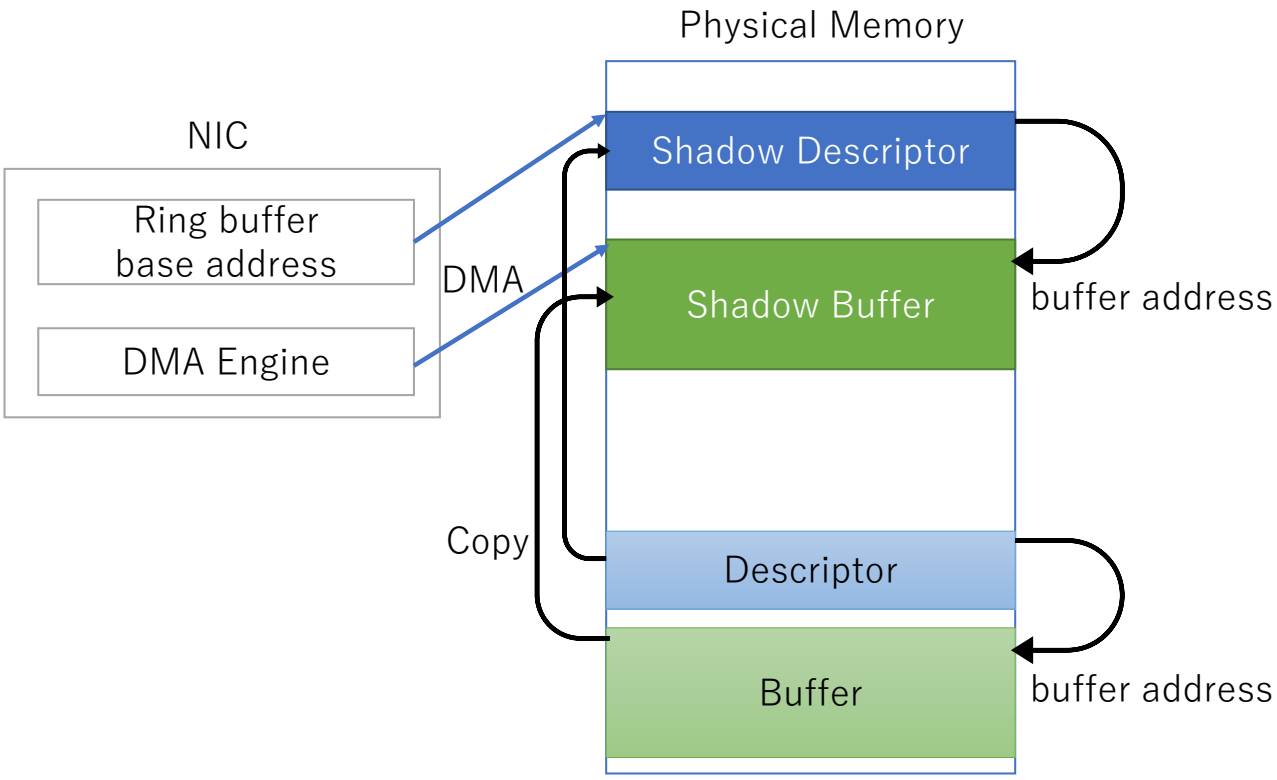


Figure 6.4: Descriptor shadowing

We modified some Linux-related parts of the ubpf and embedded them into BitVisor. The BPF interpreter takes two arguments, i.e., a BPF program, which specifies the filtering policy, and a packet data buffer that can be manipulated by the BPF program. When the hypervisor intercepts an I/O packet, we run the BPF program with the intercepted packet data. Here, the packet is discarded when the BPF program returns 0; otherwise, the packet is accepted and transmitted. BPF programs can be set independently for both transmission and reception directions.

6.3.3 Creating BPF Programs

LLVM has an eBPF backend as of version 3.8, and we can write an eBPF program in C and compile it using clang. Listing 6.1 shows an example filtering program that uses the BPF Compiler Collection (bcc) [186], which provides a helper function to compile an eBPF program in a Python script. This program creates a BPF program that filters packets whose destination IP is 192.168.20.1, and then send it to the policy server (192.168.20.51).

```
prog = compile_program("""// eBPF program
int entry(u8* pkt){
  struct eth_t *ether_hdr = (struct eth_t*) pkt;
  int type = bpf_ntohs(ether_hdr->type);
  if (type == 0x0800){ // IP
    struct ip_t *ip_hdr = (struct ip_t*)
          ((u8*)ether_hdr + sizeof(struct eth_t));
    if (ip_hdr->src == 0x0114A8C0) // 192.168.20.1
      return 0; // drop
  }
  return 1; // accept
}""")
send_program(prog, "192.168.20.51", 11111)
```

Listing 6.1: Example filtering code

6.3.4 Policy Server

BitVisor has TCP/IP server functionality based on lwIP [199], and we created a policy server using this mechanism. There are two ways to use the lwIP functionality in BitVisor. One way is to assign a NIC to the hypervisor exclusively. The NIC used by the hypervisor is concealed and cannot be accessed from the guest OS. The other way is to duplicate network packets and process them using both the guest OS and lwIP in the hypervisor. Although this method adds additional packet processing computations, it requires only a single NIC. Either method can be employed depending on the given situation.

In the hypervisor, the policy server listens to a specific port and waits for the policy setting packet, which comprises the filter type and the BPF program. When receiving a policy setting packet, the filter server first verifies the BPF program using the verifier. When verification is complete, the filter server sets the new BPF policy for transmission (if the filter type is 0) or reception (if the filter type is 1). Otherwise, the policy setting packet is discarded. Here, JIT compiling can be enabled depending on the configuration.

Note that the current implementation does not employ any authentication mechanism. In addition, we require a secure connection between the policy server and the administrator's management machine. We can use a transport layer security connection to support both.

6.4 Evaluation

In this section, we first present the results of a proof-of-concept experiment, then the results of a performance evaluation.

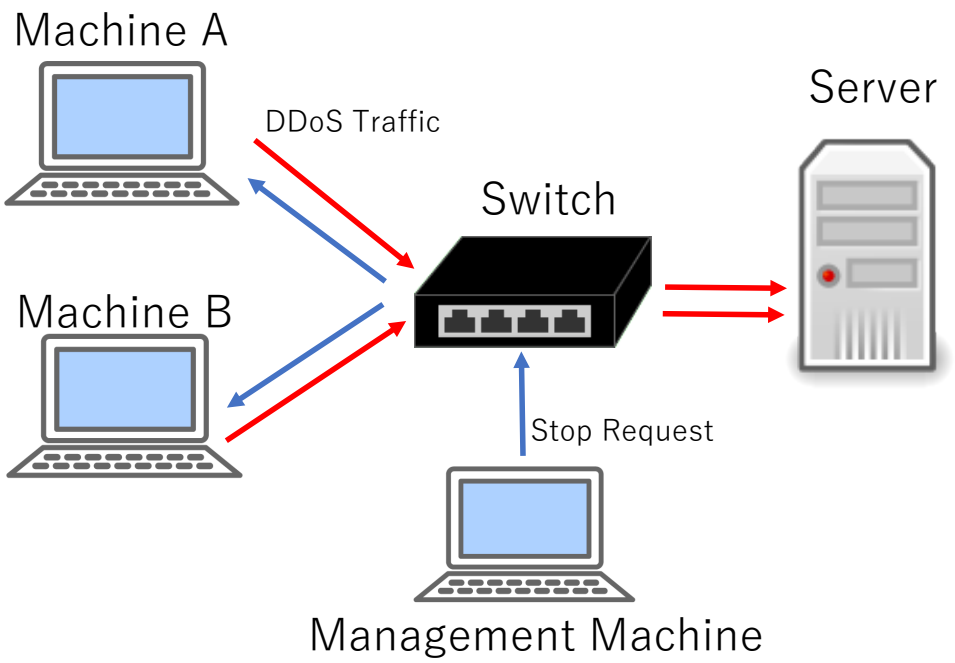


Figure 6.5: Settings of Proof-of-concept Experiment

Table 6.1: Machine Specifications

	CPU	Mem	NIC	OS
Server	i7-7700	16GB	Intel 82574L	Linux 4.13
Machine A	i7-4690K	16GB	Intel I218-V	Linux 3.19
Machine B	i7-2600K	16GB	Intel 82589V	Windows 10
Management Machine	i5-4278U	8GB	BCM57766	Linux 4.13

6.4.1 Proof-of-concept Experiment

We conducted a proof-of-concept experiment to demonstrate the effectiveness of the proposed scheme. Figure 6.5 shows the experimental settings. Here, machine A, machine B, the server machine, and the management machine are connected via the same switch. Table 6.1 shows the specifications of each machine. We installed the proposed scheme on machines A and B with JIT compiling enabled. Since each machine has only a single NIC, BitVisor shares the NIC with the guest OS and uses it as a lwIP server. The IP addresses of machine A and its policy server are `192.168.20.10` and `192.168.20.11`, respectively. The IP addresses of the machine B and its policy server are `192.168.20.20` and `192.168.20.21`, respectively. The policy server listens to port 11111 on each machine. The server runs the Apache HTTP server, and its IP address is 192.168.20.1.

Approximately 10 seconds after the experiment began, machines A and B generated enor-

mous HTTP requests using Apache Bench [223]. Approximately 30 seconds after the start of the measurement, the management machine sent a filtering policy to machines A and B that was designed to stop all packets whose destination IP address was 192.168.20.1 (server machine; port 80) . Approximately 40 seconds after the start of the measurement, the management machine reset the filtering policy of machines A and B by sending a BPF program that accepted all packets.

Figure 6.6 shows the number of HTTP requests and CPU utilization (average of all cores) of the server. The server received approximately 10,000 requests per second when machines A and B began generating requests. The number of HTTP requests and CPU utilization decreased sharply at 30 seconds, which indicates that filtering had stopped packet transmissions. From 40 seconds to the end of the measurement period, the number of HTTP requests was approximately 10,000, which is similar to the number of request sent from 10 to 20 seconds. This indicates that packet transmissions were enabled again by resetting the filter policy. These results confirm that packet transmission can be suppressed by setting a new policy, which can be reset by the proposed scheme.

6.4.2 Performance Evaluation

We measured the throughput and latency between machine A and the server to evaluate the overhead of the proposed scheme. To compare the results, we also measured the throughput and latency of a bare metal system and KVM. When measuring KVM, we created a single VM and allocated the same number of cores and memory capacity to the VM.

6.4.2.1 Throughput

We used the following `netperf` [178] command to measure throughput.

```
$ netperf -l 3 -H <ipaddr> -t TCP_STREAM -- -m <MTU>
```

Note that machine A ran the netperf server during the measurement procedure. When measuring the proposed scheme, we set the filter to stop all packets whose destination IP address was 192.168.20.50 (port 80). This filtering setting allowed machine A to send packets to the server.

Figure 6.7 shows the measurement results. Note that all measurements were performed three times, and the average score is plotted in the figure. As can be seen, all throughput measurement results were approximately 930 Mbps and the proposed scheme did not degrade throughput.

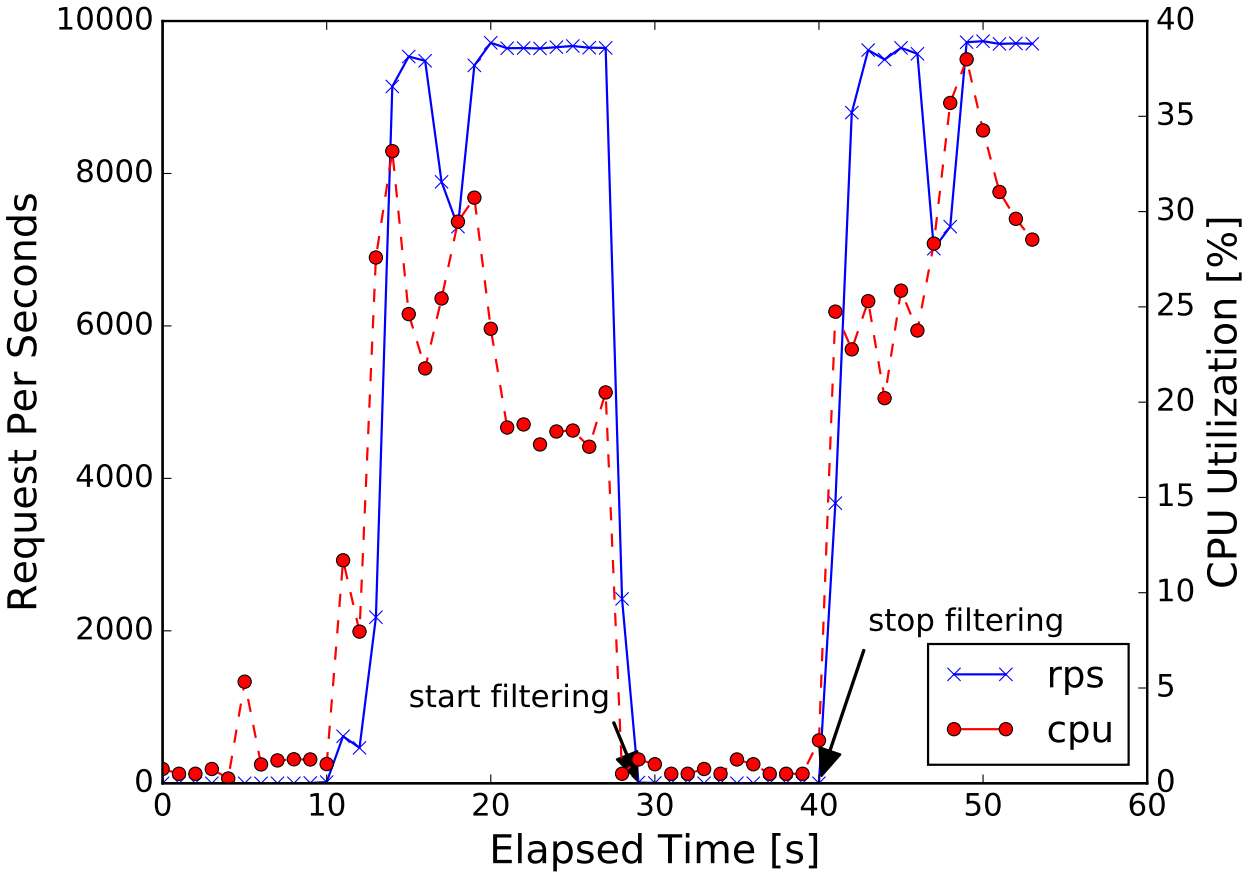


Figure 6.6: Results of Proof-of-concept Experiment

6.4.2.2 Latency

We measured latency using `ping` command. To evaluated the impact of the filter size (number of IP addresses to check), we measured the proposed scheme with different filtering settings. Figure 6.8 shows a boxplot of the latency. We measured the latency 30 times in each experiment. The numbers in the parentheses indicate the number of IP addresses to check. For example, `proposed (100)` means that the filter checked the destination IP address of the packets did not match any 100 addresses in the filter program.

The median latency values were 203, 241 (proposed (1)), and 312 µs for the bare metal system, the proposed scheme, and KVM, respectively. The proposed scheme has lower latency than KVM. As can be seen, there is little difference between the proposed schemes with different filter settings. This implies that the overhead of the proposed scheme mainly come from the introducing parapass-through architecture to intercept I/Os. The filtering itself performs efficiently.

Note that we also measured the throughput of the proposed method under the same filtering

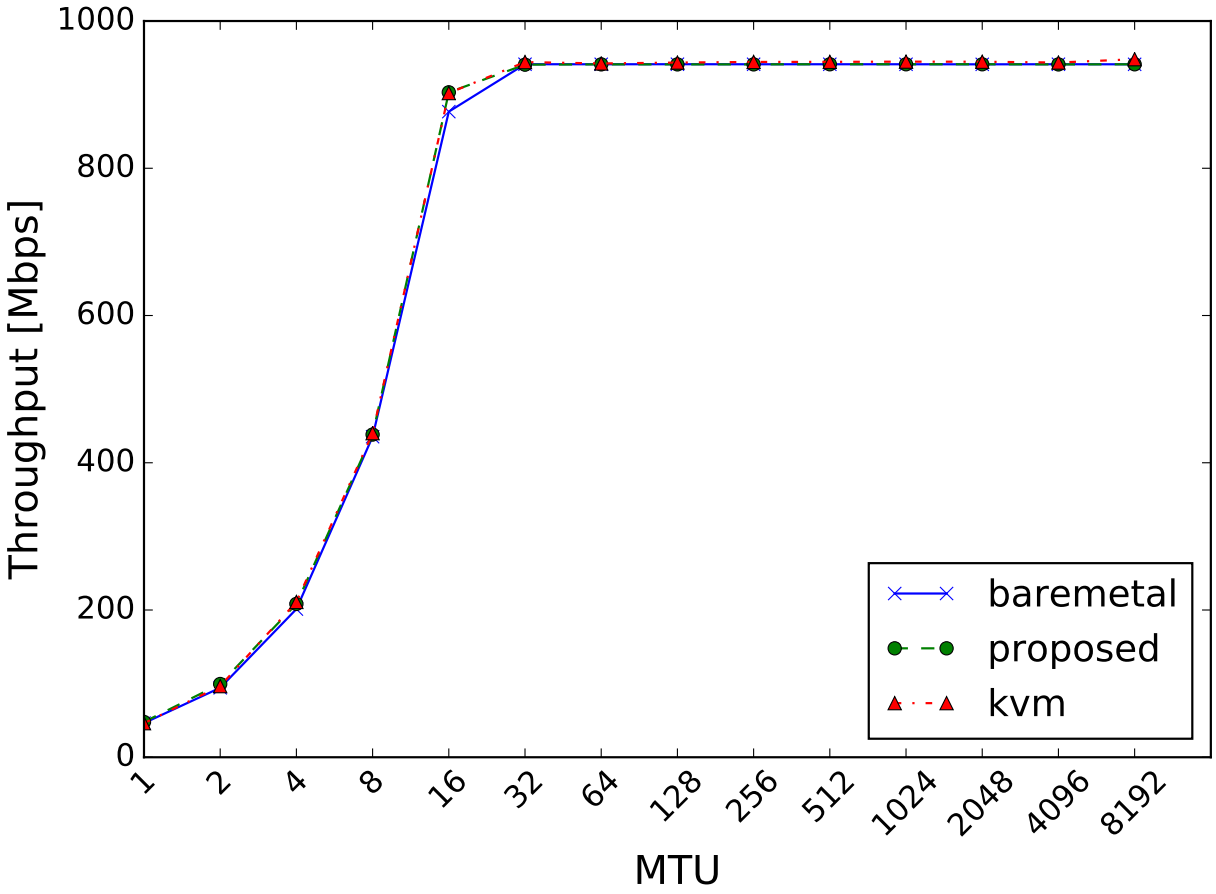


Figure 6.7: Throughput

settings. The throughput did not change regarding to the number of IP addresses to check.

6.5 Related Work of Source Side DDoS Protection

AVDOS [28], VMWall [31], xFilter [83], and AL-SAFE [103] all employ a hypervisor to enforce packet filtering in the attacking machine. These methods also propose a DDoS attack detection mechanism. While AVDOS uses only packet information, the other three methods utilize VMI for fine-grained filtering. In addition, these methods use the Xen [9] or KVM [21] hypervisors. The main difference of our study from these works is that our research goal is to create a lightweight and flexible scheme with which a system administrator can suppress packet transmission from managed machines. In addition, AVDOS, VMWall, and xFilter do not have a functionality to change policies externally. Note that the target of xFilter and AL-SAFE is the IaaS cloud. While we do not perform VMI, packet-based filtering is sufficient to stop DDoS attack packets and has a performance advantage.

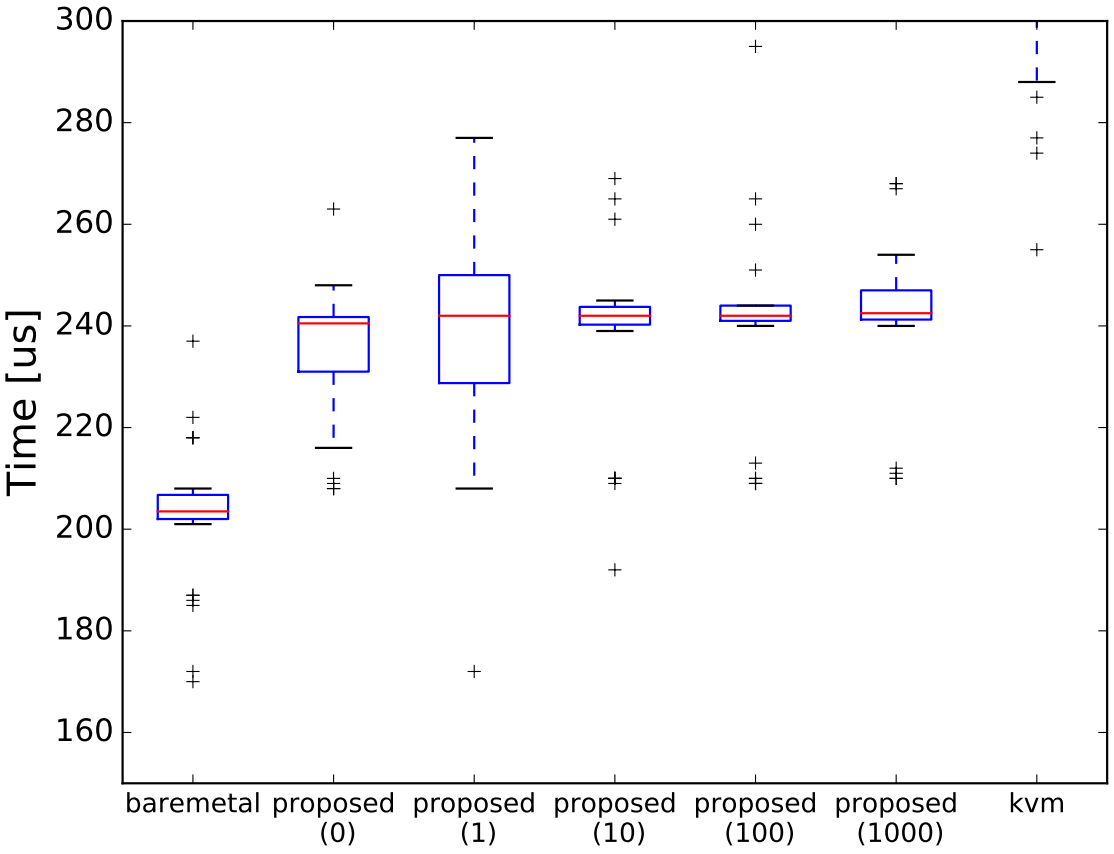


Figure 6.8: Ping Latency

KVM has a filtering mechanism called nwfilter [194] provided by libvirt, and nwfilter uses the Linux iptables [202] mechanism to filter packets. However, as the experimental results show, KVM incurs high virtualization overhead and is not suitable for personal computers. Major cloud vendors provide packet filtering services for their cloud instances. For example, the Amazon Elastic Compute Cloud uses a hypervisor with a firewall to enhance security [164]. Here, the communication of the guest OS passes through the hypervisor and firewall before reaching the external network. However, this method can only be used within Amazon AWS, and its design and implementation are not open to the public. In addition, it is not available for personal use.

Software Defined Networking (SDN) is a technology that can flexibly change network configurations and settings through software. Examples of such techniques include OpenFlow [205] and VMWare NSX [227]. Many studies have explored detection and defense methods against DDoS attacks using SDN. For example, [39] performed some pioneering research on DDoS attack detection and prevention using an OpenFlow switch. By implementing OpenFlow switch

functionality in the hypervisor, such methods can be used as source-side DDoS attack prevention systems. In fact, Open vSwitch [206] provides OpenFlow switch implementations for KVM and Xen. However, they are too complicated and heavyweight for personal computers.

Some studies have worked on detection and dynamic filtering of DDoS attacks on the attacker side. For example, to prevent spoofed packets from being transmitted, Network Ingress Filtering [4] confirms that the IP address of the packet sender is valid in the network at the edge router. D-WARD [11, 17] records bidirectional network traffic and compares the flow rate to a predetermined normal flow rate model. If the flow rates differ, an attack is assumed and filtering is performed. MULTOPS [7] and TOPS [8] use the ratio of the network flow rate in both the transmitting and receiving directions in routers as a DDoS attack detection method. Here, packet filtering is performed when one of flow rates is extremely large. [47] uses confidense-based filtering method and [106, 120] uses machine learning techniques to detect DDoS attacks in the cloud. Note that these detection mechanisms are orthogonal to and can be used in conjunction with the proposed scheme.

6.6 Summary

In this chapter, we have proposed a reliable, lightweight, transparent, and flexible DDoS attack prevention scheme that can be used by a system administrator to easily suppress packet transmissions from managed machines. We employ a thin hypervisor that can enforce packet filtering based on a filtering policy. The filtering policy is described as a executable code with restricted ISA. Safe execution is achieved by statically verifying the code in advance. The filtering policy can be dynamically changed by sending new policy to the policy server in the hypervisor. To make the proposed scheme lightweight and transparent, it uses a parapass-through hypervisor that intercepts only network I/Os. We implemented the proposed scheme using BitVisor and the eBPF. The experimental results show that the proposed scheme demonstrates negligible overhead relative to both latency and throughput.

7 Conclusion

In this thesis, we discussed optimizing virtualization for functional requirements through several use cases.

First, we optimized nested virtualization for hypervisor device driver testing. Focusing on the fact that the security features required by normal virtualization are unnecessary for testing purposes, we eliminated nested page shadowing. The performance experiments showed that the proposed method had a much lower overhead than the traditional nested virtualization scheme and could test the hypervisor device drivers in close to the real environment. Second, we presented the efficient IOMMU virtualization method for device protection. We achieved higher performance than a regular vIOMMU by only shadowing the necessary area for protection. These studies showed that we could gain significant performance improvements by limiting the functions to be virtualized. On the contrary, this means that current nested virtualization and IOMMU virtualization still have a large overhead. Improving these performances while keeping the functionality is one of the important future works. There is a limit to performance improvement in software alone. Research at the hardware architecture design level will be necessary in this regard.

We also presented a detailed performance evaluation of the NUMA-visible virtual machine on Linux. The evaluations revealed several problems with NUMA scheduling. We fixed the incorrect paravirtualization feature that causes severe performance degradation. Experimental results suggested that there is still room for improvement in NUMA scheduling performance.

Finally, we proposed a method to improve the flexibility of hypervisors without compromising performance by using a secure and lightweight language virtual machine. An an example of the use of the language virtual machine, we present source side DDoS prevention scheme using virtualization. The experimental results showed that the proposed scheme demonstrates negligible overhead relative to both latency and throughput. Using a secure and lightweight language virtual machine is a promising approach to increase hypervisor's flexibility, and finding other applications also would become an important work.

Bibliography

[1] Gerald J. Popek and Robert P. Goldberg. "Formal Requirements for Virtualizable Third Generation Architectures". In: *Communications of the ACM* 17.7 (1974), pp. 412–421. DOI: 10.1145/361011.361073.

[2] Alan Demers, Dan Greene, Carl Hauser, Wes Irish, John Larson, Scott Shenker, Howard Sturgis, Dan Swinehart, and Doug Terry. "Epidemic Algorithms for Replicated Database Maintenance". In: *Proceedings of the 6th Annual ACM Symposium on Principles of Distributed Computing (PODC'87)*. 1987, pp. 1–12. DOI: 10.1145/41840.41841.

[3] Steven McCanne and Van Jacobson. "The BSD Packet Filter: A New Architecture for User-level Packet Capture". In: *Proceedings of the USENIX Winter 1993*. Vol. 46. USENIX Association, 1993. URL: https://dl.acm.org/doi/10.5555/1267303.1267305.

[4] Paul Ferguson and Daniel Senie. "Network Ingress Filtering: Defeating Denial of Service Attacks which Employ IP Source Address Spoofing". In: *RFC 2827* (2000). URL: https://tools.ietf.org/html/rfc2827.

[5] Yin Zhang and Vern Paxson. "Detecting Stepping Stones". In: *Proceedings of the 9th USENIX Security Symposium (SEC'00)*. USENIX Association, 2000. URL: https://www.usenix.org/conference/9th-usenix-security-symposium/detecting-stepping-stones.

[6] Peter. M. Chen and Brian. D. Noble. "When Virtual is Better Than Real". In: *Proceedings of the 8th Workshop on Hot Topics in Operating Systems (HotOS'01)*. Institute of Electrical and Electronics Engineers, 2001, pp. 133–138. DOI: 10.1109/HOTOS.2001.990073.

[7] Thomer M Gil and Massimiliano Poletto. "MULTOPS: A Data-structure for Bandwidth Attack Detection". In: *Proceedings of the 10th USENIX Security Symposium (SEC'01)*. USENIX Association, 2001, pp. 23–38. URL: https://www.usenix.org/conference/10th-usenix-security-symposium/multops-data-structure-bandwidth-attack-detection.

[8] Samuel Abdelsayed, David Glimsholt, Christopher Leckie, Simon Ryan, and Samer Shami. "An Efficient Filter for Denial-of-service Bandwidth Attacks". In: *Proceedings of the 2003 IEEE Global Telecommunications Conference (GLOBECOM'03)*. Vol. 3. Institute of Electrical and Electronics Engineers, 2003, pp. 1353–1357. DOI: 10.1109/GLOCOM.2003.1258459.

[9] Paul Barham, Boris Dragovic, Keir Fraser, Steven Hand, Tim Harris, Alex Ho, Rolf Neugebauer, Ian Pratt, and Andrew Warfield. "Xen and the Art of Virtualization". In: *Proceedings of the 10th ACM Symposium on Operating Systems Principles (SOSP'03)*. 2003, pp. 164–177. DOI: 10.1145/945445.945462.

[10] Tal Garfinkel and Mendel Rosenblum. "A Virtual Machine Introspection Based Architecture for Intrusion Detection". In: *Proceedings of the 2003 Network and Distributed Systems Security Symposium (NDSS'03)*. 2003, pp. 191–206. URL: https://www.ndss-symposium.org/ndss2003/virtual-machine-introspection-based-architecture-intrusion-detection/.

[11] Jelena Mirkovic, Gregory. Prier, and Peter Reiher. "Source-end DDoS Defense". In: *Proceedings of the 2nd IEEE International Symposium on Network Computing and Applications (NCA'03)*. Institute of Electrical and Electronics Engineers, 2003, pp. 171–178. DOI: 10.1109/NCA.2003.1201153.

[12] Joshua LeVasseur, Volkmar Uhlig, Jan Stoess, and Stefan Götz. "Unmodified Device Driver Reuse and Improved System Dependability via Virtual Machines". In: *Proceedings of the 6th USENIX Conference on Operating Systems Design and Implementation (OSDI'04)*. USENIX association, 2004. URL: https://www.usenix.org/conference/osdi-04/unmodified-device-driver-reuse-and-improved-system-dependability-virtual-machines.

[13] David E. Lowell, Yasushi Saito, and Eileen J. Samberg. "Devirtualizable Virtual Machines Enabling General, Single-Node, Online Maintenance". In: *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*. Boston, MA, USA: Association for Computing Machinery, 2004, pp. 211–223. DOI: 10.1145/1024393.1024419.

[14] Jelena Mirkovic and Peter Reiher. "A Taxonomy of DDoS Attack and DDoS Defense Mechanisms". In: *SIGCOMM Comput. Commun. Rev.* 34.2 (2004), pp. 39–53. DOI: 10.1145/997150.997156.

[15] Nick L. Petroni Jr., Timothy Fraser, Jesus Molina, and William A. Arbaugh. "Copilot - a Coprocessor-based Kernel Runtime Integrity Monitor". In: *Proceedings of the 13th Conference on USENIX Security Symposium (SEC'04)*. USENIX Association, 2004. URL: http://dl.acm.org/citation.cfm?id=1251375.1251388.

[16] Volkmar Uhlig, Joshua LeVasseur, Espen Skoglund, and Uwe Dannowski. "Towards Scalable Multiprocessor Virtual Machines". In: *Proceedings of the 3rd Conference on Virtual Machine Research And Technology Symposium*. USENIX Association, 2004. URL: http://dl.acm.org/citation.cfm?id=1267242.1267246.

[17] Jelena Mirkovic and Peter Reiher. "D-WARD: A Source-end Defense against Flooding Denial-of-service Attacks". In: *IEEE Transactions on Dependable and Secure Computing* 2.3 (2005), pp. 216–232. DOI: 10.1109/TDSC.2005.35.

[18] PCI-SIG. *PCI-SIG ENGINEERING CHANGE NOTICE PCI Express Access Control Services (ACS)*. 2005.

[19] Thomas Ball, Ella Bounimova, Byron Cook, Vladimir Levin, Jakob Lichtenberg, Con McGarvey, Bohus Ondrusek, Sriram K. Rajamani, and Abdullah Ustuner. "Thorough Static Analysis of Device Drivers". In: *Proceedings of the 1st ACM SIGOPS/EuroSys European Conference on Computer Systems (EuroSys'06)*. Association for Computing Machinery, 2006, pp. 73–85. DOI: 10.1145/1217935.1217943.

[20] Andreas Johansson, Neeraj Suri, and Brendan Murphy. "On the Selection of Error Model(s) for OS Robustness Evaluation". In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. Institute of Electrical and Electronics Engineers, 2007, pp. 502–511. DOI: 10.1109/DSN.2007.71.

[21] Avi Kivity, Yaniv Kamay, Dor Laor, Uri Lublin, and Anthony Liguori. "KVM: the Linux Virtual Machine Monitor". In: *Proceedings of the 2007 Linux Symposium*. 2007, pp. 225–230. URL: https://www.kernel.org/doc/ols/2007/ols2007v1-pages-225-230.pdf.

[22] Manuel Mendonca and Nuno Neves. "Robustness Testing of the Windows DDK". In: *Proceedings of the 37th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'07)*. Institute of Electrical and Electronics Engineers, 2007, pp. 554–564. DOI: 10.1109/DSN.2007.85.

[23] Hendrik Post and Wolfgang Küchlin. "Integrated Static Analysis for Linux Device Driver Verification". In: *Proceedings of the 2007 Integrated Formal Methods (IFM'07)*. Springer Berlin Heidelberg, 2007.

[24] Joanna Rutkowska. "Beyond The CPU: Defeating Hardware Based RAM Acquisition". In: (2007).

[25] Arvind Seshadri, Mark Luk, Ning Qu, and Adrian Perrig. "SecVisor: A Tiny Hypervisor to Provide Lifetime Kernel Code Integrity for Commodity OSes". In: *Proceedings of 21st ACM SIGOPS Symposium on Operating Systems Principles (SOSP'07)*. Association for Computing Machinery, 2007, pp. 335–350. DOI: 10.1145/1294261.1294294.

[26] Thomas Witkowski, Nicolas Blanc, Daniel Kroening, and Georg Weissenbacher. "Model Checking Concurrent Linux Device Drivers". In: *Proceedings of the 22nd IEEE/ACM International Conference on Automated Software Engineering (ASE'07)*. Association for Computing Machinery, 2007, pp. 501–504. DOI: 10.1145/1321631.1321719.

[27] Thomas Friebel and Sebastian Biemueller. "How to Deal with Lock Holder Preemption". In: *Xen Summit North America*. 2008.

[28] Sanjam Garg and Huzur Saran. "Anti-DDoS Virtualized Operating System". In: *Proceedings of the 3rd International Conference on Availability, Reliability and Security (ARES'08)*. Institute of Electrical and Electronics Engineers, 2008, pp. 667–674. DOI: 10.1109/ARES.2008.120.

[29] Man-Lap Li, Pradeep Ramachandran, Swarup Kumar Sahoo, Sarita V. Adve, Vikram S. Adve, and Yuanyuan Zhou. "Understanding the Propagation of Hard Errors to Software and Implications for Resilient System Design". In: *Proceedings of the 13th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'08)*. Association for Computing Machinery, 2008, pp. 265–276. DOI: 10.1145/1346281.1346315.

[30] Rusty Russell. "virtio: Towards a De-facto Standard for Virtual I/O Devices". In: *ACM SIGOPS Operating Systems Review* 42.5 (2008), pp. 95–103. DOI: 10.1145/1400097.1400108.

[31] Abhinav Srivastava and Jonathon Giffin. "Tamper-resistant, Application-aware Blocking of Malicious Network Connections". In: *Proceedings of the 11th International Symposium on Recent Advances in Intrusion Detection (RAID'08)*. Springer, 2008, pp. 39–58. DOI: 10.1007/978-3-540-87403-4_3.

[32] Asim Kadav, Matthew J. Renzelmann, and Michael M. Swift. "Tolerating Hardware Device Failures in Software". In: *Proceedings of the ACM SIGOPS 22Nd Symposium on Operating Systems Principles (SOSP'09)*. Association for Computing Machinery, 2009, pp. 59–72. DOI: 10.1145/1629575.1629582.

[33] Julia L. Lawall, Julien Brunel, Nicolas Palix, Rene Rydhof Hansen, Henrik Stuart, and Gilles Muller. "WYSIWIB: A Declarative Approach to Finding API Protocols and Bugs in Linux Code". In: *Proceedings of the 2009 IEEE/IFIP International Conference on Dependable Systems Networks (DSN'09)*. Institute of Electrical and Electronics Engineers, 2009, pp. 43–52. DOI: 10.1109/DSN.2009.5270354.

[34] PCI-SIG. *Address Translation Services Specification Revision 1.0*. 2009.

[35] Takahiro Shinagawa, Hideki Eiraku, Kouichi Tanimoto, Kazumasa Omote, Shoichi Hasegawa, Takashi Horie, Manabu Hirano, Kenichi Kourai, Yoshihiro Oyama, Eiji Kawai, Kenji Kono, Shigeru Chiba, Yasushi Shinjo, and Kazuhiko Kato. "BitVisor: A Thin Hypervisor for Enforcing I/O Device Security". In: *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'09)*. Association for Computing Machinery, 2009, pp. 121–130. DOI: 10.1145/1508293.1508311.

[36] Chuliang Weng, Zhigang Wang, Minglu Li, and Xinda Lu. "The Hybrid Scheduling Framework for Virtual Machine Systems". In: *Proceedings of the 2009 ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'09)*. Association for Computing Machinery, 2009. DOI: 10.1145/1508293.1508309.

[37] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. "The Turtles Project: Design and Implementation of Nested Virtualization". In: *Proceedings of the 9th USENIX Conference on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, 2010, pp. 423–436. URL: http://dl.acm.org/citation.cfm?id=1924943.1924973.

[38] Muli Ben-Yehuda, Michael D. Day, Zvi Dubitzky, Michael Factor, Nadav Har'El, Abel Gordon, Anthony Liguori, Orit Wasserman, and Ben-Ami Yassour. "The Turtles Project: Design and Implementation of Nested Virtualization". In: *Proceedings of the 9th USENIX Symposium on Operating Systems Design and Implementation (OSDI'10)*. USENIX Association, 2010. URL: https://www.usenix.org/conference/osdi10/turtles-project-design-and-implementation-nested-virtualization.

[39] Rodrigo Braga, Edjard Mota, and Alexandre Passito. "Lightweight DDoS Flooding Attack Detection using NOX/OpenFlow". In: *Proceedings of the 2010 IEEE Local Computer Network Conference (LCN'10)*. Institute of Electrical and Electronics Engineers, 2010, pp. 408–415. DOI: 10.1109/LCN.2010.5735752.

[40] Yosuke Chubachi, Takahiro Shinagawa, and Kazuhiko Kato. "Hypervisor-based Prevention of Persistent Rootkits". In: *Proceedings of the 2010 ACM Symposium on Applied Computing (SAC'10)*. Association for Computing Machinery, 2010, pp. 214–220. DOI: 10.1145/1774088.1774131.

[41] Volodymyr Kuznetsov, Vitaly Chipounov, and George Candea. "Testing Closed-source Binary Device Drivers with DDT". In: *Proceedings of the 2010 USENIX Conference on USENIX Annual Technical Conference (ATC'10)*. USENIX Association, 2010, pp. 1–12. URL: http://dl.acm.org/citation.cfm?id=1855840.1855852.

[42] Lorenzo Martignoni, Aristide Fattori, Roberto Paleari, and Lorenzo Cavallaro. "Live and Trustworthy Forensic Analysis of Commodity Production Systems". In: *Proceedings of the 13th Recent Advances in Intrusion Detection (RAID'10)*. Springer, 2010, pp. 297–316. DOI: 10.1007/978-3-642-15512-3_16.

[43] Jiang Wang, Angelos Stavrou, and Anup Ghosh. "HyperCheck: A Hardware-Assisted Integrity Monitor". In: *Proceedings of the 13th Recent Advances in Intrusion Detection (RAID'10)*. Springer, 2010, pp. 158–177. DOI: 10.1007/978-3-642-15512-3_9.

[44] Sidney Amani, Leonid Ryzhyk, Alastair F. Donaldson, Gernot Heiser, Alexander Legg, and Yanjin Zhu. "Static Analysis of Device Drivers: We Can Do Better!" In: *Proceedings of the 2nd Asia-Pacific Workshop on Systems (APSys'11)*. Association for Computing Machinery, 2011, 8:1–8:5. DOI: 10.1145/2103799.2103809.

[45] Nadav Amit, Muli Ben-Yehuda, Dan Tsafrir, and Assaf Schuster. "vIOMMU: Efficient IOMMU Emulation". In: *Proceedings of the 2011 USENIX Annual Technical Conference (ATC'11)*. USENIX Association, 2011. URL: https://www.usenix.org/conference/usenixatc11/viommu-efficient-iommu-emulation.

[46] Saketh Bharadwaja, Weiqing Sun, Mohammed Niamat, and Fangyang Shen. "Collabra: A Xen Hypervisor Based Collaborative Intrusion Detection System". In: *Proceedings of the 8th International Conference on Information Technology: New Generations*. 2011, pp. 695–700. DOI: 10.1109/ITNG.2011.123.

[47] Qi Chen, Wenmin Lin, Wanchun Dou, and Shui Yu. "CBF: A Packet Filtering Method for DDoS Attack Defense in Cloud Environment". In: *Proceedings of the 9th International Conference on Dependable, Autonomic and Secure Computing (DSN'11)*. Institute of Electrical and Electronics Engineers, 2011, pp. 427–434. DOI: 10.1109/DASC.2011.86.

[48] Khaled Z. Ibrahim, Steven Hofmeyr, and Costin Iancu. "Characterizing the Performance of Parallel Applications on Multi-socket Virtual Machines". In: *Proceedings of the 11th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid'11)*. Institute of Electrical and Electronics Engineers, 2011, pp. 1–12. DOI: 10.1109/CCGrid.2011.50.

[49] Vladimir V. Rubanov and Eugene A. Shatokhin. "Runtime Verification of Linux Kernel Modules Based on Call Interception". In: *2011 Fourth IEEE International Conference on Software Testing, Verification and Validation (ICST'11)*. Institute of Electrical and Electronics Engineers, 2011, pp. 180–189. DOI: 10.1109/ICST.2011.20.

[50] Fernand Lone Sang, Vincent Nicomette, and Yves Deswarte. "I/O Attacks in Intel PC-based Architectures and Countermeasures". In: *Proceedings of the 2011 First SysSec Workshop*. Institute of Electrical and Electronics Engineers, 2011. DOI: 10.1109/SysSec.2011.10.

[51] Orathai Sukwong and Hyong S. Kim. "Is Co-scheduling Too Expensive for SMP VMs?" In: *Proceedings of the 6th European Conference on Computer Systems (EuroSys'11)*. Association for Computing Machinery, 2011. DOI: 10.1145/1966445.1966469.

[52] Jiang Wang, Fengwei Zhang, Kun Sun, and Angelos Stavrou. "Firmware-Assisted Memory Acquisition and Analysis Tools for Digital Forensics". In: *Proceedings of the 2011 6th IEEE International Workshop on Systematic Approaches to Digital Forensic Engineering (SADFE'11)*. Institute of Electrical and Electronics Engineers, 2011, pp. 1–5. DOI: 10.1109/SADFE.2011.7.

[53] Luwei Cheng and Cho-Li Wang. "vBalance: Using Interrupt Load Balance to Improve I/O Performance for SMP Virtual Machines". In: *Proceedings of the 3rd ACM Symposium on Cloud Computing (SoCC'12)*. Association for Computing Machinery, 2012. DOI: 10.1145/2391229.2391231.

[54] Vitaly Chipounov, Volodymyr Kuznetsov, and George Candea. "The S2E Platform: Design, Implementation, and Applications". In: *ACM Trans. Comput. Syst.* 30.1 (2012), 2:1–2:49. DOI: 10.1145/2110356.2110358.

[55] Alexander Kudryavtsev, Vladimir Koshelev, and Arutvun Avetisyan. "Modern HPC Cluster Virtualization Using KVM and Palacios". In: *Proceedings of the 19th International Conference on High Performance Computing (HPC'12)*. Institute of Electrical and Electronics Engineers, 2012, pp. 1–9. DOI: 10.1109/HiPC.2012.6507495.

[56] Hyungon Moon, Hojoon Lee, Jihoon Lee, Kihwan Kim, Yunheung Paek, and Brent Byunghoon Kang. "Vigilare: Toward Snoop-based Kernel Integrity Monitor". In: *Proceedings of the 12th ACM Conference on Computer and Communications Security (CCS'12)*. Association for Computing Machinery, 2012, pp. 28–37. DOI: 10.1145/2382196.2382202.

[57] Tilo Müller, Benjamin Taubmann, and Felix C. Freiling. "OS-Independent Software-Based Full Disk Encryption Secure against Main Memory Attacks". In: *Proceedings of the 2012 Applied Cryptography and Network Security (ACNS'12)*. Springer, 2012, pp. 66–83. DOI: 10.1007/978-3-642-31284-7_5.

[58] Yushi Omote, Yosuke Chubachi, Takahiro Shinagawa, Tomohiro Kitamura, Hideki Eiraku, and Katsuya Matsubara. "Hypervisor-based Background Encryption". In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC'12)*. Association for Computing Machinery, 2012, pp. 1829–1836. DOI: 10.1145/2245276.2232073.

[59] Yoshihiro Oyama, Tran Truong Duc Giang, Yosuke Chubachi, Takahiro Shinagawa, and Kazuhiko Kato. "Detecting Malware Signatures in a Thin Hypervisor". In: *Proceedings of the 27th Annual ACM Symposium on Applied Computing (SAC'12)*. SAC '12. New York, NY, USA: Association for Computing Machinery, 2012, pp. 1807–1814. DOI: 10.1145/2245276.2232070.

[60] Sankaralingam Panneerselvam and Michael M. Swift. "Chameleon: Operating System Support for Dynamic Processors". In: *Proceedings of the 17th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'12)*. Association for Computing Machinery, 2012. DOI: 10.1145/2150976.2150988.

[61] K. T. Raghavendra, S. Vaddagiri, N. Dadhania, and J. Fitzhardinge. "Paravirtualization for Scalable Kernel-Based Virtual Machine (KVM)". In: *Proceedings of the 2012 IEEE International Conference on Cloud Computing in Emerging Markets (CCEM'12)*. Institute of Electrical and Electronics Engineers, 2012, pp. 1–5. DOI: 10.1109/CCEM.2012.6354619.

[62] Matthew J. Renzelmann, Asim Kadav, and Michael M. Swift. "SymDrive: Testing Drivers Without Devices". In: *Proceedings of the 10th USENIX Conference on Operating Systems Design and Implementation (OSDI'12)*. USENIX Association, 2012, pp. 279–292. URL: http://dl.acm.org/citation.cfm?id=2387880.2387908.

[63] Varshapriya Shakti D Shekar B B Meshram. "Device Driver Fault Simulation using KEDR". In: *International Journal. of Advanced Research in Computer Engineering & Technology*. Vol. 1. 4. 2012, pp. 580–584.

[64] Joe Sylve. "Lime – Linux Memory Extractor". In: *Proceedings of the 7th ShmooCon Conference*. 2012.

[65] Zongwei Zhou, Virgil D. Gligor, James Newsome, and Jonathan M. McCune. "Building Verifiable Trusted Path on Commodity x86 Computers". In: *Proceedings of the 32nd IEEE Symposium on Security and Privacy (S&P'12)*. Institute of Electrical and Electronics Engineers, 2012. DOI: 10.1109/SP.2012.42.

[66] Ŝtefan Balogh and Miroslav Mydlo. "New Possibilities for Memory Acquisition by Enabling DMA Using Network Card". In: *2013 IEEE 7th International Conference on Intelligent Data Acquisition and Advanced Computing Systems (IDAACS'13)*. Vol. 02. 2013, pp. 635–639. DOI: 10.1109/IDAACS.2013.6663002.

[67] Kai Cong, Fei Xie, and Li Lei. "Symbolic Execution of Virtual Devices". In: *Proceedings of the 13th International Conference on Quality Software (QSIC'13)*. Institute of Electrical and Electronics Engineers, 2013, pp. 1–10. DOI: 10.1109/QSIC.2013.44.

[68] Nadav Har'El, Abel Gordon, Alex Landau, Muli Ben-Yehuda, Avishay Traeger, and Razya Ladelsky. "Efficient and Scalable Paravirtual I/O System". In: *Presented as part of the 2013 USENIX Annual Technical Conference (ATC'13)*. USENIX, 2013. URL: https://www.usenix.org/conference/atc13/technical-sessions/presentation/har%7B%5Ctextquoteright%7Del.

[69] Hwanju Kim, Sangwook Kim, Jinkyu Jeong, Joonwon Lee, and Seungryoul Maeng. "Demand-based Coordinated Scheduling for SMP VMs". In: *Proceedings of the 18th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'13)*. Association for Computing Machinery, 2013. DOI: 10.1145/2451116.2451156.

[70] Hojoon Lee, HyunGon Moon, DaeHee Jang, Kihwan Kim, Jihoon Lee, Yunheung Paek, and Brent ByungHoon Kang. "KI-Mon: A Hardware-assisted Event-triggered Monitoring Platform for Mutable Kernel Object". In: *Proceedings of the 22nd USENIX Security Symposium (SEC'13)*. USENIX Association, 2013, pp. 511–526. URL: https://www.usenix.org/conference/usenixsecurity13/technical-sessions/presentation/lee.

[71] Jiannan Ouyang and John R. Lange. "Preemptable Ticket Spinlocks: Improving Consolidated Performance in the Cloud". In: *Proceedings of the 9th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'13)*. Association for Computing Machinery, 2013. DOI: 10.1145/2451512.2451549.

[72] Xiang Song, Jicheng Shi, Haibo Chen, and Binyu Zang. "Schedule Processes, Not VCPUs". In: *Proceedings of the 4th Asia-Pacific Workshop on Systems (APSys'13)*. Association for Computing Machinery, 2013. DOI: 10.1145/2500727.2500736.

[73] Patrick Stewin. "A Primitive for Revealing Stealthy Peripheral-Based Attacks on the Computing Platform's Main Memory". In: *Proceedings of the 16th Research in Attacks, Intrusions, and Defenses (RAID'13)*. Springer, 2013, pp. 1–20. DOI: 10.1007/978-3-642-41284-4_1.

[74] VMWare. *The CPU Scheduler in VMware vSphere® 5.1*. https://www.vmware.com/content/dam/digitalmarketing/vmware/en/pdf/techpaper/vmware-vsphere-cpu-sched-performance-white-paper.pdf (Visted on 2022-3-02). 2013.

[75] Stefan Winter, Michael Tretter, Benjamin Sattler, and Neeraj Suri. "simFI: From single to simultaneous software fault injections". In: *2013 43rd Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'13)*. Institute of Electrical and Electronics Engineers, 2013, pp. 1–12. DOI: 10.1109/DSN.2013.6575310.

[76] Cong Xu, Sahan Gamage, Hui Lu, Ramana Kompella, and Dongyan Xu. "vTurbo: Accelerating Virtual Machine I/O Processing Using Designated Turbo-Sliced Core". In: *Proceedings of the the 2013 USENIX Annual Technical Conference (ATC'13)*. USENIX, 2013. URL: https://www.usenix.org/conference/atc13/technical-sessions/presentation/xu.

[77] Saman Taghavi Zargar, James Joshi, and David Tipper. "A Survey of Defense Mechanisms against Distributed Denial of Service (DDoS) Flooding Attacks". In: *IEEE Commun. Surveys & Tutorials* 15.4 (2013), pp. 2046–2069. DOI: 10.1109/SURV.2013.031413.00127.

[78] Liu Ziyi, Lee Jong Hyuk, Zeng Junyuan, Wen Yuanfeng, Lin Zhiqiang, and Shi Weidong. "CPU Transparent Protection of OS Kernel and Hypervisor Integrity with Programmable DRAM". In: *ACM SIGARCH Computer Architecture News* (2013). URL: https://dl.acm.org/doi/abs/10.1145/2508148.2485956.

[79] Nuttapong Chakthranont, Phonlawat Khunphet, Ryousei Takano, and Tsutomu Ikegami. "Exploring the Performance Impact of Virtualization on an HPC Cloud". In: *Proceedings of the 6th IEEE International Conference on Cloud Computing Technology and Science (CloudCom'14)*. Institute of Electrical and Electronics Engineers, 2014, pp. 426–432. DOI: 10.1109/CloudCom.2014.71.

[80] Xiaoning Ding, Phillip B. Gibbons, Michael A. Kozuch, and Jianchen Shan. "Gleaner: Mitigating the Blocked-Waiter Wakeup Problem for Virtualized Multicore Applications". In: *Proceedings of the 2014 USENIX Annual Technical Conference (ATC'18)*. USENIX Association, 2014, pp. 73–84. URL: https://www.usenix.org/conference/atc14/technical-sessions/presentation/ding.

[81] Sahan Gamage, Cong Xu, Ramana Rao Kompella, and Dongyan Xu. "vPipe: Piped I/O Offloading for Efficient Data Movement in Virtualized Clouds". In: *Proceedings of the 2014 ACM Symposium on Cloud Computing (SoCC'14)*. Association for Computing Machinery, 2014. DOI: 10.1145/2670979.2671006.

[82] Daehee Jang, Hojoon Lee, Minsu Kim, Daehyeok Kim, Daegyeong Kim, and Brent Byunghoon Kang. "ATRA: Address Translation Redirection Attack against Hardware-based External Monitors". In: *Proceedings of the 21st ACM Conference on Computer and Communications Security (CCS'14)*. ACM, 2014, pp. 167–178. DOI: 10.1145/2660267.2660303.

[83] Kenichi Kourai, Takeshi Azumi, and Shigeru Chiba. "Efficient and Fine-Grained VMM-Level Packet Filtering for Self-Protection". In: *Int. J. Adapt. Resilient Auton. Syst.* 5.2 (2014), pp. 83–100. DOI: 10.4018/ijaras.2014040105.

[84] Ming Liu and Tao Li. "Optimizing Virtual Machine Consolidation Performance on NUMA Server Architecture for Cloud Workloads". In: *Proceeding of the ACM/IEEE 41st International Symposium on Computer Architecture (ISCA'14)*. Institute of Electrical and Electronics Engineers, 2014, pp. 325–336. DOI: 10.1109/ISCA.2014.6853224.

[85] Gábor Pék, Andrea Lanzi, Abhinav Srivastava, Davide Balzarotti, Aurélien Francillon, and Christoph Neumann. "On the Feasibility of Software Attacks on Commodity Virtual Machine Monitors via Direct Device Assignment". In: *Proceedings of the 9th ACM Symposium on Information, Computer and Communications Security (AsiaCCS'14)*. Association for Computing Machinery, 2014. DOI: 10.1145/2590296.2590299.

[86] He Sun, Kun Sun, Yuewu Wang, Jiwu Jing, and Sushil Jajodia. "TrustDump: Reliable Memory Acquisition on Smartphones". In: *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS'14)*. Springer, 2014, pp. 202–218. DOI: 10.1007/978-3-319-11203-9_12.

[87] Fengwei Zhang, Haining Wang, Kevin Leach, and Angelos Stavrou. "A Framework to Secure Peripherals at Runtime". In: *Proceedings of the 19th European Symposium on Research in Computer Security (ESORICS'14)*. Springer, 2014, pp. 219–238. DOI: 10.1007/978-3-319-11203-9_13.

[88] Kai Cong, Li Lei, Zhenkun Yang, and Fei Xie. "Automatic Fault Injection for Driver Robustness Testing". In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis (ISSTA'15)*. Association for Computing Machinery, 2015, pp. 361–372. DOI: 10.1145/2771783.2771811.

[89] Takaaki Fukai, Yushi Omote, Takahiro Shinagawa, and Kazuhiko Kato. "OS-Independent Live Migration Scheme for Bare-Metal Clouds". In: *Proceedings of the 2015 IEEE/ACM 8th International Conference on Utility and Cloud Computing (UCC'15)*. 2015, pp. 80–89. DOI: 10.1109/UCC.2015.23.

[90] Moshe Malka, Nadav Amit, Muli Ben-Yehuda, and Dan Tsafrir. "rIOMMU: Efficient IOMMU for I/O Devices That Employ Ring Buffers". In: *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. Association for Computing Machinery, 2015, pp. 355–368. DOI: 10.1145/2694344.2694355.

[91] Moshe Malka, Nadav Amit, and Dan Tsafrir. "Efficient Intra-Operating System Protection Against Harmful DMAs". In: *Proceedings of the 13th USENIX Conference on File and Storage Technologies (FAST'15)*. USENIX Association, 2015. URL: https://www.usenix.org/conference/fast15/technical-sessions/presentation/malka.

[92] T. Miao and H. Chen. "FlexCore: Dynamic virtual machine scheduling using VCPU ballooning". In: *Tsinghua Science and Technology* 20.1 (2015), pp. 7–16. DOI: 10.1109/TST.2015.7040515.

[93] Benoît Morgan, Éric Alata, Vincent Nicomette, Mohamed Kâaniche, and Guillaume Averlant. "Design and Implementation of a Hardware Assisted Security Architecture for Software Integrity Monitoring". In: *Proceedings of 2015 IEEE 21st Pacific Rim International Symposium on Dependable Computing (PRDC'15)*. Institute of Electrical and Electronics Engineers, 2015, pp. 189–198. DOI: 10.1109/PRDC.2015.46.

[94] Yushi Omote, Takahiro Shinagawa, and Kazuhiko Kato. "Improving Agility and Elasticity in Bare-metal Clouds". In: *Proceedings of the 20th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. Association for Computing Machinery, 2015, pp. 145–159. DOI: 10.1145/2694344.2694349.

[95] Yushi Omote, Takahiro Shinagawa, and Kazuhiko Kato. "Improving Agility and Elasticity in Bare-metal Clouds". In: *Proceedings of the Twentieth International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'15)*. 2015, pp. 145–159. DOI: 10.1145/2775054.2694349.

[96] Sankaralingam Panneerselvam, Michael Swift, and Nam Sung Kim. "Bolt: Faster Reconfiguration in Operating Systems". In: *Proceedings of the 2015 USENIX Annual Technical Conference (USENIX ATC'15)*. USENIX, 2015. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/panneerselvam.

[97] Omer Peleg, Adam Morrison, Benjamin Serebrin, and Dan Tsafrir. "Utilizing the IOMMU Scalably". In: *Proceedings of the 2015 USENIX Annual Technical Conference (ATC'15)*. USENIX Association, 2015. URL: https://www.usenix.org/conference/atc15/technical-session/presentation/peleg.

[98] Stephen M. Trimberger. "Three Ages of FPGAs: A Retrospective on the First Thirty Years of FPGA Technology". In: *Proceedings of the IEEE* 103.3 (2015), pp. 318–331. DOI: 10.1109/JPROC.2015.2392104.

[99] Fengwei Zhang, Kevin Leach, Angelos Stavrou, Haining Wang, and Kun Sun. "Using Hardware Features for Increased Debugging Transparency". In: *Proceedings of the 36th IEEE Symposium on Security and Privacy (S&P'15)*. Institute of Electrical and Electronics Engineers, 2015. DOI: 10.1109/SP.2015.11.

[100] Ning Zhang, Kun Sun, Wenjing Lou, Y. Thomas Hou, and Sushil Jajodia. "Now You See Me: Hide and Seek in Physical Address Space". In: *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security (AsiaCCS'15)*. ACM, 2015, pp. 321–331. DOI: 10.1145/2714576.2714600.

[101] Jia-Ju Bai, Yu-Ping Wang, Jie Yin, and Shi-Min Hu. "Testing Error Handling Code in Device Drivers Using Characteristic Fault Injection". In: *Proceedings of the 2016 USENIX Conference on Usenix Annual Technical Conference (ATC'16)*. USENIX Association, 2016, pp. 635–647. URL: https://www.usenix.org/conference/atc16/technical-sessions/presentation/bai.

[102] Luwei Cheng, Jia Rao, and Francis C. M. Lau. "vScale: Automatic and Efficient Processor Scaling for SMP Virtual Machines". In: *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*. Institute of Electrical and Electronics Engineers, 2016. DOI: 10.1145/2901318.2901321.

[103] Anna Giannakou, Louis Rilling, Jean-Louis Pazat, and Christine Morin. "AL-SAFE: A Secure Self-adaptable Application-level Firewall for IaaS Clouds". In: *Proceedings of the 8th IEEE International Conference on Cloud Computing Technology and Science (CloudCom'16)*. Institute of Electrical and Electronics Engineers, 2016, pp. 383–390. DOI: 10.1109/CloudCom.2016.0067.

[104] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. "Opportunistic Spinlocks: Achieving Virtual Machine Scalability in the Clouds". In: *SIGOPS Oper. Syst. Rev.* 50.1 (2016), pp. 9–16. DOI: 10.1145/2903267.2903271.

[105] Lazaros Koromilas, Giorgos Vasiliadis, Elias Athanasopoulos, and Sotiris Ioannidis. "GRIM: Leveraging GPUs for Kernel Integrity Monitoring". In: *Proceedings of the 19th Research in Attacks, Intrusions, and Defenses (RAID'16)*. Springer International Publishing, 2016, pp. 3–23. URL: https://link.springer.com/chapter/10.1007/978-3-319-45719-2_1.

[106] Raneel Kumar, Sunil Pranit Lal, and Alok Sharma. "Detecting Denial of Service Attacks in the Cloud". In: *Proceedings of the 14th International Conference on Dependable, Autonomic and Secure Computing, 14th International Conference on Pervasive Intelligence and Computing, 2nd International Conference on Big Data Intelligence and Computing and Cyber Science and Technology Congress (DASC/PiCom/DataCom/CyberSciTech'16)*. Institute of Electrical and Electronics Engineers, 2016, pp. 309–316. DOI: 10.1109/DASC-PICom-DataCom-CyberSciTec.2016.70.

[107] Jean-Pierre Lozi, Baptiste Lepers, Justin Funston, Fabien Gaud, Vivien Quéma, and Alexandra Fedorova. "The Linux Scheduler: a Decade of Wasted Cores". In: *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*. Association for Computing Machinery, 2016, pp. 1–16. DOI: 10.1145/2901318.2901326.

[108] Alex Markuze, Adam Morrison, and Dan Tsafrir. "True IOMMU Protection from DMA Attacks: When Copy is Faster than Zero Copy". In: *Proceedings of the 21st International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'16)*. ACM, 2016. DOI: 10.1145/2872362.2872379.

[109] Jiannan Ouyang, John R. Lange, and Haoqiang Zheng. "Shoot4U: Using VMM Assists to Optimize TLB Operations on Preempted vCPUs". In: *Proceedings of the12th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'16)*. Association for Computing Machinery, 2016. DOI: 10.1145/2892242.2892245.

[110] Syed Asif Raza Shah, Amol Hindurao Jaikar, Sangwook Bae, and Seo-Young Noh. "Improve Performance and Throughput of VMs for Scientific Workloads in a Cloud Environment". In: *Proceedings of the 2016 International Conference on Platform Technology and Service (PlatCon'16)*. IEEE, 2016, pp. 1–6. DOI: 10.1109/PlatCon.2016.7456802.

[111] Chad Spensky, Hongyi Hu, and Kevin Leach. "LO-PHI: Low-Observable Physical Host Instrumentation for Malware Analysis". In: *Proceedings of the 2016 Network and Distributed System Security Symposium (NDSS'16)*. Internet Society, 2016. URL: https://www.ndss-symposium.org/wp-content/uploads/2017/09/lo-phi-low-observable-physical-host-instrumentation-malware-analysis.pdf.

[112] Satoru Takekoshi, Takahiro Shinagawa, and Kazuhiko Kato. "Testing Device Drivers Against Hardware Failures in Real Environments". In: *Proceedings of the 31st Annual ACM Symposium on Applied Computing (SAC'16)*. Association for Computing Machinery, 2016, pp. 1858–1864. DOI: 10.1145/2851613.2851740.

[113] Boris Teabe, Alain Tchana, and Daniel Hagimont. "Application-specific Quantum for Multi-core Platform Scheduler". In: *Proceedings of the 11th European Conference on Computer Systems (EuroSys'16)*. Association for Computing Machinery, 2016. DOI: 10.1145/2901318.2901340. URL: http://doi.acm.org/10.1145/2901318.2901340.

[114] Jason Wang and Peter Xu. "Vhost and VIOMMU". In: *KVM Forum 2016*. 2016.

[115] Fengwei Zhang and Hongwei Zhang. "SoK: A Study of Using Hardware-assisted Isolated Execution Environments for Security". In: *Proceedings of the 2016 Hardware and Architectural Support for Security and Privacy (HASP'16)*. ACM, 2016, pp. 1–8. DOI: 10.1145/2948618.2948621.

[116] Eric Auger. "vIOMMU/ARM: Full Emulation and virtio-iommu Approaches". In: *KVM Forum 2017*. 2017.

[117] Edouard Bugnion, Jason Nieh, and Dan Tsafrir. "Hardware and Software Support for Virtualization". In: *Synthesis Lectures on Computer Architecture* 12.1 (2017). DOI: 10.2200/S00754ED1V01Y201701CAC038. URL: https://www.morganclaypool.com/doi/abs/10.2200/S00754ED1V01Y201701CAC038.

[118] Ronny Chevalier, Maugan Villatel, David Plaquin, and Guillaume Hiet. "Co-processor-based Behavior Monitoring: Application to the Detection of Attacks Against the System Management Mode". In: *Proceedings of the 33rd Annual Computer Security Applications Conference (ACSAC'17)*. ACM, 2017, pp. 399–411. DOI: 10.1145/3134600.3134622.

[119] Takaaki Fukai, Satoru Takekoshi, Kohei Azuma, Takahiro Shinagawa, and Kazuhiko Kato. "BMCArmor: A Hardware Protection Scheme for Bare-Metal Clouds". In: *Proceedings of the 9th IEEE International Conference on Cloud Computing Technology and Science (CloudCom'17)*. Institute of Electrical and Electronics Engineers, 2017, pp. 322–330. DOI: 10.1109/CloudCom.2017.43.

[120] Zecheng He, Tianwei Zhang, and Ruby B. Lee. "Machine Learning Based DDoS Attack Detection from Source Side in Cloud". In: *Proceedings of the 4th International Conference on Cyber Security and Cloud Computing (CSCloud'17)*. Institute of Electrical and Electronics Engineers, 2017, pp. 114–120. DOI: 10.1109/CSCloud.2017.58.

[121] Manabu Hirano, Takuma Tsuzuki, Seishiro Ikeda, Naoga Taka, Kenji Fujiwara, and Ryotaro Kobayashi. "WaybackVisor: Hypervisor-Based Scalable Live Forensic Architecture for Timeline Analysis". In: *Proceedings of 7th International Symposium on Trust, Security and Privacy for Emerging Applications (TSP'17)*. Springer, 2017, pp. 219–230. DOI: 10.1007/978-3-319-72395-2_21.

[122] Jaeseong Im, Jongyul Kim, Jonguk Kim, Seongwook Jin, and Seungryoul Maeng. "On-Demand Virtualization for Live Migration in Bare Metal Cloud". In: *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*. Association for Computing Machinery, 2017, pp. 378–389. DOI: 10.1145/3127479.3129254.

[123] Filipe Manco, Costin Lupu, Florian Schmidt, Jose Mendes, Simon Kuenzer, Sumit Sati, Kenichi Yasukata, Costin Raiciu, and Felipe Huici. "My VM is Lighter (and Safer) Than Your Container". In: *Proceedings of the 26th Symposium on Operating Systems Principles (SOSP'17)*. Association for Computing Machinery, 2017. DOI: 10.1145/3132747.3132763.

[124] Aravinda Prasad, K Gopinath, and Paul E. McKenney. "The RCU-Reader Preemption Problem in VMs". In: *Proceedings of the 2017 USENIX Annual Technical Conference (ATC'17)*. USENIX Association, 2017, pp. 265–270. URL: https://www.usenix.org/conference/atc17/technical-sessions/presentation/prasad.

[125] Zhengwei Qi, Chengcheng Xiang, Ruhui Ma, Jian Li, Haibing Guan, and David S. L. Wei. "ForenVisor: A Tool for Acquiring and Preserving Reliable Data in Cloud Live Forensics". In: *IEEE Transactions on Cloud Computing* 5.3 (2017), pp. 443–456. DOI: 10.1109/TCC.2016.2535295.

[126] Kun Suo, Yong Zhao, Jia Rao, Luwei Cheng, Xiaobo Zhou, and Francis C. M. Lau. "Preserving I/O Prioritization in Virtualized OSes". In: *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*. Association for Computing Machinery, 2017. DOI: 10.1145/3127479.3127484.

[127] Boris Teabe, Vlad Nitu, Alain Tchana, and Daniel Hagimont. "The Lock Holder and the Lock Waiter Pre-emption Problems: Nip Them in the Bud Using Informed Spinlocks (I-Spinlock)". In: *Proceedings of the 12th European Conference on Computer Systems (EuroSys'17)*. ACM, 2017, pp. 286–297. DOI: 10.1145/3064176.3064180.

[128] G. Wang, L. Zhang, and W. Xu. "What Can We Learn from Four Years of Data Center Hardware Failures?" In: *2017 47th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN'17)*. Institute of Electrical and Electronics Engineers, 2017, pp. 25–36. DOI: 10.1109/DSN.2017.26.

[129] Kenichi Yasukata, Felipe Huici, Vincenzo Maffione, Giuseppe Lettieri, and Michio Honda. "HyperNF: Building a High Performance, High Utilization and Fair NFV Platform". In: *Proceedings of the 2017 Symposium on Cloud Computing (SoCC'17)*. Association for Computing Machinery, 2017. DOI: 10.1145/3127479.3127489.

[130] Jeongseob Ahn, Chang Hyun Park, Taekyung Heo, and Jaehyuk Huh. "Accelerating Critical OS Services in Virtualized Systems with Flexible Micro-sliced Cores". In: *Proceedings of the 13th European Computing Systems Conference (EuroSys'18)*. Association for Computing Machinery, 2018. DOI: 10.1145/3190508.3190521.

[131] Justinien Bouron, Sebastien Chevalley, Baptiste Lepers, Willy Zwaenepoel, Redha Gouicem, Julia Lawall, Gilles Muller, and Julien Sopena. "The Battle of the Schedulers: FreeBSD ULE vs. Linux CFS". en. In: *Proceedings of the 2018 USENIX Annual Technical Conference (ATC'18)*. USENIX Association, 2018, pp. 85–96. URL: https://www.usenix.org/conference/atc18/presentation/bouron.

[132] Guilherme Cox, Zi Yan, Abhishek Bhattacharjee, and Vinod Ganapathy. "Secure, Consistent, and High-Performance Memory Snapshotting". In: *Proceedings of the 8th ACM Conference on Data and Application Security and Privacy (CODASPY'18)*. ACM, 2018, pp. 236–247. DOI: 10.1145/3176258.3176325.

[133] Weiwei Jia, Cheng Wang, Xusheng Chen, Jianchen Shan, Xiaowei Shang, Heming Cui, Xiaoning Ding, Luwei Cheng, Francis C. M. Lau, Yuexuan Wang, and Yuangang Wang. "Effectively Mitigating I/O Inactivity in vCPU Scheduling". In: *Proceedings of the 2018 USENIX Annual Technical Conference (ATC'18)*. USENIX, 2018. URL: https://www.usenix.org/conference/atc18/presentation/jia.

[134] Sanidhya Kashyap, Changwoo Min, and Taesoo Kim. "Scaling Guest OS Critical Sections with eCS". In: *Proceedings of the 2018 USENIX Annual Technical Conference (ATC'18)*. USENIX Association, 2018, pp. 159–172. URL: https://www.usenix.org/conference/atc18/presentation/kashyap.

[135] O. Kilic, S. Doddamani, A. Bhat, H. Bagdi, and K. Gopalan. "Overcoming Virtualization Overheads for Large-vCPU Virtual Machines". In: *Proceedings of the IEEE 26th International Symposium on Modeling, Analysis, and Simulation of Computer and Telecommunication Systems (MASCOTS'18)*. Institute of Electrical and Electronics Engineers, 2018. DOI: 10.1109/MASCOTS.2018.00042.

[136] Alex Markuze, Igor Smolyar, Adam Morrison, and Dan Tsafrir. "DAMN: Overhead–Free IOMMU Protection for Networking". In: *Proceedings of the 23rd International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'18)*. ACM, 2018, pp. 301–315. DOI: 10.1145/3173162.3173175.

[137] Masanori Misono, Masahiro Ogino, Takaaki Fukai, and Takahiro Shinagawa. "FaultVisor2: Testing Hypervisor Device Drivers Against Real Hardware Failures". In: *Proceedings of the 10th International Conference on Cloud Computing Technology and Science (CloudCom'18)*. Institute of Electrical and Electronics Engineers, 2018, pp. 204–211. DOI: 10.1109/CloudCom2018.2018.00048.

[138] Masanori Misono, Kaito Yoshida, Juho Hwang, and Takahiro Shinagawa. "Distributed Denial of Service Attack Prevention at Source Machines". In: *Proceedings of the 16th International Conference on Dependable, Autonomic and Secure Computing (DASC'18)*. Institute of Electrical and Electronics Engineers, 2018, pp. 488–495. DOI: 10.1109/DASC/PiCom/DataCom/CyberSciTec.2018.00096.

[139] Benoît Morgan, Éric Alata, Vincent Nicomette, and Mohamed Kaâniche. "IOMMU Protection against I/O Attacks: a Vulnerability and a Proof of Concept". In: *Journal of the Brazilian Computer Society* 24.1 (2018). DOI: 10.1186/s13173-017-0066-7.

[140] Iori Yoneji, Takaaki Fukai, Takahiro Shinagawa, and Kazuhiko Kato. "Unified Hardware Abstraction Layer with Device Masquerade". In: *Proceedings of the 33rd Annual ACM Symposium on Applied Computing (SAC'18)*. New York, NY, USA: Association for Computing Machinery, 2018, pp. 1102–1108. DOI: 10.1145/3167132.3167250.

[141] Fengwei Zhang, Kevin Leach, Angelos Stavrou, and Haining Wang. "Towards Transparent Debugging". In: *IEEE Transactions on Dependable and Secure Computing* 15.2 (2018), pp. 321–335. DOI: 10.1109/TDSC.2016.2545671.

[142] Ahmad Atamli, Giuseppe Petracca, and Jon Crowcroft. "IO-Trust: An Out-of-band Trusted Memory Acquisition for Intrusion Detection and Forensics Investigations in Cloud IOMMU Based Systems". In: *Proceedings of the 14th International Conference on Availability, Reliability and Security (ARES'19)*. ACM, 2019, 45:1–45:6. DOI: 10.1145/3339252.3340511.

[143] Bao Bui, Djob Mvondo, Boris Teabe, Kevin Jiokeng, Lavoisier Wapet, Alain Tchana, Gaël Thomas, Daniel Hagimont, Gilles Muller, and Noel DePalma. "When eXtended Para - Virtualization (XPV) Meets NUMA". In: *Proceedings of the 14th European Conference on Computer Systems (EuroSys'19)*. Association for Computing Machinery, 2019, pp. 1–15. DOI: 10.1145/3302424.3303960.

[144] Michael Kiperberg, Roee Leon, Amit Resh, Asaf Algawi, and Nezer Zaidenberg. "Hypervisor-assisted Atomic Memory Acquisition in Modern Systems". In: *Proceedings of the 5th International Conference on Information Systems Security and Privacy (ICISSP'19)*. SCITEPRESS Science and Technology Publications, 2019. DOI: 10.5220/0007566101550162.

[145] Hao Li, Xuefei Xu, Jinkui Ren, and Yaozu Dong. "ACRN: A Big Little Hypervisor for IoT Development". In: *Proceedings of the 15th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments (VEE'19)*. Association for Computing Machinery, 2019, pp. 31–44. DOI: 10.1145/3313808.3313816.

[146] A. Theodore Markettos, Colin Rothwell, Brett F. Gutstein, Allison Pearce, Peter G. Neumann, Simon W. Moore, and Robert N. M. Watson. "Thunderclap: Exploring Vulnerabilities in Operating System IOMMU Protection via DMA from Untrustworthy Peripherals". In: *Proceedings of the 2019 Network and Distributed Systems Security Symposium (NDSS'19)*. Internet Society, 2019. URL: https://www.ndss-symposium.org/ndss-paper/thunderclap-exploring-vulnerabilities-in-operating-system-iommu-protection-via-dma-from-untrustworthy-peripherals/.

[147] Lei Zhou, Jidong Xiao, Kevin Leach, Westley Weimer, Fengwei Zhang, and Guojun Wang. "Nighthawk: Transparent System Introspection from Ring -3". In: *Proceedings of the 24th European Symposium on Research in Computer Security (ESORICS'19)*. Springer, 2019, pp. 217–238. DOI: 10.1007/978-3-030-29962-0_11.

[148] Jingde Chen, Subho S. Banerjee, Zbigniew T. Kalbarczyk, and Ravishankar K. Iyer. "Machine Learning for Load Balancing in the Linux Kernel". In: *Proceedings of the 11th ACM SIGOPS Asia-Pacific Workshop on Systems (APSys'20)*. Association for Computing Machinery, 2020, pp. 67–74. DOI: 10.1145/3409963.3410492.

[149] Patrick Jauernig, Ahmad-Reza Sadeghi, and Emmanuel Stapf. "Trusted Execution Environments: Properties, Applications, and Challenges". In: *IEEE Security Privacy* 18.2 (2020), pp. 56–60. DOI: 10.1109/MSEC.2019.2947124.

[150] Tobias Latzo, Julian Brost, and Felix Freiling. "BMCLeech: Introducing Stealthy Memory Forensics to BMC". In: *Forensic Science International: Digital Investigation* 32 (2020), p. 300919. DOI: 10.1016/j.fsidi.2020.300919.

[151] Baptiste Lepers, Redha Gouicem, Damien Carver, Jean-Pierre Lozi, Nicolas Palix, Maria-Virginia Aponte, Willy Zwaenepoel, Julien Sopena, Julia Lawall, and Gilles Muller. "Provable Multicore Schedulers with Ipanema: Application to Work Conservation". In: *Proceedings of the 15th European Conference on Computer Systems (EuroSys'20)*. Association for Computing Machinery, 2020. DOI: 10.1145/3342195.3387544.

[152] Suravee Suthikulpanit and Wei Huang. "AMD-vIOMMU: A Hardware-assisted Virtual IOMMU Technology". In: *KVM Forum 2020*. 2020.

[153] Kun Tian, Yu Zhang, Luwei Kang, Yan Zhao, and Yaozu Dong. "coIOMMU: A Virtual IOMMU with Cooperative DMA Buffer Tracking for Efficient Memory Management in Direct I/O". In: *Proceedings of the 2020 USENIX Annual Technical Conference (ATC'20)*. USENIX Association, 2020. URL: https://www.usenix.org/conference/atc20/presentation/tian.

[154] Nezer Jacob Zaidenberg, Michael Kiperberg, Raz Ben Yehuda, Roee Leon, Asaf Algawi, and Amit Resh. "Hypervisor Memory Introspection and Hypervisor Based Malware Honeypot". In: *Proceedings of the 5th Information Systems Security and Privacy (ICISSP'20)*. Springer, 2020, pp. 317–334. DOI: 10.1007/978-3-030-49443-8_15.

[155] Runhua Zhang, Alan L. Cox, and Scott Rixner. "Virtflex: Automatic Adaptation to NUMA Topology Change for OpenMP Applications". en. In: *Proceedings of 16th International Workshop on OpenMP*. Springer, 2020, pp. 212–227. DOI: 10.1007/978-3-030-58144-2_14.

[156] Markuze Alex, Shay Vargaftik, Gil Kupfer, Boris Pismeny, Nadav Amit, Adam Morrison, and Dan Tsafrir. "Characterizing, Exploiting, and Detecting DMA Code Injection Vulnerabilities in the Presence of an IOMMU". In: *Proceedings of the 16th European Conference on Computer Systems (EuroSys'21)*. ACM, 2021, pp. 395–409. DOI: 10.1145/3447786.3456249.

[157] Shih-Wei Li, Xupeng Li, Ronghui Gu, Jason Nieh, and John Zhuang Hui. "Formally Verified Memory Protection for a Commodity Multiprocessor Hypervisor". In: *Proceedings of the 30th USENIX Security Symposium (SEC'21)*. USENIX Association, 2021. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/li-shih-wei.

[158] Shih-Wei Li, Xupeng Li, John Zhuang Huiand Jason Nieh, and Ronghui Gu. "A Secure and Formally Verified Linux KVM Hypervisor". In: *Proceedings of the 42nd IEEE Symposium on Security and Privacy (S&P'21)*. Institute of Electrical and Electronics Engineers, 2021. DOI: 10.1109/SP40001.2021.00049.

[159] Masanori Misono and Takahiro Shinagawa. "POSTER: OS Independent Fuzz Testing of I/O Boundary". In: *Proceedings of the 2021 ACM Conference on Computer and Communications Security (CCS'21)*. Association for Computing Machinery, 2021. DOI: 10.1145/3460120.3485359.

[160] Raz Ben Yehuda, Erez Shlingbaum, Yuval Gershfeld, Shaked Tayouri, and Nezer Jacob Zaidenberg. "Hypervisor Memory Acquisition for ARM". In: *Forensic Science International: Digital Investigation* 37 (2021), p. 301106. DOI: 10.1016/j.fsidi.2020.301106.

[161] Masanori Misono, Toshiki Hatanaka, and Takahiro Shinagawa. "DMAFV: Testing Device Drivers against DMA Faults". In: *Proceedings of the 37th ACM/SIGAPP Symposium On Applied Computing (SAC 2022)*. Association for Computing Machinery, 2022. DOI: 10.1145/3477314.3507082.

[162] Thomas Van Strydonck, Aïna Linn Georges, Armaël Guéneau, Alix Trieu, Amin Timany, Frank Piessens, Lars Birkedal, and Dominique Devriese. "Proving Full-System Security Properties Under Multiple Attacker Models on Capability Machines". In: *Proceedings of the 35th IEEE Computer Security Foundations Symposium (CSF'22)*. Institute of Electrical and Electronics Engineers, 2022.

[163] *Amazon EC2 Instance Types*. URL: https://aws.amazon.com/ec2/instance-types/ (visited on 11/13/2021).

[164] Amazon Web Service. *Introduction to AWS security processes revision June 2016*. URL: https://d0.awsstatic.com/whitepapers/Security/Intro_Security_Practices.pdf (visited on 11/13/2021).

[165] *AMD I/O Virtualization Technology (IOMMU) Specification*. URL: https://www.amd.com/en/support/tech-docs/amd-io-virtualization-technology-iommu-specification (visited on 11/13/2021).

[166] *AMD Secure Encrypted Virtualization (SEV)*. URL: https://developer.amd.com/sev/ (visited on 11/13/2021).

[167] ARM. *TrustZone*. URL: https://www.arm.com/products/security-on-arm/trustzone (visited on 11/13/2021).

[168] *Arm Confidential Compute Architecture*. URL: https://www.arm.com/why-arm/architecture/security-features/arm-confidential-compute-architecture (visited on 11/13/2021).

[169] *Arm System Memory Management Unit Architecture Specification, SMMU Architecture Version 3*. URL: https://developer.arm.com/documentation/ihi0070/latest (visited on 11/13/2021).

[170] *Arm TrustZone Technology*. URL: https://developer.arm.com/ip-products/security-ip/trustzone (visited on 11/13/2021).

[171] Gal Beniamini. *Over The Air - Vol. 2, Pt. 1: Exploiting The Wi-Fi Stack on Apple Devices*. URL: https://googleprojectzero.blogspot.com/2017/09/over-air-vol-2-pt-1-exploiting-wi-fi.html (visited on 11/13/2021).

[172] *CFS Scheduler - The Linux Kernel documentaion*. URL: https://www.kernel.org/doc/html/latest/scheduler/sched-design-CFS.html (visited on 11/13/2021).

[173] *Cgroups - The Linux Kernel documentaion*. URL: https://www.kernel.org/doc/Documentation/cgroup-v1/cgroups.txt (visited on 11/13/2021).

[174] Docker. *Empowering App Development for Developers — Docker*. URL: https://www.docker.com/ (visited on 03/02/2022).

[175] *Features/VT-d – QEMU*. URL: https://wiki.qemu.org/Features/VT-d (visited on 11/13/2021).

[176] *ftrace – Function Tracer*. URL: https://www.kernel.org/doc/Documentation/trace/ftrace.txt (visited on 11/13/2021).

[177] *futex(2) – Linux manual page*. URL: https://man7.org/linux/man-pages/man2/futex.2.html (visited on 11/13/2021).

[178] HewlettPackard. *Netperf*. URL: https://github.com/HewlettPackard/netperf (visited on 11/13/2021).

[179] IEEE. *IEEE Copyright Policy*. URL: https://www.ieee.org/publications/rights/copyright-policy.html (visited on 11/13/2021).

[180] igel. *vThrii Seamless Provisioning*. URL: https://www.igel.co.jp/en/solution/ (visited on 11/13/2021).

[181] *Intel 64 and IA-32 Architectures Software Developer's Manual Volume 3 System Management Mode*. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-sdm.html (visited on 11/13/2021).

[182] *Intel Architecture Memory Encryption Technologies Specification Revision 1.3*. URL: https://software.intel.com/content/dam/develop/external/us/en/documents-tps/multi-key-total-memory-encryption-spec.pdf (visited on 11/13/2021).

[183] *Intel Software Guard Extensions*. URL: https://software.intel.com/content/www/us/en/develop/topics/software-guard-extensions.html (visited on 11/13/2021).

[184] *Intel Trust Domain Extensions (Intel TDX)*. URL: https://software.intel.com/content/www/us/en/develop/articles/intel-trust-domain-extensions.html (visited on 11/13/2021).

[185] *Intel Virtualization Technology for Directed I/O Architecture Specification*. URL: https://software.intel.com/content/www/us/en/develop/download/intel-virtualization-technology-for-directed-io-architecture-specification.html (visited on 11/13/2021).

[186] IO Visor. *BCC BPF compiler collection*. URL: https://www.iovisor.org/technology/bcc (visited on 11/13/2021).

[187] IO Visor. *iovisor/ubpf: Userspace eBPF VM*. URL: https://github.com/iovisor/ubpf (visited on 11/13/2021).

[188] *iommu/vt-d: Do not Enable ATS for Untrusted Devices*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/torvalds/linux.git/commit/?id=fb58fdcd295b914ece1d829b24df00a17a9624bc (visited on 11/13/2021).

[189] J. Axboe. *fio*. URL: https://github.com/axboe/fio (visited on 11/13/2021).

[190] *JSON-RPC 2.0 Specification*. URL: https://www.jsonrpc.org/specification (visited on 11/13/2021).

[191] *kernel/git/mason/schbench.git*. URL: https://git.kernel.org/pub/scm/linux/kernel/git/mason/schbench.git/ (visited on 11/13/2021).

[192] Alexander Khalimonenko, Oleg Kupreev, and Kirill Ilganaev. *DDoS Atacks in Q3 2017*. URL: https://securelist.com/ddos-attacks-in-q3-2017/83041/ (visited on 11/13/2021).

[193] Andy Klein. *Backblaze Hard Drive Stats for 2017*. URL: https://www.backblaze.com/blog/hard-drive-stats-for-2017/ (visited on 11/13/2021).

[194] *libvirt: Network filters*. URL: https://libvirt.org/formatnwfilter.html (visited on 11/13/2021).

[195] *libvirt: The virtualization API*. URL: https://libvirt.org/ (visited on 11/13/2021).

[196] Linux Kernel Documentation. *Fault Injection Capabilities Infrastructure*. URL: https://www.kernel.org/doc/Documentation/fault-injection/fault-injection.txt (visited on 11/13/2021).

[197] Linux Kernel Documentation. *VFIO - "Virtual Function I/O"*. URL: https://www.kernel.org/doc/Documentation/vfio.txt (visited on 11/13/2021).

[198] *Linux socket filtering aka Berkeley packet filter (BPF)*. URL: https://www.kernel.org/doc/Documentation/networking/filter.txt (visited on 11/13/2021).

[199] *lwIP - a lightweight TCP/IP stack - summary*. URL: http://savannah.nongnu.org/projects/lwip (visited on 11/13/2021).

[200] Microsoft. *Driver Verifier*. URL: https://docs.microsoft.com/en-us/windows-hardware/drivers/devtest/driver-verifier (visited on 11/13/2021).

[201] *NAS Parallel Benchmarks*. URL: https://www.nas.nasa.gov/publications/npb.html (visited on 11/13/2021).

[202] *netfilter*. URL: http://www.netfilter.org/ (visited on 11/13/2021).

[203] *numactl/numactl: NUMA support for Linux*. URL: https://github.com/numactl/numactl (visited on 11/13/2021).

[204] *OMP_WAIT_POLICY*. URL: https://www.openmp.org/spec-html/5.1/openmpse64.html (visited on 11/13/2021).

[205] Open Networking Foundation. *OpenFlow*. URL: https://www.opennetworking.org/sdn-resources/openflow (visited on 11/13/2021).

[206] *Open vSwitch*. URL: http://openvswitch.org (visited on 11/13/2021).

[207] Oracle. *Java*. URL: https://www.java.com/en/ (visited on 03/02/2022).

[208] *Paravirtualized KVM features*. URL: https://www.qemu.org/docs/master/system/i386/kvm-pv.html (visited on 03/02/2022).

[209] *PCIe Screamer R02*. URL: https://docs.lambdaconcept.com/screamer/older_versions.html (visited on 11/13/2021).

[210] *perf-bench(1) – Linux manual page*. URL: https://man7.org/linux/man-pages/man1/perf-bench.1.html (visited on 11/13/2021).

[211] *Platform Runtime Mechanism Specification Version: 1.0*. URL: https://uefi.org/sites/default/files/resources/Platform%5C%20Runtime%5C%20Mechanism%5C%20-%5C%20with%5C%20legal%5C%20notice.pdf (visited on 11/13/2021).

[212] *QEMU*. URL: https://www.qemu.org/ (visited on 11/13/2021).

[213] K. T. Raghavendra. *Paravirtualized ticket spinlocks*. URL: https://lwn.net/Articles/552696/ (visited on 11/13/2021).

[214] Björn Ruytenberg. *Breaking Thunderbolt Protocol Security: Vulnerability Report*. URL: https://thunderspy.io/assets/reports/breaking-thunderbolt-security-bjorn-ruytenberg-20200417.pdf (visited on 11/13/2021).

[215] *Scheduler Domains - The Linux Kernel documentaion*. URL: https://www.kernel.org/doc/Documentation/scheduler/sched-domains.txt (visited on 11/13/2021).

[216] *SCREAMER M.2 USB-C (R04)*. URL: https://shop.lambdaconcept.com/home/43-screamer-m2.html (visited on 11/13/2021).

[217] *siemens/jailhouse: Linux-based partitioning hypervisor*. URL: https://github.com/siemens/jailhouse (visited on 05/06/2021).

[218] *SMI Transfer Monitor (STM)*. URL: https://software.intel.com/content/www/us/en/develop/articles/smi-transfer-monitor-stm.html (visited on 11/13/2021).

[219] *src/southbridge/intel/common/finalize.c – coreboot – Gitile*. URL: https://review.coreboot.org/plugins/gitiles/coreboot/+/refs/heads/4.12_branch/src/southbridge/intel/common/finalize.c (visited on 11/13/2021).

[220] Standard Performance Evaluation Corporation. *SPEC CPU2017*. URL: https://www.spec.org/cpu2017/ (visited on 11/13/2021).

[221] Alexei Starovoitov. *[RFC,net-next,08/14] bpf: add eBPF verifier*. URL: https://patchwork.kernel.org/patch/4438881/ (visited on 11/13/2021).

[222] *TCPDUMP&LIBPCAP public repository*. URL: http://www.tcpdump.org (visited on 11/13/2021).

[223] The Apache Software Foundation. *ab - Apache HTTP server benchmarking tool*. URL: https://httpd.apache.org/docs/2.4/programs/ab.html (visited on 11/13/2021).

[224] *The Kernel's Command-line Parameters*. URL: https://www.kernel.org/doc/html/v5.13/admin-guide/kernel-parameters.html (visited on 11/13/2021).

[225] *The PARSEC Benchmark Suite*. URL: https://parsec.cs.princeton.edu/ (visited on 11/13/2021).

[226] *ufrisk/pcileech: Direct Memory Access (DMA) Attack Software*. URL: https://github.com/ufrisk/pcileech (visited on 11/13/2021).

[227] VMWare. *VMware NSX*. URL: http://www.vmware.com/products/nsx.html (visited on 11/13/2021).

[228] VMWare. *vSphere Hypervisor*. URL: https://www.vmware.com/products/vsphere-hypervisor.html (visited on 11/13/2021).

[229] *What is Intel Management Engine?* URL: https://www.intel.com/content/www/us/en/support/articles/000008927/software/chipset-software.html (visited on 11/13/2021).