國立臺灣大學電機資訊學院資訊工程學系

碩士論文

Department of Computer Science and Information Engineering

College of Electrical Engineering and Computer Science

National Taiwan University

Master's Thesis

提升 KVM 效能監測單位虛擬化之效能

Enhancing Performance of KVM Performance Monitoring Unit Virtualization

陳建豪

Jian-Hao Chen

指導教授: 黎士瑋 博士

Advisor: Shih-Wei Li, Ph.D.

中華民國 113 年 8 月

August 2024

doi:10.6342/NTU202403471

國立臺灣大學碩士學位論文
口試委員會審定書
MASTER'S THESIS ACCEPTANCE CERTIFICATE
NATIONAL TAIWAN UNIVERSITY

提升 KVM 效能監測單位虛擬化之效能

Enhancing Performance of KVM Performance Monitoring Unit Virtualization

本論文係陳建豪君（學號 R11922014）在國立臺灣大學資訊工程學系完成之碩士學位論文，於民國 113 年 7 月 8 日承下列考試委員審查通過及口試及格，特此證明。

The undersigned, appointed by the Department of Computer Science and Information Engineering on 8 July 2024 have examined a Master's thesis entitled above presented by CHEN, JIAN-HAO (student ID: R11922014) candidate and hereby certify that it is worthy of acceptance.

口試委員 Oral examination committee:

黎士瑋

（指導教授 Advisor）

洪士灝 洪鼎詠

系主任/所長 Director: 陳祝嵩

致謝

謝謝黎士瑋教授這兩年來的指點，尤其是在最後的論文撰寫階段。儘管我不善於把成果轉化為有力的文字，黎士瑋教授仍然致力於協助我完成這篇論文。除了黎士瑋教授之外，也感謝撥冗出席口試的洪士灝教授和洪鼎詠助研究員，謝謝兩位口試委員在口試過程中的指點以及祝福。此外，也要感謝實驗室的同學還有其他朋友在研究過程中的陪伴，我想如果沒有你們的話研究的過程就會變得很孤單，而我自己一個人或許就沒辦法好好撐完這段路。最後，我也要感謝我的家人讓我能夠無後顧之憂的完成碩士學位。

陳建豪

中華民國一百一十三年七月

doi:10.6342/NTU202403471

摘要

當今的處理器會透過特殊的性能分析硬體將架構相關的事件的資訊提供給使用者。具體而言，大部分的處理器架構都有提供效能監測單位 (PMU) 來記錄並且回報硬體事件的資訊。此外，有些供應商亦會在處理器上提供其他專門的性能分析功能，例如 Intel 處理器提供的最後分支紀錄 (LBR) 能夠提供最近執行的分支指令的詳細資訊。軟體開發者可以利用這些處理器提供的功能來對自己的程式做性能分析，並且根據分析的結果來優化程式。由於開發者越來越傾向於把程式部署於雲端服務所提供的虛擬機器，主流的虛擬機器監測器會虛擬化這些性能分析硬體，並且將虛擬的性能分析硬體提供給虛擬機器使用。然而，目前針對性能分析硬體的虛擬化會導致虛擬機器頻繁的退出到虛擬機器監測器，這也大幅提升了在虛擬機器中使用性能分析硬體的開銷。因此，這篇論文旨在優化 Intel 處理器的性能分析硬體的虛擬化的性能。我們在確保性能分析硬體被正確地多工複用的前提下虛擬機器直接存取性能分析硬體。此外，我們也把性能分析硬體產生的中斷直接傳遞給虛擬機器以降低頻繁取樣下造成的陷入。我們對我們在 Linux KVM 虛擬機器監測器上的實作原型進行評估，結果顯示我們能夠顯著地提高性能分析硬體如 PMU 的虛擬化效能。

關鍵字：性能分析、作業系統、虛擬化、KVM

doi:10.6342/NTU202403471

Abstract

Modern processors expose architectural events to users via special profiling features. Specifically, most architectures provide a performance monitoring unit (PMU) to record and report hardware events. Besides, vendors like Intel support customized features like Last Branch Record (LBR), which profiles the execution of branch instructions. Developers utilize these features to profile their programs' execution and analyze performance. As programs are increasingly deployed to virtual machines (VMs) running on the cloud, commodity hypervisors expose virtual profiling hardware to VMs. However, the current virtualization support incurs frequent VM exits to the hypervisor, causing significant overhead to VMs using the profiling hardware. This thesis aims to optimize the virtualization performance of hardware profiling features on Intel processors. We multiplex the profiling hardware to safely enable direct access from the VM. Further, we pass through interrupts generated by the profiling hardware to VMs to avoid traps caused by frequent samplings. Evaluation of our prototype for the Linux KVM hypervisor shows that our

doi:10.6342/NTU202403471

approach significantly improved the virtualization performance of PMU.

Keywords: Profiling, Operating Systems, Virtualization, KVM

doi:10.6342/NTU202403471

Contents

doi:10.6342/NTU202403471

doi:10.6342/NTU202403471

doi:10.6342/NTU202403471

List of Figures

doi:10.6342/NTU202403471

List of Tables

doi:10.6342/NTU202403471

Chapter 1 Introduction

Performance is a significant concern in software development. For instance, a web developer aims to minimize server latency to enhance user experience. To optimize a program's performance, developers often profile it to analyze its execution. Modern processors feature profiling hardware to log and report architectural events during execution. Most architectures provide a Performance Monitoring Unit (PMU) [2, 8]. Further, hardware vendors like Intel support customized profiling features, such as the Last Branch Record (LBR) [10]. A developer can utilize the profiling hardware to acquire information about software execution, such as the occurrence of last-level misses and branch misprediction or the most recently executed branch instructions on the processor. Previous work [14] leveraged architectural events exposed by the profiling hardware to recognize program bottlenecks. Utility software such as the Linux perf tool [11] and Intel Vtune [7] has been introduced to assist users in configuring the profiling hardware and retrieving the architectural profiling information.

There has been a growing shift of software deployments from in-house servers to virtual machines (VMs) running on the cloud. To support profiling in the cloud VM environment, commodity hypervisors like Linux KVM [9] were extended to provide virtual profiling hardware to VMs. Nevertheless, the current virtualization support incurs frequent VM exits to the hypervisor, causing significant overhead to VMs using the profiling

doi:10.6342/NTU202403471

hardware. For example, KVM employs trap-and-emulate to virtualize VM access to the PMU and traps performance monitoring interrupts (PMI) to the hypervisor. Additionally, KVM implements complex operations to emulate PMU accesses and handle PMIs to ensure the physical resources of PMU are properly shared by every process on the host and every virtual machine. This can drastically slow down the VM's program to be profiled and potentially lead to unstable or even inaccurate profiling results.

Previous works [5, 15] have attempted to optimize the virtualization performance of PMU by granting VM direct access to the hardware. This allows VMs to configure the PMU and retrieve profiling event states without trapping such accesses to the hypervisor. However, these works did not optimize traps resulting from PMIs, the interrupts generated by the profiling hardware. The hardware sends PMIs to notify the processor when an event counter overflows to support interrupt-based sampling. PMIs are crucial because the hardware will reset the counter to zero after the overflow. Upon receiving the PMI, the software retrieves counters from the hardware and reset their values. Shortening the delivery latency of PMIs leads to better accuracy in profiling results within a fixed time. We found that PMI handling in KVM causes significant delivery latency to VMs, resulting in poor profiling accuracy.

This thesis aims to optimize the virtualization performance of hardware profiling features on KVM for Intel processors due to their wide adoption. Specifically, we focus on reducing the virtualization overhead of PMU accesses and PMI while retaining functionality and security. To optimize PMU virtualization, instead of employing trap-and-emulate, we leverage a similar approach to previous work [5, 15] and allow VMs to directly access the PMU. This eliminates most VM exits when profiling a program in VMs. We extended KVM to multiplex the PMU states of VMs and the hypervisor host to ensure multiple

doi:10.6342/NTU202403471

entities can safely share and use the hardware. Further, we pass through PMIs to VMs to avoid traps caused by frequent samplings. This allows VMs to handle PMIs without hypervisor intervention. We ensure that PMIs are delivered to the intended entity.

We modified KVM in mainline Linux v6.9 to incorporate the proposed optimizations. Evaluating our KVM prototype shows that the passthrough optimization reduced the cost of micro-level operations that access the PMU by more than 20× and PMI delivery overhead by 8×. Additionally, we ran Linux perf tool in a VM to profile the performance of an Nginx server. We demonstrated that the optimized KVM significantly reduced the sampling overhead, retaining the performance of the profiling workloads.

In summary, this thesis makes the following contributions: (1) we identify the sources of profiling overhead in VMs in the current KVM implementation; the profiling overhead primarily stems from the cost of KVM emulating VM accesses to PMU and handling PMIs. (2) we introduce PMU and PMI passthrough to enhance virtualization performance while preserving functionality and security.

The rest of the thesis will be organized as follows. We first discuss background chapter 2, followed by design in chapter 3. Evaluation of our implementation will be discussed in chapter 4. Related work and future work will be discussed in chapter 5 and chapter 6 respectively. We conclude the thesis in chapter 7.

Chapter 2 Background

2.1 Intel VMX

Intel introduced Virtual Machine Extension (VMX) to support unmodified virtual machines. Intel VMX adds two orthogonal operation modes for CPU execution levels (i.e., ring 0, 1, 2, 3): VMX root and non-root. The hypervisor executes in VMX root operation to fully control the hardware and deprivileges VMs in VMX non-root operation, where sensitive instructions, e.g., I/O instructions and read (RDMSR) or write (WRMSR) to model-specific registers (MSRs) and events, such as external interrupts, cause the processor to trap to the hypervisor in VMX root operation, a transition known as a VM exit. The hypervisor handles VM exits and executes the VM entry instructions to resume the VM.

Intel VMX incorporates an in-memory data structure called the Virtual Machine Control Structure (VMCS). VMCS contains the CPU states of the hypervisor host and VM in the host and VM state area. When a hypervisor enters the VM, VMX saves the hypervisor's states in the host state area and restores the VM states to the hardware from the guest state area. Contrarily, VMX saves VM states to the guest state area on a VM exit and restores the host states from the VMCS to the hardware. Hypervisors associate a VMCS for each virtual CPU (vCPU). Hypervisors configure VMCS's VM control fields to control the VM's behaviors in the non-root operation. For instance, a hypervisor can grant a VM the privilege to handle specific operations, such as exceptions, I/O instructions, and external interrupts, without causing VM exits. Additionally, a hypervisor can configure a VMCS to automatically load or store registers during VM entry and exit.

2.2 Intel Performance Monitoring Unit

Intel PMU is a per-CPU-core module with several programmable MSRs and performance monitoring counters (PMCs). The latter counts the architectural events on the core. The PMU includes two MSRs: IA32_PERF_GLOBAL_CTRL (referred to as CTRL) and IA32_PERF_GLOBAL_STATUS (referred to as STATUS), respectively, for software to enable/disable and get the status of each PMC. A PMC also consists of two MSRs, IA32_A_PMC and IA32_PERFEVTSEL (referred to as EVTSEL). IA32_A_PMC includes the event count and a bit width value. A PMC overflow occurs when the event count value of IA32_A_PMC exceeds the bit width value. On the other hand, the software programs the EVTSEL to enable or disable the associated PMC and specify the event type and the privilege level (i.e., ring 0 or ring 1-3) where the event is to be counted.

2.3 Performance Monitoring Interrupt

The software programs EVTSEL to raise a performance monitoring interrupt when IA32_A_PMC overflows. Upon receiving the PMI, the software reads the counter value, resets the counter value and re-enables the counter after an overflow. The x86 processors deliver the PMI as a non-maskable interrupt (NMI). Therefore, PMIs cannot be masked by clearing the interrupt flag (IF) of the RFLAGS register; instead, the software can configure

doi:10.6342/NTU202403471

the local APIC to mask PMIs. If the hardware is configured to generate PMI, it sends a signal to the local APIC to inform that a PMI should be delivered. When local APIC receives the signal, it first checks whether the PMI is masked. If the PMI is not masked, the local APIC obtains the delivery mode of the PMI, masks the subsequent PMIs, and delivers the PMI to the processor.

2.4 Intel Last Branch Record

Intel Last Branch Record (LBR) is a per-CPU-core module that records recent branch instructions to a set of MSRs. The LBR consists of several entries; each entry logs the source and destination address of a taken branch instruction to MSR_LBR_FROM and MSR_LBR_TO. Current Intel processor implementations include no more than 32 LBR entries. When all entries are used, LBR overwrites the least recently used entry with the currently taken branch instruction information. This causes data loss if the software does not retrieve the LBR record before the replacement. To resolve the issue, the software can configure a PMC to generate a PMI before the hardware replaces an entry, allowing the software to back up LBR records prior to replacement.

2.4.1 Performance Counters for Linux

The Performance Counters for Linux (PCL) is a kernel subsystem for performance analysis in Linux. The PCL abstracts the underlying performance monitoring hardware (PMU in our case) to a performance monitoring event (the perf_event structure) and provides a complete framework that includes a perf_event_open() system call, a PMI handler, and supports context switching PMU hardware. In Linux, the software uses the profiling

doi:10.6342/NTU202403471

hardware via the PCL. If the software tries to access profiling hardware directly without assistance of PCL, not only PMI would not be correctly handled but also the states of PMU hardware would not be correctly saved and restored on context switch, resulting in erroneous profiling result.

2.5 PMU and LBR Virtualization in KVM

KVM uses trap-and-emulate to virtualize VMs' accesses to the PMU MSRs. It sets up the VMCS to trap VMs' MSR reads and writes to the profiling hardware to emulate the MSR accesses. The accesses are then redirected to virtual PMU, which consists of in-memory data structures recording the virtual states of PMU, e.g., the value of CTRL and EVTSEL. KVM emulates the intended operation of an MSR access by analyzing and matching its semantics. We list the recognized semantics and the associated emulation in KVM.

- Write to PMU MSRs: KVM updates the MSR value to the virtual PMU.
- Start/Stop PMC: This is a special case for writing CTRL and EVTSEL. When a VM writes CTRL or EVTSEL, KVM checks on the value of CTRL and EVTSEL. If the virtual PMU is not associated to a perf_event and the value of virtual CTRL and EVTSEL indicates that a PMC is enabled, KVM registers a perf_event to host PCL and associate the perf_event to the virtual PMU, which utilizes host PCL to (1) setup PMU MSRs according to virtual PMU (2) save and restore the states PMU MSR when the VM is context switched. Contrarily, if the virtual PMU is associated to a perf_event and the value of virtual CTRL and EVTSEL indicates that a PMC is disabled, KVM detach the perf_event from the virtual PMU and deregisters the

perf_event.

- Read from PMU MSRs: KVM retrieves the MSR value from the virtual PMU and returns the value to VMs.

- Read PMU counters: This is a special case for reading IA32_A_PMC. If the virtual PMU is associated to a perf_event, KVM retrieves the value from the perf_event and updates the virtual PMU. Otherwise, KVM simply retrieves the value from the virtual PMU.

The emulation incurs overhead to VMs' accesses to the PMU MSRs. Even worse, such MSR accesses occur frequently in PCL. Specifically, PCL reads and writes these PMU MSRs when

- registering/deregistering a performance monitoring event

- reading/resetting the counter value and re-enabling the counter in the PMI handler

- saving/restoring the state of PMU MSRs of a process

Consequently, profiling with PCL in VMs can be much slower than on bare metal.

PMI handling in KVM. A PMI causes VM exit when the processor is in VMX non-root operation, and KVM eventually invokes the PMI handler from the host PCL to handle the PMI. The handler invokes callbacks from KVM to update the virtual PMU. If KVM decides that the VM should be interrupted by the events from the virtual PMU hardware, it injects a virtual PMI into the VM. Eventually, the PMI handler from the guest PCL is invoked to update the states of perf_event in guest PCL. The whole PMI handling process

doi:10.6342/NTU202403471

in KVM involves the host PCL, guest PCL, and KVM, thus imposing significant overhead compared to bare metal.

2.6 Non-maskable Interrupt

An NMI is a type of interrupt that users cannot disable or ignore using the processor's interrupt masking mechanisms (e.g., set the interrupt disable flag). In addition to PMIs, the hardware could send NMIs to signal critical events requiring high-priority handling and trigger debugging and diagnosing.

NMIs can be generated by hardware. Some specialized hardware includes a watchdog timer that can be programmed to send NMIs to ensure the system remains alive and responsive. If the watchdog timer expires, the system is considered down and the hardware could take the configured actions such as rebooting the system. A typical example is the Intelligent Platform Management Interface (IPMI), a hardware protocol that defines interfaces for remote system management and monitoring independent of the monitored machine. The IPMI watchdog timer includes a pre-timeout interrupt requesting the monitored system dump the stack trace and other information for troubleshooting.

NMIs can also be generated by software. Specifically, the software could configure local APIC to send an inter-processor interrupt (IPI) to target processors. Additionally, the software could configure local APIC to specify the delivery mode of the IPI, while NMI is one of the options. With this mechanism, the Linux kernel could send an NMI to request target processors to dump their stack trace for troubleshooting. Furthermore, when the Linux kernel receives the command to reboot or shut down, it might send an NMI to all other processors to forcefully stop their execution.

doi:10.6342/NTU202403471

2.6.1 NMI Handling

In Linux, the kernel traverses the list of registered NMI handlers upon receiving an NMI. Because there are lots of sources of NMIs, either software or hardware must provide information about NMIs. Each NMI handler checks the information to identify an NMI's purpose before executing their underlying logic to handle the NMI. If an NMI is not registered by any registered NMI handlers, it is regarded as unknown. We provide a case study for some common NMI handlers and their associated information checks found in Linux kernel.

- The PMI handler checks a global counter maintained by PCL and STATUS. The counter records the number of registered performance monitoring events in the kernel, and STATUS shows which PMCs overflow. If there are no registered performance monitoring events in the kernel or STATUS indicates that no PMC overflows, the PMI handler would not deem the NMI as a PMI and simply skip the real handling logics.
- The NMI handlers to dump stack trace and stop CPU's execution are similar. They both checks a global bitmap maintained by Linux kernel. Before a kernel thread sends NMI to target processors, it updates the bitmap so that only bits associated with target processors are set. Upon receiving an NMI, the NMI handlers to dump stack trace and stop CPU's execution checks if the corresponding bit of current processor is set in the bitmap. If not, they skip the underlying logics.
- The NMI handler for IPMI watchdog checks several software flags in the driver. These software flags are set upon driver initialization and user command. Besides,

there is a flag in IPMI hardware that indicates the occurrence of pre-timeout interrupt. The software could consult the IPMI hardware about the flag.

2.6.2 NMI Blocking and PMI Masking

To avoid nested NMI, when an Intel processor receives an NMI, the following NMIs will be blocked on that processor until an interrupt return instruction, *IRET*, is executed. In Linux, preemption and interrupts are disabled until all NMI handlers complete their work, ensuring the atomic execution of NMI handling due to the criticality of NMIs. After all NMI handlers complete their work, an *IRET* is executed to return from the NMI handlers and disable NMI blocking, allowing the processor to subsequently receive NMIs. Given the atomicity of NMI handlers, NMI blocking is always disabled outside of NMI handlers.

As mentioned in section 2.3, the local APIC masks PMIs before raising them to the processor. The PMI handler in PCL unmasks PMI after finishing its work. Similar to NMI blocking, PMI masking is always disabled outside of the PMI handler since the PMI handler is executed atomically.

Figure 2.1 illustrates the status of NMI and PMI in a scenario where a program causes PMC overflows, subsequently triggering a PMI signal sent as an NMI. Upon receiving the PMI, the hardware automatically blocks the NMI and masks the PMI from the processor core. Once the PMI handler completes its task, it unmasks the PMI. Similarly, after all NMI handlers have finished, the kernel executes the *IRET* instruction to unblock the NMI. In Linux, the NMI handling process is atomic, ensuring that NMI blocking and PMI masking are always disabled when the software is running.

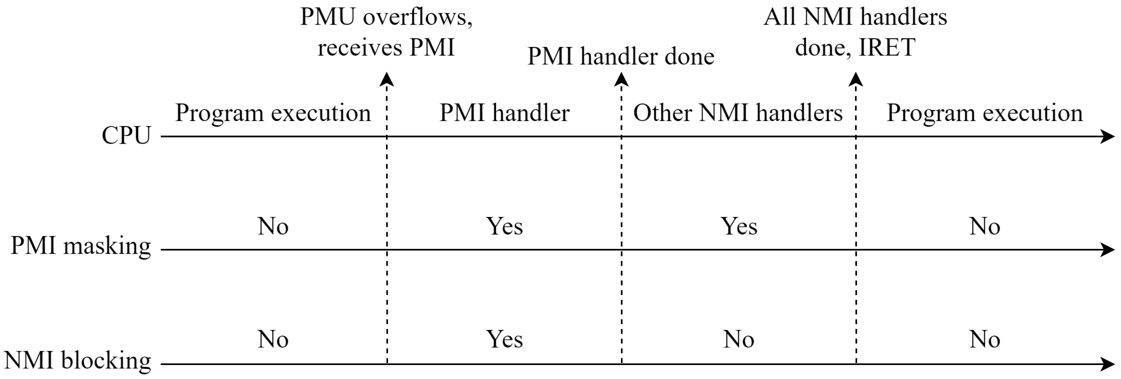

Figure 2.1: Timeline of NMI blocking and PMI masking

doi:10.6342/NTU202403471

Chapter 3 Design

As mentioned in section 2.5, MSR accesses and PMI in virtual machines cause VM exits. These VM exits and the subsequent KVM emulation work result in significant overhead. Intuitively, the virtualization overhead can be reduced by eliminating the VM exits and KVM's emulation. This thesis proposes two techniques, *MSR passthrough*, and *NMI passthrough*, to eliminate the VM exits due to MSR accesses and PMI handling to optimize the performance of PMU virtualization. We ensure the optimizations preserve the security and functionalities of the host and guest OS kernel.

3.1 MSR passthrough

As mentioned in section 2.5, KVM's trap-and-emulate incurs significant overhead to VMs' accesses to PMU MSRs, which turns out to slow down profiling in VMs. To reduce the overhead of such MSR accesses, we configure VMCS to allow VMs to access the hardware's PMU MSRs directly. Specifically, we clear corresponding bits for PMU MSRs in the MSR bitmap in VMCS to disable VM exits caused by VMs executing RDMSR/ WRMSR against these MSRs.

MSR passthrough eliminates the MSR emulation overhead in KVM for managing metadata and VM states. It allows the guest PCL to manage the PMU hardware. Un-

doi:10.6342/NTU202403471

like KVM, which registers performance monitoring events from the host PCL to facilitate PMU virtualization, the host PCL is detached from the vCPU thread that uses the monitoring hardware when employing the MSR passthrough. This means the host PCL neither saves nor restores PMU MSR states on the hardware when the host Linux context switches the vCPU thread. Our design delivers two goals.

- **P1**: Isolate host and VM MSR states.
- **P2**: Minimize performance impact of the isolation mechanism.

To achieve **P1**, we extended KVM to context switch the hardware MSRs for PMU between the host and VMs. The VMCS supports automatically context switching MSRs between the host and VMs. Specifically, on a VM exit, Intel VMX saves the MSR states from the hardware to the VMCS and restores the host MSR states from the VMCS to the hardware; VMX performs the opposite operations on VM entries. On the evaluated Intel hardware, more than 10 relevant MSRs must be saved when used by the VM and restored when entering the VM. Relying on VMCS for context switching these MSRs could induce significant performance overhead.

To ensure **P2**, we register callback functions to Linux's preempt notifier to context switch MSRs whenever Linux schedules in or out a vCPU thread. The *sched_out* callback saves the VM's MSR values from the hardware when a vCPU thread is to be scheduled out. The *sched_in* callback restores the saved MSR values to the hardware when a vCPU thread is to be scheduled. Deferring the context switch for performance monitoring MSRs is feasible because the host or other vCPUs do not use the MSRs before they are re-scheduled. Compared to the VMCS-based approach, the deferral-based approach intuitively context switches MSRs only when the vCPU does not use the states or when

doi:10.6342/NTU202403471

the vCPU gets executed, thus resulting in better performance.

We defer context switching all PMU MSRs except for CTRL MSR, which we configure the VMCS to save and restore on VM exits and entries. This is intended to isolate the VM and the host's PMU usage. We aim to prevent hardware events in the hypervisor from affecting the PMU state (i.e., IA32_A_PMC) accessed by the VM. If all MSR save and restore are deferred, PMU will keep counting and recording after the VM exits. To address the issue, we disable the PMU on a VM exit after VMCS performs the context switch of CTRL. This ensures that the instructions executed in KVM do not affect the PMU state and LBR records. Nevertheless, this renders profiling KVM with PMU impossible. We deem it a reasonable trade-off as most users are unlikely to profile KVM when VM actively profiles with MSR passthrough.

3.2 PMI passthrough

As mentioned in section 2.5, the PMI handling process in KVM brings significant overhead. To optimize the performance of PMI handling, we disabled NMI exiting in the VM execution control field of VMCS. We configured the NMI setting because, as mentioned earlier, PMIs are delivered as NMIs on Intel hardware. An NMI will not cause VM exit when the processor is in VMX non-root operation. In other words, we pass through the PMI to VMs, allowing them to handle the PMI without KVM's intervention. Since the virtual machine can handle NMI without VM exit, overhead exists in the original KVM implementation, including running host NMI handlers and NMI injection are removed.

Our design ensures the following properties to safely enable PMI passthrough.

doi:10.6342/NTU202403471

- **P1**: NMIs are handled by the intended entity.

- **P2**: NMI states of an entity should not affect the states of other entities.

3.2.1 Property P1

As mentioned above, disabling NMI exit in VMCS grants the VM the privilege to handle the NMI directly. However, since NMI is widely used in the operating system, there is a risk that an NMI not intended for the VM could be mistakenly handled by it. This misrouting can lead to functionality and security issues. We provide a case study about various NMI usages from section 2.6. We discuss scenarios when the host NMIs are being delivered to VMs and the potential side effects if this occurs. We then discuss the mechanism we introduce to ensure property **P1**.

For PMIs delivered as NMIs, since we context-switch MSRs, the PMCs only function after VM entry with MSR passthrough. Consequently, every PMI that arrives in VMX non-root operation is always targeted at the guests. Therefore, it is impossible for a PMI intended for the host to be mistakenly handled by the guest. However, other NMIs listed in subsection 2.6.1 are prone to the misrouting. Hence, we propose 2 different mechanisms to solve the issue.

We first assume that NMI passthrough is used by a cooperative and benign guest VM. Developers who deploy applications or services to VMs may leverage NMI passthrough to acquire profiling information. In a scenario in which a VM employs NMI passthrough when a VM receives an unknown NMI, the NMI should be targeted at the host. Thus, we expose a hypercall for VMs to invoke and inform KVM that they have received an unknown NMI and to redirect the NMI back to the host. When KVM receives the hypercall,

doi:10.6342/NTU202403471

it invokes the NMI handlers in the host kernel to handle the NMI.

We also consider a non-cooperative or malicious guest VM which would not inform KVM of the unknown NMI. Recall that each NMI handler checks the information in software or hardware to identify an NMI's purpose. We observe that these information are cleared or reset only if the underlying logic of the associated NMI handler is executed. For instance, only the NMI handler to dump stack trace would clear the bits in the bitmap indicating the processors that should dump their stack trace. If a target processor somehow misses the NMI, the associated bit is not cleared until it receives next NMI and executes the underlying logics. Consequently, we could check these information in the host kernel when some VMs enable NMI passthrough to identify if there are lost NMIs intended for the host. Once the host kernel detects lost NMIs by checking these information, it invokes NMI handlers instantly in the host kernel to handle it. Moreover, these information are either stored in the host kernel or hardware. The VMs could not access data in the host kernel unless there are bugs in the host kernel. Accesses to the hardware, e.g., IPMI, are typically accomplished by I/O instructions, which are intercepted by KVM by default. That is, KVM is able to detect such hardware accesses and analyze the semantics. As a result, the malicious guest VMs could not modify these information to confuse the host kernel.

Performance Event Skid A Performance Event Skid [3] refers to the lag between the occurrence of a performance event and the time it is observed or delivered to the targeted entity (e.g., a CPU processor or monitoring software). The delivery of a PMI is an example of a performance event skid. Specifically, a delay could exist between when a PMC overflows and when a processor receives the PMI. This delay can cause issues in a

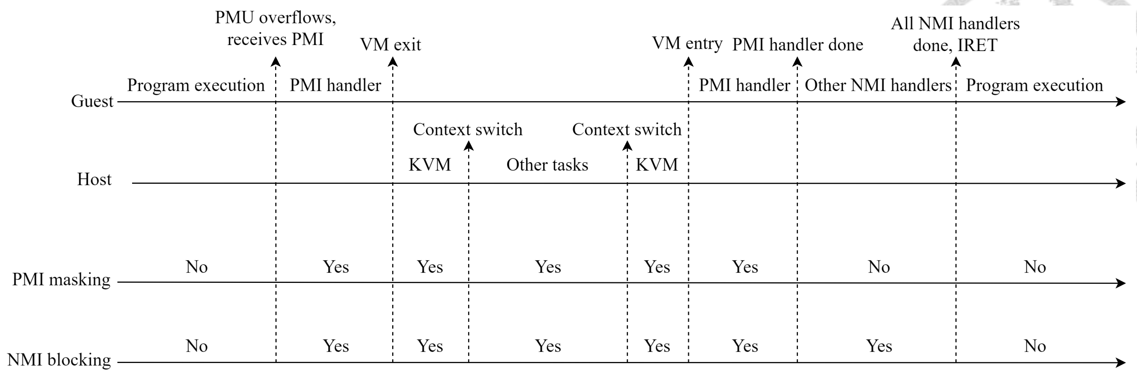

Figure 3.1: Original timeline of NMI blocking and PMI masking with passthrough

virtualization context with NMI passthrough.

When a VM adopts NMI passthrough, all PMIs are expected to be sent to and handled by the VM when PMC overflows result from hardware events during the VM's execution. However, we found that the host kernel sometimes receives NMIs intended for PMIs that should be directed to VMs, thus reporting the receipt of unknown NMIs. This occurs because VM exits may happen within a performance event skid. For instance, an external timer interrupt may arrive at the processor during a performance event skid of PMIs, causing a VM exit and resulting in the hypervisor or the host kernel receiving the PMI. The host cannot handle the PMI because it is unaware of the VMs' performance monitoring events.

To resolve the issue of the host mistakenly receiving the PMI, we register an NMI handler in the host kernel. This handler sets a flag to inform the KVM to inject a PMI into the guest on the next VM entry.

3.2.2 Property P2

The diagram in Figure 3.1 shows that the status of NMI and PMI could be incorrectly prolonged with NMI passthrough, violating **P2**. The design of NMI passthrough ensures

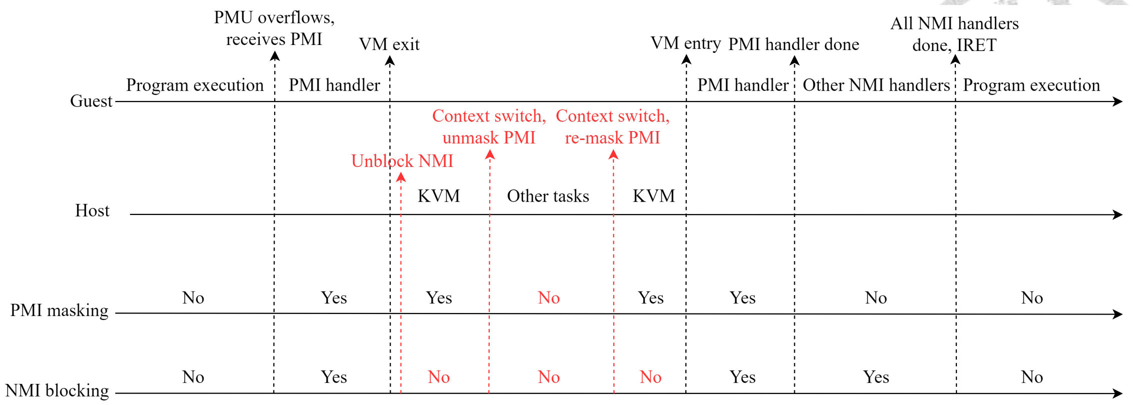

Figure 3.2: Timeline of NMI blocking and PMI masking with passthrough after securing P2

that **P2** is preserved when NMIs and PMIs are raised to the processor when VMs employ the mechanism are running. Specifically, we ensure these interrupts' blocking or masking states do not affect the host's or other VMs' execution. The diagram in Figure 3.2 shows the status of NMI and PMI after **P2** is secured. We discuss the detail of NMI and PMI below.

NMI Blocking. As discussed in section 2.6, on Intel hardware, when an NMI is sent to a processor, the hardware blocks subsequent NMIs from the processor until an *IRET* instruction is executed.

KVM configures VMCS to enable NMI-exiting and intercepts all NMIs sent to the processor during VM execution by default. Upon receiving an NMI, KVM invokes NMI handlers in the host Linux kernel and determines whether to inject an NMI into the VM. The host Linux handles NMIs atomically, and thus NMI blocking is guaranteed to be disabled after all NMI handlers in the host Linux kernel are finished. However, when NMIs are routed directly to VMs with NMI passthrough, the guest kernel cannot handle NMIs atomically. This is because, during the execution of a VM's NMI handler, VM exits can occur due to events like timer interrupts or the execution of sensitive instructions in

NMI handlers, such as those performing memory-mapped I/Os. Since KVM must handle these events, the trapping prolongs the duration that NMIs are blocked on the processor, potentially causing NMI loss and compromising the host kernel's profiling functionalities. Due to NMI blocking, the host's processes or threads that preempt the trapping vCPU thread cannot receive PMIs. Consequently, the profiling result could be erroneous.

To resolve the issue caused by NMI blocking, we check the *blocking by NMI* bit in the VMCS to identify whether NMI blocking is effective. If this bit is set, we unblock NMI by executing the *NMI unblock helper* shown in Listing 1 in KVM outside an interrupt handler. The helper executes *IRET* to unblock the NMI. In essence, *IRET* pops the return address, code segment selector, RFLAGS, old stack pointer, and stack segment selector from the stack and continues the execution. Directly executing this instruction could corrupt KVM and lead to malfunction. Thus, we craft a fake interrupt stack to make *IRET* act as a *NOOP* instruction; in other words, *IRET* causes no effect on the processor's states after execution except for NMI blocking. Also, NMI blocking is automatically re-enabled on VM entry once *blocking by NMI* bit is set in the VMCS. Consequently, we could ensure that the status of NMI blocking is same as before VM exit.

```
static void x86_unblock_nmi(void)
{
    asm(
        "movq %%rsp, %%rax\n\t" // push updates rsp, save rsp first
        "pushq $0x18\n\t"       // ss == 0x18
        "pushq %%rax\n\t"       // rsp
        "pushfq\n\t"            // rflags
        "pushq $0x10\n\t"       // cs == 0x10
        "pushq $1f\n\t"         // rip
        "iretq\n\t"
        "1: nop\n\t"
        :
        :
        : "%rax", "cc", "memory");
    return;
}
```

Listing 1: NMI unblock helper

PMI Masking. When PMIs are passed through to the VM, we must ensure the VM can unmask them. This presents two challenges. First, VMs cannot unmask PMIs on their own. Unmasking PMI requires an update to the local APIC on the hardware, which cannot be done without a VM exit. On the other hand, KVM emulates APIC operations by reflecting updates to a mocked APIC in memory; KVM does not update the local APIC hardware. On KVM, VMs cannot unmask PMIs in the hardware's local APIC. Since PMI is masked, the processor cannot receive subsequent PMIs. To resolve this issue, we modify the APIC emulation of KVM. If NMI passthrough is enabled and the VM attempts to unmask PMI, KVM unmasks PMI in physical local APIC on behalf of the VM.

Second, similar to the issue of NMI blocking, interrupting the VM's PMI handling process delays unmasking PMIs with NMI passthrough and compromising functionalities that depend on PMIs. To resolve the issue, we manually unmask PMI when a vCPU thread is scheduled out, ensuring that PMI masking does not affect other tasks on the host. We also record whether PMI was masked before the vCPU thread was scheduled out. If it was masked, we re-mask PMI when the vCPU thread is scheduled back in. In summary, we ensure that (1) PMI is always unmasked for other host tasks and (2) PMI remains masked if it was masked on the last VM exit.

doi:10.6342/NTU202403471

Chapter 4 Evaluation

We added and modified 465 lines of C code to KVM in the mainline Linux v6.9 to implement the proposed MSR passthrough and NMI passthrough.

4.1 Experimental Setup

All experiments are conducted on a physical machine with 2 Intel Xeon Silver 4114 10-core CPUs @2.20GHz and 192 GB DDR4 RAM. We use Ubuntu 20.04 with Linux kernel v5.5.0 for host operating system for evaluation on bare metal. On the other hand, we use Ubuntu 22.04 with Linux kernel v6.9 for host operating system and Ubuntu 20.04 with Linux kernel v5.5.0 for guest operating system for evaluation in virtual machines. The virtual machines are launched by QEMU v8.0.0 with KVM acceleration. The vCPU of the virtual machines are passthroughed from host, and the amount of vCPU is 2 in the evaluation of subsection 4.3.1 and 1 for others, and the memory size of the virtual machines is 4GB.

Table 4.1: Consumed CPU Cycles for MSR Accesses. RO represents the MSR is read only.

	KVM		Passthrough		Bare Metal	
	Read	Write	Read	Write	Read	Write
EVTSEL	3134	3200	143	219	107	186
IA32_A_PMC	3128	3202	127	208	90	175
CTRL	3076	3134	123	168	87	134
STATUS	3120	RO	122	RO	87	RO

4.2 Performance

4.2.1 MSR Accesses

In this section, we show the elapsed cycles for a single RDMSR/WRMSR instruction on PMU and LBR MSRs. We use KVM-unit-tests [4] to obtain elapsed cycles for RDMSR/WRMSR in a virtual machine. As for bare metal, we execute instructions identical to KVM-unit-tests in a custom system call. The result is shown in Table 4.1.

In Table 4.1, we can see that KVM incurs a significant overhead for a virtual machine to access PMU MSRs. The overhead mainly results from the trap-and-emulate process of KVM. Moreover, it's worth mentioning that both EVTSEL and CTRL are written to 0 in the tests. In this case, KVM does not register or deregister any event to host PCL. That is, the number of elapsed cycles would be much larger for KVM when the special case of writing EVTSEL and CTRL mentioned in section 2.5 occurs because the emulation process of KVM turns out to involve complex operations in host PCL.

On the other hand, with MSR passthrough and proper context switching of MSR, a virtual machine can directly access PMU and LBR MSRs without KVM interposition, which minimizes the overhead. On average, an MSR read/write operation takes 4.5%/

doi:10.6342/NTU202403471

7.1% of elapsed cycles of the original KVM implementation with MSR passthrough, which shows the effectiveness of MSR passthrough in reducing the overhead of MSR accesses in a virtual machine.

4.2.2 Simple Program Using PCS

We measured the execution time of a simple program that uses PCL in VM and on bare metal to show that MSR passthrough and NMI passthrough play a role in reducing the overhead. Specifically, the program registers a performance monitoring event to count branch instructions in ring 1-3 and runs an empty for loop with N iterations (i.e., N branch instructions are executed). The performance monitoring event is configured to generate a PMI every M branch instructions. Both M and N are adjustable. The program is expected to receive $\frac{N}{M}$ PMIs during its execution. The code snippet is given in section A.1. We first measured the execution time of the program without PMIs by setting M larger than N. The execution time is approximately 0.2 ms both in VM and bare metal, and we deemed this as the basic execution time of the program. The results are shown in Table 4.2. Note that N is fixed to 100000 in this section.

Table 4.2: Execution Time of the Program Using PCL, N = 100000

	Number of PMIs	KVM	Only MSR Passthrough	MSR + NMI Passthrough	Bare Metal
M = 200000	0	0.200 ms	0.200 ms	0.200 ms	0.200 ms
M = 10000	10	0.510 ms	0.340 ms	0.271 ms	0.234 ms
M = 1000	100	2.635 ms	1.281 ms	0.547 ms	0.406 ms
M = 100	1000	24.181 ms	10.472 ms	3.276 ms	2.153 ms

In Table 4.2, we can see that the KVM adds considerable overhead to PMU virtualization. On bare metal, handling 10/100/1000 PMIs incurs 0.034/0.206/1.953 ms of overhead to the overhead. Handling 10/100/1000 PMIs in a VM governed by KVM incurs

Table 4.3: Consumed CPU Cycles of a single VM exit.

	KVM	NMI Passthrough	Overhead
CPUID instruction	2981	3216	235 (7.9%)
VMCALL instruction	2951	3174	223 (7.6%)
INL instruction	3221	3453	232 (7.2%)
OUTL instruction	3231	3464	233 (7.2%)

0.31/2.435/23.981 ms of overhead to the program, which is about 9.12/11.82/12.28× of the overhead on bare metal. As mentioned earlier, KVM uses trap-and-emulate to virtualize VMs' accesses to the PMU MSRs, adding significant overhead to profiling with PCL in VMs. Since MSR passthrough allows VMs to access PMU MSRs directly, the overhead of KVM is reduced by about 55% with MSR passthrough alone. In addition to PMU MSR accesses, PMI handling in KVM also incurs significant overhead to PMU virtualization. As mentioned earlier, a PMI causes VM exit, and KVM eventually invokes the PMI handler from the host' s PCL to handle the PMI. Subsequently, KVM injects a PMI to the VM, and ultimately the PMI handler from the VM's PCL is invoked. This lengthy PMI handling process accounts for a great proportion of the overhead. With NMI passthrough, the PMI handling process is simplified to invoking the PMI handler from the VM's PCL. Compared to enabling MSR passthrough alone, enabling MSR passthrough and NMI passthrough simultaneously reduces the overhead of handling 10/100/1000 PMI by 49.3%/67.9%/70.1%. If compared to KVM, enabling MSR passthrough and NMI passthrough simultaneously even reduces the overhead of handling 10/100/1000 PMI by 77.1%/85.7%/87.2%. In conclusion, we successfully reduced the profiling overhead in VMs with MSR passthrough and NMI passthrough.

doi:10.6342/NTU202403471

4.2.3 VM Exit Overhead of NMI passthrough

In section 3.2, we propose several mechanisms to safely enable PMI passthrough, some of which are enforced after every VM exit. In this section, we measure the overhead results from these mechanisms. Specifically, the overhead below consists of the following checks:

- NMI information
- Performance event skid
- NMI blocking

Note that PMI masking is not included because PMI masking is checked on context switch instead of VM exit. We used KVM-unit-tests to obtain elapsed cycles of a single VM exit. The result is shown in Table 4.3. The first column shows the reason of VM exit.

Table 4.3 shows that we add about 230 cycles to every VM exit to safely enable NMI passthrough. Originally, KVM configures VMCS to enable NMI-exiting and intercepts all NMI. The required works, e.g., unblock NMI, inject NMI to VMs, are guaranteed to be carried out once an NMI arrives at the processor. Consequently, KVM does not need to enforce these checks. On the contrary, since host could be unaware of arrival of an NMI, enabling NMI passthrough could prevent host from taking correct action to handle an NMI. Hence, we must enforce these checks to ensure the functionality of host. Although these checks add overhead to VM exit, we consider it a reasonable trade-off given the improvement on PMI handling. Moreover, it's unnecessary to enable NMI passthrough throughout a VM's lifetime. It's sufficient to enable NMI passthrough only when the VMs need to perform profiling. In this case, we suggest the hypervisors expose a hypercall

doi:10.6342/NTU202403471

for the VMs to enable and disable NMI passthrough, removing the constant performance downgrade.

4.3 Real-World Usage

Generally, any use case on bare metal is also feasible in a virtualized environment with our work. In this section, we discuss on 2 possible real-world use cases.

4.3.1 Linux Perf Tool

An intuitive case is to use Linux perf tool [11] to profile and optimize programs in a virtualized environment such as cloud. Thus, we experimented with simulating profiling with the Linux perf tool in the cloud.

Simple Workload We crafted a simple workload which executes empty for loops with different iterations in different functions and profile the workload with Linux Perf Tool. The workload is shown in Listing 2. The execution time of each function is proportional to the iterations executed by itself and all of its callees. For instance, function a() executes 2×10^8 iterations and calls function aa() which executes 10^8 iterations. The number of iterations in the workload is 10^9. Thus, the execution time that function a() accounts for is $\frac{10^8+2\times10^8}{10^9} = 30\%$. We sampled the elapsed CPU cycles in ring 1-3 and capture stack traces for the workload, which is further used to generate distribution of execution time. We also measured the execution time of the workload with and without profiling.

As shown in Table 4.4 and Table 4.5, both KVM and passthrough can generate correct profiling results. However, it costs 12.7% more overhead for KVM to finish the profiling.

doi:10.6342/NTU202403471

```c
#define ITER 100000000
#define loop(x) for (int i = 0; i < x; i++)

void aa(void)  { loop(ITER); }
void a(void)   { loop(2 * ITER); aa(); }
void bbb(void) { loop(ITER); }
void bb(void)  { loop(2 * ITER); bbb(); }
void b(void)   { loop(1 * ITER); bb(); }
void c(void)   { loop(3 * ITER); }
int main() {
    a(); b(); c();
    return 0;
}

```

Listing 2: Source Code of the Simple workload

With passthrough optimization, profiling and optimization of a program is accelerated, saving developers' time. This also implies that KVM could generate incomplete or inaccurate profiling result within the same time interval compared to passthrough optimization.

Nginx We run an nginx server in the virtual machine and profile the nginx worker process by attaching the Linux perf tool to the nginx worker process. We sample the elapsed CPU cycles in rings 1-3 with a frequency of 4000 Hz (i.e., approximately 4000 PMI will be generated per second, which is the default frequency of Linux perf tool). Then, we use wrk [13] to stress the nginx server for 30 seconds and obtain the throughput. We configure wrk to use 1 threads and open 16 connections simultaneously. We run nginx and wrk either in the same virtual machine for virtualized cases or in the same physical machine for bare metal.

Table 4.4: Time Distribution of Each Function Generated by Linux Perf Tool

	KVM	Passthrough	Bare Metal	Expected
a	31.82%	31.26%	29.58%	30%
aa	9.72%	9.82%	9.72%	10%
b	39.02%	39.29%	39.59%	40%
bb	29.27%	29.47%	29.14%	30%
bbb	9.76%	9.81%	9.71%	10%
c	29.15%	29.44%	30.82%	30%
main	100%	100%	100%	100%

Table 4.5: Execution time of Crafted Workload

	KVM	Passthrough	Bare Metal
Without Profiling	2.36 s	2.36 s	1.92 s
With Profiling	2.84 s	2.54 s	2.06 s
Overhead	20.3%	7.6%	7.3%

Table 4.6: Statistics of wrk Benchmark without Profiling

	KVM	Passthrough	Bare Metal
Average Latency	488.86 us	494.02 us	417.52 us
Maximum Latency	3.80 ms	3.03 ms	2.76 ms
Requests per Second	32535.12	32120.10	36042.20
Transferred Data per Second	2.45 GB	2.42 GB	2.72 GB

Table 4.6 shows the statistics of wrk benchmark without profiling. It's worth mentioning that the throughput of our work is slightly lower than that of the original KVM implementation. The main reason is that we enable MSR passthrough throughout the lifetime of a vCPU, which requires MSR context switching and thus introducing a little overhead (about 1.3%).

Table 4.7 shows the statistics of wrk benchmark with nginx worker process profiled by Linux perf tool. Profiling nginx worker process in a virtual machine with the original KVM implementation induces about 15.3% overhead, which results from abundant VM exits due to MSR accesses and PMI and consequent emulation. As mentioned earlier, such overhead could lead to incomplete or inaccurate profiling result within the same time interval compared to bare metal and passthrough optimization. Also, such overhead can

Table 4.7: Statistics of wrk Benchmark with Linux Perf Tool Profiling

	KVM	Passthrough	Bare Metal
Average Latency	577.32 us	505.49 us	424.90 us
Maximum Latency	19.04 ms	4.08 ms	2.13 ms
Requests per Second	27573.13	31538.83	35943.40
Transferred Data per Second	2.08 GB	2.38 GB	2.71 GB

hinder real performance issue in the program if the program itself is quite latency-sensitive and the profiling overhead results to a timeout. On the other hand, profiling nginx worker process in a virtual machine with MSR and NMI passthrough induces only 1.8% overhead. Although it's still larger than the 0.3% overhead on bare metal, we eliminate VM exits due to MSR accesses and PMI with MSR and NMI passthrough and significantly reduce the profiling overhead caused by KVM emulation. This also implies that we reduce the required time to generate reliable profiling result and the chance to reach the timeout of the program compared to KVM.

In conclusion, we drastically reduces the overhead of profiling a program in a virtual machine and generates stable and reliable profiling result in a virtual machine. Contrarily, the original KVM implementation imposes significant overhead on profiling and could generate less reliable profiling result and may even obscure the underlying performance issues, which hinders the optimization progress in a virtual machine.

4.3.2 Fuzzing

Fuzzing refers to the process that provides random input data to a target program and check if crash or memory leakage occurs. Recently, coverage-guided fuzzing, which takes code coverage as a hint to generate input data, has been the mainstream of research on fuzzing. [6] is a state-of-the-art coverage-guided fuzzer targeted at operating system kernel. Based on [6], [12] moves the fuzzing process into a virtual machine and takes advantage of virtual machine snapshot to accelerate kernel device driver fuzzing.

One common metric of code coverage is branch coverage, i.e., the taken branches divided by the total branches, since more branch coverage means that more basic blocks are

doi:10.6342/NTU202403471

executed. From our perspectives, LBR can help and improve coverage-guided fuzzing. First, it requires instrumentation to obtain code coverage of a program. However, instrumentation may be infeasible. Instrumentation inserts instructions to collect code coverage, which enlarges program size. If the disk size is limited, instrumentation should be avoided. In this case, LBR would be a good alternative to instrumentation to collect branch coverage since it requires only MSR writes to enable LBR and MSR reads to obtain the branch information. Besides, a coverage-guided fuzzer may also take LBR record as a metric in addition to branch coverage. One main advantage of LBR is we can know the execution order of each branch instructions, which is not shown by the branch coverage. The order may help the fuzzer understand more about the execution of the target program and thus the fuzzer can generate input data much more accurately, potentially enhancing the fuzzing efficiency. Also, LBR logs more miscellaneous attributes of branch instructions such as elapsed cycles, whether it's a misprediction, etc., all of which provide more information about the program execution to the fuzzer. Hence, we managed to enable LBR and obtain complete LBR record for every executed system call with [12], which involves PMU and LBR virtualization in ring 0. We port our work backward to v4.18.20 to match the kernel version of [12]. Table 4.8 shows the number of executed test cases (i.e., system calls) within 3 hours of Agamotto with and without LBR enabled of 7 kernel drivers. Since PMU virtualization of v4.18.20 is much slower than that of v4.18.20 and LBR virtualization is not supported in v4.18.20, we could not collect data for KVM.

As mentioned in section 2.4, we utilized PMU to generate a PMI every 32 taken branch instructions to avoid LBR buffer from being overwritten by subsequent branch instructions. We read LBR buffer and stored the content of LBR buffer in the kernel in the guest PMI handler. We implemented an ioctl() handler in KVM to pass the content of

Table 4.8: Number of Executed System Calls of Agamotto

Driver	Agamotto	Agamotto + LBR	Deduction
ar5523	7561	5624	1937 (25.6%)
btusb	8077	5889	2188 (27.1%)
go7007	7731	6243	1488 (19.2%)
pn533	7118	5774	1344 (18.9%)
rsi	7046	5571	1475 (20.1%)
si470x	7452	5754	1698 (22.8%)
usx2y	7226	5587	1639 (22.7%)
Average Deduction			22.3%

LBR buffer from the VM to QEMU, which is required for [12]. As shown in Table 4.8, all the operations above brought about 22.3% overhead in total to [12], which is acceptable if the LBR record can help the fuzzer generate input data more accurately. Note that we enabled LBR in every test case in the experiment, but it may be sufficient to enable LBR only in a few test cases. For instance, enable LBR on test cases mutated from a test case in the corpus since the execution flow would be very similar to the original one. Thus, the overhead can be smaller if LBR is selectively enabled.

It's also feasible to enable and collect LBR as a hint before the fuzzing process. A common issue for fuzzing is that the fuzzer struggles to generate test cases to pass specific branches, which limits the effectiveness of fuzzing. Thus, one may run the fuzzing process on the initial corpus (i.e., test cases) with LBR enabled in advance to identify branches that could hinder the fuzzing process so that the fuzzer can generate or mutate a test case that passes the branch more efficiently with the information.

4.4 Functionality and Security

In section 3.1 and section 3.2, we discussed properties to be secured to safely enable MSR passthrough and NMI passthrough. We conducted several experiments to prove that the properties are successfully secured. In this section, we discuss on the detail and the result of the conducted experiments.

4.4.1 MSR passthrough

In section 3.1, we introduced 2 goals for MSR passthrough:

- **P1**: Isolate host and VM MSR states.
- **P2**: Minimize performance impact of the isolation mechanism.

Since **P2** is not relevant to functionality, we would only discuss **P1** here. We conducted 2 experiments to show that **P1** is secured. In the first experiment, we launched 2 VMs and forced them to run on the same physical CPU. We wrote an MSR involved in MSR passthrough in the first VM and wrote the same MSR in the second VM. We then read the MSR in the first VM. Without MSR context switching in section 3.1, the first VM read the value written by the second VM, showing that the MSRs are not properly isolated and multiplexed between multiple VMs. Contrarily, the first VM read the value written by itself with MSR context switching in section 3.1, showing that we successfully isolate and multiplex the MSRs between multiple VMs.

The second experiment is quite similar to the previous one, except that the second VM is replaced by a host process. We performed identical steps to above and obtained the

doi:10.6342/NTU202403471

same result, showing that we also properly isolate and multiplex the MSRs between VMs and host processes. In conclusion, we successfully secure **P1**.

4.4.2 NMI passthrough

In section 3.2, we introduced 2 properties that must be secured:

- **P1**: NMIs are handled by the intended entity.
- **P2**: NMI states of an entity should not affect the states of other entities.

In this section, we prove that both **P1** and **P2** are secured with our design.

P1. We conducted the following experiments to show that NMIs are handled by the intended entity. In cases of performance event skid, we utilize guest PCL in the VM to generate a fixed amount of PMIs. Originally, it's possible for a PMI to be mistakenly received by host due to performance event skid. In this case, the underlying logic of host PMI handler may not be executed since host PCL is detached from the vCPU thread when employing MSR passthrough, leaving PMI masked in local APIC. As a result, the VM cannot receive subsequent PMIs. We observed that it is common for VMs to be impacted by performance event skid. However, with our solution in section 3.2, we could correctly detect performance event skid and inject NMI back to VMs, and VMs are never affected by the performance event skid.

In cases of VMs mistakenly handling NMIs intended for host, we conducted experiments to show that the problem is solvable for both benign and malicious VMs. In this experiment, we used NMIs to dump stack trace as an example. We sent NMIs to dump

doi:10.6342/NTU202403471

stack trace to the processor that the VM ran on and checked if host could always detect it. For benign VMs, we invoked the hypercall mentioned in section 3.2 when the VM encounters an unknown NMI. For malicious VMs, we relied on NMI information checks mentioned in section 3.2 after VM exit. Specifically, we checked the global bitmap indicating the processors that should dump their stack trace and reported the occurrence of an NMI if the associated bit of the processor that the VM ran on was set in the global bitmap. As a result, our design successfully detects the NMIs to dump stack trace for both benign and malicious VMs.

P2. We conducted the following experiments to show that the status of NMI blocking and PMI masking matches to Figure 3.2. In cases of NMI blocking, we ran a VM and utilize guest PCL to generate a PMI in the VM. We invoked a special hypercall in the VM's NMI handler to force a VM exit before the VM finishes all the NMI handlers. In the special hypercall handler, we utilized host PCL to generate a PMI. The implementation of the special hypercall handler is given in section A.2. Without invoking *NMI unblock helper* in Listing 1, NMI is blocked when KVM is handling the hypercall (as KVM in Figure 3.1). In this case, the processor could not receive the PMI, and the host PCL could not update the counter value. Consequently, the host PCL gave a wrong counter value. Contrarily, our design invokes *NMI unblock helper* in Listing 1 after VM exit if it detects a VM exit within guest NMI handlers. In this case, the process received the PMI correctly, and the host PCL updated the counter value in the PMI handler. As a result, the host PCL gave the correct counter value, showing that we successfully avoid the NMI blocking from being prolonged.

In cases of PMI masking, we ran a VM and pin the VM to a specific physical CPU.

doi:10.6342/NTU202403471

We ran another process, P_{Host}, which executed the program used in subsection 4.2.2 in host with M = 10000, N = 100000000. We make N much larger compared to subsection 4.2.2 to prolong the execution time of P_{Host}. As mentioned in subsection 4.2.2, P_{Host} is expected to receive $\frac{N}{M}$ = 10000 PMIs. We forced P_{Host} to run on the same physical CPU as the VM. After P_{Host} started its execution, we masked PMI in the VM and waited for P_{Host} to complete its execution.

Without unmasking PMI during context switch, the status of PMI masking is similar to Figure 3.1. Before PMI is masked by the VM, P_{Host} could receive PMIs normally. After that, PMI is masked for the remaining execution time of P_{Host} (as other tasks in Figure 3.1), preventing P_{Host} from receiving subsequent PMIs. As a result, P_{Host} received less than 10000 PMIs. Contrarily, our design unmask PMI on context switch and the status of PMI masking is similar to Figure 3.2. Therefore, PMI is unmasked throughout the execution of P_{Host} as Figure 2.1 shows. As a result, P_{Host} could always receive 10000 PMIs, showing that we successfully prevent the PMI masking from being prolonged.

doi:10.6342/NTU202403471

Chapter 5 Related Work

KVM [9], as a baseline of our implementation, has provided a trap-and-emulate method to virtualize PMU MSR accesses, which incurs significant overhead in a virtual machine. Additionally, PMI causes VM exit in KVM by default, which causes KVM to invoke NMI handler in host kernel and then inject NMI to virtual machine. The PMI handling process of KVM brings lots of overhead. On the other hand, our work manages to reduce the overhead of PMU virtualization of KVM by simultaneously enabling MSR passthrough and NMI passthrough, both of which makes impact on the performance of using PMU in a virtual machine.

[5] implements MSR passthrough to accelerate PMU virtualization and proposes two timing to context switch MSRs: (1) VM entry and VM exit (2) vCPU thread switch. The former is achieved by VMCS as mentioned in section 3.1. The latter context switches MSR values only when the KVM switches from one vCPU to another vCPU, which is totally different from context switching MSR on process context switching (i.e., via Linux preempt notifier or host PCL). Specifically, if a context switch occurs between a vCPU and a host process, [5] does not context switch MSRs. In this case, the instructions executed by host process affect the PMC counters. Consequently, the MSR states are not isolated between host and VMs, violating **P1** mentioned in section 3.1. Contrarily, our work context switches CTRL by the VMCS and other MSRs by Linux preempt notifier, striking

a balance between overhead and functionality. Additionally, [5] lacks performance comparison to the PMU virtualization of KVM, while we conduct a thorough experiment to show the effectiveness of MSR passthrough against the KVM implementation. Also, PMI handling process of [5] is identical to KVM. Thus, the overhead for PMI handling in [5] is still considerable. On the other hand, our work implements NMI passthrough together with MSR passthrough to further enhance the performance of PMU virtualization, which brings significant performance improvement.

[15] also managed to accelerate PMU virtualization by MSR passthrough. Compared to [5] and our work, [15] only context switches MSRs with VMCS (i.e., on every VM entry and VM exit), which can incur more overhead than context switching MSR with Linux preempt notifier as we do. Besides, [15] does not propose NMI passthrough to enhance the performance of PMI handling. Instead, [15] configure local APIC so that PMI is delivered as a maskable interrupt instead of an NMI before VM entry and implement a PMI handler to inject PMI to the virtual machine. In this case, When a PMI (delivered as a maskable interrupt) arrives in VMX non-root operation, VM exit occurs and KVM invoke the PMI handler provided by [15]. As a result, KVM injects a PMI to guest, but the injected PMI is delivered as an NMI. However, PMI still cause VM exit and requires KVM to inject PMI in [15], while our work not only eliminates VM exit due to PMI but also spare the work to inject a PMI.

[1] grants virtual machines power to handle maskable interrupts without VM exits. Although the concept of [1] is similar to NMI passthrough, there are several key differences between maskable interrupt and NMI which causes different challenges for NMI passthrough. Firstly, all NMI sources share a single interrupt handler. Unlike [1] can determine each maskable interrupt should or should not cause VM exit by maintaining

doi:10.6342/NTU202403471

a shadow IDT, we cannot simply have PMI handled without VM exit while other NMI sources still cause VM exit. Besides, there are no mechanisms to set the processor affinity of NMI, and thus we cannot redirect critical NMIs to a dedicated processor. Last but not least, there are some additional side effects for NMI such as NMI blocking. In section 4.4, we addressed these challenges and verified that our solutions are feasible.

doi:10.6342/NTU202403471

Chapter 6 Future Work

Fine-Grained MSR Passthrough Currently, we enable MSR passthrough throughout a virtual machine's lifetime, which introduces the demand to context switch MSRs and thus a little overhead. It would be better to enable MSR passthrough in a more fine-grained manner in three aspects: virtual machine, amount of MSR and time.

In terms of virtual machine, we can make whether to enable MSR passthrough a parameter for virtual machine creation so that the VM which does not need to utilize PMU can be free from the overhead of context switch MSRs. Moreover, as mentioned in section 3.1, we cannot profile KVM with PMU with MSR passthrough. With the ability to optionally enable MSR passthrough for a VM, we can choose to disable MSR passthrough when we want to profile KVM.

In terms of amount of MSR, we can let VMs specify the MSRs to be passthroughed. For instance, if a VM only needs to utilize one PMC within PMU, other PMC MSRs within PMU should not be passthroughed. In this case, we reduce the amount of MSRs to be context switched, lowering the overhead of MSR passthrough.

In terms of time, if MSR passthrough is enabled on demand like NMI passthrough, we only need to context switch MSRs when MSR passthrough is enabled. Thus, context switch MSR is only needed when the virtual machine is utilizing PMU, removing the unnecessary overhead when the virtual machine does not need to access PMU. However, guest PCL performs several checks to obtain information about PMU when the guest kernel is booting. If MSR passthrough is not enabled when guest is booting, some checks in guest PCL may fail, preventing VMs from utilizing PMU and other profiling hardware. Thus, it's challenging to enable MSR passthrough in a fine-grained manner in terms of time. We deem this a problem to be solved in future.

Alternative to NMI Currently, we do not change the default delivery mode of PMI and thus PMI is delivered as an NMI. Thus, we disable NMI-exiting in VMCS to reduce the overhead of PMI. Given NMI is used for some critical condition, it's an option to deliver PMI as a maskable interrupt like [15] does. In this case, we can combine the effort of [1] to passthrough only the vector of PMI. However, Linux kernel assumes that PMI is delivered as an NMI. Thus, guest kernel must be modified to correctly handle PMI in this case. Also, this removes the ability to profile the performance of interrupt handler since PMI cannot be handled within an interrupt handler if it's delivered as a maskable interrupt.

doi:10.6342/NTU202403471

Chapter 7 Conclusions

This thesis presents and implements MSR passthrough and NMI passthrough in KVM v6.9 to enhance performance of KVM PMU virtualization. In essence, MSR passthrough and NMI passthrough allow virtual machines to access PMU MSRs and handle PMIs respectively, which significantly reduces the overhead of VM exits and complex emulation due to accesses to PMU MSRs and PMI handling in virtual machines. Although MSR passthrough and NMI passthrough brings several side effects which could lead to functionality issues and security issues, we propose and implement solutions to these side effects. With these solutions, MSR passthrough and NMI passthrough will not compromise functionality and security of host and guest. Besides, we evaluate the performance of MSR passthrough and NMI passthrough and compare the performance to the original KVM implementation. As a result, we significantly improve the performance of accesses to PMU MSRs and PMI handling in virtual machines and thus reduce the profiling overhead in virtual machines. Additionally, we show the practicality of MSR passthrough and NMI passthrough by profiling nginx in a virtual machine. Last but not least, we conduct several experiments to verify the effectiveness our solutions to the side effects.

doi:10.6342/NTU202403471

References

[1] N. Amit, A. Gordon, N. Har'El, M. Ben-Yehuda, A. Landau, A. Schuster, and D. Tsafrir. Bare-metal performance for virtual machines with exitless interrupts. Commun. ACM, 59(1):108–116, dec 2015.

[2] Arm. Coresight architecture. https://developer.arm.com/Architectures/CoreSight

[3] D. Bakhvalov. Understanding performance events skid. https://easyperf.net/blog/2018/08/29/Understanding-performance-events-skid, 2018.

[4] K. Developers. Kvm-unit-tests. https://www.linux-kvm.org/page/KVM-unit-tests, 2020.

[5] J. Du, N. Sehrawat, and W. Zwaenepoel. Performance profiling of virtual machines. In Proceedings of the 7th ACM SIGPLAN/SIGOPS International Conference on Virtual Execution Environments, VEE '11, page 3–14, New York, NY, USA, 2011. Association for Computing Machinery.

[6] Google. Syzkaller. https://github.com/google/syzkaller, 2019.

[7] Intel. Intel vtune. https://www.intel.com/content/www/us/en/developer/tools/oneapi/vtune-profiler.html.

[8] Intel. Perfmon events. https://perfmon-events.intel.com/.

doi:10.6342/NTU202403471

[9] A. Kivity, Y. Kamay, D. Laor, U. Lublin, and A. Liguori. kvm: the linux virtual machine monitor. In Proceedings of the Linux symposium, volume 1, pages 225–230. Dttawa, Dntorio, Canada, 2007.

[10] A. Kleen. An introduction to last branch records. https://lwn.net/Articles/680985/, 2016.

[11] linux perf maintainer. Perf wiki. https://perf.wiki.kernel.org/index.php/Main_Page.

[12] D. Song, F. Hetzelt, J. Kim, B. B. Kang, J.-P. Seifert, and M. Franz. Agamotto: Accelerating kernel driver fuzzing with lightweight virtual machine checkpoints. In 29th USENIX Security Symposium (USENIX Security 20), pages 2541–2557. USENIX Association, Aug. 2020.

[13] G. Will. wrk - a http benchmarking tool. https://github.com/wg/wrk, 2012.

[14] A. Yasin. A top-down method for performance analysis and counters architecture. In 2014 IEEE International Symposium on Performance Analysis of Systems and Software (ISPASS), ISPASS 2014, pages 35–44, Monterey, CA, USA, March 2014.

[15] X. Zhang. Kvm: x86/pmu: Introduce passthrough vpm. https://lwn.net/Articles/959653, 2024.

doi:10.6342/NTU202403471

Appendix A — Code Snippets For Performance Analysis

A.1 Simple Program Using PCL

```
#define M 100
#define N 100000
int perf_event_open(struct perf_event_attr *attr, pid_t pid, int cpu,
    int group_fd, unsigned long flags)
{
    return syscall(SYS_perf_event_open, attr, pid, cpu, group_fd, flags);
}

int start_nmi_passthrough(void)
{
    int ret;
    asm volatile("vmcall" : "=a"(ret) : "a"(13));
    return ret;
}

int end_nmi_passthrough(void)
{
    int ret;
    asm volatile("vmcall" : "=a"(ret) : "a"(14));
    return ret;
}

int main() {
    int perf_fd;
    long long count;
    struct timespec ts_start, ts_end, ts_diff;
    struct perf_event_attr attr = {
        .type = PERF_TYPE_HARDWARE,
        .size = sizeof(sturct perf_event_attr),
        .config = PERF_COUNT_HW_BRANCH_INSTRUCTIONS,
        .exclude_kernel = 1,
        .disabled = 1,
        .sample_period = M,
    };

    clock_gettime(CLOCK_MONOTONIC, &ts_start);
    start_nmi_passthrough();
    perf_fd = perf_event_open(&attr, 0, -1, -1, 0);
    ioctl(perf_fd, PERF_EVENT_IOC_ENABLE, 0);
    for (int i = 0; i < N; i++) {}
    ioctl(perf_fd, PERF_EVENT_IOC_DISABLE, 0);
    end_nmi_passthrough();
    clock_gettime(CLOCK_MONOTONIC, &ts_end);
    ts_diff = diff(ts_start, ts_end);
    read(perf_fd, &count, sizeof(count));
    printf("count = %lld, time = %lds + %ldns\n", count, ts_diff.tv_sec, ts_diff.tv_nsec);
    close(perf_fd);
    return 0;
}
```

A.2 Hypercall to Test NMI Unblocking

```
static int hc_test_nmi_unblocking(void)
{
    volatile int i;
    u32 intr;
    u64 value, enabled, running;
    struct perf_event *event;
    struct perf_event_attr attr = {
        .type = PERF_TYPE_HARDWARE,
        .size = sizeof(sturct perf_event_attr),
        .config = PERF_COUNT_HW_BRANCH_INSTRUCTIONS,
        .exclude_user = 1,
        .disabled = 1,
        .sample_period = 1000
    };

    // Make sure NMI blocking is in effect
    intr = vmcs_read32(GUEST_INTERRUPTIBILITY_INFO);
    assert(intr & GUEST_INSR_STATE_NMI);
```

```c

    // Register sampling event
    event = perf_event_create_kernel_counter(&attr, -1, current, NULL, NULL);
    perf_event_enable(event);
    for (i = 0; i < 1001; i++) {}
    perf_event_disable(event);
    value = perf_event_read_value(event, &enabled, &running);
    pr_info("[Test NMI Unblock] count = %llu\n", value);

    return 0;
}
```

doi:10.6342/NTU202403471